
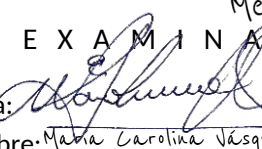
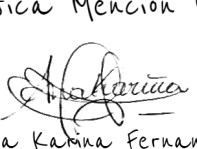


FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

FRAMEWORK PARA LA IMPLEMENTACIÓN DE
HERRAMIENTAS DE SOFTWARE DE SIMULACIÓN

Este Jurado; una vez realizado el examen del presente trabajo
ha evaluado su contenido con el resultado: Veinte (20) puntos

J U R A D O E X A M I N A D O R		
	Mención Honorífica	Mención Publicación
Firma: 	Firma: 	Firma: 
Nombre: <u>Rafael Lara</u>	Nombre: <u>María Carolina Vásquez</u>	Nombre: <u>Ana Karina Fernandes</u>

REALIZADO POR

Rolando José Andrade Fernández

PROFESOR GUÍA

Ing. María Carolina Vásquez García

FECHA

Octubre de 2021



**FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA**

**FRAMEWORK PARA LA IMPLEMENTACIÓN DE
HERRAMIENTAS DE SOFTWARE DE SIMULACIÓN**

TRABAJO ESPECIAL DE GRADO

Presentado ante la

UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Como parte de los requisitos para optar al título de

INGENIERO EN INFORMÁTICA

REALIZADO POR

Rolando José Andrade Fernández

PROFESOR GUÍA

Ing. María Carolina Vásquez García

FECHA

Octubre de 2021

*En un mundo donde todo parecía creado,
decidí crear nuevos mundos.*

Agradecimientos

Durante mi vida, mi carrera y el desarrollo de mi tesis, encontré a varias personas que me apoyaron y me llevaron hasta acá, estoy seguro que si empiezo a enumerar nombres y apellidos, alguno me falta, siempre falta, sin embargo, tengo el método perfecto para que no me falte ningún nombre, un agradecimiento personal que requiere de su introspección.

Si está leyendo esto y me apoyó en mi dura travesía, de verdad no tengo agradecimientos suficientes para darle, así que le prometo ser el mejor Ingeniero de la historia para que pueda decir con orgullo que estuvo ahí para mí, que llegué ahí gracias a usted; familiar, amigo o profesor, le estaré eternamente agradecido.

Rolando Andrade

Índice General

Sinopsis	xvi
Introducción	xvii
1. El problema	1
1.1. Planteamiento del problema	1
1.2. Solución propuesta	3
1.3. Objetivos	5
1.3.1. Objetivo general	5
1.3.2. Objetivos específicos	5
1.4. Alcance	6
1.5. Limitaciones	7
1.6. Justificación	8
2. Marco teórico	10
2.1. Marco conceptual	10
2.1.1. Sistemas	10
2.1.1.1. Componentes de los sistemas	11
2.1.1.2. Sistemas de eventos discretos y continuos	11
2.1.2. Modelos	12
2.1.2.1. Tipos de modelos	12
2.1.2.2. Clasificación de modelos de simulación	12
2.1.3. Simulación de eventos discretos	13
2.1.3.1. Representación del tiempo	13

2.1.3.2.	Programación de eventos y avance del tiempo	13
2.1.4.	Teoría de colas	14
2.1.4.1.	Elementos de un sistema de cola	14
2.1.4.2.	Notación	15
2.1.5.	Estructuras de datos	16
2.1.5.1.	Árbol	16
2.1.5.2.	<i>Heap</i>	16
2.1.5.3.	<i>Set</i>	17
2.1.5.4.	Arreglos asociativos	18
2.1.6.	Arquitectura de software orientada a objetos	18
2.1.7.	<i>Frameworks</i>	18
2.1.7.1.	Tipos de <i>frameworks</i>	19
2.1.7.2.	Componentes de un <i>framework</i>	19
2.1.7.3.	Inversión de control e inyección de dependencias	20
2.2.	Antecedentes	21
2.2.1.	Estudio de los sistemas como modelos	21
2.2.2.	Orígenes de la simulación por computadora	22
2.2.3.	Surgimiento de <i>frameworks</i> de simulación	23
2.2.4.	Implementación de herramientas de simulación	26
2.2.4.1.	DesmoJ	26
2.2.4.2.	SimPy	27
2.3.	Herramientas, lenguajes y <i>frameworks</i>	28
2.3.1.	Lenguajes de programación	28
2.3.1.1.	Java	28
2.3.1.2.	Python	29
2.3.1.3.	TypeScript	29
2.3.1.4.	C++	30
2.3.2.	Herramientas y <i>frameworks</i>	31
2.3.2.1.	Vue.js	31
2.3.2.2.	UML	31

2.3.2.3. Socket.IO	31
2.3.2.4. PyPI	31
2.3.2.5. DeepSource	32
2.3.2.6. Better Code Hub	32
2.3.2.7. Sphinx	32
3. Marco metodológico	33
3.1. Metodología de desarrollo	33
3.1.1. Etapa de análisis y planificación	34
3.1.1.1. Fase de investigación	34
3.1.1.2. Fase de definición del problema	35
3.1.1.3. Fase de definición del plan de desarrollo	35
3.1.2. Etapa de diseño general	35
3.1.2.1. Fase de definición de arquitectura	35
3.1.2.2. Fase de definición de estructuras de datos	35
3.1.2.3. Fase de definición de mecanismos de comunicación	35
3.1.3. Etapa de diseño y desarrollo	36
3.1.3.1. Fase de diseño detallado	36
3.1.3.2. Fase de implementación	36
3.1.3.3. Fase de instanciación y pruebas	36
3.1.4. Prototipo	37
3.1.5. Etapa de evaluación y validación	37
3.1.5.1. Fase de uso del <i>framework</i>	37
3.1.5.2. Fase de validación de resultados	37
3.1.6. Versión final	38
3.1.6.1. Despliegue del <i>framework</i>	38
3.1.6.2. Despliegue del simulador	38
3.1.6.3. Generación de página de referencia	38
3.1.7. Justificación de la metodología	38
3.1.7.1. Ciclo de vida	38
3.1.7.2. Enfoque inicial en la investigación	39

3.1.7.3.	Características del proyecto	39
3.1.7.4.	Adaptabilidad	39
3.2.	Metodología de gestión de trabajo	39
3.2.1.	Principios de Kanban	40
3.2.2.	Prácticas de Kanban	40
3.2.3.	Adaptación de la metodología	41
3.2.4.	Justificación de la metodología	42
3.2.4.1.	Carencias de la metodología de desarrollo	42
3.2.4.2.	Facilidad de implementación	42
3.2.4.3.	Herramientas existentes	42
3.3.	Metodología de evaluación y validación	42
3.3.1.	Evaluación del <i>framework</i>	43
3.3.1.1.	Comportamiento e inversión de control	43
3.3.1.2.	Cerrado a la modificación	43
3.3.1.3.	Seguimiento de buenas prácticas de Ingeniería de Software	44
3.3.2.	Validación de resultados	44
3.3.3.	Justificación de la metodología	44
3.3.3.1.	Objetividad y reproducibilidad	44
3.3.3.2.	Generalización	45
3.3.3.3.	Relación con los objetivos de la investigación	45
4.	Desarrollo de la investigación	46
4.1.	Definición de la investigación	46
4.1.1.	Etapas de análisis y planificación	46
4.1.2.	Etapas de diseño general	46
4.1.2.1.	Fase de definición de arquitectura	46
4.1.2.2.	Fase de definición de estructuras de datos	48
4.1.2.3.	Fase de definición de mecanismos de comunicación	48
4.2.	Desarrollo de los prototipos	50
4.2.1.	Prototipo 1. Autómata celular lineal	50

4.2.1.1.	Fase de diseño detallado	50
4.2.1.2.	Fase de implementación	51
4.2.1.3.	Fase de instanciación y pruebas	51
4.2.2.	Prototipo 2. Autómata celular en malla	51
4.2.2.1.	Fase de diseño detallado	51
4.2.2.2.	Fase de implementación	53
4.2.2.3.	Fase de instanciación y pruebas	54
4.2.3.	Prototipo 3. Sistema de fábrica de discos	54
4.2.3.1.	Fase de diseño detallado	54
4.2.3.2.	Fase de implementación	59
4.2.3.3.	Fase de instanciación y pruebas	60
4.2.4.	Prototipo 4. Simulador de fábrica de discos	60
4.2.4.1.	Fase de diseño detallado	60
4.2.4.2.	Fase de implementación	64
4.2.4.3.	Fase de instanciación y pruebas	65
4.2.5.	Prototipo 5. Interfaz gráfica de simulador de redes de cola	65
4.2.5.1.	Fase de definición de arquitectura	65
4.2.5.2.	Fase de diseño detallado	66
4.2.5.3.	Fase de implementación	68
4.2.6.	Prototipo 6. Emisor de entidades y fuente	70
4.2.6.1.	Fase de diseño detallado	70
4.2.6.2.	Fase de implementación	72
4.2.6.3.	Fase de instanciación y pruebas	73
4.2.7.	Prototipo 7. Servidor y distribuciones matemáticas	73
4.2.7.1.	Fase de diseño detallado	74
4.2.7.2.	Fase de implementación	75
4.2.7.3.	Fase de instanciación y pruebas	77
4.2.8.	Prototipo 8. Simulador red de colas (consola)	77
4.2.8.1.	Etapas de diseño general	77
4.2.8.2.	Fase de diseño detallado	78

4.2.8.3. Fase de implementación	80
4.2.8.4. Fase de instanciación y pruebas	81
4.2.9. Prototipo 9. Conexión con el <i>kernel</i>	82
4.2.9.1. Fase de definición de arquitectura	82
4.2.9.2. Fase de diseño detallado	83
4.2.9.3. Fase de implementación	84
4.2.10. Prototipo 10. Simulador final	85
4.2.10.1. Fase de implementación	85
4.2.10.2. Despliegue del <i>framework</i>	87
4.2.10.3. Despliegue del simulador	87
4.2.10.4. Generación de página de referencia	88
5. Resultados obtenidos	89
5.1. Análisis de resultados	89
6. Conclusiones y recomendaciones	95
6.1. Conclusiones	95
6.2. Recomendaciones	96
6.2.1. Recomendaciones sobre el <i>framework</i>	96
6.2.2. Recomendaciones sobre el simulador de red de colas	97
Referencias	99
Anexo A. Desarrollo	104
A.1. Evolución del algoritmo de enrutamiento	104
A.1.1. Prototipo 2. Autómata celular lineal	104
A.1.2. Prototipo 3. Sistema de fábrica de discos	105
A.1.3. Prototipo 8. Simulador de red de colas (consola)	107
A.2. Evaluación y validación de prototipo	108
A.2.1. Prototipo 1. Autómata celular lineal	108
A.2.1.1. Fase de uso del <i>framework</i>	108
A.2.1.2. Fase de validación de resultados	108

A.2.2. Prototipo 2. Autómata celular en malla	109
A.2.2.1. Fase de uso del framework	109
A.2.2.2. Fase de validación de resultados	109
A.2.3. Prototipo 3. Sistema de fábrica de discos	111
A.2.3.1. Fase de uso del framework	111
A.2.3.2. Fase de validación de resultados	111
A.2.4. Prototipo 4. Simulador de fábrica de discos	112
A.2.4.1. Fase de uso del framework	112
A.2.4.2. Fase de validación de resultados	112
A.2.5. Prototipo 6. Emisor de entidades y fuente	113
A.2.5.1. Fase de uso del framework	113
A.2.5.2. Fase de validación de resultados	113
A.2.6. Prototipo 7. Servidor y distribuciones matemáticas	114
A.2.6.1. Fase de uso del framework	114
A.2.6.2. Fase de validación de resultados	114
A.2.6.3. Segunda fase de validación de resultados	115
A.2.7. Prototipo 8. Simulador de red de colas (consola)	116
A.2.7.1. Fase de uso del framework	116
A.2.7.2. Fase de validación de resultados	117
A.2.8. Evaluación y validación de la investigación	121
A.2.8.1. Comportamiento e inversión de control	121
A.2.8.2. Cerrado a la modificación	122
A.2.8.3. Seguimiento de buenas prácticas de Ingeniería de Software	122
A.2.9. Validación de resultados	125
Anexo B. Juego de la vida	126
B.1. Definición	126
B.2. Leyes genéticas de Conway	126
B.2.1. Supervivencia	126
B.2.2. Fallecimiento	126

B.2.3. Nacimiento	126
Anexo C. Nuevos paradigmas y futuro de la investigación	127
C.1. Implementación de los <i>frameworks</i>	127
C.2. Computación paralela	128
C.3. Combinación de modelos de simulación	129
C.4. Inteligencia Artificial	130
Anexo D. Uso del <i>framework</i>	131
D.1. Instalación	132
D.2. Creación un autómatas celular lineal	132
D.2.1. Reglas	132
D.2.2. Implementación de las celdas	133
D.2.3. Implementación del autómatas	135
D.2.4. Corrida de la simulación	136
D.3. Creación del Juego de la Vida de Conway	137
D.3.1. Implementación de las celdas	137
D.3.2. Implementación del autómatas	138
D.3.3. Corrida de la simulación	140
D.4. Creación de un simulador de eventos discretos	140
D.4.1. Reglas	141
D.4.2. Creación de las estaciones	141
D.4.3. Creación del generador de piezas	144
D.4.4. Creación del sumidero	146
D.4.5. Creación de la fábrica	146
D.4.6. Diseño y simulación de la red	147
D.4.6.1. Añadir expresiones	148
Anexo E. Imágenes de la página de referencia	149
Anexo F. Imágenes del simulador	152

Índice de Tablas

4.1.	Propiedades de la fuente	73
4.2.	Propiedades del servidor	76
A.1.	Validación de salidas a la tercera iteración.	108
A.2.	Salidas obtenidas	112
A.3.	Salidas obtenidas	113
A.4.	Resultados de los experimentos	113
A.5.	Resultados generales de los experimentos	114
A.6.	Muestras de experimento 3	114
A.7.	Muestras de experimento 4	114
A.8.	Muestras de experimento 5	114
A.9.	Muestras de experimento 3	115
A.10.	Muestras de experimento 4	115
A.11.	Muestras de experimento 5	115
A.12.	Prueba de medias para muestras no apareadas de experimento 3	115
A.13.	Prueba de medias para muestras no apareadas de experimento 4	116
A.14.	Prueba de medias para muestras no apareadas de experimento 5	116
A.15.	Vectores de entrada de simulación de red sencilla	117
A.16.	Vectores de entrada de simulación de rutas booleanas	117
A.17.	Vectores de entrada de simulación de selección de rutas	118
A.18.	Vectores de entrada de simulación de elementos en paralelo	118
A.19.	Muestras de simulación de red sencilla	118
A.20.	Muestras de simulación de rutas booleanas en sumidero 1 y 2	119
A.21.	Muestras de simulación de selección de rutas en sumidero 1 y 2	119

A.22. Muestras de simulación de elementos en paralelo en sumidero 1 y 2	119
A.23. Prueba de medias para muestras no apareadas de simulación de red sencilla	119
A.24. Prueba de medias para muestras no apareadas de simulación de rutas booleanas	120
A.25. Prueba de medias para muestras no apareadas de sumidero 1 durante simulación de selección de rutas	120
A.26. Prueba de medias para muestras no apareadas de sumidero 2 durante simulación de selección de rutas	120
A.27. Prueba de medias para muestras no apareadas de sumidero 1 durante simulación de elementos en paralelo	121
A.28. Prueba de medias para muestras no apareadas de sumidero 2 durante simulación de elementos en paralelo	121

Índice de Ilustraciones

2.1.	Elementos de una cola	15
2.2.	Operaciones entre conjuntos	18
2.3.	Componentes de un <i>framework</i>	20
2.4.	Sim One. Uno de los primeros modelos híbridos	22
2.5.	Representación del sistema dinámico como mecanismo de transición de estado	23
3.1.	Diagramas de los modelos metodológicos empleados.	34
3.2.	Ejemplo de tablero Kanban	41
3.3.	Ejemplo de resultado de revisión de código	41
4.1.	Componentes del sistema	47
4.2.	Arquitectura del simulador.	47
4.3.	Resumen de etapa de diseño general	49
4.4.	Diseño de célula del autómatas	50
4.5.	Autómata celular lineal	51
4.6.	Diseño para el prototipo 2	52
4.7.	Ejemplo de red cíclica	53
4.8.	Diseño del prototipo 3.	55
4.9.	Extracción del evento inminente de la FEL	57
4.10.	Principales conjuntos del sistema dinámico	57
4.11.	Modelos afectados por una transición	58
4.12.	Modelo de línea de ensamblaje	60
4.13.	Diseño de control de simulación	61
4.14.	Diseño de motor de simulación y reportes	62

4.15.	Diagrama de secuencia (simplificado) del ciclo de simulación	63
4.16.	Componentes del simulador	65
4.17.	Módulos del <i>store</i>	66
4.18.	Diagrama de clases del modelador	66
4.19.	Vista de simulación de Simio	67
4.20.	Vista de resultados de Simio	68
4.21.	Representación de los elementos del simulador	69
4.22.	Vista inicial de simulación	69
4.23.	Propiedad de entidad	70
4.24.	Diseño de emisor de entidades	71
4.25.	Diseño de expresiones	71
4.26.	Extensión de los <i>buffers</i>	72
4.27.	Ejemplo de disciplinas	72
4.28.	Diseño de la cola de procesamiento	74
4.29.	Visualización de las distribuciones.	75
4.30.	Pasos en transición externa.	76
4.31.	Pasos en transición autónoma	76
4.32.	Nueva clase ruta.	79
4.33.	Algoritmo de selección por defecto	79
4.34.	Redes simuladas	81
4.35.	Arquitectura del simulador	82
4.36.	Diseño del <i>kernel</i> del simulador	83
4.37.	Listado de expresiones	85
4.38.	Vista de reportes	86
A.1.	Salida inicial	110
A.2.	Salida final	110
A.3.	<i>Checks</i> de la <i>release</i>	122
A.4.	Análisis del software de Better Code Hub	123
A.5.	Análisis del software de DeepSource.	124
D.1.	Imagen del cuaderno en Colaboratory.	131

D.2.	Autómata celular lineal	132
D.3.	Sistema de dos estaciones	140
E.1.	Página de inicio	149
E.2.	Página de información de proyecto	150
E.3.	Página de ejemplo de implementación	150
E.4.	Página de referencia de un módulo	151
E.5.	Fórmulas matemáticas de referencia	151
F.1.	Simulador luego de correr una simulación	152
F.2.	Barra de estado de una simulación	152
F.3.	Ajuste de velocidad de simulación	153
F.4.	Inspector de propiedades y expresiones	153
F.5.	Selector de tiempo de simulación	154
F.6.	Arrastre de componentes de simulación	154
F.7.	Creación de rutas	155
F.8.	Entidades en <i>buffer</i> en tiempo real	155
F.9.	Eliminación de componentes	156
F.10.	Simulación paso por paso	156
F.11.	Guardar el modelo diseñado	156
F.12.	Importar el modelo guardado	157
F.13.	Estadísticas de simulación	157

Sinopsis

La simulación permite experimentar con modelos que representan parte o la totalidad de un sistema. El estudio por modelos dio origen a los modelos matemáticos y estos a su vez a los modelos de simulación. La evolución de los sistemas de simulación, también ha significado un avance en la manera en la que se desarrollan, siendo los elementos comunes y repetitivos los que fomentaron la creación de *frameworks* para la implementación de herramientas de simulación.

La presente investigación busca dar una posible solución al problema causado por el costo de las licencias de los simuladores comerciales, la dificultad para pagarlas, la piratería generada por esas dificultades y la cantidad limitada de opciones que faciliten el desarrollo de herramientas de simulación enfocadas a un ámbito general.

Para ello se propuso la elaboración de un *framework* para la implementación de herramientas de simulación, con la definición de los módulos que forman parte del dominio de las simulaciones y la creación de un simulador de redes de cola que valide su utilidad.

La investigación fue llevada a cabo a través de una metodología basada en el desarrollo evolutivo, asumiendo prácticas y principios de Kanban para la gestión de trabajo, y haciendo uso de herramientas de software y estadística para evaluar y validar los resultados de la investigación.

Tras la realización de diez (10) prototipos funcionales, se logró obtener y validar un *framework* que se adaptara a los objetivos asumidos en la investigación, tomando en cuenta las limitaciones y alcance establecido.

Introducción

La simulación es un área apasionante que crece en torno a la constante evolución en los estudios de la matemática e informática con el fin de entender cada vez más el mundo y los sistemas que lo conforman.

Para la realización de simulaciones apoyadas por computadora se requiere del uso de simuladores, que se pueden adquirir o desarrollar dependiendo de las características intrínsecas de cada investigación y del equipo responsable de efectuarla.

A lo largo de la historia, la necesidad de implementar herramientas de simulación ha permitido encontrar patrones y elementos comunes que fomentan el establecimiento de un marco de trabajo para los mismos. La estandarización de los componentes de una simulación ha dado paso al desarrollo de *frameworks* que facilitan la realización de algunos procesos del dominio.

Los *frameworks* son herramientas clave en el desarrollo de sistemas informáticos, aumentando la productividad y disminuyendo tiempos de entrega mediante la reutilización de diseños y código.

La presente investigación busca desarrollar un *framework* para la implementación de herramientas de simulación, pues es una excelente manera de utilizar varios de los conocimientos pertenecientes a la Ingeniería Informática como la Ingeniería de Software o la Investigación de Operaciones, a la vez que se resuelve un problema que impacta en muchas otras áreas e individuos interesados en todo el mundo.

Con el fin de alcanzar este objetivo, se estudiaron los conceptos y fundamentos del área que llevaron a la definición de los problemas actuales y la propuesta de una solución a los mismos, que fue desarrollada a lo largo de todas las fases de la investigación.

En la investigación fue utilizada una metodología basada en el desarrollo evolutivo junto a principios y prácticas definidas en Kanban [1] para la gestión de trabajo. Por otra parte, para la evaluación y validación objetiva de la investigación, se definió una metodología con los dominios y factores que se consideraron relevantes para el estudio de los resultados, tales como las características del *framework* y la equivalencia de sus salidas con las obtenidas en otras herramientas o de manera teórica.

El trabajo se presenta organizado en seis capítulos, cada uno dedicado a una fase del estudio y en los que se desarrolla los siguientes contenidos:

- **Capítulo 1. El problema:** describe el problema abordado en la investigación y la solución propuesta al mismo. Define los objetivos, su alcance y limitaciones.
- **Capítulo 2. Marco teórico:** especifica los conceptos, herramientas e historia de proyectos similares en los que se fundamentan las decisiones y prácticas tomadas durante la investigación.
- **Capítulo 3. Marco metodológico:** detalla y justifica las metodologías empleadas, así como las fases y etapas que guiaron el desarrollo de la investigación.
- **Capítulo 4. Desarrollo de la investigación:** explica el proceso realizado y las decisiones tomadas a lo largo de la investigación que dieron origen a los resultados obtenidos.
- **Capítulo 5. Resultados obtenidos:** pormenoriza los resultados obtenidos a fin de analizarlos y justificar la razón de los mismos.
- **Capítulo 6. Conclusiones y recomendaciones:** expone las conclusiones a las que llegó la investigación, además de las recomendaciones que pueden apoyar la realización de futuros trabajos en áreas similares o con referencia a este.

Capítulo 1

El problema

En este capítulo se expone detalladamente el planteamiento del problema que dio origen a la solución propuesta. Una vez definido el problema y la solución, se enumeran los objetivos, se describe su alcance, además de presentar las limitaciones que se tomaron en la investigación. Finalmente, se justifica la relevancia e impacto de la investigación realizada en las áreas de estudio involucradas.

1.1. Planteamiento del problema

Una simulación es la imitación del funcionamiento de un proceso o sistema del mundo real a lo largo del tiempo con el fin de llegar a conclusiones que permitan identificar sus características [2]. La simulación ha ido evolucionando a lo largo de los años estableciéndose como un área de estudio muy importante en la actualidad, cuyas aplicaciones han revolucionado las metodologías de experimentación científica, permitiendo la demostración y el descubrimiento de soluciones a problemas complejos.

En el mercado existe una gran variedad de herramientas de simulación usadas en el ambiente académico, empresarial y científico como: Packet Tracer¹ para la simulación de sistemas de redes, Multisim² para la simulación de circuitos electrónicos, o Simio³

¹ Sitio web de Packet Tracer: <https://www.netacad.com/es/courses/packet-tracer>

² Sitio web de Multisim: <https://www.ni.com/es-cr/support/downloads/software-products/download.multisim.html>

³ Sitio web de Simio: <https://www.simio.com/>

para la simulación de una inmensurable cantidad de procesos y sistemas.

Así mismo, los simuladores suelen venir con restricciones de licencia a un costo y un uso específico, que limitan sus capacidades o incluso privan su uso. Por ejemplo, según Simio [3], el desembolso de las universidades por la adquisición sus licencias puede llegar a USD 120.000,00 anuales por el uso del software para un laboratorio de 50 personas, aunque por convenios se puede dar la subvención de parte o la totalidad de los mismos, bajo ciertas restricciones como la complejidad de los modelos o el uso exclusivo para la educación prohibiendo su utilización legal en el ámbito de investigación.

Las principales instituciones educativas de Venezuela se encuentran en una situación crítica presupuestaria que limitan su funcionamiento [4], por lo que es probable que se vean obligadas a prescindir de estas herramientas para el aprendizaje e investigación, dejando rezagadas sus competencias en el área.

Los costos de licencia no solo afectan a las universidades, sino que también a los estudiantes, pues pese a que la mayoría de los sistemas de simulación poseen períodos de prueba o funcionalidades limitadas, no son suficientes, por lo que se tiene que adquirir una licencia ya sea personal o de estudiante para realizar trabajos de estudio independiente fuera de las instalaciones de la universidad. Por ejemplo, la versión libre de Simio permite realizar modelos muy sencillos de manera gratuita, pero para la realización de proyectos de simulación más complejos la oferta de la empresa es una licencia de USD 25,00 anuales para estudiantes [3], lo cual es extremadamente costoso para la mayoría de las personas en Venezuela.

La necesidad de las funcionalidades ofrecidas por las herramientas en algunos casos incita a la piratería del software por parte de los estudiantes y profesores, pues a pesar de que se distribuya una copia legal entre ellos, la difusión de la misma ya viola de por sí los términos de la licencia, lo que implica un uso deshonesto de las herramientas que afectan a los desarrolladores del software y a la imagen de la institución, pudiendo llegar incluso a disputas legales [5].

Si las instituciones educativas empiezan a prescindir de las herramientas de simulación debido a sus costos, algunas empresas en el país también se verán

afectadas, ya que, les costará mucho más encontrar personal capacitado en el área.

Con la necesidad de abordar el área de la simulación desde un punto de vista económico, se podrían desarrollar herramientas de simulación propias de cada institución aplicando ingeniería inversa a las herramientas existentes y haciendo énfasis en las funcionalidades que se desean, sin embargo, los *frameworks* y librerías de código abierto existentes para el desarrollo de herramientas de simulación están destinadas a procesos específicos como DESMO-J, o están especificadas para un desarrollo a un muy bajo nivel como SimPy.

El problema de las herramientas destinadas a un fin específico es su dificultad para modificarlas para otros fines, lo que llevaría el uso de distintas herramientas para distintos proyectos. En el otro extremo, la falta de abstracción de una herramienta conlleva a la reinención de procesos de manera repetitiva y artesanal, dejando de lado la ingeniería que involucra el proceso de gestión y desarrollo de proyectos de tecnologías de la información.

El costo de licencias, la dificultad para pagarlas, la piratería generada por esas dificultades, aunado a la ausencia de opciones que faciliten el desarrollo de herramientas de simulación enfocadas a un ámbito general, sin llegar a carecer de una abstracción que permita la reutilización de módulos entre proyectos; conforma un problema de gran interés, debido a las consecuencias catastróficas que podría acarrear para el estudio, la práctica y la investigación de la simulación.

Ante esto, se puede apreciar la necesidad de investigar y desarrollar una alternativa que sirva como solución a los problemas planteados, surgiendo la iniciativa de investigación desarrollada en el presente proyecto.

1.2. Solución propuesta

Para dar respuesta a las necesidades establecidas en el apartado anterior, se propone desarrollar un *framework* para la implementación de herramientas de simulación.

El ambiente de desarrollo deberá estar construido siguiendo un paradigma orientado

a objetos utilizando Python 3 como lenguaje de programación. La inversión de control se prevé que no sea total, ya que se desea que el usuario puede elegir cuándo llamar a las utilidades provistas por el marco de trabajo, tomando el control durante la simulación, donde los distintos procesos pertenecientes al *framework* realizan las llamadas a la implementación —por parte del usuario— de los métodos abstractos definidos en el marco de trabajo propuesto.

A grandes rasgos, el dominio del software a implementar está conformado por cinco módulos: un módulo encargado de ejecutar los procesos de simulación, otro módulo responsable de definir el sistema dinámico, un módulo para definir y crear los modelos, un módulo para controlar las simulaciones, y un módulo para la generación de reportes. Adicionalmente, se espera la inclusión de un módulo compartido de utilidades no necesariamente propias del dominio tales como: definiciones de clases y funciones, librerías matemáticas y herramientas de monitoreo.

Al separar el entorno de desarrollo en módulos, se permite el reemplazo y la agregación de nuevos módulos con mayor facilidad. La división por funcionalidades del dominio de la simulación es bastante útil debido a la gran variedad de soluciones que se pueden adoptar para cada contexto y herramientas de simulación a desarrollar.

Siguiendo las órdenes del módulo de control, el módulo de simulación es el encargado de realizar las operaciones de cómputo del próximo estado de la simulación, guardando los resultados obtenidos con el apoyo del módulo de reportes. El cómputo de un nuevo estado surge de la conexión de modelos definidos y existentes bajo un mismo sistema dinámico, que invierte el control y llama a las funciones definidas por los usuarios en los modelos.

Puesto que el área es bastante compleja, el desarrollo del marco de trabajo está dirigido hacia la simulación de sistemas de eventos discretos, dando la oportunidad de construir varias herramientas de simulación de este tipo.

Para validar las facilidades y ventajas del entorno de trabajo, se propuso una prueba de implementación con un software de simulación de redes de colas. Dicho simulador estaría compuesto de dos partes, una interfaz gráfica y un *kernel*.

El *kernel* es la implementación de las funcionalidades que ofrece el ambiente de desarrollo, haciendo uso de las abstracciones y agregando funcionalidades orientadas a un simulador de redes de cola.

Por su parte, la interfaz de usuario estaría implementada haciendo uso de Vue 2 y TypeScript, lo que le permitiría funcionar en los navegadores más recientes. La aplicación debería estar en constante comunicación con el *kernel* por medio de *WebSockets* para la definición de modelos y corrida de simulaciones.

1.3. Objetivos

1.3.1. Objetivo general

Desarrollar un *framework* para la implementación de herramientas de software de simulación.

1.3.2. Objetivos específicos

1. Determinar la arquitectura, estándares, normas, patrones y tácticas a ser implementados por el ambiente de desarrollo.
2. Desarrollar un módulo de simulación.
3. Desarrollar un módulo de sistema dinámico.
4. Desarrollar un módulo de definición de modelos de simulación.
5. Desarrollar un módulo de control de simulaciones.
6. Desarrollar un módulo de reporte de simulaciones.
7. Desarrollar los mecanismos de comunicación y coordinación entre los módulos.
8. Desarrollar una aplicación de simulación que implemente las funcionalidades ofrecidas por el ambiente de trabajo.
9. Documentar las distintas funcionalidades del ambiente de desarrollo en una referencia para programadores.

1.4. Alcance

1. Determinar la arquitectura, estándares, normas, patrones y tácticas a ser implementados por el ambiente de desarrollo.

Se define el tipo de entorno de trabajo a implementar, la estructura y organización de los componentes, los patrones y tácticas de instanciación, comunicación y recuperación a implementar en la fase de desarrollo, y los paradigmas de simulación a utilizar como base de proyecto de simulación.

2. Desarrollar un módulo de simulación.

El módulo de simulación es el encargado de realizar las operaciones de cómputo del próximo estado con base en las reglas definidas por el sistema dinámico. No se considerarán cálculos en paralelo, debido a que estos implican algoritmos distintos de enrutamiento de las entradas hacia los modelos en red [6]. El simulador se limitará a los sistemas de eventos discretos, pues son los más comunes en el dominio del problema a resolver.

3. Desarrollar un módulo de sistema dinámico.

El módulo de sistema dinámico permite al usuario definir y agrupar las entidades, establecer las conexiones de entrada y salida entre las mismas, y contiene las funciones para la programación y consulta de eventos. Será utilizado por el módulo de simulación para llevar a cabo las transiciones de estado y obtener las salidas del sistema. Utilizará el paradigma de lista de eventos futuros.

4. Desarrollar un módulo de definición de modelos de simulación.

El módulo de definición de modelos es el encargado de la construcción de los mismos. Su fin es permitir al usuario crear y establecer las propiedades y relacionar las entidades pertenecientes al sistema dinámico. Desde el punto de vista del *framework*, las entidades serán vistas como nodos, y las rutas entre las mismas como conexiones; siendo el programador el responsable de extender

estos conceptos para ajustarse a distintos sistemas dinámicos e interfaces de usuario en la implementación de herramientas de simulación.

5. Desarrollar un módulo de control de simulaciones.

El módulo de control es el encargado de manipular el comportamiento de la simulación, como iniciar, detener, salir, aumentar y reducir frecuencia de la misma.

6. Desarrollar un módulo de reporte de simulaciones.

Se limitará a mostrar el histórico de las salidas de cada iteración obtenidos como resultado de los eventos simulados. Los datos se almacenan en estructuras de datos en memoria.

7. Desarrollar los mecanismos de comunicación y coordinación entre los módulos.

Los módulos deben comunicarse entre ellos para lograr realizar el proceso de simulación.

8. Desarrollar una aplicación de simulación que implemente las funcionalidades ofrecidas por el ambiente de trabajo.

La aplicación de simulación permite validar el potencial uso del ambiente de desarrollo y su alcance estará determinado por la creación, simulación y reporte de resultados de algunos modelos de redes de colas.

9. Documentar las distintas funcionalidades del ambiente de desarrollo en una referencia para programadores.

Se utilizarán herramientas de generación automática de documentación. El idioma empleado será el inglés.

1.5. Limitaciones

Algunas limitaciones que tienen un impacto en las decisiones a tomar en el desarrollo proyecto son:

- El *framework* está desarrollado en Python 3.9, haciendo uso de los tipos derivados de PEP 484.
- El estilo de código a emplear en el *framework* está guiado por las definiciones de PEP 8.
- El paradigma de programación empleado para el desarrollo del *framework* será orientado a objetos.
- El simulador está desarrollado en Vue 2 y TypeScript.
- El simulador requiere de conexión a internet y el uso de un navegador de escritorio basado en el motor de renderizado Chromium.
- El simulador solo implementará y simulará los componentes de tipo fuente, servidor, sumidero y ruta.
- La interfaz del simulador estará implementada en inglés.

1.6. Justificación

Un ambiente de desarrollo para la implementación de herramientas de software de simulación tiene la capacidad de ser una solución bastante interesante ante el problema planteado.

En la administración, ventas, procesos, infraestructura, auditoría, educación y un sinnúmero de ramas más, se utiliza la simulación y estadística para realizar proyecciones y evaluar el rendimiento de las actividades realizadas.

Para llevar a cabo las simulaciones los usuarios optan por adquirir un simulador o por desarrollar uno propio. Entre las ventajas de utilizar uno propio estaría la capacidad de adaptarlo específicamente a las necesidades requeridas e incluso poder ahorrar costos por licencias. Por otro lado, es posible que para proyectos muy específicos no exista el software de simulación requerido y el desarrollo de uno desde cero sea muy complicado, como por ejemplo un simulador de ambientes de

conducción para vehículos autónomos o un motor de videojuegos, tomando importancia el uso de los *frameworks*.

Los entornos de trabajo facilitan el desarrollo de aplicaciones de software relativamente estandarizadas, proporcionando el diseño y componentes auxiliares. Su uso permite la reducción de la complejidad del desarrollo de aplicaciones de software, aumenta la facilidad para realizar cambios, propicia el uso de una arquitectura predefinida, y ayuda a los desarrolladores a enfocarse en implementar las funcionalidades propias de su negocio [7].

Un entorno de desarrollo permite tener una misma base para realizar los distintos proyectos de simulación que puedan surgir, permitiendo reutilizar módulos y enfocándose solo en las nuevas funcionalidades. Un ejemplo de las ventajas que puede traer este marco de trabajo se puede ver en los problemas que atacan la mayoría de simuladores, ya que, la mayoría de las veces no requieren cambios en las operaciones de cómputo, reportes o control, lo que suele variar es la interfaz y el tipo de sistema que se desea simular, por lo que existe la posibilidad de volver a usar los otros módulos ya existentes sin necesidad de realizar ningún cambio, y los que cambiaron, no hace falta crearlos desde cero, sino que al formar parte de un ambiente de desarrollo están abiertos a la extensión para soportar las nuevas funcionalidades [6].

Pese a que la repetición es frecuente en las herramientas de simulación, hacer del *framework* completamente modular lo hace mucho más flexible a los cambios, pudiendo adaptar al ambiente de desarrollo a otros contextos y soluciones, como lo puede ser un módulo de simulación que realice operaciones paralelas en vez de secuenciales, un módulo de reporte que almacene en una base de datos en lugar de estructuras en memoria, y un sinfín de posibilidades que otorga la abstracción del ambiente.

El uso de un *framework* que pueda ser usado para próximos proyectos contribuye al seguimiento de buenas prácticas de Ingeniería de Software, generando productos estandarizados, robustos y repetibles, que ayuden en un futuro al estudio e investigación en el área de la simulación.

Capítulo 2

Marco teórico

En este capítulo se describen los conceptos, herramientas y la historia de proyectos similares que fundamentan las decisiones y prácticas tomadas durante el desarrollo de la investigación.

2.1. Marco conceptual

Para entender algunos términos y decisiones tomadas en el desarrollo del proyecto, se presentan a continuación algunas definiciones relevantes para la investigación.

2.1.1. Sistemas

Un sistema es “conjunto de cosas que ordenadamente relacionadas entre sí contribuyen a determinado objeto” [6].

Es común que los sistemas sean afectados por el ambiente donde se encuentran. Para el estudio de sistemas se crea un límite del sistema, donde se agrupan todos los factores en factores propios y externos. Un mismo factor, puede ser propio o externo para un mismo sistema dependiendo del límite del sistema establecido en el caso de estudio [2].

2.1.1.1. Componentes de los sistemas

Con base en lo establecido por Banks et al. [2], los sistemas están compuestos por entidades, actividades, estados y eventos.

Las entidades son los objetos de interés de estudio en el sistema. Son elementos únicos definidos por atributos, expresados como variables locales. Una colección de entidades de un sistema, puede ser un subconjunto de entidades pertenecientes a otros sistemas. Una entidad puede ser dinámica si se mueve a través del sistema, o estática si se encarga de servir a otras entidades [2][8].

Una actividad representa un período de tiempo de una longitud determinada en la que ocurren los eventos [2]. Una actividad puede ser detenida antes de su ejecución por un tiempo específico llamado retraso. La suma de actividades y retrasos, se denomina eventos [8].

Por su parte, el estado de un sistema es una colección de variables que permiten describir al sistema en cualquier momento. La determinación de las variables de estado es una de las principales metas de la simulación. Dependiendo de la naturaleza de los cambios de la variable de estado, un sistema puede ser discreto o continuo [8].

Finalmente, un evento es una ocurrencia instantánea que puede llegar a afectar el estado del sistema. Los eventos pueden ser endógenos si son propios del sistema, o exógenos si son provenientes del ambiente [8].

2.1.1.2. Sistemas de eventos discretos y continuos

Los sistemas de eventos se pueden categorizar como discretos o continuos. Un sistema discreto es uno en el que el estado de sus variables cambia de manera instantánea solo en un conjunto discretos de puntos en el tiempo mientras que un sistema continuo es uno en el que el estado de sus variables cambia continuamente a lo largo del tiempo [2].

2.1.2. Modelos

El estudio de sistemas puede realizarse experimentando con el sistema completo, pero a veces puede resultar complicado o incluso imposible, por lo que se suele construir un modelo que represente al sistema que se desea estudiar de manera fiel. Crear un modelo, permite estudiar el sistema en caso de que no exista, sea complicado o costoso experimentar con él [2].

2.1.2.1. Tipos de modelos

Los modelos pueden ser matemáticos o físicos.

Banks et al. plantea que los modelos matemáticos son representados por notación simbólica y ecuaciones matemáticas. Los modelos de simulación son un caso particular de modelos matemáticos [2].

Por su parte, los modelos físicos son representaciones reales de una parte o la totalidad de un sistema. Ejemplos de estos modelos pueden ser modelos a escala o pruebas en túneles de viento [9].

Para hallar solución a los modelos matemáticos, se puede realizar un estudio analítico de las ecuaciones o crear un modelo de simulación [9].

2.1.2.2. Clasificación de modelos de simulación

Según su tipo, un modelo de simulación puede ser estático o dinámico, estocástico o determinístico.

Un modelo es estático si representa al sistema en un punto específico en el tiempo, mientras que se califica como dinámico si representa como cambia el sistema sobre el tiempo [2].

Por otro lado, el modelo es determinístico si no posee variables aleatorias, por lo que todas sus entradas son conocidas y todas sus salidas son constantes. Mientras tanto, un modelo se categoriza como estocástico en caso de que posea al menos una variable aleatoria generando así salidas aleatorias [2].

2.1.3. Simulación de eventos discretos

“Una simulación es una imitación de una operación de un proceso o sistema del mundo real a través del tiempo. Sin importar que se haga a mano o a computadora, la simulación incurre en la generación de un histórico artificial de un sistema y en la observación de ese histórico artificial para bosquejar inferencias respecto a las características de operación del sistema real” [2].

La simulación de eventos discretos es el modelo de un sistema en el que el estado de sus variables cambia solo en un conjunto de puntos discretos de tiempo donde hay la ocurrencia de algún evento [2].

2.1.3.1. Representación del tiempo

El reloj de simulación es una variable de unidad implícita, que indica el valor actual del tiempo simulado. Su unidad es la misma que la de sus entradas [9].

Para avanzar el reloj de simulación, existen dos métodos ampliamente extendidos, el avance por una unidad de tiempo fija, y el avance por el tiempo del próximo evento [9].

2.1.3.2. Programación de eventos y avance del tiempo

Los eventos tienen un tiempo en el que la simulación los debe ejecutar. Banks et al. [2] propone agrupar al conjunto de eventos en una lista llamada lista de eventos futuros¹ (FEL).

La lista de eventos futuros está ordenada por los tiempos de ejecución de los eventos de manera cronológica. El primer evento de la lista, se llama evento inminente. El reloj avanza saltando por los tiempos programados a su vez que se ejecutan los eventos siguiendo el siguiente algoritmo² [2]:

1. Remover de la lista el evento inminente.
2. Igualar el reloj a el tiempo del evento inminente.

¹ Llamada en inglés *Future Event List*.

² El nombre del algoritmo es: *Event Scheduling/Time Advance Algorithm*.

3. Ejecutar el evento inminente. Modificar el estado del sistema o las propiedades de las entidades.
4. Programar nuevos eventos si es necesario y agregarlos a la FEL.
5. Actualizar las estadísticas.

2.1.4. Teoría de colas

“La teoría de colas es una disciplina, dentro de la investigación operativa, que tiene por objeto el estudio y análisis de situaciones en las que existen entes que demandan cierto servicio, de tal forma que dicho servicio no puede ser satisfecho instantáneamente, por lo cual se provocan esperas” [10].

2.1.4.1. Elementos de un sistema de cola

Según Abad [10] un sistema de cola posee los siguientes elementos:

- Una fuente que representa al conjunto de individuos que pueden llegar a solicitar el servicio.
- El cliente que representa a todo individuo de la fuente que solicita servicio.
- La cola como representación de los clientes que hacen espera. La cola posee una capacidad máxima de clientes y una disciplina. La disciplina de una cola puede ser:
 - FIFO: el primero en llegar es el primero en ser atendido.
 - LIFO: atiende primero al cliente que llegó de último.
 - RSS: selecciona al cliente aleatoriamente.
 - RR: cada cliente recibe un tiempo de servicio de manera secuencial.
- El mecanismo de servicio es el procedimiento por el cual se le da servicio a los clientes que lo solicitan.

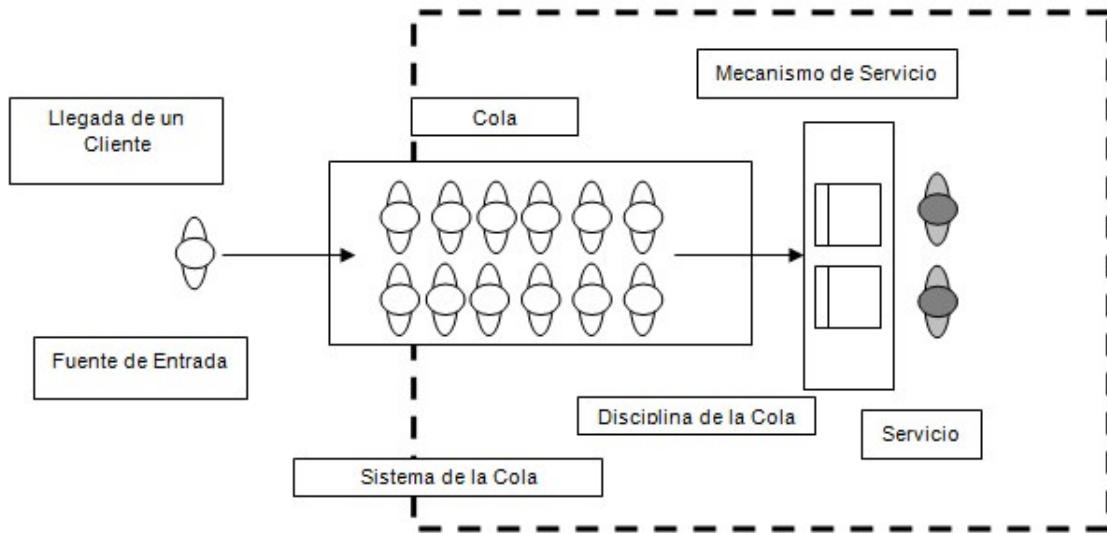


Figura 2.1: Elementos de una cola. Extraído de Santiago [11].

2.1.4.2. Notación

A continuación se presenta la notación de los conceptos más importantes de sistemas de colas usados para la investigación [10]:

- L : cantidad media de clientes en el sistema.
- L_q : cantidad media de clientes en la cola.
- W : tiempo medio de clientes en el sistema.
- W_q : tiempo medio de clientes en la cola.
- A : distribución del tiempo entre llegadas.
- B : distribución del tiempo de servicio.
- K : capacidad de la cola.
- H : tamaño de la fuente.
- Z : disciplina de la cola.

2.1.5. Estructuras de datos

Algunas estructuras de datos que influyeron en el diseño de la solución propuesta se detallan en el siguiente apartado.

2.1.5.1. Árbol

Un árbol es una estructura de datos abstracta y no lineal que almacena los elementos de manera jerárquica. Está definido por un conjunto de nodos con una relación padre/hijo con las siguientes propiedades [12]:

- Si el árbol T no está vacío, tiene un único nodo que no tiene padre, llamado nodo raíz de T .
- Para cada nodo v de T distinto del nodo raíz, tiene un único nodo padre w , y se dice que cada nodo con padre w es hijo de w .

Adicionalmente, existen otras relaciones entre nodos. Los nodos con el mismo padre se denominan nodos hermanos, y si un nodo no posee hijos se dice que es un nodo externo u hoja [12].

Un árbol binario es un árbol ordenado que se caracteriza por tener nodos que como máximo posean dos hijos llamados hijo izquierdo e hijo derecho, donde el hijo izquierdo precede en orden al hijo derecho [12].

2.1.5.2. Heap

Un *heap* o montículo es una estructura de datos conformada por un arreglo de objetos y que puede verse como un árbol binario casi completo³ [13].

Existen dos tipos de *heap*, los *min-heap* y *max-heap*, dependiendo del cumplimiento por parte de los nodos del árbol de una propiedad de satisfacción dada por una desigualdad [13]. En el caso de un *min-heap* A la propiedad de satisfacción para cada nodo i es:

³ Casi todos los nodos poseen cero o dos hijos

$$A[PADRE(i)] \leq A[i] \quad (2.1)$$

Debido a esta propiedad, se sabe que el nodo raíz siempre será el que tenga menor valor entre los nodos dentro del *heap*.

Los *heaps* son particularmente apropiados para las colas de prioridad puesto que cumple con las siguientes ventajas [6]:

- El número máximo de entradas del *heap* es conocido y equivalente al número de modelos atómicos en el sistema, por lo que se puede implementar dentro de un arreglo de dimensiones determinadas.
- Se reduce el costo computacional de mover elementos dentro de la estructura de datos.

2.1.5.3. Set

Los *sets* o conjuntos son estructuras de datos caracterizadas por ser una colección de elementos distinguibles entre sí [13]. Esta propiedad permite evitar repetidos en una lista.

Con base en lo documentado por Cormen et al. [13], los conjuntos permiten operaciones de unión, intersección y diferencia:

La intersección entre un conjunto A y B es el conjunto:

$$A \cap B = \{x : x \in A \text{ y } x \in B\} \quad (2.2)$$

La unión entre un conjunto A y B es el conjunto:

$$A \cup B = \{x : x \in A \text{ o } x \in B\} \quad (2.3)$$

La diferencia entre un conjunto A y B es el conjunto:

$$A - B = \{x : x \in A \text{ y } x \notin B\} \quad (2.4)$$

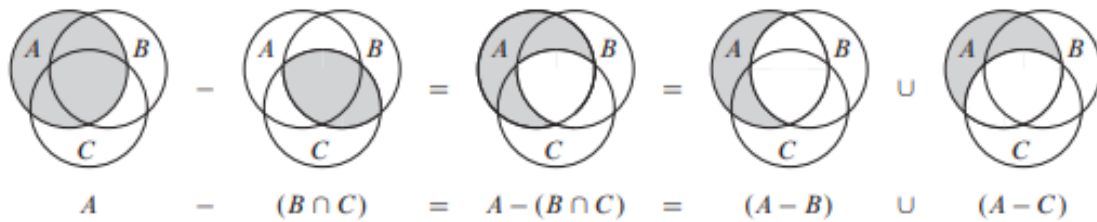


Figura 2.2: Operaciones entre conjuntos. Extraído de Cormen [13].

2.1.5.4. Arreglos asociativos

Los arreglos asociativos o *maps*, son estructuras de datos abstractas en las que una única clave está asociada a algún valor. Su implementación viene dada por los diccionarios también conocidos como tablas *hash* [12].

En este tipo de estructura el orden de los elementos no es de principal relevancia. Su principal ventaja radica en su facilidad de uso y rapidez de sus métodos, dando el mejor tiempo de ejecución para los casos esperados⁴ [12].

2.1.6. Arquitectura de software orientada a objetos

La arquitectura de software orientada a objetos está basada en clases y objetos como principales bloques primitivos de construcción de software. Los objetos son instancias de las clases y por lo tanto, representan instancias de un concepto de dominio [14].

2.1.7. Frameworks

Un *framework*, ambiente de desarrollo, entorno de desarrollo, marco de trabajo o entorno de trabajo; modela un dominio específico o un aspecto importante del mismo [14].

Los ambientes de desarrollo representan el dominio a través de un diseño abstracto, basado en clases abstractas o interfaces, definiendo además cómo las

⁴ $O(1)$ como tiempo de ejecución en notación *big-Oh*

mismas colaboran entre sí durante el tiempo de ejecución. Actúa como un esqueleto que determina cómo los objetos se relacionan unos con otros [14].

El principal objetivo de los *frameworks* es permitir la reutilización y así aumentar la productividad, proveyendo clases abstractas e implementaciones concretas, con la definición de primitivas que delegan la implementación crítica a las subclases [14].

2.1.7.1. Tipos de *frameworks*

Según Riehle [14] dependiendo de su forma de uso, un ambiente de desarrollo puede ser extendido como caja blanca o usado como caja negra.

Los *frameworks* son de caja negra cuando su uso viene dado por la composición de objetos, mientras que es de caja blanca cuando se extiende por herencia de clases. A su vez, el mismo autor afirma que la mayoría de los ambientes de desarrollo son en realidad de caja gris al mezclar ambos métodos para su uso.

2.1.7.2. Componentes de un *framework*

Los principales componentes de un ambiente de desarrollo son las ranuras, métodos de anclaje y los puntos congelados, definidos de la siguiente forma [7]:

Las ranuras o *hot spots* son “clases o métodos abstractos que deben ser ejecutados o puestos en ejecución. Estos lugares indican donde puede extenderse el *framework* por los usuarios del mismo”.

Los métodos de anclaje o *hook methods* son “elementos públicos de los *framework* que representan recursos sobre los que los clientes pueden construir métodos. Constituyen las operaciones a los que el usuario tendrá acceso, para la utilización de las clases concretas aportadas por el *framework*”.

Los puntos congelados o *frozen spots* son “elementos inmutables, pertenecientes a la estructura interna o esqueleto. Es código puesto en ejecución que llamará a uno o más puntos calientes proporcionados por el ejecutor. Constituirán por tanto el núcleo, infraestructura y diseño interno del marco de trabajo.”

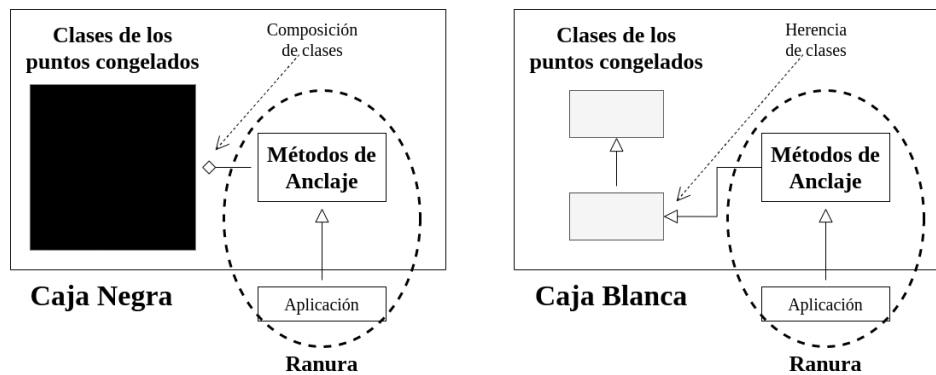


Figura 2.3: Componentes de un *framework*. Adaptado de Parsons et al. [15].

2.1.7.3. Inversión de control e inyección de dependencias

Según Fowler [16] la inversión de control (IoC) es un fenómeno típico al extender un *framework*, siendo una de las principales características de los mismos.

El mismo autor resalta que durante la ejecución del software, los métodos implementados por el usuario, son usados por el *framework* en vez de la aplicación, invirtiendo el control, como si el mismo *framework* fuera el programa principal.

Por otra parte, la inyección de dependencias es un patrón donde las clases trabajan con métodos pertenecientes a abstracciones cuyas implementaciones son dadas por un ensamblador⁵, que se encarga de proveer (inyectar) la dependencia correcta a cada una por interfaz, constructor o por métodos de asignación [17].

Existe una confusión entre la inversión de control y la inyección de dependencias debido al uso extendido de los contenedores IoC⁶, que tienen la tarea de inyectar dependencias, cuando la inversión de control va más allá [17].

El concepto de inversión de control es demasiado general y ha sido motivo de discusión a lo largo del tiempo ante la duda de los aspectos de software cuyo control es recomendable invertir [16].

La inyección de dependencias es una forma de conseguir la inversión de control, ya que permite trabajar con abstracciones de implementación hecha por el usuario [16].

⁵ Cualquier herramienta que genere instancias, como puede ser a través de *plugins* por un contenedor IoC o la instanciación directa en el mismo programa principal.

⁶ *Frameworks* encargados de gestionar las dependencias de los módulos de un proyecto.

No obstante, no es la única manera de conseguir inversión de control. Una alternativa es la creación de una clase plantilla, donde una superclase define el flujo del programa y las subclasses sobrescriben los métodos o implementan los métodos abstractos que la superclase llama [16].

2.2. Antecedentes

Los antecedentes de la investigación van desde los orígenes de la simulación, hasta el desarrollo de *frameworks* y librerías que se usan en la actualidad para la implementación de herramientas de simulación y que sirvieron de inspiración o base.

2.2.1. Estudio de los sistemas como modelos

Es común que los sistemas sean afectados por el ambiente donde se encuentra. Para el estudio de sistemas se crea un límite del sistema, donde se agrupan todos los factores en factores propios y externos. Un mismo factor puede ser propio o externo para un mismo sistema dependiendo del límite del sistema establecido en el caso de estudio [2].

Dependiendo de la naturaleza de los cambios de estado, los sistemas han podido ser catalogados como discretos o continuos, pese a que en la naturaleza es casi imposible encontrar alguno que sea perteneciente completamente a uno de estos grupos [8].

Ante el problema de estudiar los sistemas complejos, con el tiempo se empezaron a desarrollar sistemas más sencillos que los representaran de manera fiel, surgiendo los modelos. Crear un modelo, ha permitido a lo largo de la historia estudiar los sistemas en caso de que no existan o sea costoso experimentar con ellos [2].

El estudio por modelos dio origen a los modelos matemáticos y físicos, donde los modelos matemáticos son representados por notación simbólica y ecuaciones matemáticas [2], mientras que los físicos son representaciones reales de una parte o la totalidad de un sistema⁷ [9].

⁷ Como lo pueden ser maquetas, simulaciones por túneles de viento, colisionador de partículas, etc.

2.2.2. Orígenes de la simulación por computadora

Mucho antes de que existieran las computadoras digitales, computadoras analógicas fueron utilizadas para ayudar a los cálculos en física y matemática. Con el tiempo, tras muchos avances tecnológicos durante los siglos XIX y XX, las computadoras fueron mejorando en su rapidez y precisión en la realización de operaciones complejas, siendo las necesidades militares de la Segunda Guerra Mundial, las que finalmente aceleraron el paso de las computadoras análogas a digitales, iniciando la Era de la Información que se extiende desde 1970 hasta el presente [18].

Los orígenes de la simulación por computadora se remontan a las simulaciones de vuelo, ya que, a principio del siglo XIX el 90 % de las catástrofes aéreas eran responsabilidad del piloto. De esta forma, en 1931 se patenta un dispositivo de entrenamiento para aviadores⁸, siendo el primer simulador con una utilidad, logrando entrenar a más de 10.000 soldados de los aliados durante la Segunda Guerra Mundial en operaciones de vuelo y aterrizaje con clima desfavorable [18].

En cuanto a la guerra, desde la antigüedad el hombre desarrolló juegos de guerra, como medio de estrategia y simulación. Siendo el escenario bélico de mediados del siglo XX, el detonante para la realización de simuladores computarizados [19].

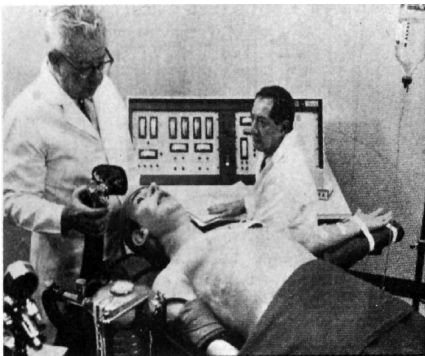


Figura 2.4: Sim One fue uno de los primeros modelos híbridos. Extraído de [20].

Pese a que la medicina se valió durante muchos años de modelos físicos como maniqués y estructuras anatómicas, a finales de los años 60 fue pionera en la simulación de alta precisión, surgiendo el primer simulador de anestesia Sim One, que era un maniquí dirigido por computadora a escala real representando a un paciente. Los futuros avances en el área se trasladaron a otras áreas de la salud como la veterinaria y la odontología [18][19].

A medida que la tecnología fue avanzando, más métodos y procesos fueron trasladados a entornos virtuales o basados por

⁸ El nombre de la patente fue *Combination Training Device for Student Aviators and Entertainment*, registrada en septiembre de 1931.

computadora [19], dando origen a simuladores de todo tipo y tareas, como el estudio del cosmos, simulación de partículas, análisis de procesos productivos e incluso desarrollo de videojuegos.

2.2.3. Surgimiento de *frameworks* de simulación

La flexibilidad de las herramientas de simulación, hizo que se incluyera en dominios del conocimiento muy variados, lo que dio origen a distintos desarrollos.

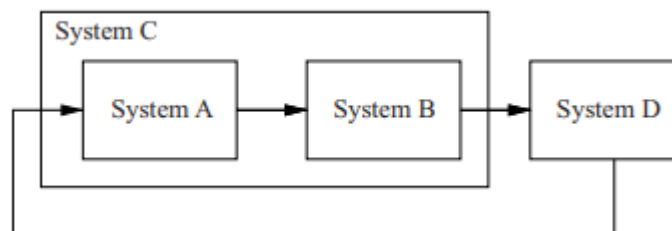
Los conceptos de modelado monolítico de simuladores que fueron empleados en las primeras herramientas de simulación, cambiaron de paradigma rápidamente a uno modular debido a la necesidad de dividir las operaciones complejas en piezas manejables para tratar los problemas uno a la vez, además de permitir la reutilización de parte o la totalidad de los componentes del modelo a desarrollar [6].

A medida que se desarrollaban más herramientas de simulación, se descubrió que todos tenían elementos en común, pudiendo ser representados principalmente por un sistema dinámico, un software para construir los modelos, un motor de simulación para computar las entradas y salidas de la simulación, y un medio de control y observación de simulaciones [6].

La caracterización del sistema dinámico es el marco conceptual que engloba a todos los *frameworks* de simulación, pues es el conjunto de reglas para determinado dominio.



(a) Modelo atómico.



(b) Modelo de red.

Figura 2.5: Representación del sistema dinámico como mecanismo de transición de estado realizada por Nutaro [6].

Al lograr abstraer el concepto de sistema dinámico a través de un mecanismo de transición de estado para el sistema como el de la Figura 2.5.a, y entenderse que la interconexión de modelos en red del sistema es invariante a sus dinámicas, se ha podido crear un mismo marco de trabajo para los distintos tipos de simulación existentes [6].

En la Figura 2.5.b se puede ver como un modelo de red, tiene el mismo comportamiento que un modelo atómico (mostrado en la Figura 2.5.a): entrada, estado y salida. Donde el sistema C, comprende los sistemas A y B, dando una salida hacia D, que también es una de sus entradas, que enruta a A. Al no depender de la dinámica, los modelos pueden ser discretos o continuos sin implicar un cambio en el *framework*.

Con esa base conceptual, se puede establecer las siguientes definiciones para la simulación de eventos discretos:

- Estado s : conjunto de variables que permiten describir al sistema.
- Llegada de evento e : tiempo en el que se realiza un cambio sobre el estado. Puede ser externo si el evento proviene de otro sistema, o autónomo si surge del mismo.
- Trayectoria X : entrada recibida por el sistema.
- Trayectoria Y : salida del sistema.
- Función de salida λ : mapea el estado del sistema en una trayectoria de salida Y .

$$\lambda : S \longrightarrow Y \quad (2.5)$$

- Función de avance de tiempo $ta(s)$: tiempo en el que el sistema emite un evento autónomo.

$$ta : S \longrightarrow R_0^\infty \quad (2.6)$$

$$R_0^\infty = \{x : x \in \mathbb{R} \cap x \geq 0\} \cup \infty$$

- Transición interna o autónoma $\delta_{int}(s)$: cambia el estado s con la llegada de un evento autónomo a un estado s' .

$$\delta_{int} : S \longrightarrow S' \quad (2.7)$$

- Estado total actual (s, e) : un sistema permanece en un estado s hasta que transcurre $ta(s)$ o hasta que una entrada intermedia en tiempo e lleve al sistema a un nuevo estado.
- Conjunto de estado total actual Q : es el conjunto de los distintos estado total actual (s, e) que han sido efectuados antes de $ta(s)$.

$$Q = \{(s, e) : s \in S \cap 0 \leq e \leq ta(s)\} \quad (2.8)$$

- Transición externa $\delta_{ext}((s, e), x)$: cambia el estado total actual del sistema (s, e) por la llegada de un evento externo con una trayectoria X .

$$\delta_{ext} : Q \times X \longrightarrow S' \quad (2.9)$$

- Transición confluyente $\delta_{con}(s)$: caso especial de una transición externa, donde el tiempo de llegada e equivale al tiempo de avance de tiempo $ta(s)$, por lo que se ejecutan la transición interna y externa a la vez.

$$\delta_{con} : S \times X \longrightarrow S' \quad (2.10)$$

2.2.4. Implementación de herramientas de simulación

En cuanto a ambientes de desarrollo, lenguajes y librerías previamente desarrolladas para la implementación de herramientas de simulación se puede destacar la influencia de DesmoJ y SimPy para el desarrollo del proyecto.

2.2.4.1. DesmoJ

DesmoJ es un conocido ambiente de desarrollo de simulación de eventos discretos desarrollado en Java. Entre las ventajas que provee están [21]:

- Abstracciones de modelo, entidad, evento y proceso.
- Componentes como colas, muestreo de distribuciones y contenedores de datos.
- Elementos de infraestructura como *scheduler*, lista de eventos y reloj.

Para la ejecución de simulaciones, se define un modelo y un experimento. En el modelo se describe el sistema y en el experimento los elementos de infraestructura.

Con base en Alaggia [21] el algoritmo de simulación usado por el *framework* sigue los siguientes pasos:

1. Se crea una instancia del modelo a simular⁹.
2. Se crea una instancia de experimento.
3. Se conecta el modelo con el experimento.
4. Desde el modelo se invoca la operación de inicio del experimento.
 - 4.1. El experimento invoca el método para programar los primeros eventos del modelo.
 - 4.2. Se invocan varios métodos intermedios para iniciar la simulación.
 - 4.3. Procesa el siguiente evento o proceso de la simulación mientras sea posible.

⁹ La clase primero se debe definir a través de la extensión del modelo abstracto.

2.2.4.2. SimPy

Por otro lado, Briceño [22] tras analizar los antecedentes de los distintos ambientes de desarrollo y paquetes orientados a resolver problemas de simulación, indica que estos son muy costosos y difíciles de modificar, por lo que sugiere el uso de SimPy, que es gratis y al ser de código abierto, es fácil de extender hacia nuevas funcionalidades.

El propio proyecto de SimPy [23] se define como un “marco de simulación de eventos discretos basado en Python estándar”. Se basa en un paradigma funcional, proveyendo al programador de una gran cantidad de recursos para hacer software de simulación de manera muy rápida.

SimPy está compuesto por módulos de simulación, trazas, velocidad de la simulación, realización de simulación paso a paso, gráfica de resultados, interfaces gráficas y formato de visualización [22].

Uno de los algoritmos posibles para realizar una simulación es el siguiente [22]:

1. Importar la librería de funciones.
2. Definir una clase de procesos.
3. Formular un modelo.
 - 3.1. Inicializa las funciones de ejecución de SimPy.
 - 3.2. Genera una o más instancias de la clase de procesos definida.
 - 3.3. Activa las instancias.
 - 3.4. Comienza la ejecución de la simulación.
4. Definir los valores de las entradas de la simulación.
5. Ejecutar el experimento.
6. Analizar los resultados.

2.3. Herramientas, lenguajes y *frameworks*

A continuación se presenta una breve descripción de las principales herramientas, lenguajes y *frameworks* que influyeron en la investigación.

2.3.1. Lenguajes de programación

Java, Python, TypeScript y C++ son los lenguajes de programación más usados a lo largo de la carrera, por lo que fueron objeto de investigación con el fin de seleccionar el mejor para el desarrollo del proyecto.

2.3.1.1. Java

Java es un lenguaje multipropósito de alto nivel, basado en clases y orientado a objetos. Las aplicaciones compiladas con el lenguaje generan unos archivos llamados *bytecode* que en teoría pueden correr en cualquier máquina virtual de Java (JVM) [24].

Entre las ventajas del lenguaje se encuentran:

- Portabilidad casi garantizada al correr en la JVM.
- Gran cantidad de librerías.
- Tipado fuerte.

No obstante, el lenguaje posee ciertos inconvenientes:

- No es posible acceder a la documentación y sitios oficiales de consulta y descarga de Oracle desde Venezuela.
- Sus *frameworks* para la creación de interfaces gráficas son difíciles de extender para lograr vistas y componentes complejos.
- Al ser basado en clases se limita su flexibilidad de uso, pudiendo llegar a complicar el diseño del software de manera forzada.

2.3.1.2. Python

Python es un poderoso lenguaje de programación interpretado¹⁰. Tiene estructuras de datos de alto nivel eficientes y soporta el desarrollo por el paradigma de programación orientado a objetos. Tiene una sintaxis elegante que lo hacen ideal para la realización de *scripts* y aplicaciones para la mayoría de plataformas [25].

Algunas ventajas relevantes para el proyecto son:

- Entre los lenguajes elegidos es el que menos vocabulario, longitud de programa, tiempo estimado de desarrollo, volumen de código, esfuerzo y tiempo de programación requiere, según el análisis comparativo de Abdulkareem y Abboud [26].
- Gran cantidad de librerías.
- Permite delegar funciones que requieren mejor uso de recursos y menores tiempos de ejecución a funciones y estructuras escritas en C.

Por otro lado, las falencias más evidentes son:

- Es relativamente lento en tiempos de ejecución.
- Excesivo uso de memoria por los tipos y estructuras de datos que define por defecto.
- Librerías de interfaces gráficas deficientes.

2.3.1.3. TypeScript

JavaScript (JS) es un lenguaje de programación ligero, interpretado, o compilado justo-a-tiempo¹¹ con funciones de primera clase¹². JavaScript es un lenguaje de programación basado en prototipos, multiparadigma, de un solo hilo, dinámico, con soporte para programación orientada a objetos, imperativa y declarativa [27].

¹⁰ La mayoría de sus instrucciones son directamente ejecutadas por un intérprete.

¹¹ El compilador puede ser usado durante la ejecución.

¹² Las funciones se tratan como variables.

Por su parte, TypeScript es un superconjunto tipado de JavaScript que no altera la base del mismo, pero que añade nuevas reglas sobre el uso de los valores para prevenir errores comunes. Una vez que el compilador de TypeScript termina de validar el código, elimina los tipos, generando archivos de JavaScript “plano” sin las definiciones de tipos asignadas a los valores [28].

Algunas ventajas relevantes para el proyecto son:

- Versatilidad de uso para distintas plataformas.
- Gran cantidad de librerías.
- Excelentes librerías para la creación de interfaces gráficas.

Entre sus desventajas se encuentra:

- Que sea un lenguaje de programación de un solo hilo puede llegar a ser limitante.
- Distintos errores en cálculo de operaciones y fallos en librerías numéricas estándar.

2.3.1.4. C++

C++ es un lenguaje de programación compilado, es decir que la fuente es procesada por un compilador para producir archivos de código de objetos trasladables, que combinados con un *linker* genera un programa ejecutable, que es creado para el software y el hardware específico, por lo que no es portable a otros dispositivos [29].

Entre sus ventajas está:

- Permite optimizar el uso de recursos y tiempos de ejecución de manera sencilla comparado con los otros lenguajes evaluados.
- Tipado fuerte.
- Excelentes librerías para creación de interfaces gráficas.

Por otro lado, el lenguaje sufre las siguientes carencias:

- Los ejecutables generados no son portables.

- Bajo nivel de abstracción comparado con las otras alternativas, lo que puede alargar tiempos de desarrollo, complicar la implementación y dificultar la mantenibilidad del software.

2.3.2. Herramientas y *frameworks*

El conjunto de herramientas estudiado para la realización del trabajo de grado se describe a continuación:

2.3.2.1. Vue.js

Vue es un *framework* progresivo¹³ para la construcción de interfaces de usuario. Está diseñado para ser adoptado progresivamente en los proyectos. Pese a que su librería núcleo está enfocada en la capa de presentación, es perfectamente capaz de proveer aplicaciones completas en combinación con otras librerías de soporte [30].

2.3.2.2. UML

El lenguaje de modelado unificado¹⁴ (UML) es un lenguaje de modelado formal, conciso, comprensivo, escalable y estándar, para el desarrollo de sistemas [31].

2.3.2.3. Socket.IO

Socket.IO es una librería que permite la comunicación bidireccional y basada en eventos en todas las plataformas y enfocado en la fiabilidad y rapidez [32].

La facilidad que da para utilizar *sockets* desde el navegador y en el servidor, la hace una buena herramienta de comunicación entre aplicaciones.

2.3.2.4. PyPI

El índice de paquetes de Python¹⁵ (PyPI) es un repositorio de software para el lenguaje de programación Python que ayuda a encontrar e instalar software

¹³ Es posible incluirlo en cualquier etapa del desarrollo sin importar la base existente.

¹⁴ *The Unified Modeling Language* en inglés

¹⁵ Traducción al español de *The Python Package Index*.

desarrollado y compartido por los miembros de la comunidad de Python [33].

2.3.2.5. DeepSource

DeepSource es una herramienta que permite configurar un proyecto para automatizar la revisión de código objetiva de los elementos que conforman el código, además de realizar correcciones automáticas a algunas faltas comunes de rendimiento, seguridad y prácticas en una gran variedad de lenguajes, lo que permite despachar código de mejor calidad mucho más rápido [34].

2.3.2.6. Better Code Hub

Better Code Hub es una herramienta que verifica que una base de código en GitHub cumpla con diez (10) lineamientos de Ingeniería de Software, con el fin de seguir las mejores prácticas de la industria y ofrecer consejos sobre las mejoras a realizar [35].

2.3.2.7. Sphinx

Sphinx es una herramienta que permite crear de manera sencilla documentación inteligente y estética originalmente para proyectos en Python. Hace uso del lenguaje de marcado ReStructuredText para generar documentos en varios formatos [36].

Capítulo 3

Marco metodológico

En el presente capítulo se describe y justifica las metodologías de desarrollo, gestión de trabajo y evaluación, usadas a lo largo del proyecto.

3.1. Metodología de desarrollo

Se consideró una metodología basada en un modelo de entrega evolutiva propuesto por Aguilar [37] adaptado a un modelo metodológico de desarrollo de marcos de trabajo [7] y haciendo uso de algunas prácticas y principios de la metodología Kanban aplicados al desarrollo de software [1].

El proyecto fue pautado para una duración veinticuatro (24) semanas, distribuidas en cuatro (4) semanas con etapas solapadas que desembocan en diez (10) ciclos de dos (2) semanas, con la generación de un prototipo tras cada iteración. No obstante, con el fin de realizar una revisión exhaustiva de la investigación y mejorar el producto final, se agregaron dos (2) iteraciones más.

La entrega evolutiva se caracteriza por la emisión continua de prototipos, que son mostrados al cliente para luego ser refinados a partir de los comentarios de los mismos como se muestra en la Figura 3.1.a.

Dada la naturaleza de la investigación, en algunos ciclos se decidió atravesar por todas las etapas de la metodología como en la Figura 3.1.b, pasando a ser un prototipado evolutivo en vez de ceñirse fielmente a la metodología principal. No

obstante, Aguilar afirma que es natural que la entrega evolutiva tome la forma del prototipado evolutivo si la planificación es abierta a la recepción de cambios [37].

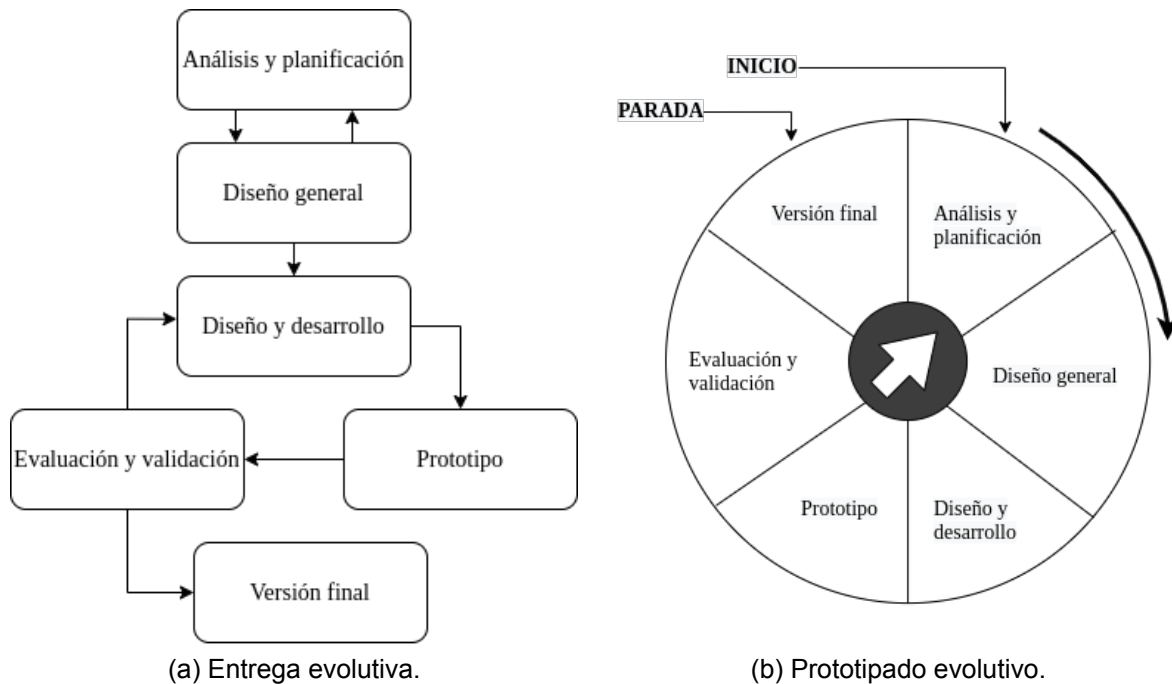


Figura 3.1: Diagramas de los modelos metodológicos empleados. De autoría propia, adaptado de Aguilar [37].

La metodología está conformada por las etapas de análisis y planificación, diseño general, diseño y desarrollo, prototipo, evaluación y validación, y entrega de versión final, que se definen a continuación:

3.1.1. Etapa de análisis y planificación

”El objetivo de toda planeación es la de clarificar el problema a solucionar, definir el producto a obtener o servicio a proporcionar, estimar los costos económicos en que se va a incurrir, así como los recursos humanos y de cualquier otro tipo que se requieran para alcanzar la meta” [37].

3.1.1.1. Fase de investigación

En la fase de investigación se realiza todo lo correspondiente al marco teórico del problema con el fin de obtener conceptos útiles, hallar soluciones y clarificar los

problemas a afrontar en cada etapa.

3.1.1.2. Fase de definición del problema

El objetivo de esta fase es clarificar el problema a solucionar y definir los prototipos a generar, refinando cada vez más la solución a desarrollar [37].

3.1.1.3. Fase de definición del plan de desarrollo

En la definición del plan de desarrollo se determinan las actividades a realizar a lo largo del proyecto mediante el establecimiento de épicas basadas en los objetivos, con un tiempo y duración estimados para realizarlas [37]. Todas las actividades son agregadas a un tablero para la gestión de flujos de trabajo.

3.1.2. Etapa de diseño general

El propósito de esta etapa es desarrollar propuestas del sistema que satisfagan los requerimientos de los objetivos propuestos [38].

3.1.2.1. Fase de definición de arquitectura

Es esta fase se especifica la organización del software, considerando los componentes que lo integran, así como las interfaces y comportamientos que caracterizan a estos componentes [39].

3.1.2.2. Fase de definición de estructuras de datos

Establece las estructuras de datos a usar tentativamente por cada módulo o funcionalidad que generan restricciones en las entradas y salidas de información [38].

3.1.2.3. Fase de definición de mecanismos de comunicación

La fase busca definir los mecanismos de comunicación, interacción y colaboración entre los componentes [39]. Contribuye a identificar las principales estrategias para

manipular el ambiente de desarrollo por medio de la definición de nuevas estructuras y la creación métodos que cumplan los requerimientos de información de los procesos modelados.

3.1.3. Etapa de diseño y desarrollo

Para cada módulo identificado en la etapa de diseño global, se realiza un proceso de diseño específico que deriva en la implementación y pruebas de software de cada uno [37].

3.1.3.1. Fase de diseño detallado

Tiene como tarea definir para cada componente de manera específica: sus entradas y salidas, cómo el usuario y los demás componentes interactúan con él, las clases existentes y las secuencias y condiciones que cumple cada proceso [39].

3.1.3.2. Fase de implementación

Tras el respectivo diseño específico de los elementos pertenecientes al objetivo planteado para la iteración, se pasa a la fase de implementación de la solución donde se aplican las especificaciones definidas en las fases anteriores. De manera adicional, en esta etapa se contempla la documentación del código.

3.1.3.3. Fase de instanciación y pruebas

Dependiendo de las funcionalidades a probar, se realizan pruebas unitarias, extremo a extremo y aceptación por medio de la instanciación, que para los elementos de caja negra se realiza a través de la configuración e inyección de elementos propios de la librería, mientras que los de caja blanca por medio de la extensión [7]. Por otro lado, también se decidió probar el seguimiento de buenas prácticas de Ingeniería de Software a través de DeepSource y Better Code Hub.

3.1.4. Prototipo

Al finalizar el ciclo se obtiene un prototipo. El objetivo del prototipo es realizar una validación y evaluación con el fin de mejorar la solución y verificar el cumplimiento de las funcionalidades planificadas para determinada iteración.

3.1.5. Etapa de evaluación y validación

Es una revisión del cumplimiento de los requerimientos y funcionalidades esperados para determinada iteración, además de generar retroalimentación sobre los resultados obtenidos.

3.1.5.1. Fase de uso del *framework*

Implica una evaluación de las facilidades que otorga el *framework* para la elaboración de una herramienta de simulación de redes de cola.

Al no ser sencillo elaborar un marco objetivo sobre la facilidad de uso del ambiente de desarrollo, la metodología a emplear para evaluar el *framework* será su capacidad para simular y permitir la creación de los componentes de redes de cola de tipo: fuente, servidor, sumidero y ruta.

3.1.5.2. Fase de validación de resultados

La fase se basa en el uso del software creado con el *framework* a fin de identificar fallas en el funcionamiento del mismo.

La metodología a emplear para la validación será la realización de experimentos de simulación en herramientas existentes o con resultados conocidos, y el software creado, con el objetivo de comparar sus salidas. En caso de la inclusión de valores aleatorios, se realizarán varias corridas a fin de determinar la equivalencia de las muestras a partir de una prueba de hipótesis para muestras independientes.

3.1.6. Versión final

Es el punto de parada de la investigación y el momento en el que el prototipo se convierte en producto de ingeniería [37].

3.1.6.1. Despliegue del *framework*

Durante el despliegue del marco de trabajo se empaqueta el proyecto Python con el fin de obtener los archivos de distribución que puedan ser subidos a PyPI con el fin de facilitar su acceso por otros usuarios.

3.1.6.2. Despliegue del simulador

Consiste en crear los archivos de distribución del simulador a fin de que pueda ser ejecutado bajo un ambiente de producción.

3.1.6.3. Generación de página de referencia

Al finalizar la documentación de todo el ambiente de desarrollo se genera la documentación con Sphinx, que convierte los comentarios en código a una página de referencia.

3.1.7. Justificación de la metodología

A continuación se presentan los principales motivos por los cuales se consideró el uso de este tipo de metodología:

3.1.7.1. Ciclo de vida

Al ser una investigación, se puede aprovechar la entrega evolutiva como método para obtener mayor retroalimentación de las hipótesis planteadas a través de entregables constantes, además de permitir dividir el desarrollo, fijar objetivos y planificar cómodamente, siendo flexible a la introducción de nuevos elementos y funcionalidades.

3.1.7.2. Enfoque inicial en la investigación

Al principio del proyecto, la metodología de desarrollo se enfoca en la investigación antes que la entrega de prototipos tempranos, lo que permite conocer el dominio del problema de mejor manera, siendo algo imprescindible al momento de desarrollar un *framework*.

3.1.7.3. Características del proyecto

El proyecto es un trabajo de investigación que culmina con la generación de un prototipo, por lo que más allá de una metodología de desarrollo de prototipo, es necesario considerar también las etapas de investigación y documentación.

La metodología adaptada aprovecha las ventajas en mejora y adaptabilidad ante cambios de las metodologías ágiles, prescindiendo de los principios y prácticas de otras metodologías ágiles más completas.

Al ser un proyecto desarrollado de manera individual, varios de los artefactos y prácticas de otras metodologías de desarrollo en equipo no agregan tanto valor al proyecto, pues en su mayoría tienen como objetivo apoyar la comunicación y el trabajo colaborativo, factores que no aplican al proyecto actual.

3.1.7.4. Adaptabilidad

Es común la transformación de la metodología a un modelo de prototipado evolutivo si la planificación es abierta a la recepción de cambios [37]. Esta flexibilidad la hace una excelente metodología para el desarrollo de un prototipo basado en una investigación, ya que contempla la dualidad entre un desarrollo puro, y la inclusión de etapas de investigación en el desarrollo.

3.2. Metodología de gestión de trabajo

La metodología empleada para la gestión de trabajo se fundamenta en varios principios y prácticas definidos en Kanban.

3.2.1. Principios de Kanban

Los principios básicos de Kanban son [1]:

- Empezar con lo que se tiene, el proceso actual.
- Estar de acuerdo con perseguir un desarrollo evolutivo abierto al cambio y la mejora.
- Respeto a los roles y responsabilidades del equipo.

La metodología empieza con el proceso existente, sin la necesidad de requerir grandes cambios en el mismo, buscando la mejora continua y rapidez ante el cambio. No se puede hablar de un equipo al tratarse de una investigación individual, sin embargo, dentro del flujo del desarrollo se asumen varios roles y se manejan ciertas herramientas, cuya responsabilidad es necesaria respetar para lograr una buena gestión de las actividades.

3.2.2. Prácticas de Kanban

A partir de los principios descritos, surgen las siguientes prácticas [1]:

- Visualizar el trabajo y el flujo del mismo.
- Limitar el trabajo en progreso¹ (WIP).
- Gestionar el flujo.
- Mejorar por medio de modelos y uso del método científico.

Kanban se basa con el concepto del trabajo en progreso y en cómo este debe ser limitado. De este forma, evita el inicio de nuevos procesos sin haber hecho entrega o retirada de una parte del trabajo en realización. Los límites WIP es lo que define qué se puede hacer en el proceso de desarrollo y la cantidad de actividades simultáneas en progreso por cada estado del flujo de trabajo [40].

¹ Traducción al español de *Work-In-Progress*

3.2.3. Adaptación de la metodología

El flujo de trabajo está visualizado y gestionado a través de un tablero dividido en cuatro bandejas: "Por Hacer", "Haciendo", "Revisión" y "Finalizado" como se puede ver en la Figura 3.2.

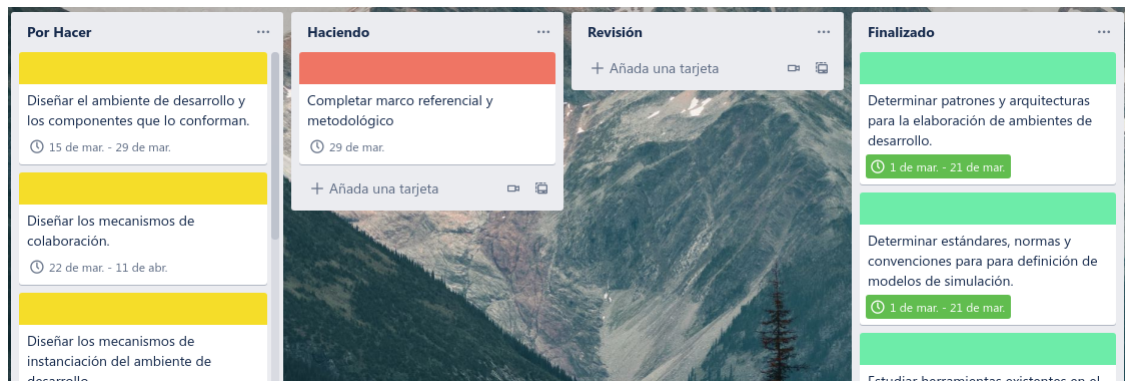


Figura 3.2: Ejemplo de tablero Kanban. De autoría propia.

En la bandeja de "Por hacer" se ubican en un principio épicas con las actividades requeridas para cumplir con los objetivos. En la planificación por iteración, se convierten las épicas en actividades más reducidas. Al momento de empezar una actividad atómica, esta pasa a la columna a "Haciendo" si no supera el límite WIP de dos actividades. En caso de abandonar una actividad se archiva o vuelve a mover a "Por hacer" y de finalizarse se mueve a "Revisión".

Una vez finalizadas las actividades correspondientes a una épica, se procede a asegurar el cumplimiento de lo establecido en la misma por medio de la evaluación y validación del modelo creado, al mismo tiempo que se aplican o ignoran los consejos dados por las herramientas de revisión de código DeepSource y Better Code Hub.

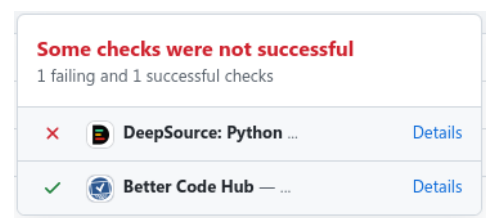


Figura 3.3: Ejemplo de resultado de revisión de código. De autoría propia.

Acatadas o desestimadas las revisiones y consejos dados por las herramientas, las actividades finalizadas pasan de manera permanente a la bandeja "Finalizado", existiendo la posibilidad de crear nuevas tareas en el caso de existir una hipótesis de

mejora sobre algún apartado en el prototipo.

3.2.4. Justificación de la metodología

A continuación se presentan los principales motivos por los cuales se consideró el uso de este tipo de metodología:

3.2.4.1. Carencias de la metodología de desarrollo

La metodología de desarrollo elegida no posee ningún método para la gestión de trabajo en el proyecto.

Es recomendable la aplicación de una metodología de gestión de trabajo, ya que permite saber el estado en el que se encuentra el proyecto y gestionar las tareas del mismo; evitando problemas de organización, planificación y cumplimiento de requerimientos.

3.2.4.2. Facilidad de implementación

El principio de "iniciar con el proceso existente" hace a la metodología perfectamente adaptable a las otras metodologías adoptadas para realizar la investigación.

3.2.4.3. Herramientas existentes

La experiencia y comodidad en el uso de herramientas que facilitan la aplicación de varias de las prácticas de la metodología y su integración con otras herramientas usadas en el desarrollo, fue de especial relevancia para la selección de la metodología.

3.3. Metodología de evaluación y validación

La evaluación y validación del *framework* tiene como función verificar el cumplimiento de los objetivos y obtener las recomendaciones de mejora de los prototipos desarrollados. El ambiente de desarrollo es usado por los usuarios para

generar salidas, obteniéndose dos dominios a estudiar cuyas metodologías se exponen a continuación:

3.3.1. Evaluación del *framework*

Debido a que evaluar el uso de una herramienta puede resultar subjetivo, la metodología empleada para lograrlo involucra el estudio de las características del ambiente de desarrollo.

3.3.1.1. Comportamiento e inversión de control

Evaluar el comportamiento del marco de trabajo consiste en estudiar los procedimientos predeterminados que enmarca y en cómo el usuario participa en la ejecución del mismo por medio de la inversión de control.

El factor a evaluar es la existencia o no de inversión de control, como característica primordial para considerar a la herramienta desarrollada como un *framework*.

Para ello se considera como inversión de control: cualquier momento durante la ejecución del software en el que el ambiente de desarrollo toma el control y realiza llamadas a métodos implementados por los usuarios, ya sea por extensión o por inyección de dependencias.

3.3.1.2. Cerrado a la modificación

Otra de las características de los ambientes de desarrollo, es que los mismos están cerrados a la modificación de su base de código, siendo adaptado por los usuarios por medio de la extensión.

Para evaluarlo, se propuso desarrollar un simulador de redes de cola a partir de la extensión del *framework* como generalización a todas las posibles herramientas de simulación.

Con el fin de demostrar la capacidad de extender el marco de desarrollo para adaptarse al dominio de simuladores de redes de cola sin necesidad de la modificación de la base de código, se implementarán y simularán con el mismo, los

elementos de tipo: fuente, servidor, sumidero y ruta.

3.3.1.3. Seguimiento de buenas prácticas de Ingeniería de Software

Para evaluar el seguimiento de buenas prácticas de Ingeniería de Software se utilizarán las medidas, métricas, reportes, consejos y mejoras proveídas por DeepSource y Better Code Hub.

3.3.2. Validación de resultados

Al realizar procesos que involucran salidas, es necesario validar los resultados de las operaciones que ejecuta el *framework*, con el fin de garantizar su correcto funcionamiento.

Para lograrlo, se realizarán simulaciones con los prototipos por un tiempo determinado y se registrarán los resultados obtenidos de la simulación. Posteriormente, los resultados son comparados con los resultados obtenidos, ya sea de manera teórica o por simulación con otras herramientas.

Con valores constantes, basta con realizar una simulación y realizar el análisis comparativo. Por otra parte, para evaluar las simulaciones con parámetros aleatorios, se tomarán muestras de quince elementos y se realizará una prueba estadística para muestras con datos independientes con el fin de determinar si la media del prototipo y la media esperada son equivalentes.

3.3.3. Justificación de la metodología

A continuación se presentan los principales motivos por los cuales se consideró el uso de este tipo de metodología:

3.3.3.1. Objetividad y reproducibilidad

El estudio de los factores seleccionados para la aplicación de la metodología, bajo las restricciones impuestas por la misma, puede ser realizado por cualquier investigador para obtener los mismos resultados.

3.3.3.2. Generalización

Las características de los simuladores permite saber la capacidad del *framework* para implementar cualquier tipo de simulador de eventos discretos a través de la elaboración de un simulador específico, sin necesidad de estudiar los infinitos simuladores que se podrían construir.

Por otra parte, el simulador de redes de cola evaluado considera únicamente los elementos de fuente, servidor, sumidero y ruta, ya que en base al marco teórico, se puede considerar al resto de elementos como una extensión de los mismos.

3.3.3.3. Relación con los objetivos de la investigación

Pese a la generalización hecha hacia los simuladores de redes de cola, la metodología cumple con los parámetros suficientes para determinar el cumplimiento de los objetivos establecidos en la investigación.

Capítulo 4

Desarrollo de la investigación

Este capítulo explica el proceso realizado y las decisiones tomadas a lo largo de la investigación que dieron origen a los resultados y descubrimientos obtenidos, mediante la aplicación de la metodología propuesta.

4.1. Definición de la investigación

4.1.1. Etapa de análisis y planificación

A partir de la investigación realizada, cuyos resultados más relevantes se pueden encontrar en el marco teórico, se definió el problema y la solución planteada en el primer capítulo. Posteriormente, se creó el plan de desarrollo con base en diez (10) prototipos y una etapa de diseño general de la solución.

4.1.2. Etapa de diseño general

4.1.2.1. Fase de definición de arquitectura

En esta fase se diseñó un modelo inicial de los componentes que conforman al *framework*. Este modelo se presenta en la Figura 4.1.

Se pensó un módulo de control cuya función fuera manejar la simulación, el módulo de simulación ejecutaba las transiciones y obtenía las salidas del sistema dinámico, para guardarlas con el apoyo del módulo de reportes.

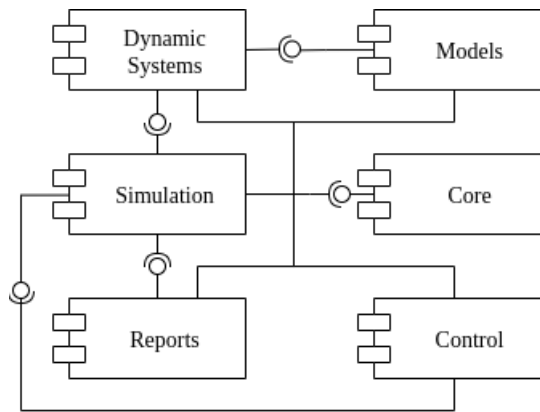


Figura 4.1: Componentes del sistema. De autoría propia.

Por su parte, el módulo de sistema dinámico se ideó con la premisa de que el usuario definiría e incluiría modelos en el mismo a través de la extensión, encargándose de enrutar las entradas y salidas a medida que realiza las transiciones de cada uno.

Ante la posible existencia de elementos como bus de eventos, herramientas de monitoreo, definiciones de clases y funciones compartidas, etc., se agregó al diseño un módulo compartido que provee funcionalidades a todos los módulos.

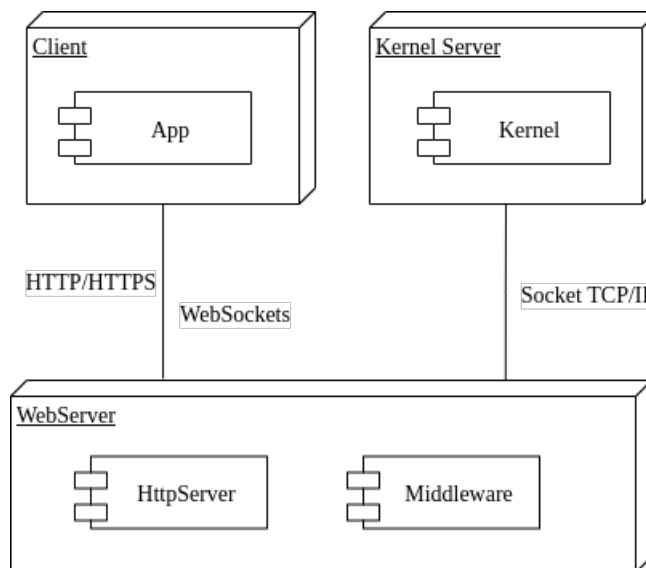


Figura 4.2: Arquitectura del simulador. De autoría propia.

En cuanto a la arquitectura física del simulador que implemente el marco de trabajo, se pensó un sistema compuesto de tres capas: un *kernel* comunicado con una capa intermedia a través de *sockets* TCP/IP y la capa intermedia comunicada a una interfaz

de usuario a través del protocolo *WebSockets* y peticiones/respuestas HTTP/HTTPS como se ve en la Figura 4.2.

4.1.2.2. Fase de definición de estructuras de datos

Entre las estructuras de datos consideradas en el diseño general está:

- **Sets:** el conjunto se eligió para colecciones de elementos cuyo orden no importa y que se pueda garantizar que solo exista uno en la colección:
 - Modelos y rutas del sistema dinámico.
 - Próximos modelos inmediatos a ejecutar un evento.
- **Arreglos asociativos:** el arreglo asociativo se consideró para colecciones de elementos que se desearan relacionar entre sí:
 - Entradas y salidas a los sistemas: el modelo como clave y las entradas o salidas del modelo como valor.
 - Reportes de las simulaciones: el tiempo como clave, y las salidas en ese tiempo como valor.
- **Heaps:** la pila de tipo *min-heap*, se pensó como estructura de datos para elementos agrupados en una cola de prioridad, siendo la lista de eventos futuros la única funcionalidad prevista para esto.
- **Lista:** para cualquier otra colección de datos que no cumpliera con ninguna de las condiciones anteriores, como elemento por defecto se eligió la lista.
- **Expresión:** estructura de datos abstracta pensada para soportar todo tipo de entradas como parámetro de simulación al implementar un método para evaluarlo.

4.1.2.3. Fase de definición de mecanismos de comunicación

Como mecanismos de comunicación entre sistemas se eligió el uso de *sockets* debido a su naturaleza bidireccional donde el servidor puede comunicarse con el

cliente sin necesidad de que este realice una petición.

En cuanto a la comunicación entre módulos, se eligió la composición por medio de la inyección de dependencias a través de constructor haciendo uso de un contenedor IoC, además de la emisión y recepción de eventos a través de un bus de eventos.

No obstante, tras las primeras pruebas previas al desarrollo, se desestimó el uso del contenedor IoC, debido a que no se encontró una ventaja significativa que compensara el esfuerzo requerido para la integración y configuración del marco de trabajo con el *framework* de inyección de dependencias estudiado.

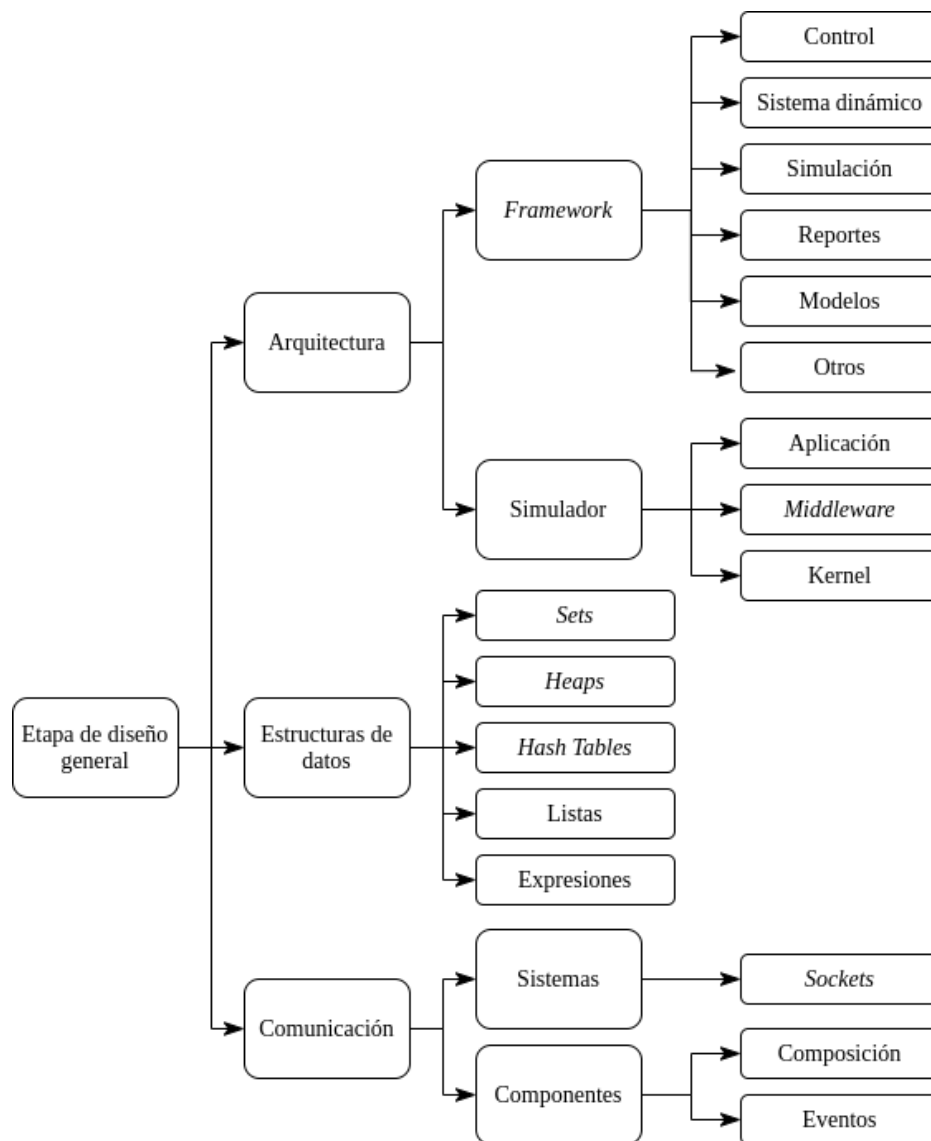


Figura 4.3: Resumen de etapa de diseño general. De autoría propia.

4.2. Desarrollo de los prototipos

4.2.1. Prototipo 1. Autómata celular lineal

El primer prototipo consistió en desarrollar un modelo atómico para sistemas de eventos de tiempo discreto¹, siendo los autómatas celulares un perfecto ejemplo de este tipo de sistemas.

4.2.1.1. Fase de diseño detallado

Al ser un modelo de tiempo discreto, el tiempo de llegada e es constante, por lo que, tanto las transiciones autónomas como externas suceden al mismo tiempo, siendo confluentes.

Con este fundamento, se diseñó la clase abstracta *AtomicModel* con un atributo privado *state* de tipo *ModelState*² para guardar el estado del modelo.

El modelo atómico tiene dos métodos abstractos protegidos: el método *_output_function* que mapea el estado basado en la ecuación 2.5 y el método *_state_transition* que realiza la transición de estado basado en la transición confluyente de la ecuación 2.10.

Adicionalmente, la clase abstracta tiene dos métodos públicos, ideados para llamar a la implementación de los métodos abstractos realizada por el usuario en las subclases.

Esta clase es extendida por una clase concreta *Cell* que implementa los métodos abstractos y cuyo estado es una variable booleana que indica si está viva o no la célula, cambiando el tipo *ModelState* a *bool*.

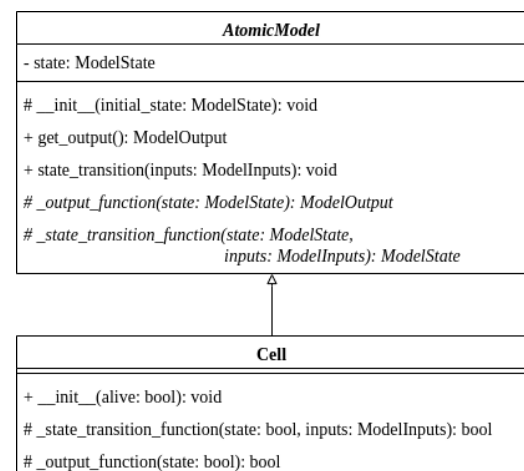


Figura 4.4: Diseño de célula del autómata. De autoría propia.

¹ Los eventos de tiempo discreto pueden verse como eventos discretos con una función de tiempo constante para todos los elementos pertenecientes al sistema discreto.

² Alias para el tipo de dato *Any*.

4.2.1.2. Fase de implementación

Para la clase abstracta *AtomicModel*, se implementó *get_output* para que retorne el valor de la llamada al método abstracto *_output_function* con el parámetro estado, al mismo tiempo que el método *state_transition* se implementó para asignar al estado del modelo el resultado de la llamada a el método abstracto *_state_transition* con los parámetros estado y entrada.

Por su parte, la clase *Cell* implementó los métodos abstractos para devolver el estado como función de salida, y devolver la entrada³ como función de transición.

4.2.1.3. Fase de instanciación y pruebas

Para la realización de las pruebas, se creó una prueba unitaria donde se instanciaron tres células: viva, muerta, viva. Dentro de un ciclo de tres iteraciones obtuvo la salida de todos los modelos, y se realizó la transición de estado de cada uno insertando como entradas las salidas generadas por modelo directamente anterior, como se ve en la Figura 4.5.

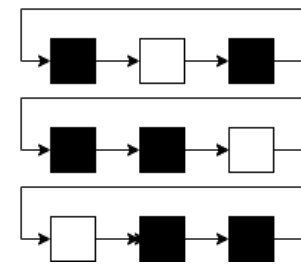


Figura 4.5: Autómata celular lineal. De autoría propia.

4.2.2. Prototipo 2. Autómata celular en malla

El segundo prototipo consistió en juntar los distintos modelos atómicos para eventos de tiempo discreto bajo un mismo sistema dinámico, a fin de extender el *framework* para simular un autómata celular con una disposición en forma de malla o cuadrícula, con las reglas del "Juego de la vida" de Conway que se encuentran en el Anexo B.

4.2.2.1. Fase de diseño detallado

El prototipo se diseñó como se muestra en la Figura 4.6, modificando clases existentes y creando nuevas de acuerdo a las necesidades que surgieron y cuyo razonamiento se describe a continuación:

³ Como la entrada es un arreglo asociativo, en realidad devuelve el valor de la única clave del diccionario.

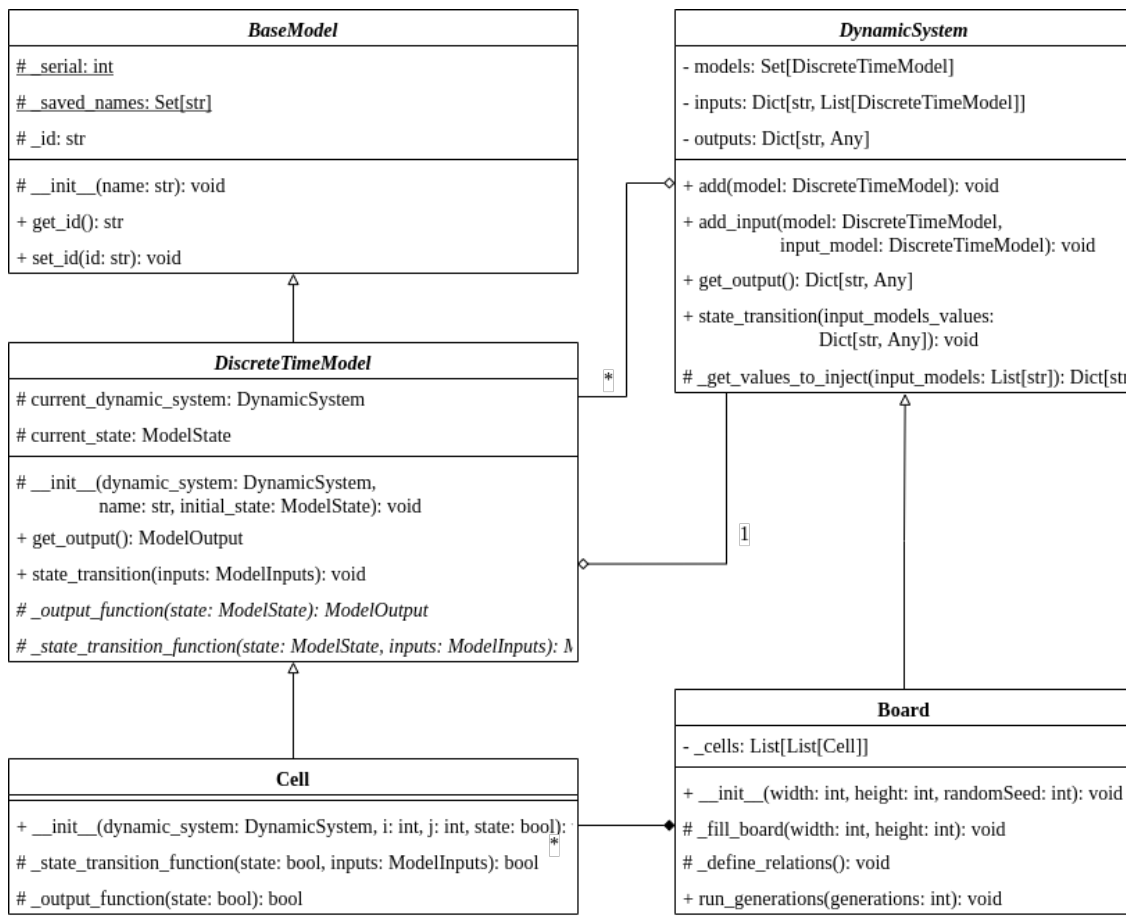


Figura 4.6: Diseño para el prototipo 2. De autoría propia.

Al existir varios modelos a simular, se decidió crear una clase *BaseModel* que es la encargada de otorgar un identificador a cada uno. Posee un conjunto de nombres de clases como atributo privado estático que evita la repetición de identificadores.

Se cambió el nombre de la clase *AtomicModel* a *DiscreteTimeModel*, pues el modelo dejó de ser atómico al permitir la composición de estos, y se incluyó en el constructor un parámetro para la inyección del sistema dinámico y la definición del identificador.

Se creó la clase *DynamicSystem* para representar a los sistemas dinámicos. Posee dos atributos: los modelos registrados en el sistema y la última salida del sistema dinámico, siendo agregado posteriormente un tercer atributo para almacenar los modelos que generan una entrada a determinado modelo a fin de enrutar las entradas para ejecutar una transición de estado.

Los métodos públicos del sistema dinámico permiten agregar un modelo al sistema

dinámico, agregar un modelo de entrada a un modelo perteneciente al sistema dinámico, calcular la salida del sistema dinámico y calcular el siguiente estado del sistema dinámico.

El algoritmo de transición de estado del sistema dinámico, primero ejecuta la transición de estado de los modelos que reciben una entrada externa. Posteriormente, para cada uno de los modelos que reciben una entrada de algún modelo y que no hayan pasado por una transición externa, obtiene las salidas de sus modelos de entrada, y las inyecta para realizar una transición de estado. Este algoritmo fue implementado como se ve en el Código Anexo 23, siendo la base para todos los algoritmos siguientes.

4.2.2.2. Fase de implementación

Se implementaron los cambios y mejoras requeridos haciendo énfasis en la integración del sistema dinámico y el algoritmo de enrutamiento de las entradas diseñado.

Para lograr el correcto funcionamiento del marco de trabajo, se tuvo que oficializar como regla para uso del *framework*⁴ la necesidad de computar la salida del estado dinámico antes de ejecutar una transición de estado para la realización de transiciones autónomas o confluentes.

La separación de transición de estado y función de salida, evita problemas de dependencias en entradas⁵ y permite el cómputo de sistemas donde un modelo aparece más de una vez, ya que, al no ejecutar una transición de estado inmediata a la salida del modelo se garantiza que solo se ejecutará una vez la transición, sin importar que el modelo esté referenciado más de una vez en la red.

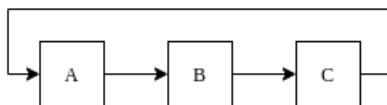


Figura 4.7: Ejemplo de red cíclica. De autoría propia.

⁴ En el primer prototipo no se podía considerar regla, pues era el único modo de realizar la transición al no existir conexiones entre modelos que permitiera la interacción directa entre ellos.

⁵ El modelo que ejecutará la transición ya tendrá sus entradas definidas, sin importar que el estado de sus modelos de entrada haya sido afectado en la misma iteración.

Un ejemplo de lo descrito anteriormente se puede ver en la Figura 4.7, donde A depende de C, pero por transitividad C depende de A. Al computar la salida de primero, se obtiene a partir del mapeo de sus estados el resultado para A, B y C, los cuales son usados por las transiciones, cubriendo las dependencias y evitando la llamada cíclica de los modelos.

4.2.2.3. Fase de instanciación y pruebas

En esta fase se creó una prueba unitaria donde se instanció un tablero de 10x10 para correr diez iteraciones con una semilla igual a 42 para permitir la reproducción de resultados aleatorios.

El tablero extiende del sistema dinámico. Crea las células aleatoriamente inyectándose como sistema dinámico y estableciéndolas como células vivas ($P(X < 0.5)$) o muertas ($P(X \geq 0.5)$), para luego definir las relaciones de entrada entre las celdas instanciadas.

4.2.3. Prototipo 3. Sistema de fábrica de discos

Como tercer prototipo se planteó un simulador de eventos discretos basado en el modelo de línea de ensamblaje de ejemplo de Nutaro [6]. La idea principal era ajustar los componentes creados a la simulación de modelos de eventos discretos a través de un algoritmo de programación de eventos.

4.2.3.1. Fase de diseño detallado

El principal problema a enfrentar durante el diseño de este prototipo era la necesidad de soportar los sistemas de eventos discretos, por lo que se propuso el diseño de la Figura 4.8, donde se pueden ver algunos cambios respecto al prototipo anterior y la inclusión de un sistema dinámico.

La clase *BaseModel* se decidió renombrar a *Entity*, ya que solo poseía características para otorgar entidad a los modelos, siendo posible la inclusión de entidades distintas a los modelos en el futuro.



Figura 4.8: Diseño del prototipo 3. De autoría propia.

Extendiendo de la clase abstracta *Entity*, se decidió crear un modelo abstracto con el nombre *BaseModel*, que posee un sistema dinámico y un estado, proveyendo los métodos para interactuar con ellos. A su vez, el modelo incluye dos métodos abstractos para obtener la salida y realizar la transición de estado. Se decidió no implementarlos

directamente, pues de esta manera se expone la interfaz para realizar estas tareas sin definir un método definitivo para lograrlo.

Extendiendo a *BaseModel* se cambió *DiscreteTimeModel* para que se convirtiera en *DiscreteEventModel*, para ello se dividió la transición de estado en externa (2.9), autónoma (2.7) y confluyente (2.10), creando los métodos abstractos para las funciones de transición externa e interna, e implementando los métodos abstractos del modelo base con este fin.

Adicionalmente, se agregó un método con el fin de obtener el tiempo para la emisión de un evento autónomo a través de la ejecución de la función de avance de tiempo (2.6) representada por un método abstracto.

Para facilitar la programación y eliminación de eventos de un modelo de eventos discretos, se idearon los métodos públicos *schedule* y *unschedule*.

En cuanto al sistema dinámico, se decidió abstraer las funcionalidades de salida y transición, debido a que su implementación podía verse afectada por el tipo de sistema dinámico. Con esto en cuenta, surgió la idea de la clase abstracta *BaseDynamicSystem* a partir de la clase *DynamicSystem* elaborada anteriormente, donde se implementa el comportamiento fijo del sistema dinámico como agregar modelos y enlazarlos, mientras que la salida y transición se definen como métodos abstractos.

Extendiendo la clase, se creó *DiscreteEventDynamicSystem*, donde se esperaba implementar el algoritmo de enrutamiento y salida del nuevo sistema dinámico para eventos discretos. Como los eventos deben ser programados, se decidió que el sistema dinámico estuviera compuesto por un programador de eventos *Scheduler*.

La clase *Scheduler* está pensada para manejar la lista de eventos futuros presentada en el apartado 2.1.3.2. La lista de eventos futuros es un *min-heap* de modelos programados (*ScheduledModel*), que poseen un modelo cualquiera y un tiempo para ser programados con el que son ordenados en el montículo.

Al considerar la FEL como un *min-heap*, se optimiza la inserción y extracción de los elementos con prioridad. El programador de eventos dispone de los métodos para agregar o eliminar del *heap*, obtener los modelos con un evento inminente y actualizar los tiempos de la lista.

El sistema dinámico puede ejecutar una transición de estado en cualquier tiempo. Las transiciones pueden originarse por una entrada a uno de los modelos del sistema dinámico, un evento autónomo o ambas.

Al igual que el algoritmo anterior, para realizar una transición, primero se ejecutan las transiciones externas y luego las autónomas.

En primer lugar, se le resta el tiempo de llegada del evento a la lista de eventos futuros.

Los modelos afectados por una entrada externa, forman un conjunto *ModelosEntrada*. Todos los modelos de este conjunto, ya ejecutaron su transición, siendo externa en caso de no tener programado algún evento para el tiempo determinado, o confluyente en caso contrario.

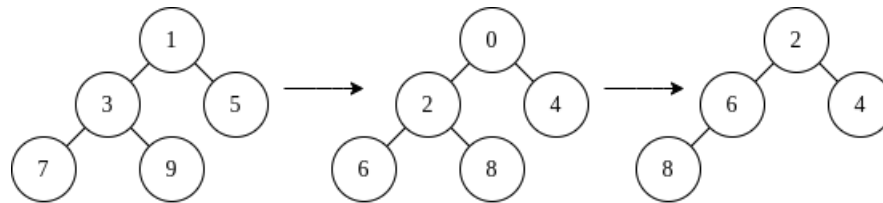


Figura 4.9: Extracción del evento inminente de la FEL. De autoría propia.

Si el tiempo de un evento inminente es cero, significa que hay eventos autónomos que requieren ejecutarse, por lo que hay un conjunto dentro de los modelos del sistema dinámico que van a ejecutar una transición que se decidió llamar *ModelosAutónomos* extraído de la FEL como se muestra en la Figura 4.9.

Como puede haber modelos que recibieron una entrada externa al sistema dinámico en su tiempo inminente, los conjuntos *ModelosEntrada* y *ModelosAutónomos* pueden tener una intersección (zona gris en la Figura 4.10), siendo necesario eliminar todos los modelos de entrada del conjunto de modelos autónomos para no repetir la transición ejecutada anteriormente.

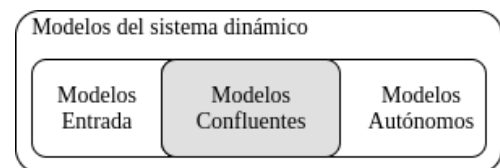


Figura 4.10: Principales conjuntos del sistema dinámico. De autoría propia.

$$ModelosAutónomosFaltantes = ModelosAutónomos - ModelosEntrada \quad (4.1)$$

Como la transición interna de los modelos implica la liberación de una salida, existe un conjunto de modelos que van a ser afectados por la transición de los modelos autónomos llamado *ModelosAfectados*, que reciben una o varias entradas de los modelos autónomos. Este conjunto puede tener elementos del conjunto modelos de entrada y autónomos, por lo que los autónomos faltantes puros⁶ se definen como la diferencia entre los autónomos faltantes y modelos afectados.

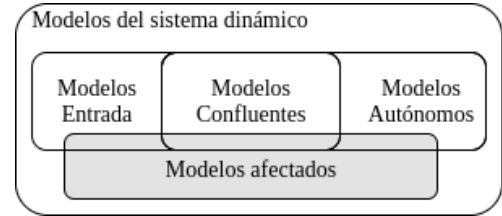


Figura 4.11: Modelos afectados por una transición. De autoría propia.

$$ModelosAutónomosPuros = ModelosAutónomosFaltantes - ModelosAfectados \quad (4.2)$$

Para todos los modelos que ejecutarán una transición autónoma i , se debe obtener su salida y modelos a los que le genera entrada. Como un modelo afectado puede ser un modelo de entrada, y para estos ya se definió una transición, es necesario removerlos.

$$Salidas_i = ModelosSalida_i - ModelosEntrada \quad (4.3)$$

Una vez obtenidas las salidas afectadas, se les asigna el valor de entrada. Como un modelo puede tener varios modelos de entrada, como estructura de datos para la entrada se eligió un arreglo asociativo con el modelo generador como clave y la entrada que genera el modelo como valor.

⁶ No recibirán entrada externa, por lo que solo ejecutarán una transición interna.

```

1 Entrada de modelos afectados: {
2     Modelo Afectado A: {
3         Modelo de Entrada X: Valor X,
4         Modelo de Entrada Y: Valor Y
5     },
6     Modelo Afectado B: {
7         Modelo de Entrada X: Valor X,
8         Modelo de Entrada Z: Valor Z
9     }
10 }

```

Código 4.1: Ejemplo de entradas de modelos afectados. De autoría propia.

Una vez calculadas las entradas se computa la transición de estado de los modelos autónomos puros sin entradas, y los modelos afectados con las entradas para modelos afectados calculadas.

4.2.3.2. Fase de implementación

Los cambios en la estructura fueron realizados sin mayor inconveniente, siendo las nuevas estructuras y algoritmos lo más complicado de implementar.

En Python no existe el *heap* como estructura de datos válida. Para usar *heaps* se utilizó la librería *heapq*, que provee los algoritmos de montículos sobre una lista.

Para obtener una vista de los próximos eventos a ejecutar sin alterar la FEL⁷, fue necesario crear copias de la lista y aplicar iterativamente (mientras el tiempo del evento sea equivalente al mínimo) el algoritmo de extracción del elemento con mayor prioridad.

Con el fin de manipular la operaciones dentro del *framework* de manera más sencilla, se sobrescribieron varios operadores de algunas clases como `__eq__` para las equivalencias⁸ y `__lt__` para la desigualdad⁹ "menor que".

⁷ La librería utiliza la referencia directa de la lista, por lo que cualquier método modifica al objeto original.

⁸ Buscando equivalencias entre atributos en vez de referencias en memoria.

⁹ Perfecto para los algoritmos de inserción y eliminación del *heap*.

4.2.3.3. Fase de instanciación y pruebas

Para las pruebas se hizo un modelo basado en el modelo de línea de ensamblaje de ejemplo de Nutaro que se presenta a continuación [6]:

En este modelo se tiene una prensa con un tiempo de servicio uniforme de un segundo (1 seg) y un taladro con tiempo de servicio uniforme de dos segundos (2 seg) conectados entre sí como muestra la Figura 4.12.

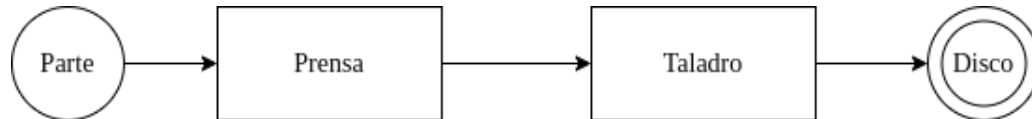


Figura 4.12: Modelo de línea de ensamblaje. Basado en Nutaro [6].

Cada estación solo puede procesar un elemento a la vez; durante una transición externa, encolan los elementos y disminuyen su tiempo de procesamiento, mientras que en una transición autónoma desencolan un elemento y restablecen el tiempo de procesamiento.

Para implementarlo se extendió la clase *DiscreteEventDynamicSystem* y en su implementación se agregó una instancia de *Press* y *Drill*, clases que extienden de *DiscreteEventModel* y que implementan las reglas descritas.

En el proceso de prueba se realizó un proceso iterativo de obtención de salida y ejecución de transición de estado del sistema dinámico hasta que no hubiera eventos pendientes en la FEL y cuyos resultados pueden ser consultados en la Tabla Anexa A.2.

4.2.4. Prototipo 4. Simulador de fábrica de discos

Con el *framework* listo para definir los sistemas dinámicos, se prosiguió con el desarrollo del marco de trabajo para la ejecución de simulaciones.

4.2.4.1. Fase de diseño detallado

Para realizar el proceso de simulación es necesario un mecanismo de control, un motor de simulación y un generador de reportes.

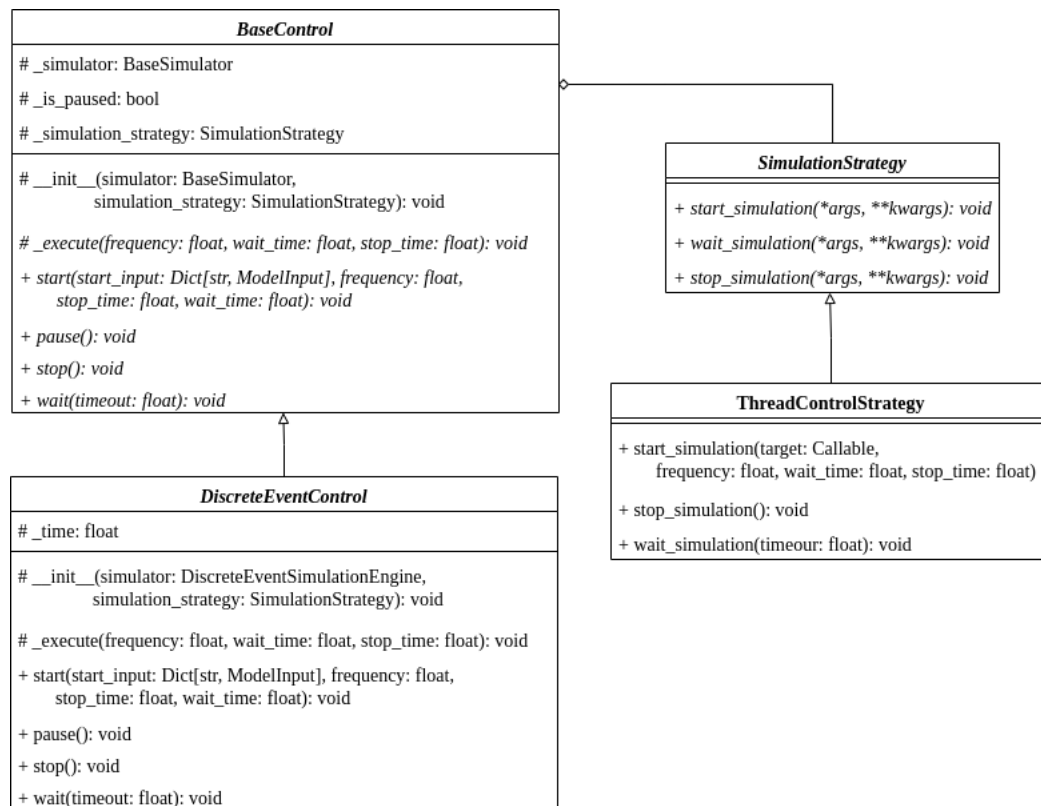


Figura 4.13: Diseño de control de simulación. De autoría propia.

La interfaz de control vista en la Figura 4.13, se definió como *BaseControl* con un simulador, una estrategia de simulación y una variable que indica si la simulación está pausada como atributos, mientras que se pensaron varios métodos abstractos para el control de la simulación.

Como se deseaba que el control fuera accesible durante la corrida de una simulación, se tomó la decisión de ejecutar las simulaciones en un proceso diferente. No obstante, posteriormente se vio que se podía lograr el mismo efecto con hilos en vez de procesos, surgiendo la clase *ThreadControlStrategy* con el objetivo de administrar hilos durante la simulación.

Como acoplar directamente un elemento de infraestructura¹⁰ a un proceso de dominio no se creía conveniente, surgió la clase abstracta *SimulationStrategy* que es usada para delegar aquellos detalles de implementación que pueden variar de acuerdo a la infraestructura.

¹⁰ Detalles de implementación. Su funcionalidad puede ser realizada por otra alternativa según convenga.

En cuanto al módulo de simulación y reportes, el diseño de clases puede ser presenciado en la Figura 4.14.

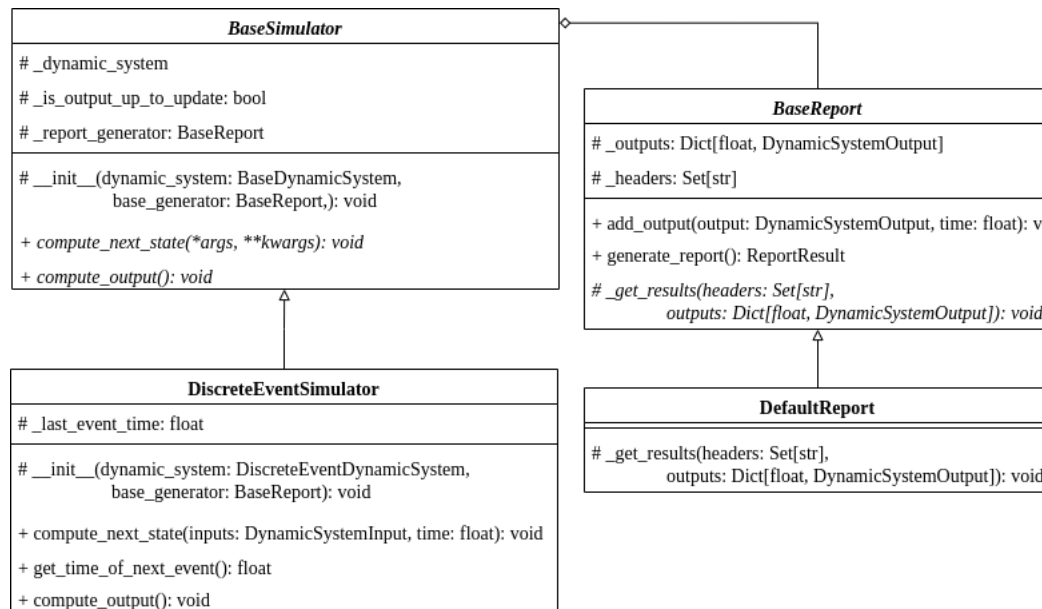


Figura 4.14: Motor de simulación y reportes. De autoría propia.

El simulador se basó en el diseño propuesto por Nutaro [6] para simuladores de eventos discretos, donde se computaba un siguiente estado del sistema dinámico, se obtenía una salida y se recibía el tiempo del evento inminente. Como se deseaba que el *framework* pudiera adaptarse a otro tipo de simulaciones, se generalizó el concepto de motor de simulación con *BaseSimulator*.

La clase abstracta define lo mínimo necesario para ejecutar todo tipo de simulaciones: un sistema dinámico para simular, un método para calcular el próximo estado y un método para obtener la salida de la simulación. De manera adicional, se aprovechó la definición para permitir la inyección de un módulo de reportes de simulación.

Por el lado del módulo de reportes, se diseñó una clase abstracta *BaseReport*, la cual posee un método público para añadir salidas en un tiempo, y otro método para generar el reporte, el cual delega la creación del reporte a un método abstracto.

El proceso de simulación se diseñó a partir del algoritmo que se seguía manualmente en los prototipos anteriores, siendo posible visualizar su automatización en el diagrama de secuencia de la Figura 4.15.

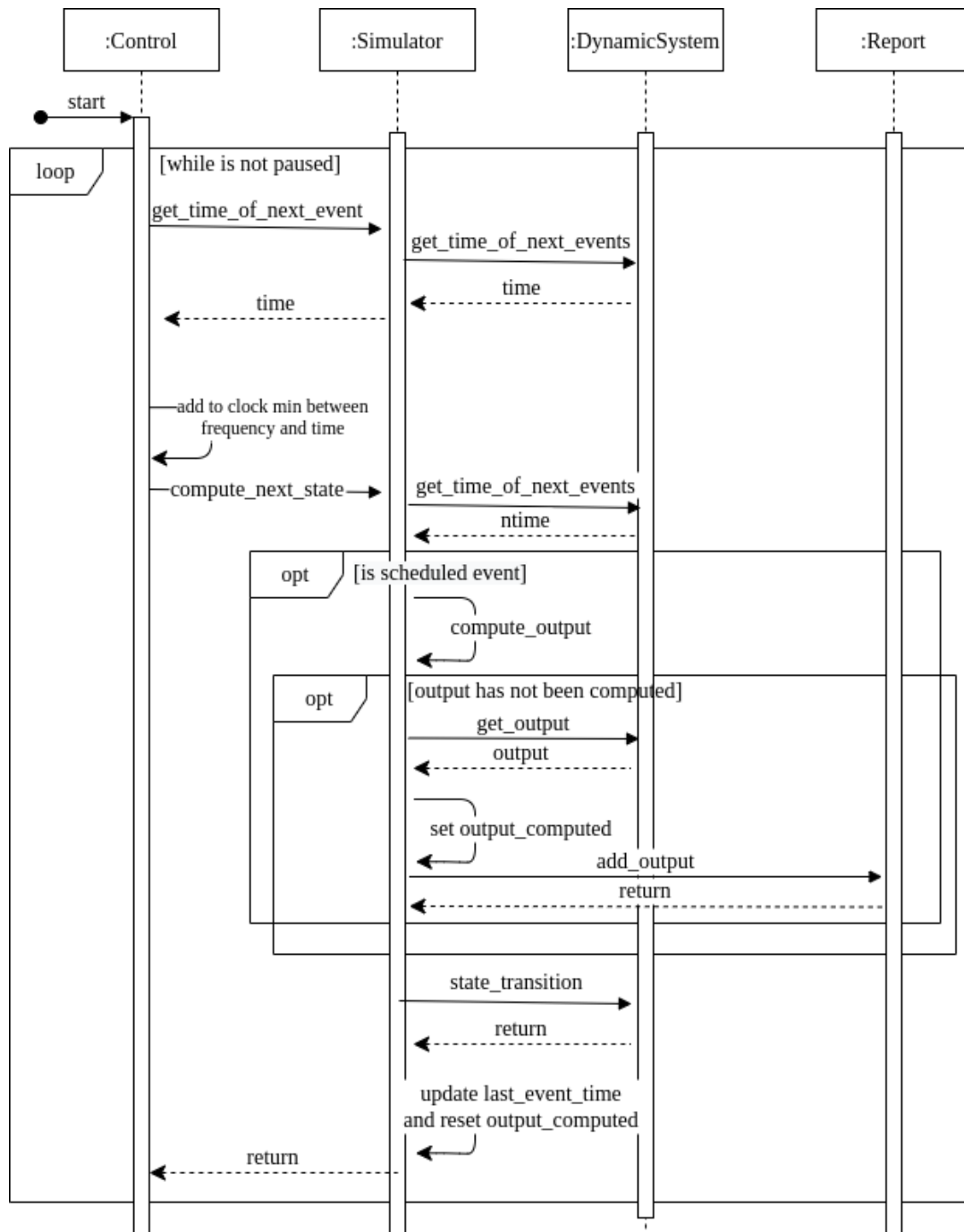


Figura 4.15: Diagrama de secuencia (simplificado) del ciclo de simulación. De autoría propia.

El módulo de control lleva un reloj que aumenta su tiempo en base a la frecuencia o el tiempo del próximo evento, dependiendo de cual sea un tiempo menor, pasando una solicitud al simulador para computar el próximo estado en el tiempo de reloj.

El simulador determina si el tiempo de llegada corresponde a un tiempo de evento

o un tiempo externo a los planificados. En caso de ser un evento planificado, computa la salida del modelo y la almacena en el módulo de reportes.

Una vez obtenida la salida, el simulador produce la transición de estado en el sistema dinámico como se hizo en el prototipo anterior, y avanza su tiempo de simulación.

4.2.4.2. Fase de implementación

La implementación no implicó grandes cambios en la base de código existente, siendo las nuevas funcionalidades casi la totalidad de las novedades introducidas.

Con el fin de manejar múltiples hilos, se utilizó la librería estándar de Python *threading*, creando un nuevo hilo con el método de ejecución de la simulación perteneciente a las subclases de *BaseControl*, y eliminando el hilo al detener la simulación para optimizar recursos y permitir la serialización de los objetos. Por su parte, el método de espera de simulación se implementó mediante la unión del hilo de simulación al hilo principal.

En cuanto a los reportes, se decidió que los reportes por defecto usaran la librería *PrettyTable*, que permite imprimir los datos por consola de manera comprensible. No obstante, se creó una clase intermedia para no acoplarse directamente a la librería en caso de que a futuro se quisiera usar otra herramienta.

Adicionalmente, se integró una librería de monitoreo para el muestreo de trazas de ejecución llamada *loguru*. Para su uso, se creó un decorador¹¹ *debug* que sirviera tanto de medio de desacople, como para facilitar la realización de registros.

Tras la realización de algunas pruebas, se vio la necesidad de crear un nuevo módulo para la definición de experimentos, con el objetivo de ensamblar los módulos. Los experimentos pueden recibir cualquier módulo en su constructor y exceptuando el módulo de sistema dinámico, puede instanciar cualquier otro módulo por defecto que se haya definido en el *framework*.

No se quiso instanciar un sistema dinámico por defecto porque el sistema dinámico es independiente a la simulación, por lo que pese a poder crear un sistema vacío, al no hacerlo, se obliga al usuario a definirlo, lo que da más control a la hora de diseñar una

¹¹ Es una función que encapsula (decora) otras funciones para la extensión de funcionalidades.

herramienta de simulación.

A su vez, los módulos de control, reporte y simulación se decidieron auto-instanciar en caso de que no sean proveídos, porque son dependientes del tipo de simulación, permitiendo la definición de un comportamiento por defecto, aún sin conocer el sistema dinámico.

4.2.4.3. Fase de instanciación y pruebas

Para las pruebas se realizó el mismo modelo de la fase 4.2.3.3, instanciando las nuevas clases para la automatización de los procesos de simulación.

Al no poder inyectar entradas al sistema dinámico una vez iniciada la simulación, se creó un tercer modelo llamado *Generator*¹² que le proporciona las entradas a *Press*.

4.2.5. Prototipo 5. Interfaz gráfica de simulador de redes de cola

Con la base de un simulador funcional, se pensó en elaborar la interfaz gráfica de un simulador de redes de cola. De esta manera, sería más sencillo identificar los elementos más importantes del simulador a fin de desarrollarlos en los futuros prototipos.

4.2.5.1. Fase de definición de arquitectura

Al ser un nuevo software se tuvo que definir la arquitectura del mismo. En la Figura 4.16 se ven los tres módulos que componen al simulador: el modelador, la aplicación y el servidor de aplicación.

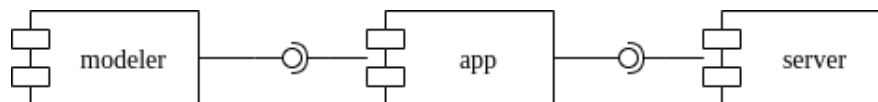


Figura 4.16: Componentes del simulador. De autoría propia.

El servidor se ideó para que tuviera la funcionalidad de servir los archivos estáticos del simulador y formar una capa intermedia entre el *kernel* y la aplicación.

¹² Sería el equivalente al modelo "parte" de la figura 4.12, el cual no se consideró en el prototipo anterior al poder insertar entradas al sistema dinámico en cualquier tiempo t .

El modelador es un componente que se pensó para crear diagramas de redes de cola, pudiendo ser de autoría propia o de terceros.

Para el manejo de datos y comunicación con el servidor, se optó por la utilización de un almacén de datos, compuesto de cuatro (4) módulos visibles en la Figura 4.17, cada uno correspondiente a los principales dominios de la aplicación.

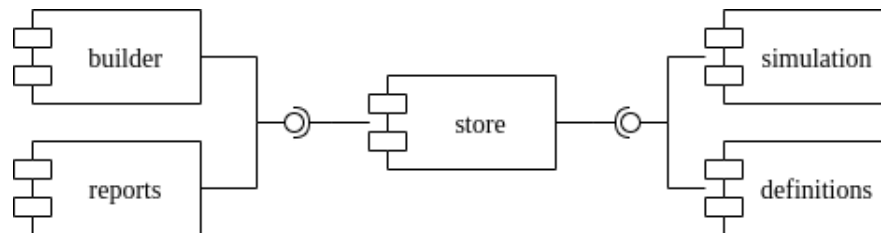


Figura 4.17: Módulos del store. De autoría propia.

4.2.5.2. Fase de diseño detallado

Al principio del proyecto se pensaba utilizar una librería de terceros para realizar el modelador, no obstante, durante la investigación no se encontró ninguna herramienta adecuada¹³ a las necesidades de la aplicación, por lo que se tuvo que diseñar rápidamente un modelador para soportar los requerimientos, surgiendo el diagrama de la Figura 4.18.

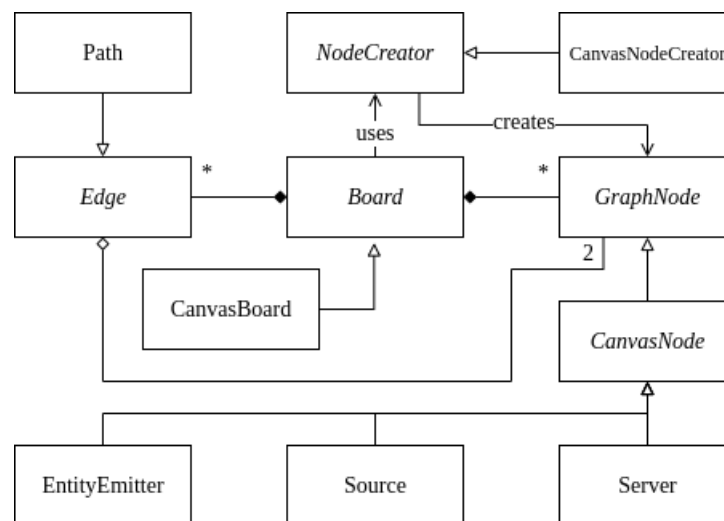


Figura 4.18: Diagrama de clases del modelador. De autoría propia.

¹³ Se buscaron herramientas para hacer diagramas que estuvieran en mantenimiento, con buena documentación y fueran completamente gratuitas.

En cuanto al dominio del modelador, se encuentran las clases abstractas *Board*, *Edge* y *GraphNode*, las cuales poseen las funcionalidades básicas para manipular nodos y rutas en redes de componentes.

Extendiendo a las clases abstractas, se pensó utilizar las implementaciones específicas para cierto tipo de infraestructura, siendo valoradas dos posibilidades para los gráficos: HTML Canvas o componentes SVG.

Como los nodos son creados en el *kernel*, para la instanciación del elemento gráfico se decidió implementar una fábrica que heredara de *NodeCreator* basado en la herramienta a elegir. La fábrica, sería pasada como estrategia creacional al tablero, cuando se desee añadir un nodo.

El tablero por su parte, manejaría todo lo referente a los nodos y rutas, el estado del tablero¹⁴, y los eventos tanto externos como internos que se ejecuten en el modelador.

En cuanto a la aplicación, no se hizo un diseño detallado significativo, siendo tomado como referencia los elementos de la interfaz de Simio Simulation Software.

La interfaz de simulación estuvo inspirada en la vista de la Figura 4.19.

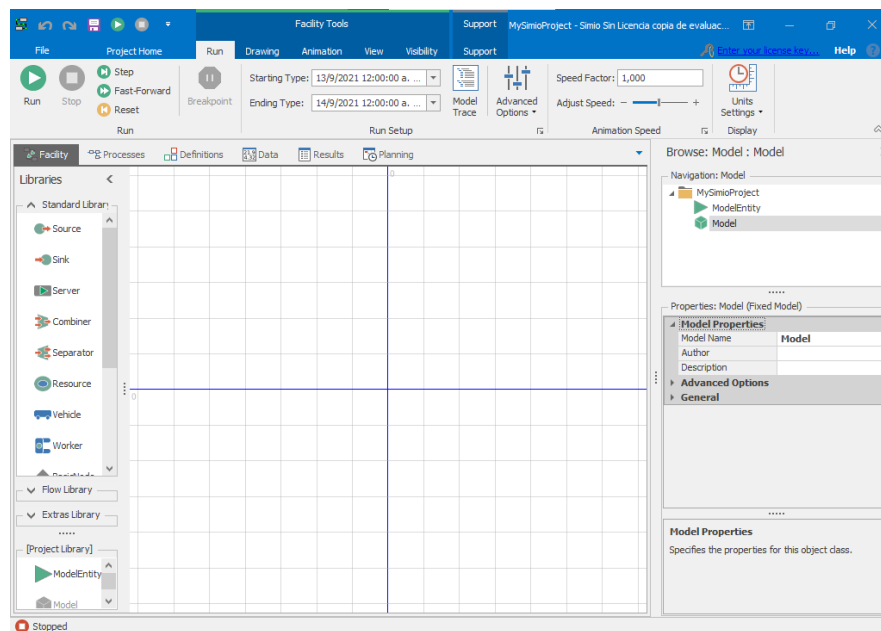


Figura 4.19: Vista de simulación de Simio. De autoría propia.

Para la vista se consideró como imprescindible la inclusión de los campos para

¹⁴ Se consideraron los estados: por defecto, eliminación de elementos y creación de rutas.

indicar el tiempo de simulación y los botones de control de la simulación. El modelador debe tener un tablero y un contenedor para seleccionar los componentes a agregar, y los componentes deben disponer de un inspector de propiedades.

La interfaz de resultados en principio se quiso parecida a la vista de *Pivot Grid* (Figura 4.20) de resultados de Simio.

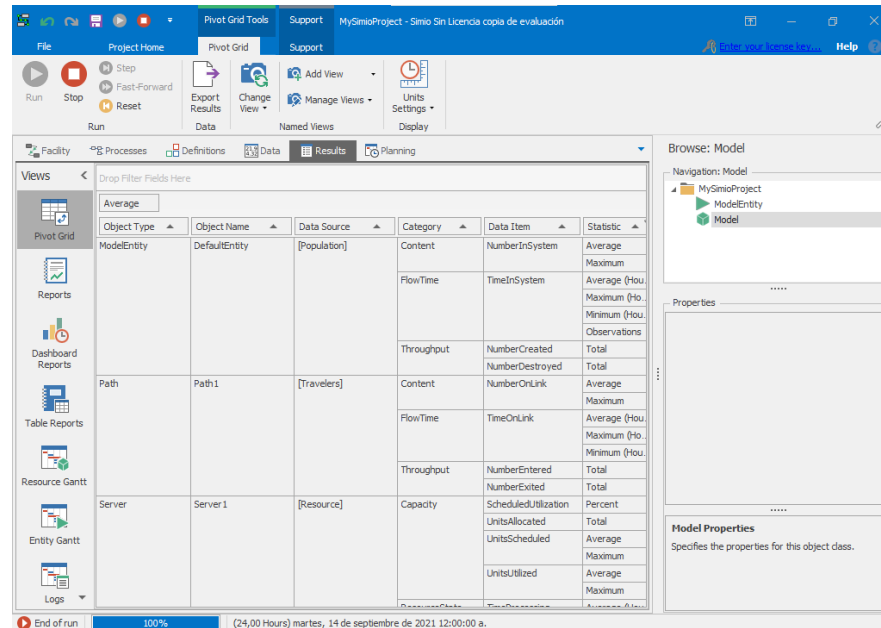


Figura 4.20: Vista de resultados de Simio. De autoría propia.

Adicionalmente, Simio dispone de otras vistas con muchas otras funcionales que no se consideraron en este primer prototipo.

4.2.5.3. Fase de implementación

La fase de implementación fue bastante compleja, estando principalmente enfocada en el modelador.

Al tablero abstracto se le agregaron funcionalidades de bus de eventos para permitir la emisión de eventos hacia la aplicación. De esta manera, el modelador es independiente de la aplicación.

La implementación del tablero se hizo con la API Canvas, debido a que se tenía amplia experiencia y proyectos base que usaban esta tecnología, lo que agilizó el desarrollo de los elementos gráficos y las acciones como dibujar, seleccionar y

arrastrar.

Los elementos creados están fuertemente influenciados por los componentes disponibles en Simio. Estos se representan de la siguiente forma:

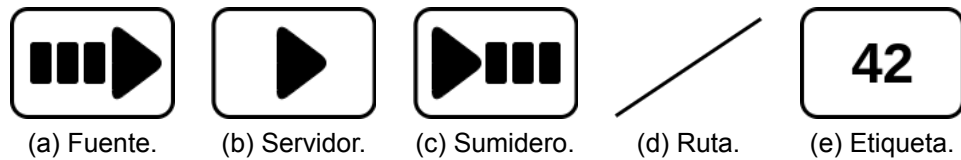


Figura 4.21: Representación de los elementos del simulador basados en Simio. De autoría propia.

Con el fin de dibujar y manipular las posiciones de los elementos del tablero se usaron las funcionalidades de *CanvasRenderingContext2D*. Es necesario controlar las posiciones de los elementos con este método o uno que lo extienda, porque el lienzo dibuja los elementos con base en este contexto, por lo que de usarse otro contexto¹⁵ para manejar las posiciones, los orígenes de renderizado¹⁶ podrían estar en distintas posiciones, dando problemas a la hora de seleccionar elementos.

En cuanto a la aplicación, se implementaron las interfaces usando Vue como *framework*, Vuetify como librería de componentes y Vuex como almacén de datos.

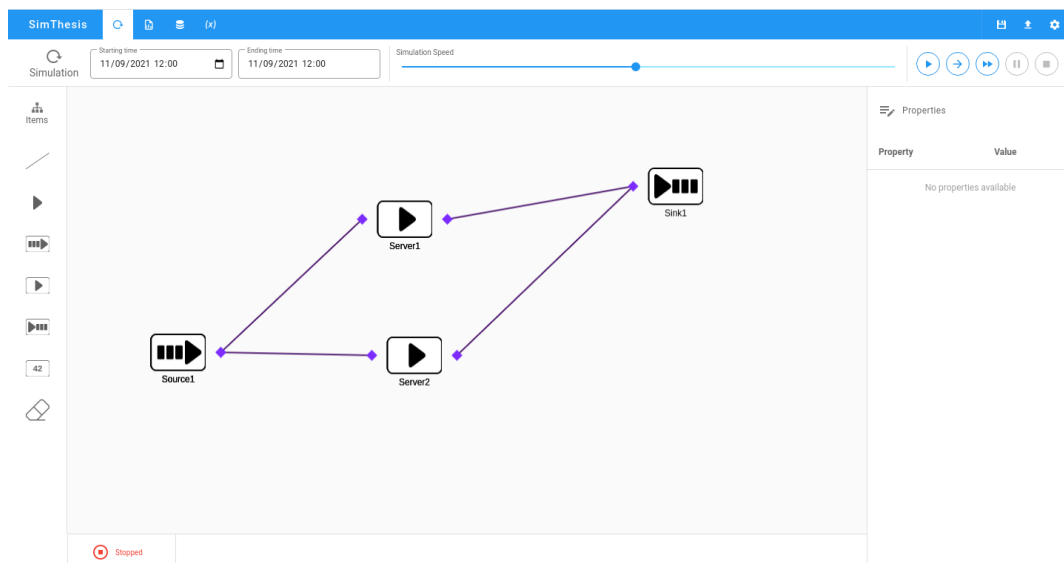


Figura 4.22: Vista inicial de simulación. De autoría propia.

¹⁵ Otro contexto podría ser el mismo componente Canvas u otra etiqueta HTML.

¹⁶ Como un plano de coordenadas cartesianas, el origen de los elementos HTML está en la esquina superior izquierda, mientras que el de *CanvasRenderingContext2D* es móvil.

La vista de simulación que se puede ver en la Figura 4.22, fue creada a partir de seis (6) componentes: el tablero al centro, la barra de elementos a la izquierda, el inspector de propiedades a la derecha, la barra de estado al pie, la barra de simulación a la cabeza, y el gestor de vistas como las pestañas por encima de la barra de simulación.

4.2.6. Prototipo 6. Emisor de entidades y fuente

Para desarrollar un simulador completamente funcional se empezó este prototipo donde se esperaba crear un generador de entidades y una fuente a partir de la extensión del *framework*.

4.2.6.1. Fase de diseño detallado

Las entidades poseen identidad y propiedades. Para corregir la ausencia de propiedades en el *framework*, se creó la clase genérica *EntityProperty* visible en la Figura 4.23, la cual sirve de contenedor para describir y guardar los valores de una propiedad.

Con el prototipo 4.2.4 se realizó un simulador con un generador, prensa y taladro. El generador sería equivalente a una fuente, por lo que fue la base para el diseño de la misma.

EntityProperty<T>
_value: T # _type: str # _category: str
__init__(value, type, category): void + get_type(): str + get_category(): str + get_value(): T + set_value(value: T): void

Figura 4.23: Propiedad de entidad. De autoría propia.

El principal problema del generador del prototipo anterior era que solo generaba un tipo de entidad, por lo que el tipo de entidad, debería convertirse en un parámetro para permitir la inclusión de tipos adicionales, surgiendo la necesidad de los emisores de entidades.

Como el comportamiento de generación de entidades no solo pertenece a los simuladores de redes de cola, se decidió incluir esta funcionalidad en el núcleo del *framework*, creando la clase abstracta *EntityEmitter* con un método para generar entidades, como se muestra en la Figura 4.24.

Extendiendo a la clase, el simulador implementó *Emitter*, permitiendo obtener las propiedades de las entidades a crear y generar nuevas entidades. Como el método

get_properties devuelve un arreglo asociativo con la referencia a las propiedades, en teoría no hace falta métodos para establecerlas, ya que un cambio del objeto obtenido representaría un cambio en el objeto original.

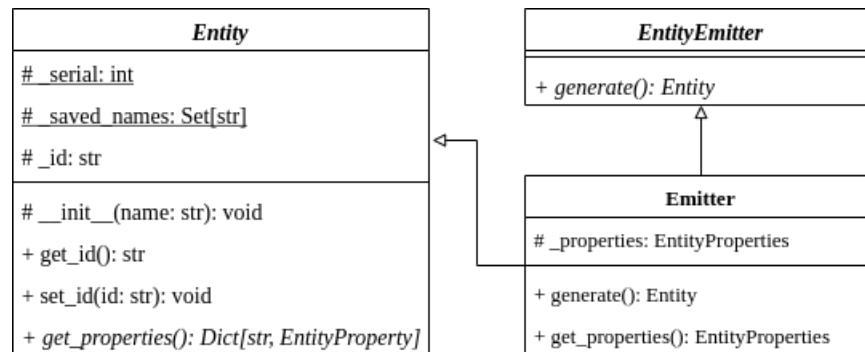


Figura 4.24: Diseño de emisor de entidades. De autoría propia.

En cuanto a la fuente, el diseño estuvo basado en el componente *Source* de Simio, descrito por la guía de referencia de uso de Simio desarrollada por la Pontificia Universidad Católica de Valparaíso [41].

Los parámetros y métodos propios del componente, fueron escogidos y codificados durante la fase de implementación y pruebas.

La mayoría de valores pertenecientes a las propiedades de los elementos de redes de cola estudiados, pueden tener varias formas de expresarse: un valor constante, una distribución, una igualdad, etc.

Ante la ausencia de las expresiones en el ambiente de desarrollo, se creó una clase abstracta *Expression* que pueda ser implementada por cualquier valor factible de ser evaluado. En esta fase del diseño, solo se consideraron las constantes creadas a través de la clase genérica *Value*, y las distribuciones aleatorias que sigan la interfaz establecida por la clase abstracta *RandomDistribution* como se muestra en la Figura 4.25.

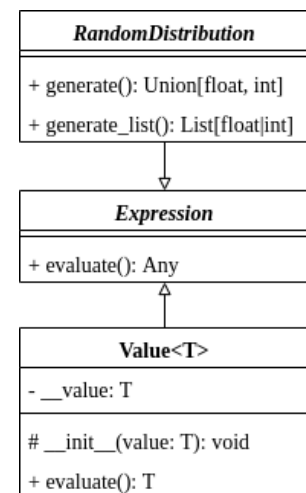


Figura 4.25: Diseño de expresiones. De autoría propia.

Durante la implementación de la fuente surgió la necesidad de una clase *Buffer*¹⁷,

¹⁷ El equivalente a una cola para el sistema desarrollado.

siendo diseñada para almacenar y manejar un grupo de entidades según una capacidad y disciplina. La clase fue extendida por una cola de entrada, salida y procesamiento.

4.2.6.2. Fase de implementación

La implementación implicó más cambios de los esperados en el *framework*, al incluir varios conceptos que no se manejaron en un principio.

Las expresiones y propiedades fueron agregadas al ambiente de desarrollo en el módulo *core*, al no considerarlos propios de un dominio específico de simulación.

Extendiendo el marco de trabajo no existió mucho problema, pudiéndose crear relativamente rápido la fuente y el emisor de entidades, siendo el principal problema a afrontar la lógica de las colas.

Durante la etapa se pensó en la existencia de tres tipos concretos de *Buffer* como se expone en la Figura 4.26.

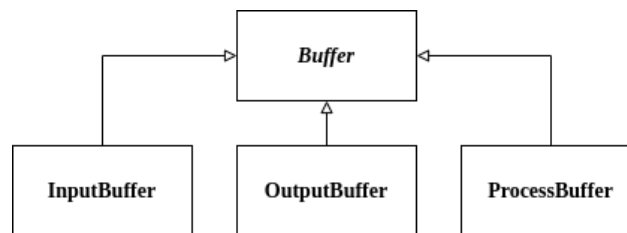


Figura 4.26: Extensión de los *buffers*. De autoría propia.

La fuente solo posee un *buffer* para llevar las salidas. No obstante, se implementaron también los *buffers* de procesamiento y entrada, debido a que el comportamiento se creía idéntico.

Al *buffer* abstracto se le implementaron los métodos para encolar, desencolar y ver el contenido. Las disciplinas implementadas sobre fueron FIFO, LIFO y aleatorio, usando una lista cuya lógica de recorridos se muestra en la Figura 4.27.

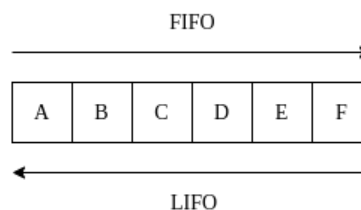


Figura 4.27: Ejemplo de disciplinas. De autoría propia.

La fuente fue creada con las siguientes propiedades:

Propiedad	Tipo	Descripción
Entity Type	EntityEmitter	Estrategia de generación de entidades.
Interarrival Time	Expression	Intervalo de tiempo entre dos llegadas sucesivas.
Entities Per Arrival	Expression	Número de entidades que serán creadas por llegada.
Time Offset	Expression	Tiempo a esperar antes de generar la primera llegada.

Tabla 4.1: Propiedades de la fuente. Adaptado de [41].

Cabe destacar que al ser propiedades, el tipo en realidad pertenece al genérico *EntityProperty<T>*. Para facilitar la definición del tipo de propiedades¹⁸ se crearon clases de apoyo que automatizan la definición del tipo, como lo pueden ser *NumberProperty* y *ExpressionProperty*.

Con el fin de facilitar el uso, los métodos de asignación de las propiedades se codificaron para aceptar tanto el tipo definido en la Tabla 4.1 como los encapsulados por clases de apoyo para propiedades.

4.2.6.3. Fase de instanciación y pruebas

Los métodos y atributos de la fuente fueron probados de manera aislada asignando un sistema dinámico y emisor de entidades de prueba con comportamientos por defecto.

Una vez arreglados todos los detalles de implementación encontrados en las pruebas, se pasó a la validación del prototipo por medio de la comparación directa con los resultados de Simio.

4.2.7. Prototipo 7. Servidor y distribuciones matemáticas

El siguiente prototipo a desarrollar con el *framework* consistió en una estación de servicio y la inclusión de distribuciones de probabilidad en la librería matemática provista por el marco de trabajo.

¹⁸ Número, booleano, expresión o cualquier otro tipo.

4.2.7.1. Fase de diseño detallado

En el diseño de los primeros prototipos no se vio la necesidad de una librería matemática incluida en el *framework*, no obstante, tras la realización de la fuente, se vio que podía ser una utilidad proveída para facilitar la reutilización.

Las distribuciones fueron diseñadas basadas en las expresiones del prototipo anterior, extendiendo la clase abstracta *RandomDistribution* como se muestra en la Figura 4.25, lo que permite generar un número o una colección de números aleatorios siguiendo algún tipo de distribución.

El diseño del servidor estuvo basado en el componente *Server* de Simio, por lo que no se tuvo que realizar una planificación exhaustiva del funcionamiento del mismo.

Durante la elaboración de pruebas, se descubrió que los servidores de Simio cuando poseen una capacidad mayor a un elemento, procesan con una disciplina en paralelo por defecto como si se tratara de un procesamiento hecho por varios servidores, siendo una política que no se había previsto con los elementos anteriores. De la misma forma, el procesamiento en paralelo y los distintos tiempos de procesamiento, obligaron a rediseñar los métodos de encolamiento y desencolamiento del *buffer* de procesamiento, pues el tiempo juega un papel fundamental. Ante estos problemas, se hizo el diseño de la Figura 4.28.

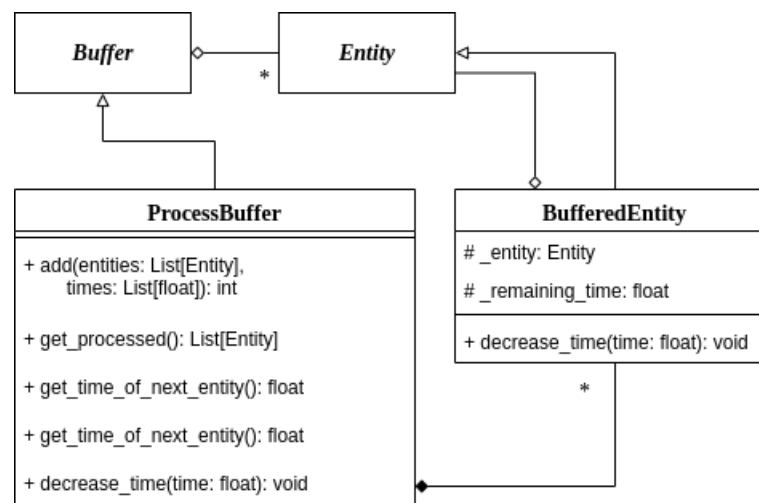


Figura 4.28: Diseño de la cola de procesamiento. De autoría propia.

Con el fin de no aplicar cambios en los otros componentes, se decidió extender la

clase *Entity* para crear la clase *BufferedEntity*, que encapsula una entidad y un tiempo. De esta manera, no hace falta editar la estructura de datos original del *buffer* que almacena entidades, y los cambios necesarios se pueden hacer en la subclase.

El planteamiento de las prioridades es idéntico al proceso realizado con la lista de eventos futuros diseñada en la sección 4.2.3.1.

La desventaja del método escogido para no alterar los elementos del prototipo anterior, es que se creó una composición fuerte entre *ProcessBuffer* y *BufferedEntity*, que pese a no afectar la investigación, crea un acoplamiento que puede afectar la modificación del simulador desarrollado, ya que de quererse usar otra estrategia para procesar las entidades, tendría que programarse nuevamente todos los métodos propios del *buffer* de procesamiento, pese a existir un algoritmo probado y funcional ya implementado para esta tarea.

4.2.7.2. Fase de implementación

Se implementaron tres (3) distribuciones aleatorias: Poisson, Exponencial y Triangular, utilizando la librería *numpy* para agilizar el desarrollo de las distribuciones y garantizar la exactitud de resultados.

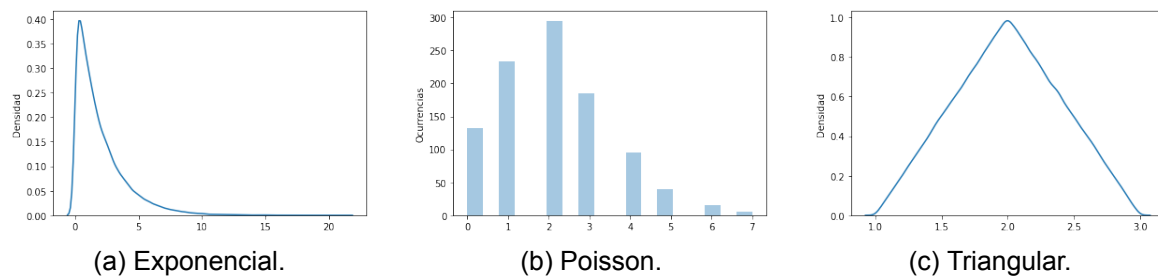


Figura 4.29: Visualización de las distribuciones. De autoría propia.

Al haber muchos tipos de distribuciones, ser un proceso muy repetitivo y a falta de tiempo para probar el funcionamiento correcto de cada una, se decidió no implementar más distribuciones que las ya realizadas.

El simulador de Simio incluye muchas más propiedades de las que se implementaron, ya que no se consideró el desarrollo de varias de las funcionalidades que soporta Simio que hacen uso de esas propiedades.

Las propiedades creadas fueron las siguientes:

Propiedad	Tipo	Descripción
Initial Capacity	Expression	Capacidad inicial del servidor.
Processing Time	Expression	Tiempo requerido para procesar cada una de las entidades.

Tabla 4.2: Propiedades del servidor. Adaptado de [41].

El estado del servidor está compuesto por los *buffers* de entrada, salida y procesamiento. El servidor mueve las entidades entre *buffers*, a través de sus transiciones de estado como se muestra en las Figuras 4.30 y 4.31.

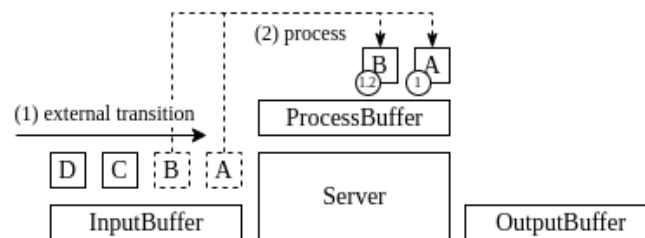


Figura 4.30: Pasos en transición externa. De autoría propia.

Al recibir una transición externa el servidor introduce las entradas en su cola de entrada (1), luego, si hubo al menos un elemento de entrada, actualiza los tiempos de procesamiento restante del *buffer* de procesamiento y verifica si se puede procesar algún elemento de la cola de entrada. Si un elemento de la cola de entrada puede procesarse, se programa un evento autónomo con un tiempo equivalente al tiempo de procesamiento (2).

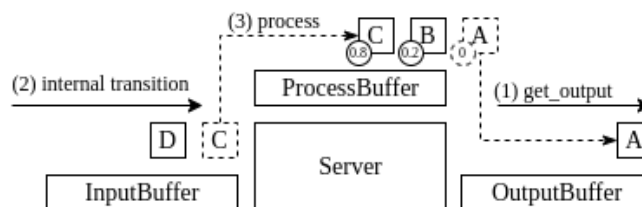


Figura 4.31: Pasos en transición autónoma. De autoría propia.

En cuanto a las transiciones autónomas, al ejecutar la función de salida, desencola los elementos inminentes y resta los tiempos de los mismos en el *buffer* de procesamiento y los introduce en la cola de salida (1), que se vacía como resultado de

la misma función. Ya ejecutando la transición interna (2), se repite el procedimiento de procesamiento (3).

4.2.7.3. Fase de instanciación y pruebas

Los métodos y atributos del servidor fueron probados de manera aislada, dejando los métodos de transición para una prueba de simulación

Una vez arreglados todos los detalles de implementación encontrados en las pruebas, se pasó a la validación del prototipo por medio de la comparación directa con los resultados de Simio.

Las validaciones resultaron correctas para las simulaciones con números enteros, pero se descubrió que el simulador ejecutaba bucles infinitos tanto por la fuente como por el servidor, en los casos donde se manejaban números con parte decimal, lo que implicaba un fallo del ambiente de desarrollo.

4.2.8. Prototipo 8. Simulador red de colas (consola)

Al principio del proyecto se planteó este prototipo para elaborar la primera versión funcional del simulador de redes de cola, con la inclusión de métodos y componentes para interactuar con el simulador. No obstante, tras los fallos encontrados en el prototipo anterior, se decidió incluir las mejoras en este prototipo debido a que los procesos involucrados en la solución estaban estrechamente relacionados con los trabajados en esta entrega.

4.2.8.1. Etapa de diseño general

El problema más urgente a solucionar del prototipo anterior correspondía a los ciclos infinitos producto de operaciones con números decimales.

Tras un extenso estudio del código se encontraron dos razones por las que surgían estos bucles: tiempos negativos y operaciones algebraicas; ambas teniendo como origen del problema la representación de los números flotantes.

Representar un número infinito de números reales en un número finito de bits obliga

a la representación aproximada de los mismos, lo que provoca varios errores conocidos dentro del área de la aritmética del ordenador [42].

En los casos donde se ejecutaba un evento inminente, durante el algoritmo de simulación diseñado (visible en la Figura 4.15), se sumaban y restaban las mismas cantidades, lo que en el apartado matemático equivale a la cancelación, pero la aproximación que generaba la computadora provocaba tiempos negativos y tiempos extremadamente reducidos, que debido a la aritmética del ordenador, se seguían reduciendo infinitamente. Para resolverlo, se decidió tomar varias acciones al respecto.

En primer lugar, donde sea que se restara el tiempo se decidió incluir una función para obtener el máximo entre cero y el tiempo evaluado, por lo que todos los tiempos producto de una operación algebraica serán neutros o positivos.

En cuanto a la representación de los datos, se decidió prescindir del tipo de dato *float*, utilizando *Decimal* en su lugar, lo que permitiría una precisión ajustable y una representación más exacta de los números. Para facilitar el intercambio de tipo en caso de ser necesario, se creó un tipo *Time*, que desacopla débilmente¹⁹ el tipo de dato, lo que lo hace fácil de intercambiar siempre y cuando solo se utilicen sus operadores y no sus métodos.

Finalmente, se eligió una precisión de seis (6) decimales para los números reales, redondeado los valores hacia el valor más cercano para aquellos números que poseen una representación con más dígitos.

4.2.8.2. Fase de diseño detallado

En cuanto al diseño específico de la etapa, se debía elaborar una forma de manejar rutas ponderadas a nivel de *framework* e incluir los métodos e interfaces para interactuar con el simulador.

Para conectar modelos entre sí nació la clase *Path*, una entidad que une dos modelos y posee una propiedad peso. Su diseño se puede apreciar en la Figura 4.32.

¹⁹ Como no sigue una interfaz, no desacopla completamente, siendo posible usar métodos que no están definidos en otros tipos de dato, pudiendo provocar fallos en caso de que se cambie.

El gran inconveniente de incluir la nueva clase, estuvo en el algoritmo de enrutamiento existente que conectaba directamente los modelos entre sí gracias a la tabla *hash* de entradas.

Se decidió reutilizar el mismo sistema dinámico, por lo que se renombró el atributo *_inputs* a *_paths*, y se modificó para que en vez de contener un conjunto de modelos, poseyera un conjunto de rutas. A su vez, se renombraron los métodos para agregar y eliminar conexiones entre modelos.

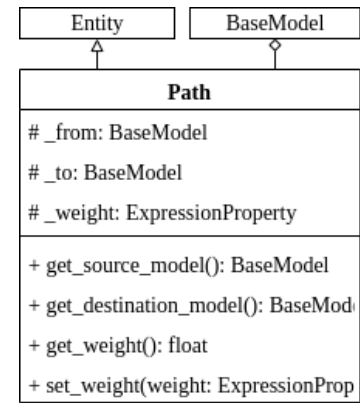


Figura 4.32: Nueva clase ruta. De autoría propia.

El algoritmo de enrutamiento seguiría siendo el mismo en palabras, cambiando únicamente la asignación del conjunto de los modelos afectados (Figura 4.11), pues ahora al existir condiciones para afectar un modelo, el conjunto de modelos de salidas no equivale al conjunto de modelos afectados, por lo que hace falta elaborar nuevos procesos.

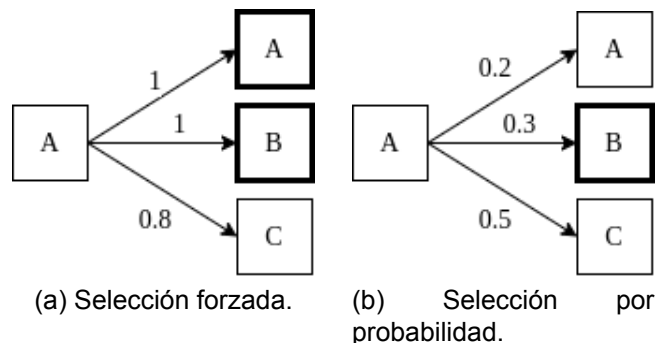


Figura 4.33: Algoritmo de selección por defecto. De autoría propia.

Con el fin de elegir las rutas, se planeó que en caso de haber una o varias rutas con un peso igual a uno²⁰ (1), los modelos afectados serán el destino de esas rutas (caso de la Figura 4.33.a), en cambio, de no haber ninguna con esas condiciones, se hace una elección aleatoria de una ruta de acuerdo a los pesos de cada una (caso de la Figura 4.33.b).

Cabe destacar que este algoritmo se puede modificar por extensión del método

²⁰ En caso de ser un peso originado por una expresión booleana, el 1 es equivalente a True.

protegido creado en el sistema dinámico.

4.2.8.3. Fase de implementación

Se realizaron los cambios diseñados y se prosiguió con la implementación de los componentes faltantes y elementos para interactuar con el simulador.

El sumidero fue implementado como un modelo de eventos discretos, cuyas entradas son recibidas por un *buffer* de entrada, y cuya eliminación es programada inmediatamente después de su llegada a la estación.

La elaboración de las simulaciones fue posible gracias a la extensión del sistema dinámico y del experimento.

Se agregaron algunos métodos por extensión en el sistema dinámico para facilitar algunos procesos como la inicialización automática de los modelos y la obtención de modelos y rutas por su identificador.

Al experimento se le añadieron por extensión métodos para crear y eliminar nodos, rutas, modificar propiedades, obtener estadísticas, entre otras funcionalidades que no están modeladas en la superclase perteneciente al *framework*.

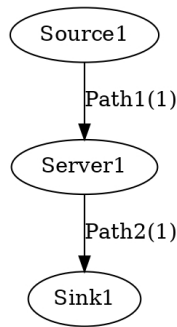
De manera adicional, se creó una clase para instanciar etiquetas, elementos que están atados a la referencia de una propiedad de cualquiera de los componentes del simulador y que sirve para accederlos de manera más rápida.

El algoritmo de enrutamiento por defecto, multiplica las entidades en los casos donde se encuentren dos o más rutas con un peso de uno (1). Esto se debe a que el algoritmo general trabaja con el enrutamiento de salidas, siendo posible su propagación a varias estaciones a la vez; no obstante, como las entidades son únicas, el enfoque general empleado es insuficiente para el tipo de herramienta desarrollada.

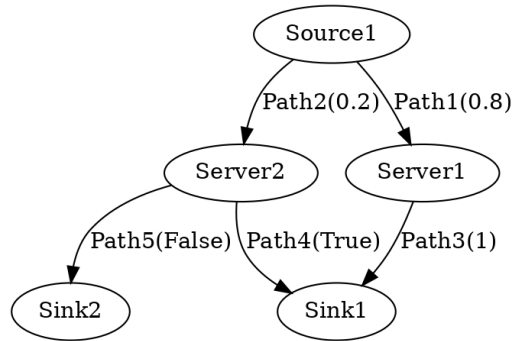
Con la finalidad de corregir el error de multiplicidad de entidades, se extendió el algoritmo de selección de rutas para que sumara los pesos de todas las rutas y los dividiera entre el total de la suma de los pesos. De esta manera, todos los pesos estará dentro del rango [0-1] y siempre se escogerá una única ruta.

4.2.8.4. Fase de instanciación y pruebas

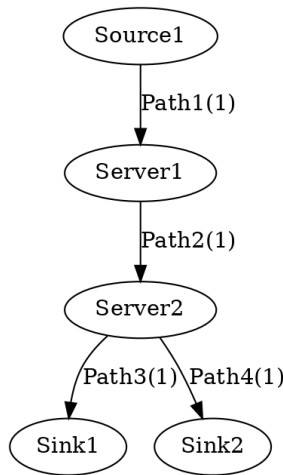
Para la validación de resultados se tuvo que volver a correr las simulaciones del prototipo anterior, y se agregaron simulaciones con los siguientes tipos de redes:



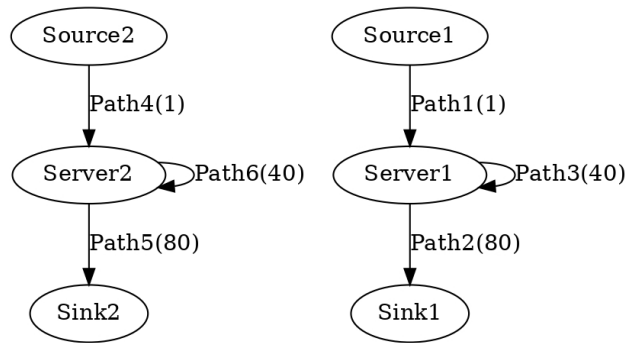
(a) Red sencilla.



(b) Rutas booleanas.



(c) Selección de rutas.



(d) Elementos en paralelo.

Figura 4.34: Redes simuladas. De autoría propia.

En primer lugar se simuló una red con una fuente, servidor y sumidero como el de la Figura 4.34.a, que al ser un modelo sencillo, permitió identificar el origen de la mayoría de problemas y errores.

Con el fin de probar el correcto funcionamiento de la selección de rutas por

expresiones booleanas y el procesamiento de varios servidores en paralelo, se creó el modelo de la Figura 4.34.b, donde se esperaba que todas las llegadas fueran dirigidas hacia el primer sumidero.

Por otro lado, como era inadmisibile la multiplicidad de entidades, se tuvo que probar el algoritmo de selección de rutas para garantizar la selección de una única ruta, por lo que se diseñó el modelo de la Figura 4.34.c, donde una entidad que sale desde el servidor debe dirigirse a uno de los dos sumideros.

Finalmente, se probó simular dos redes en paralelo, idénticas y sin conexión alguna como se muestra en la Figura 4.34.d. De manera adicional, esta prueba incluye la selección entre una ruta que emite entradas a la misma red y la ruta que dirige las entidades hacia el sumidero.

4.2.9. Prototipo 9. Conexión con el *kernel*

El simulador de redes de cola de consola implementado se buscó adaptar como *kernel* para el simulador final, siendo necesario establecer la comunicación bidireccional con la interfaz de usuario.

4.2.9.1. Fase de definición de arquitectura

Tras analizar la complejidad de desarrollar el sistema de tres capas, se simplificó el diseño de la Figura 4.2, obteniéndose el diseño de la Figura 4.35.

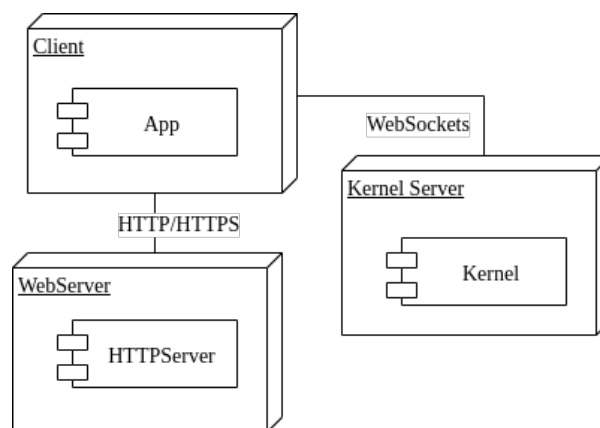


Figura 4.35: Arquitectura del simulador. De autoría propia.

La simplificación consiste en eliminar la capa intermedia, permitiendo la comunicación directa entre aplicación y *kernel* por medio de *WebSockets*.

El nuevo planteamiento se basa en que el servidor web provee los archivos de aplicación al navegador y el servidor del *kernel* es el que modela y corre las simulaciones para las distintas sesiones que se crean desde el cliente, sin necesidad de una capa intermedia, pues ambas capas trabajan sobre el mismo protocolo.

4.2.9.2. Fase de diseño detallado

El diseño de la comunicación se quiso modularizar a través de controladores, con el fin de agrupar las funciones de simulación, reporte y creación de modelos. El diagrama resultante se presenta en la Figura 4.36.

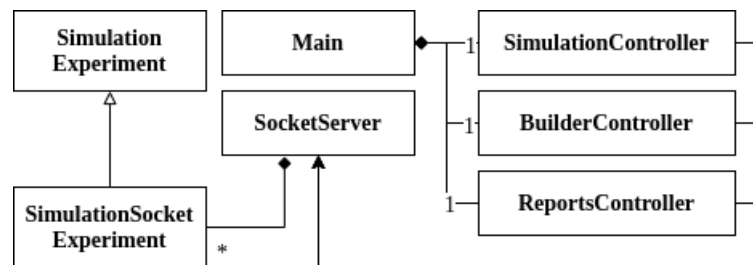


Figura 4.36: Diseño del *kernel* del simulador. De autoría propia.

Los controladores reciben y emiten salidas a través de una instancia global de un servidor de *sockets*, el cual maneja un experimento de simulación por cada cliente que se conecte con el servidor.

El experimento de simulación del prototipo anterior fue extendido para agregar funcionalidades de retransmisión de los eventos de dominio que ocurrían durante la simulación a través de *sockets*.

No se tomaron en cuenta comunicaciones por el método de difusión amplia, siendo todas las comunicaciones modeladas, guiadas por una comunicación entre un cliente específico y el servidor.

4.2.9.3. Fase de implementación

La implementación del servidor sufrió varios cambios respecto al diseño original debido a la naturaleza funcional de la librería seleccionada *Socket.IO* para crear el servidor de *sockets*.

Los controladores se implementaron por medio de la definición de métodos estáticos, decorados usando la librería. La librería maneja distintas sesiones de clientes a través de un identificador único aleatorio *sid*. Por otro lado, permite manejar salones y difusiones amplias sin esfuerzo, por lo que pese a no tomarse en cuenta este tipo de comunicaciones, pueden agregarse sin mucha dificultad.

Para cada identificador, la librería crea una clave en un arreglo asociativo que permiten almacenar variables de sesión. La única variable de sesión definida fue *experiment*, para llevar los experimentos de simulación del cliente.

Al principio de la implementación se utilizó como estrategia de despliegue *eventlet*, lo que trajo bastantes problemas debido a que no permitía la generación de hilos de la librería *threading*, siendo la empleada por el *framework* y el simulador probado.

Tras analizar las alternativas, se delegó la estrategia de despliegue a la librería *Flask*, lo que a su vez podría permitir en un futuro el soporte de peticiones REST, en conjunto con los *WebSockets*.

En el cliente se usaron las funcionalidades de *VueSocket.IO*, que automatizan los procesos de emisión y escucha de eventos de *sockets* en los distintos componentes de Vue.

Al terminar las conexión entre cliente y servidor, se encontró un defecto en el *framework* para la asignación de nombres, pues manejaba propiedades estáticas²¹ para impedir la repetición de nombres o crear seriales con autoincremento, lo que impedía mantener sesiones completamente aisladas.

Pese a que se podía resolver el problema a nivel de aplicación, pareció conveniente arreglarlo a nivel de *framework*. Ante esto, se parametrizaron las funcionalidades de almacenamiento y asignación de nombres a través de una clase *EntityManager*.

Con el fin de garantizar el correcto funcionamiento con las pruebas de las

²¹ Es una propiedad que comparten todas las instancias de una misma clase.

versiones anteriores, se hizo el parámetro opcional y de no asignarse, asigna una instancia estática de la clase.

4.2.10. Prototipo 10. Simulador final

Al tener los mecanismos de comunicación definidos, el prototipo final consistió en juntar las funcionalidades implementadas en el *kernel* y en el cliente, para construir un simulador de red de colas funcional.

4.2.10.1. Fase de implementación

A medida que se desarrollaba determinada funcionalidad, se creaba o modificaba el controlador respectivo en el *kernel* y se hacía su petición en la aplicación cliente.

Como se quería mantener equivalencia entre lo que mostraba en el cliente y sucedía en el *kernel*, se tomó la decisión de esperar la respuesta del servidor para realizar una acción en el cliente. No se descubrió ningún problema con este método, sino hasta que se trabajó con el *kernel* desplegado en un servidor remoto, donde existía un ligero retraso.

Con el fin de manejar las expresiones disponibles en el sistema, se implementó un manejador de expresiones, que mediante un arreglo asociativo, lleva el registro de las expresiones; siendo modificado con la agregación, edición o eliminación de un componente a través del experimento de simulación.

Para la edición de expresiones, el cliente solicita las expresiones al manejador y las lista. Por motivos de agilidad en el desarrollo, el listado de las expresiones mostrado en la Figura 4.37, se hizo completo en vez de separado por categoría, lo que lo hace más incómodo de usar, no obstante, el manejador de expresiones devuelve el listado en forma de tabla *hash* por lo que se podría hacer

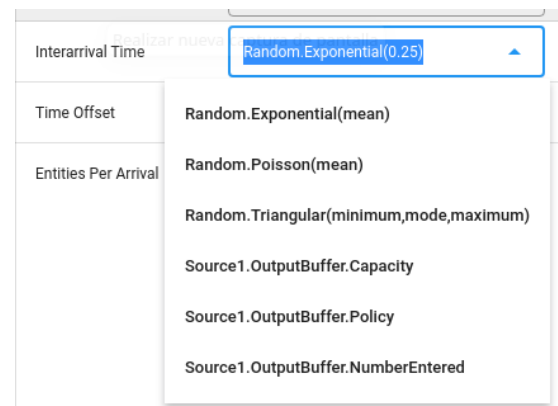


Figura 4.37: Listado de expresiones. De autoría propia.

un filtro con sus claves.

La vista de reportes esperada para el prototipo 5, fue realizada durante el desarrollo de este prototipo, quedando como se muestra en la Figura 4.38.

Object Type	Object Name	Data Source	Data Item	Statistic	Value
Source	Source1	Source1.OutputBuffer	Entered	Total	21.000000
				Maximum	1.000000
			NumberInStation	Minimum	1.000000
				Average	1.000000
				Total	21.000000
Server	Server1	Server1.InputBuffer	Entered	Total	20.000000
				Maximum	1.000000
			NumberInStation	Minimum	0.000000
				Average	0.952381
				Total	20.000000
		Server1.ProcessBuffer	Entered	Total	0.000000
				Maximum	1.000000
			NumberInStation	Minimum	1.000000
				Average	1.000000
				Total	0.000000
				Maximum	1.000000

Figura 4.38: Vista de reportes. De autoría propia.

El reporte es generado a partir de una tabla dinámica, de acuerdo al número de estadísticos, elementos y fuentes de datos por componente. La vista se actualiza cada vez que se acceda a ella.

Pese a que los modelos en el *framework* no poseen bus de datos, se le agregó a los componentes de redes de cola uno, permitiendo así emitir su estado cada vez que es alterado, pudiendo ser visible el cambio en el cliente gracias al mecanismo de redirección implementado en el experimento.

El ajuste de velocidad de simulación se hizo a través de la modificación de un parámetro propio del control de simulación que detiene el proceso de simulación un tiempo determinado.

Todos los ritmos de simulación, exceptuando el de simulación rápida, poseen su tiempo de suspensión del proceso. El simulador procesa los estados muy rápido, por lo que constantemente emite eventos y estados al cliente. Este tiempo le sirve al sistema para vaciar la cola de eventos de dominio emitidos, además que al retrasar la simulación hace que esta se ejecute en escalas de tiempo donde el usuario pueda ver

y comprender lo que sucede en el sistema.

La opción de correr la simulación rápidamente, no considera ni el tiempo máximo de reloj²², ni los tiempos de detención del proceso, lo que permite realizar las simulaciones lo más rápido que deje el computador.

4.2.10.2. Despliegue del *framework*

La creación de los archivos de distribución del *framework* se realizó haciendo uso de las utilidades del paquete de Python *build*.

La herramienta toma un archivo con los metadatos y opciones para crear los archivos de distribución. Los archivos contruidos fueron subidos a través de un cliente especializado llamado *twine* a la plataforma PyPI, con el nombre de *general-simulation-framework*²³.

4.2.10.3. Despliegue del simulador

Para desplegar el simulador se tomaron dos servidores: uno para el *kernel* y otro para servir la aplicación cliente.

Como se usó Heroku como infraestructura para desplegar el *kernel*, se tuvo que usar *gunicorn* como servidor web, atado a la estrategia de despliegue anterior, debido a que era la manera más documentada y referenciada para lograr el despliegue.

Al haber un retraso considerable de respuesta al interactuar con el *kernel*, se establecieron los parámetros del servidor en un (1) *worker*²⁴ y doce (12) hilos.

Pese a solo existir un *worker*, la mayoría de las peticiones son instantáneas por lo que el retraso entre peticiones simultáneas es nulo y no es necesario incluir más. Los hilos también representan una limitante de usuarios posibles para usar el servidor, ya que cada usuario podría llegar a ocupar hasta dos (2) hilos adicionales al hilo principal en procesos de simulación y manejo de eventos.

²² El simulador cada vez que ejecuta un ciclo de simulación, toma el mínimo entre el tiempo del próximo evento y un segundo. Al no considerarlo, salta de evento a evento.

²³ Disponible en: <https://pypi.org/project/general-simulation-framework/>

²⁴ Máxima cantidad de clientes simultáneos conectados.

El cliente fue desplegado en otro servidor, sirviendo los archivos estáticos generados por Vue a través de un servidor web desarrollado con *express*.

4.2.10.4. Generación de página de referencia

La generación de documentación se realizó con la ayuda de una extensión de Sphinx para la autogeneración de documentación llamada *apidoc*.

La herramienta generó un archivo *.rst* para todos los módulos del *framework* a partir de los comentarios escritos en el código.

A pesar de que la herramienta permitió facilitar bastante el proceso de documentación, algunos recursos generados fueron modificados manualmente para corregir algunos malentendidos y prácticas utilizadas por la herramienta, como la creación del árbol de páginas de la barra lateral de navegación y la página de inicio.

Finalmente, se utilizó Sphinx, para convertir los archivos a una página estática, que posteriormente se subió a GitHub Pages donde puede ser consultada.

Capítulo 5

Resultados obtenidos

En el presente capítulo se detallan los resultados obtenidos en cada uno de los procesos realizados durante el desarrollo de la investigación, así como se incluye un análisis para explicar la obtención de los mismos.

5.1. Análisis de resultados

Gracias al enfoque de la investigación, se obtiene como resultado una alternativa a las herramientas de apoyo para la implementación de software de simulación. A continuación, se detallan los resultados obtenidos para cada objetivo:

1. Determinar la arquitectura, estándares, normas, patrones y tácticas a ser implementados por el ambiente de desarrollo.

El *framework* implementado es de caja gris, donde existen varios componentes predefinidos para componer una aplicación y donde ciertos mecanismos son implementados por medio de la extensión.

La organización quedó dividida en seis (6) módulos, conectados entre sí por la composición de clases, cuyas instancias son proveídas por constructor a través de inyección de dependencias. No se consideró ningún patrón creacional para las instancias, siendo realizadas directamente por el usuario o por el mismo entorno de trabajo.

El paradigma de simulación del marco de trabajo está orientado a los sistemas de simulación de eventos discretos, utilizando el método de programación de eventos y avance de tiempo, a través de la lista de eventos futuros.

2. Desarrollar un módulo de simulación.

El paquete de simulación desarrollado está compuesto por una clase abstracta para definir el comportamiento común para todos los posibles tipos de simuladores a implementar con el *framework*, y una clase concreta para definir un simulador típico para las simulaciones de eventos discretos a partir de la extensión del comportamiento definido.

El simulador solicita las salidas y el cómputo del siguiente estado al sistema dinámico, siendo automatizado su control por medio del módulo de control, y guardando las salidas delegándolas al módulo de reportes.

3. Desarrollar un módulo de sistema dinámico.

El sistema dinámico define el comportamiento de un sistema para la ejecución de simulaciones, poseyendo colecciones de modelos y rutas entre modelos, además de establecer como métodos obligatorios de un sistema dinámico la obtención de una salida y el cómputo de un nuevo estado.

Bajo esa definición, se implementó una clase concreta enfocada a eventos discretos, donde se codificó un algoritmo de enrutamiento genérico.

A pesar de elegirse un comportamiento por defecto para la selección de rutas con probabilidad segura, este comportamiento al no ser excluyente, clona salidas; característica que obligó a sobrescribir el método mediante extensión durante la elaboración del simulador de redes de cola, donde este comportamiento es erróneo, ya que trabaja con entidades únicas que no se deberían clonar entre estaciones. No obstante, dada la gran cantidad de sistemas donde la difusión de salidas aplica, no se creyó conveniente cambiar el comportamiento original.

Debido a las características del enrutamiento, es posible que un método para la ejecución del mismo no sea del todo práctico, pudiendo ser mejor la inclusión

de estrategias por inyección de dependencias para definir los distintos tipos de enrutamiento.

El módulo de sistema dinámico también incluye las funcionalidades para manejar la lista de eventos futuros y las entidades programadas en la misma, siendo utilizadas por los modelos del sistema dinámico y actualizadas durante una transición de estado.

4. Desarrollar un módulo de definición de modelos de simulación.

El paquete de modelos de simulación posee la abstracción para los modelos y las definiciones de las rutas y los modelos de eventos discretos.

La interfaz de modelo permite interactuar con la salida del modelo y la transición de estado, siendo implementadas por el modelo concreto para delegar las tareas a las distintas funciones de transición de estado y salida estudiadas.

Por su parte, las rutas están descritas por una clase concreta y son definidas por el usuario o por un modelo mediante la inclusión de un modelo de entrada, uno de salida y un peso de la conexión.

El sistema dinámico almacena los modelos y sus conexiones en colecciones específicas, haciendo fácil la manipulación de los nodos y conexiones de la red.

5. Desarrollar un módulo de control de simulaciones.

El módulo de control de simulaciones está compuesto por abstracciones para el control y la ejecución de estrategias de simulación, y clases concretas que implementan funcionalidades específicas.

Todos los componentes de control pertenecientes al *framework*, poseen las funcionalidades para iniciar, pausar, detener, esperar y ejecutar una simulación. Mientras que la clase de control para simulación de eventos discretos implementa las funcionalidades y agrega otras como el control del tiempo.

Se creó una estrategia para la ejecución de simulaciones en un hilo a parte al control de simulación que corre en el hilo principal, lo que permite controlar y

ejecutar la simulación al mismo tiempo, evitando la necesidad de esperar a terminar el ciclo de simulación para llamar a algún método de control.

6. Desarrollar un módulo de reporte de simulaciones.

El paquete de reportes está compuesto por una clase abstracta para definir el comportamiento común para la generación de reportes, y una clase concreta que implementa un tipo de reporte específico.

La clase abstracta para reportes expone métodos para insertar entradas y generar reportes, mientras que el último método delega sus funcionalidades a un método abstracto.

Los reportes por defecto convierten la lista de salidas insertadas y guardadas como propiedades durante la simulación en una tabla de reporte haciendo uso de PrettyTable.

Se ha notado un retraso considerable en los tiempos de ejecución durante la generación de reportes con gran cantidad de datos. Esto puede deberse¹ a factores como la implementación poco eficiente de la conversión de las estructuras propias del reporte a tabla, o debilidades en la librería para labores de muestreo.

7. Desarrollar los mecanismos de comunicación y coordinación entre los módulos.

La mayoría de los módulos se comunican entre sí, mediante la invocación de los métodos definidos en sus interfaces de mayor nivel. De esta manera, el control maneja el simulador, el simulador el sistema dinámico y los reportes, y el sistema dinámico los modelos.

Para facilitar la coordinación entre los módulos, se creó un componente de experimentos. El paquete de experimentos está compuesto por un experimento abstracto que agrupa los módulos, y un experimento concreto con la definición de los módulos necesarios para la simulación de sistemas de eventos discretos.

¹ La cantidad de datos es el factor principal, pero si se optimizara alguno de los otros elementos es posible que exista una mejoría notable en la obtención de resultados.

Algunas funcionalidades del ambiente de desarrollo emiten eventos de dominio por medio de un bus de eventos para indicar la ocurrencia de un suceso durante o después de la simulación. Sin embargo, ningún módulo actual del *framework* escucha alguno de estos eventos, siendo realmente utilizado este mecanismo de comunicación por módulos externos al marco de trabajo, como los creados para elaborar el simulador de red de colas.

Gracias al protocolo definido por las expresiones, se permite el manejo de distintos tipos de dato con los mismos módulos. Se crearon varias expresiones concretas para instanciar distribuciones y constantes, no obstante, ha sido encontrado un fallo de seguridad en el uso de expresiones creadas por el usuario, ya que al usar la función *eval*, se permite al usuario importar y ejecutar casi cualquier librería, e incluso comandos en el sistema a través de algunas de esas librerías, por lo que no se debería usar la expresión en ambientes en producción.

8. Desarrollar una aplicación de simulación que implemente las funcionalidades ofrecidas por el ambiente de trabajo.

La aplicación desarrollada para validar el potencial uso del ambiente de desarrollo está compuesta por un *kernel* y una interfaz gráfica.

El *kernel* está basado en la aplicación de consola realizada como prototipo 8, siendo la implementación directa de las clases y prácticas impuestas por el *framework*, otorgando los puntos de entrada para su comunicación desde cualquier otro software.

Por su parte, la interfaz gráfica consulta con el servidor todas sus acciones a ejecutar, siendo un medio para dar las entradas y mostrar las salidas del *kernel*, por lo que no ejecuta ninguna operación que no sea estética o de comunicación.

Al tomar el cliente exclusivamente como fachada y medio de comunicación, existe un ligero retraso entre las acciones hechas por el usuario y la ejecución de la acción en el cliente, producto de la espera del servidor. Esta característica podría ser ignorada para usos de la herramienta en equipos locales donde el retraso es

inexistente, pero podría implicar incidencias en usabilidad para uso sobre un *kernel* remoto y malas conexiones a internet.

El manejo de excepciones es mejorable, ya que es difícil controlar completamente las entradas del usuario debido a la existencia de las expresiones. Las expresiones pueden tomar casi cualquier valor del usuario, por lo que pueden ocurrir errores debido a esta libertad que el simulador no es capaz de identificar hasta la etapa de simulación.

Durante la evaluación y validación de la herramienta, se elaboraron varias redes y se compararon los resultados con otras respuestas y soluciones, concluyendo que el simulador desarrollado permite la creación, simulación y reporte de resultados de algunos modelos de redes de colas.

A pesar de cumplir de manera correcta con los objetivos establecidos, es posible seguir mejorando el simulador con las inclusión de nuevas características, funcionalidades y componentes.

9. Documentar las distintas funcionalidades del ambiente de desarrollo en una referencia para programadores.

Los comentarios sobre métodos, clases, módulos y paquetes, fueron procesados por Sphinx para generar una página de referencia para programadores, que fue subida posteriormente a GitHub Pages.

La documentación está dividida en paquetes y módulos. Los paquetes representan la estructura de directorios que llevan a determinado módulo, y los módulos contiene la descripción de la clase propia del *framework* y un ejemplo del uso de la clase perteneciente a ese módulo, lo que permite conocer de manera general cómo implementar los componentes del marco de trabajo.

Anexo a la documentación de código y funcionalidades del ambiente de desarrollo, existe un apartado de ejemplos específicos en la página de referencia. En la sección de ejemplos se encuentran codificados variantes de los prototipos 1, 2 y 4, lo que puede ayudar a los desarrolladores conocer de manera práctica las utilidades del *framework*.

Capítulo 6

Conclusiones y recomendaciones

En este capítulo se presentan las conclusiones obtenidas tras la realización de la investigación, describiendo adicionalmente las recomendaciones para trabajos futuros en el área o basados en esta investigación.

6.1. Conclusiones

Uno de los principales objetivos de la Ingeniería de Software es guiar el desarrollo de proyectos mediante la aplicación de metodologías, estándares, normas o marcos de trabajo. Las prácticas tomadas buscan crear software de calidad que satisfagan las necesidades de los usuarios en un tiempo establecido.

La simulación ha tomado popularidad a través de los años, surgiendo nuevas herramientas adaptadas a una infinidad de contextos distintos. Sin embargo, existen componentes en común entre todos los proyectos orientados a la simulación que se pueden empaquetar en un marco de trabajo.

Gracias a la investigación realizada se ha podido desarrollar un *framework* para la implementación de herramientas de simulación, que da la posibilidad de utilizar y extender los mismos componentes para generar una gran variedad de proyectos de simulación; permitiendo prescindir de algunos conocimientos del dominio, ahorrar tiempo de programación y mantener una estructura estándar en las soluciones desarrolladas.

El ambiente de trabajo desarrollado está enfocado a la simulación de sistemas de eventos discretos, lo que ayuda implementar varias de las herramientas dentro del dominio del problema tratado, no obstante, también implica que debe ser extendido o modificado para adaptarse a otros paradigmas de simulación existentes que no se consideraron y que pueden ser útiles para muchas otras soluciones.

A lo largo del proyecto se implementaron y validaron hasta diez (10) prototipos funcionales que sirven tanto para mostrar el uso del *framework*, como para describir la evolución de la investigación, siendo el último prototipo un simulador de redes de cola sencillo, pero que es un ápice de lo que podría llegar a implementarse con una mayor cantidad de esfuerzo, tiempo y recursos.

No cabe duda de que el campo de la simulación y la implementación de herramientas para la realización de las mismas, todavía tiene mucho terreno que explorar y ramas en las que puede contribuir. Es de vital importancia continuar con la investigación en el campo e incluir disciplinas como la Ingeniería de Software, con el fin de seguir innovando en la creación de nuevas herramientas guiadas por las mejores prácticas que permitan su evolución y uso extendido por la comunidad.

6.2. Recomendaciones

Tanto el marco de trabajo como el simulador desarrollado poseen un amplio margen de mejora, pudiéndose empezar con las siguientes recomendaciones:

6.2.1. Recomendaciones sobre el *framework*

- El algoritmo de enrutamiento del sistema dinámico podría ser implementado a través de una estrategia que reciba la entrada al sistema, el tiempo, los modelos y las rutas. Esto haría posible inyectar estrategias concretas que puedan adquirir otros comportamientos y rendimientos, siendo bastante útil para optimizar los algoritmos del *framework*.
- Los reportes generados por el módulo de reportes deberían extenderse para generar estadísticas, gráficos, documentos, etc., lo que sin duda aportaría más

valor a las salidas por defecto generadas por el marco de trabajo.

- Estudiar algoritmos de enrutamiento y simulación que puedan hacer uso de la programación paralela.
 - Una alternativa sería la implementación de un algoritmo para redes disconexas como la trabajada en la figura 4.34.d, donde al no haber entradas y salidas dependientes entre algunos modelos de la red, se podría tratar una misma simulación como varias simulaciones, una en cada nodo de procesamiento, lo que hipotéticamente reduciría de manera considerable los tiempos de respuesta.
- Implementar otros paradigmas de simulación como la simulación por procesos o por actividades para cubrir un mayor rango de soluciones posibles a desarrollar.
- Extender las funcionalidades para la simulación de sistemas híbridos. Con el fin de que sea una extensión y no una modificación del código existente, se recomienda la aproximación de los sistemas continuos a sistemas de eventos discretos a través de la integración numérica y la escucha persistente de eventos desde los modelos de sistemas de eventos continuos.
- Resolver a nivel de *framework* las incidencias de seguridad provocadas por el uso de la función *eval* para las expresiones introducidas por el usuario.
- Documentar las pruebas que faltan por cubrir.
- Implementar más distribuciones matemáticas.

6.2.2. Recomendaciones sobre el simulador de red de colas

- Pulir el manejo de excepciones del simulador creado. Al permitir infinitas entradas del usuario, hay una gran cantidad de errores cuyo manejo y notificación puede ser mejorado.
- Integrar nuevos componentes al simulador.

- Herramienta para crear definiciones como funciones, constantes, variables, expresiones, etc., propias del usuario.
 - Enrutamiento de salidas por camino más cercano.
 - Herramienta para manejar los datos que simula el sistema y su orden.
 - Completar las propiedades de los componentes básicos del sistema.
 - Agregar nuevos componentes como el separador y el combinador de entidades.
 - Agregar filtros por tipo de entidad.
-
- Mejorar las estadísticas provistas por el simulador e incluir estadísticas para las rutas y entidades.
 - Permitir el uso colaborativo del simulador. Para ello, se sugiere el uso de sesiones compartidas (*rooms*) y la integración de más funciones propias de la interfaz gráfica en el *kernel*.
 - Incorporar un sistema de manejador de versiones y mecanismos para hacer y deshacer acciones ejecutadas.
 - Animar el movimiento de las entidades a través de las rutas y su paso efímero por los *buffers* de salida de los servidores.

Referencias

- [1] N. Kirovska y S. Koceski, “Usage of kanban methodology at software development teams”, *Journal of applied economics and business*, vol. 3, no. 3, pp. 25–34, 2015.
- [2] J. Banks, J. S. Carson, B. L. Nelson, y D. M. Nicol, *Discrete-event system simulation*, 4ta ed. Pearson Prentice-Hall, 2005.
- [3] “Frequently asked questions”, *Simio*, 2021. [En línea]. Disponible en: <https://www.simio.com/academics/simio-academic-software-frequently-asked-questions.php> [Accedido: 24-Sep-2021].
- [4] C. V. Inojosa, “El presupuesto 2021 deja a las universidades en la quiebra”, *Crónica Uno*, 2020. [En línea]. Disponible en: <https://cronica.uno/el-presupuesto-2021-deja-a-las-universidades-en-la-quiebra/> [Accedido: 24-Sep-2021].
- [5] “Los riesgos de la piratería de software”, *Nubelato*, 2020. [En línea]. Disponible en: <https://nubelato.com/blog/los-riesgos-la-pirateria-software/> [Accedido: 24-Sep-2021].
- [6] J. Nutaro, *Building software for simulation*. Hoboken, N.J.: John Wiley & Sons, Inc., 2011.
- [7] “Que son los frameworks”, *miGabeta*, 2017. [En línea]. Disponible en: <https://migabeta.wordpress.com/2017/01/23/que-son-los-frameworks/> [Accedido: 24-Sep-2021].
- [8] J. Banks, *Handbook of simulation: principles, methodology, advances, applications, and practice*. Hoboken, N.J.: John Wiley & Sons, Inc., 1998.
- [9] A. Law y W. Kelton, *Simulation Modeling and Analysis*. New York: McGraw-Hill,

2000.

- [10] R. C. Abad, *Introducción a la simulación y a la teoría de colas*. Netbiblo, 2002.
- [11] H. Santiago, *Teoría de Colas o de Líneas de Espera*, 2021. [En línea]. Disponible en: <https://i.imgur.com/nBNv5Nj.jpg> [Accedido: 24-Sep-2021].
- [12] R. T. Michael T. Goodrich y M. H. Goldwasser, *Data structures and algorithms in Python*. John Wiley & Sons, Inc., 2013.
- [13] T. H. Cormen, et al., *Introduction to algorithms*, 3ra ed. MIT press, 2009.
- [14] D. Riehle, *Framework design: A role modeling approach*. Diss. ETH Zurich, 2000.
- [15] D. Parsons, A. Rashid, A. Speck, y A. Telea, *A "framework" for object oriented frameworks design*. IEEE, 1999.
- [16] M. Fowler, "Inversion of control", *martinFowler.com*, 2005. [En línea]. Disponible en: <https://martinfowler.com/bliki/InversionOfControl.html> [Accedido: 24-Sep-2021].
- [17] M. Fowler, "Inversion of control containers and the dependency injection pattern", *martinFowler.com*, 2004. [En línea]. Disponible en: <https://martinfowler.com/articles/injection.html> [Accedido: 24-Sep-2021].
- [18] K. Rosen, "The history of simulation", in *The comprehensive textbook of healthcare simulation*. Springer, 2013, pp. 5–49.
- [19] M. Aebersold, "The history of simulation and its impact on the future", *AACN advanced critical care*, vol. 27, no. 1, pp. 56–61, 2016.
- [20] Rahmi M. Koç Academy of Interventional Medicine Education and Simulation, 2017. [En línea]. Disponible en: <http://aimes.org//Uploads/Files/HistoryofSimulation/7.jpg> [Accedido 24-Sep-2021].
- [21] S. Alaggia, "Relevamiento de lenguajes de simulación", 2005. [En línea]. Disponible en: https://www.fing.edu.uy/inco/cursos/simulacion/archivos/Relevamiento_de_Lenguajes_de_Simulacion.pdf [Accedido: 24-Sep-2021].
- [22] E. Briceño, "Estudio comparativo del paquete de simulación orientado a eventos discretos symPy. desarrollo de un manual de usuario con ejemplos resueltos",

trabajo de fin de grado, Universidad de Los Andes, 2007.

- [23] “Overview”, *SimPy*, 2021. [En línea]. Disponible en: <https://simpy.readthedocs.io/en/latest/> [Accedido: 24-Sep-2021].
- [24] J. Gosling, D. C. Holmes, y K. Arnold, *The Java programming language*. Addison-Wesley, 2005.
- [25] “The python tutorial”, *Python*, 2021. [En línea]. Disponible en: <https://docs.python.org/3/tutorial/index.html> [Accedido: 24-Sep-2021].
- [26] S. A. Abdulkareem y A. J. Abboud, “Evaluating python, c++, javascript and java programming languages based on software complexity calculator (halstead metrics)”, *IOP Conference Series: Materials Science and Engineering*, vol. 1076, no. 1, 2021.
- [27] “Javascript”, *MDN Web Docs*, 2021. [En línea]. Disponible en: <https://developer.mozilla.org/es/docs/Web/JavaScript> [Accedido: 24-Sep-2021].
- [28] “Typescript for the new programmer”, *TypeScript*, 2021. [En línea]. Disponible en: <https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html> [Accedido: 24-Sep-2021].
- [29] B. Stroustrup, *The C++ Programming Language*, 4ta ed. Pearson Education, Inc., 2013.
- [30] “What is vue.js?” *Vue.js*, 2021. [En línea]. Disponible en: <https://vuejs.org/v2/guide/> [Accedido: 24-Sep-2021].
- [31] R. M. Kim Hamilton, *Learning UML 2.0*. O’Reilly, 2006.
- [32] “Socket.io”, *Socket.IO*, 2021. [En línea]. Disponible en: <https://socket.io/> [Accedido: 24-Sep-2021].
- [33] “Pypi”, *Python Package Index*, 2021. [En línea]. Disponible en: <https://pypi.org/> [Accedido: 24-Sep-2021].
- [34] “Meet DeepSource. Helping developers ship good code.” *DeepSource*, 2021. [En línea]. Disponible en: <https://deepsource.io/about/> [Accedido: 24-Sep-2021].

- [35] “Spend less time fixing bugs. And more time shipping new features.” *Better Code Hub*, 2021. [En línea]. Disponible en: <https://bettercodehub.com/about> [Accedido: 24-Sep-2021].
- [36] “Welcome”, *Sphinx*, 2021. [En línea]. Disponible en: <https://www.sphinx-doc.org/en/master/> [Accedido: 24-Sep-2021].
- [37] G. Aguilar, “Ciclo de vida de desarrollo de sistemas”, en *Administración de Proyectos*, Universidad Veracruzana, 1990. [En línea]. Disponible en: <http://www.geocities.ws/gildardoaguilar/adp4.pdf> [Accedido: 24-Sep-2021].
- [38] V. Domínguez, “Incorporación de software a las organizaciones”, trabajo de investigación, Universidad Nacional de Cuyo, 2013.
- [39] R. C. María Gómez y P. González, *Fundamentos de Ingeniería de Software*. Universidad Autónoma Metropolitana, 2019.
- [40] V. Mahnič, “Applying kanban principles to software development”, *Information Technology for Practice*, vol. 2013, p. 89, 2013.
- [41] Pontificia Universidad Católica de Valparaíso, “Simio”, *Simulemos*, 2019. [En línea]. Disponible en: <https://simulemos.cl/books/simio> [Accedido: 24-Sep-2021].
- [42] D. Goldberg, “What every computer scientist should know about floating-point arithmetic”, *ACM computing surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.
- [43] R. Andrade, “Juego de la vida”, *GitHub*, 2019. [En línea]. Disponible en: <https://rolandoandrade.github.io/automata-celular/> [Accedido: 24-Sep-2021].
- [44] M. Romero, “El juego de la vida”, *Universidad Carlos III de Madrid*, 2021.
- [45] J. Y. Joshua, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, y J. E. Smith, “The future of simulation: A field of dreams”, *Computer*, vol. 39, no. 11, pp. 22–29, 2006.
- [46] G. I. Doukidis y M. C. Angelides, “A framework for integrating artificial intelligence and simulation”, *Artificial Intelligence Review*, vol. 8, no. 1, pp. 55–85, 1994.
- [47] H. J. Briegel y G. De las Cuevas, “Projective simulation for artificial intelligence”, *Scientific reports*, vol. 2, no. 1, pp. 1–16, 2012.

- [48] P. Gawłowicz y A. Zubow, “Ns-3 meets openai gym: The playground for machine learning in networking research”, in *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2019, pp. 113–120.
- [49] R. Andrade, “General simulation framework examples”, *Colab*, 2021. [En línea]. Disponible en: <https://colab.research.google.com/drive/1tJOnpvKK9jZRpqFwAezlwRv1eRthU0tP?usp=sharing> [Accedido: 24-Sep-2021].

Anexo A

Desarrollo

A.1. Evolución del algoritmo de enrutamiento

A.1.1. Prototipo 2. Autómata celular lineal

En este prototipo se creó la base para todos los algoritmos de enrutamiento desarrollados, en él se aprecia la ejecución de los modelos que reciben eventos externos y los modelos que se ven afectados por esa transición.

```

1 def __get_values_to_inject(self, input_models: List[str]) -> Dict[str, Any]:
2     if len(input_models) > 1:
3         # el modelo tiene entradas
4         values = dict()
5         for inp in input_models:
6             # busca la salida del modelo de entrada y la guarda
7             values[inp] = self.__outputs[inp]
8         return values
9     # no inyecta ninguna entrada al no tener modelos de entrada
10    return None
11
12 def state_transition(self, input_models_values: Dict[str, Any] = None):
13     input_models_values = input_models_values or {}
14     for model in input_models_values:

```

```

15     # calcula la transición de los modelos que reciben entrada
16     self.__models[model].state_transition(input__models__values[model])
17     for model in self.__inputs:
18         # calcula la transición de los modelos que son afectados por otro modelo
19         if model not in input__models__values:
20             # si no ha ejecutado una transición externa
21             vs = self.__get__values__to__inject(self.__inputs[model])
22             self.__models[model].state_transition(vs)

```

Código A.1: Algoritmo de enrutamiento inicial. De autoría propia.

El algoritmo se adapta a cualquier tipo de simulación, no obstante, no tiene implementada ninguna funcionalidad para manipular el tiempo de los eventos, siendo imprescindible para la realización de simulaciones de eventos discretos.

A.1.2. Prototipo 3. Sistema de fábrica de discos

Este prototipo se caracterizó por la inclusión del tiempo para la emisión de eventos autónomos.

```

1 def state_transition(self, input__models__values: DynamicSystemInput = None, event_time
   ↪ : float = 0):
2     # resta el tiempo
3     self.__scheduler.update_time(event_time)
4     input__models = set()
5     # ejecuta las transiciones externas
6     if input__models__values is not None:
7         for model in input__models__values:
8             self.__models[model].state_transition(input__models__values[model], event_time)
9             input__models.add(self.__models[model])
10
11     # hay transiciones autónomas para ejecutar
12     if self.__get__time__of__next__events() == 0:
13         # extrae todos los modelos autónomos

```

```

14 all_autonomous_models = self._scheduler.pop_next_models()
15 # toma los modelos autónomos faltantes
16 autonomous_models = all_autonomous_models.difference(input_models)
17 # define las entradas de los modelos
18 affected_models_inputs: Dict[str, ModelInput] = {}
19 affected_models = set()
20 # para cada modelo autónomo
21 for model in all_autonomous_models:
22     # obtiene los modelos de salida del modelo
23     outputs = self._inputs[model.get_id()].difference(input_models)
24     # para cada modelo en las salidas
25     for out in outputs:
26         # se agrega al conjunto de modelos afectados
27         affected_models.add(out)
28         # si ya tiene una entrada previa
29         if out.get_id() in affected_models_inputs:
30             # añade otra entrada
31             affected_models_inputs[out.get_id()][model.get_id()] = self._outputs[model
↪ .get_id()]
32         else:
33             # crea una nueva entrada
34             affected_models_inputs[out.get_id()] = {
35                 model.get_id(): self._outputs[model.get_id()]
36             }
37 # obtiene los modelos autónomos puros
38 autonomous_models = autonomous_models.difference(affected_models)
39 # ejecuta los modelos autónomos
40 for model in autonomous_models:
41     model.state_transition(None, event_time)
42 # ejecuta los modelos afectados
43 for model in affected_models:
44     model.state_transition(affected_models_inputs[model.get_id()], event_time)
45 # programa los eventos autónomos de nuevo

```

```

46     for model in all_autonomous_models:
47         self.schedule(model, model.get_time())

```

Código A.2: Algoritmo de enrutamiento original. De autoría propia.

Con la definición de los conjuntos de modelos afectados, autónomos y externos, y sus respectivas transiciones, este algoritmo servía para el enrutamiento eficaz de las entradas en el sistema dinámico. Sin embargo, no consideró el peso de las rutas entre los modelos, por lo que estuvo obligado a cambiar posteriormente.

A.1.3. Prototipo 8. Simulador de red de colas (consola)

Este prototipo incluyó las rutas, y con ello la finalización del algoritmo.

```

1  def _get_effective_paths(self, emitter_model: DiscreteEventModel) -> Set[Path]:
2      # el modelo emisor es el origen de alguna ruta.
3      if emitter_model in self._paths:
4          # busca las rutas con probabilidad 1 o True
5          ones = [path for path in self._paths[emitter_model] if path.get_weight() == 1]
6          # encontró ese tipo de rutas
7          if len(ones) > 0:
8              return set(ones)
9          # como no encontró, debe seleccionar una ruta por probabilidad.
10         weights = []
11         effective_path = []
12         # crea una lista de pesos y otra con las rutas
13         for path in self._paths[emitter_model]:
14             weights.append(path.get_weight())
15             effective_path.append(path)
16         # escoge una ruta al azar
17         choice = np.random.choice(len(weights), p=weights)
18         return {effective_path[choice]}
19     return set()

```

Código A.3: Selección de rutas. De autoría propia.

Con la inclusión de este algoritmo, el enturamiento está completo, sin embargo, podría mejorarse mediante la creación de una estrategia de enrutamiento, en vez de estar definida en la clase, y tener que ser modificada por extensión directa de la misma.

A.2. Evaluación y validación de prototipo

A.2.1. Prototipo 1. Autómata celular lineal

A.2.1.1. Fase de uso del framework

Existe inversión de control, ya que:

- La clase *AtomicModel* llama a métodos implementados por los usuarios a través de la extensión.
- Los métodos públicos de entrada y salida definen un comportamiento predeterminado.

Por otro lado, es imposible implementar un simulador de eventos discretos sin modificar el *framework*, ya que, solo contempla creación de modelos de sistemas de eventos de tiempo discreto, ignorando eventos que llegan en tiempos diferentes y los procesos involucrados en la simulación.

A.2.1.2. Fase de validación de resultados

El resultado esperado fue establecido por el comportamiento teórico del sistema tras tres iteraciones.

	Célula 1	Célula 2	Célula 3
Salida inicial	True	False	True
Salida esperada	False	True	True
Salida obtenida	False	True	True

Tabla A.1: Validación de salidas a la tercera iteración. De autoría propia.

Como las salidas son equivalentes, se concluye que el software está obteniendo resultados correctos.

A.2.2. Prototipo 2. Autómata celular en malla

A.2.2.1. Fase de uso del framework

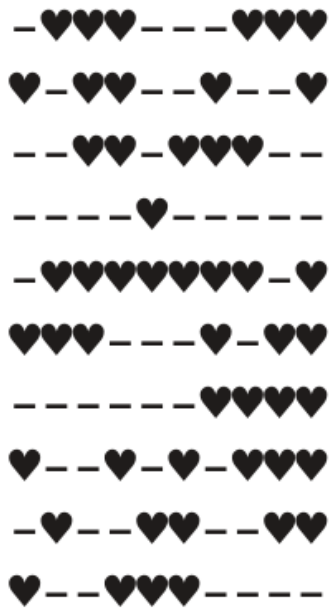
Existe inversión de control, ya que:

- Características previas.
- El sistema dinámico está proveyendo un comportamiento predefinido para la ejecución de transiciones y salidas, llegando incluso a establecer reglas para su uso.

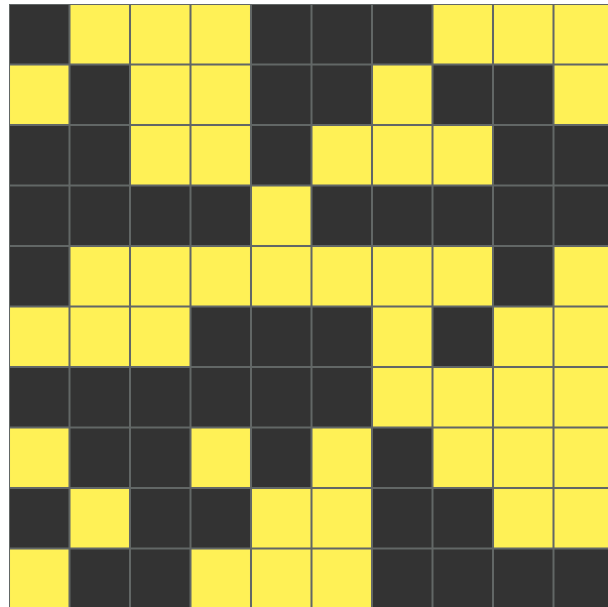
No obstante, es imposible implementar un simulador de eventos discretos sin modificación del *framework*, ya que solo contempla creación de sistemas de eventos de tiempo discreto, ignorando eventos que llegan en tiempos diferentes y los procesos involucrados en la simulación como el simulador, control o reportes.

A.2.2.2. Fase de validación de resultados

Para la validación se usó una herramienta previamente desarrollada de autoría propia [43] a fin de hallar los resultados esperados para diez generaciones.

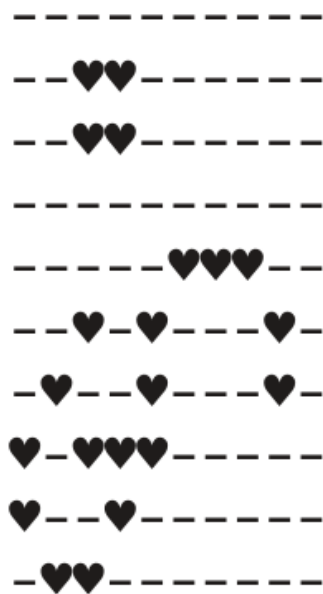


(a) Autómata implementado.

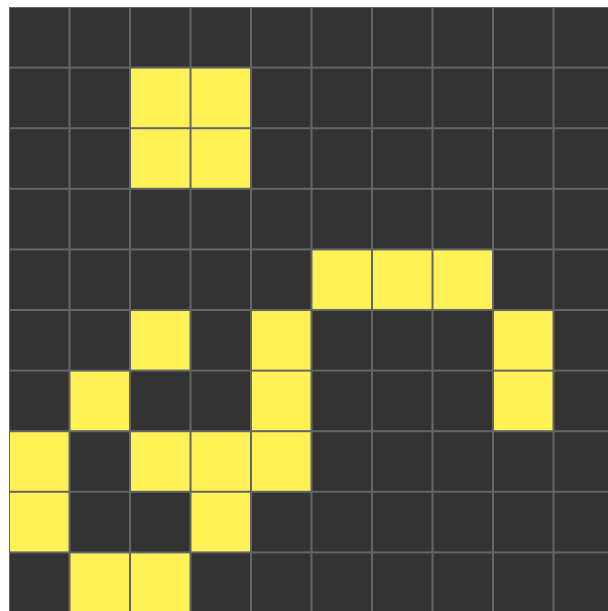


(b) Autómata de referencia (entrada manual)

Figura A.1: Salida inicial. De autoría propia.



(a) Salida obtenida



(b) Salida esperada

Figura A.2: Salida final. De autoría propia.

Como las salidas son equivalentes, se concluye que el software está obteniendo resultados correctos.

A.2.3. Prototipo 3. Sistema de fábrica de discos

A.2.3.1. Fase de uso del framework

No se ha modificado ninguna característica que influyera en las funcionalidades de inversión de control previas, por lo que existe inversión de control.

Pese a poder modelar sistemas de eventos discretos, el *framework* todavía es incapaz de simularlos sin requerir la modificación de la base de código existente.

A.2.3.2. Fase de validación de resultados

Para la validación se usaron las trazas de la simulación realizada por Nutaro [6] tras la inserción de un elemento en el tiempo cero ($t = 0$) y otras dos en el tiempo uno ($t = 1$).

Los vectores de la traza tienen la forma $(estado, tiempo)$, siendo el estado el número de elementos que requieren servicio, mientras que el tiempo, es el tiempo restante de servicio para determinado elemento.

```

1 State, t = 0, press = (0,1), drill = (0,2)
2 State, t = 0, press = (1,1), drill = (0,2)
3 State, t = 1, press = (0,1), drill = (1,2)
4 State, t = 2, press = (2,1), drill = (1,2)
5 Output, t = 3, y = 1
6 State, t = 3, press = (1,1), drill = (1,2)
7 State, t = 4, press = (0,1), drill = (2,1)
8 Output, t = 5, y = 1
9 State, t = 5, press = (0,1), drill = (1,2)
10 Output, t = 7, y = 1
11 State, t = 7, press = (0,1), drill = (0,2)

```

Código A.4: Trazas de simulación. Adaptación de Nutaro [6].

Como se puede observar, el sistema tiene una salida en $t = 3$, $t = 5$ y $t = 7$, siendo los tiempos esperados para la validación.

t	Prensa	Taladro
0	-	-
1	1	-
2	1	-
3	1	1
5	-	1
7	-	1

Tabla A.2: Salidas obtenidas. De autoría propia.

Las salidas del taladro representan las salidas del sistema. Como los tiempos de salida son equivalentes a los tiempos esperados, el prototipo está obteniendo resultados correctos.

A.2.4. Prototipo 4. Simulador de fábrica de discos

A.2.4.1. Fase de uso del framework

Existe inversión de control por los siguientes motivos:

- Permanencia de las características previas.
- El control de simulación asume los procesos de simulación.
- El software controla la generación de reportes.

Pese a estar demostrado que el software puede servir para implementar algunas herramientas de simulación, no existe evidencia suficiente para afirmar que sirva para implementar todos los tipos de simulador de eventos discretos sin modificar la base de código existente.

A.2.4.2. Fase de validación de resultados

Para la validación se usaron los resultados del prototipo anterior.

t	Generador	Prensa	Taladro
0	1	-	-
1	2	1	-
2	-	1	-
3	-	1	1
5	-	-	1
7	-	-	1

Tabla A.3: Salidas obtenidas. De autoría propia.

Las salidas del taladro representan las salidas del sistema. Como los tiempos de salida son equivalentes a los tiempos esperados, el prototipo está obteniendo resultados correctos.

A.2.5. Prototipo 6. Emisor de entidades y fuente

A.2.5.1. Fase de uso del framework

Con la realización de la fuente, está demostrado que se puede implementar el elemento sin necesidad de alterar la base de código del *framework*. Sin embargo, no hay evidencia suficiente para afirmar que sea posible implementar el resto de elementos con las mismas condiciones.

A.2.5.2. Fase de validación de resultados

Para la validación de resultados se corrió la simulación tanto en Simio como en el simulador desarrollado, obteniendo los siguientes resultados:

Vectores de entrada				Salidas servidor	
Tiempo	Interarrival time	Entities Per Arrival	Time Offset	Total esperado	Total obtenido
5	1	1	0	6	6
5	1	2	0	12	12
10	2	1	3	4	4
100	5	7	10	133	133
5	10	10	10	0	0

Tabla A.4: Resultados de los experimentos. De autoría propia

Como los valores esperados y obtenidos son iguales, el simulador está reportando

resultados de manera correcta.

A.2.6. Prototipo 7. Servidor y distribuciones matemáticas

A.2.6.1. Fase de uso del framework

Con el simulador se ha demostrado la capacidad de implementar los componentes de tipo fuente, servidor y ruta. Sin embargo, al fallar en simulaciones de números con parte decimal y no permitir las rutas ponderadas, es imposible desarrollar herramientas de simulación de eventos discretos sin alterar la base de código del *framework*.

A.2.6.2. Fase de validación de resultados

Para la validación de resultados se corrió la simulación tanto en Simio como en el simulador desarrollado, obteniendo los siguientes resultados:

Vectores de entrada						Salidas servidor	
Tiempo	Interarrival time	Entities Per Arrival	Processing Time	Capacity		Esperado	Obtenido
50	1	1	2	1000		48	48
50	1	2	2	1		24	24
50	1	1	Triangular(2, 5, 10)	1000		Tabla A.6	
50	Exponential(0.25)	1	2	1000		Tabla A.7	
50	Exponential(0.25)	Poisson(5)	Triangular(2, 5, 10)	1000		Tabla A.8	

Tabla A.5: Resultados generales de los experimentos. De autoría propia

Esperado	45	44	44	45	46	45	43	43	45	44	46	44	44	44	44
Obtenido	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabla A.6: Muestras de experimento 3. De autoría propia

Esperado	213	172	189	216	192	210	209	204	212	200	195	189	188	201	195
Obtenido	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabla A.7: Muestras de experimento 4. De autoría propia.

Esperado	957	899	864	832	813	958	763	767	962	1031	887	904	807	934	979
Obtenido	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabla A.8: Muestras de experimento 5. De autoría propia.

Para los experimentos donde se agregaron distribuciones aleatorias continuas, el simulador desarrollado entraba en bucles infinitos, por lo que no se obtuvo ningún resultado.

Al no obtener la misma salida que los valores esperados, el simulador no está generando resultados correctos.

A.2.6.3. Segunda fase de validación de resultados

Luego de corregir el prototipo anterior, se realizaron nuevamente las pruebas obteniéndose los mismos resultados de la Tabla A.5 con la diferencia de que ahora sí se obtenían muestras en los experimentos donde existían distribuciones continuas.

Esperado	45	44	44	45	46	45	43	43	45	44	46	44	44	44	44
Obtenido	43	45	44	44	44	44	44	45	44	43	45	43	45	44	43

Tabla A.9: Muestras de experimento 3. De autoría propia.

Esperado	213	172	189	216	192	210	209	204	212	200	195	189	188	201	195
Obtenido	181	195	164	201	206	195	208	205	180	200	184	188	204	183	211

Tabla A.10: Muestras de experimento 4. De autoría propia.

Esperado	957	899	864	832	813	958	763	767	962	1031	887	904	807	934	979
Obtenido	919	912	829	882	860	919	955	776	891	867	795	860	948	952	982

Tabla A.11: Muestras de experimento 5. De autoría propia.

	Esperado	Obtenido
Media	44,40	44,00
Varianza	0,83	0,57
Observaciones	15	15
Diferencia hipotética de las medias	0	
Grados de libertad	27	
Estadístico t	1,31	
Valor crítico de t (dos colas)	2,05	

Tabla A.12: Prueba de medias para muestras no apareadas de experimento 3. De autoría propia.

	Esperado	Obtenido
Media	199,00	193,67
Varianza	145,43	175,52
Observaciones	15	15
Diferencia hipotética de las medias	0	
Grados de libertad	28	
Estadístico t	1,15	
Valor crítico de t (dos colas)	2,05	

Tabla A.13: Prueba de medias para muestras no apareadas de experimento 4. De autoría propia.

	Esperado	Obtenido
Media	890,47	889,80
Varianza	6602,41	3585,60
Observaciones	15	15
Diferencia hipotética de las medias	0	
Grados de libertad	26	
Estadístico t	0,03	
Valor crítico de t (dos colas)	2,06	

Tabla A.14: Prueba de medias para muestras no apareadas de experimento 5. De autoría propia.

Como en todos los experimentos el estadístico t está entre los valores positivo y negativo del valor crítico de t para dos colas, no se rechaza la hipótesis que plantea la nula diferencia entre los resultados de Simio y el simulador desarrollado, así que el simulador puede que esté dando resultados de manera correcta.

A.2.7. Prototipo 8. Simulador de red de colas (consola)

A.2.7.1. Fase de uso del framework

Con el simulador se ha demostrado la capacidad de implementar los componentes de tipo fuente, servidor, sumidero y ruta, por lo que aunado a la existencia de inversión de control y el seguimiento de buenas prácticas de Ingeniería de Software, se ha cumplido el objetivo propuesto bajo la metodología establecida.

A.2.7.2. Fase de validación de resultados

Para la validación de resultados se realizaron varias redes donde se corrieron algunos experimentos.

Componente	Propiedad	Valor
Source1	Entities Per Arrival	Poisson(4)
	Interarrival time	Exponential(7)
Server1	Processing Time	Exponential(1)
	Capacity	1000
Path1	Weight	1
Path2	Weight	1

Tabla A.15: Vectores de entrada de simulación de red sencilla. De autoría propia.

Componente	Propiedad	Valor
Source1	Entities Per Arrival	Poisson(4)
	Interarrival time	Exponential(2)
Server1	Processing Time	Exponential(1)
	Capacity	1000
Server2	Processing Time	Triangular(1,5,10)
	Capacity	1000
Path1	Weight	0.8
Path2	Weight	0.2
Path3	Weight	1
Path4	Weight	True
Path5	Weight	False

Tabla A.16: Vectores de entrada de simulación de rutas booleanas. De autoría propia.

Componente	Propiedad	Valor
Source1	Entities Per Arrival	1
	Interarrival time	1
Server1	Processing Time	2
	Capacity	1000
Server2	Processing Time	3
	Capacity	1000
Path3	Weight	True
Path4	Weight	True

Tabla A.17: Vectores de entrada de simulación de selección de rutas. De autoría propia.

Componente	Propiedad	Valor
Source1	Entities Per Arrival	1
	Interarrival time	1
Source2	Entities Per Arrival	1
	Interarrival time	1
Server1	Processing Time	2
	Capacity	1000
Server2	Processing Time	2
	Capacity	1000
Path3 y Path6	Weight	40
Path2 y Path 5	Weight	80

Tabla A.18: Vectores de entrada de simulación de elementos en paralelo. De autoría propia.

Dados los vectores de entrada, se tomaron las siguientes muestras:

Esperado	14	25	17	15	14	8	11	10	9	7	13	11	32	14	10
Obtenido	14	22	25	12	5	15	19	5	13	7	14	11	17	10	19

Tabla A.19: Muestras de simulación de red sencilla. De autoría propia.

Esperado 1	27	27	30	63	23	65	28	37	34	57	38	34	27	31	34
Esperado 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Obtenido 1	21	36	36	25	26	33	31	45	27	47	45	32	57	59	31
Obtenido 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabla A.20: Muestras de simulación de rutas booleanas en sumidero 1 y 2. De autoría propia.

Esperado 1	8	9	8	7	10	5	11	4	8	7	8	7	6	8	9
Esperado 2	7	6	7	8	5	10	4	11	7	8	7	8	9	7	6
Obtenido 1	10	6	7	8	4	5	8	7	9	10	6	9	11	8	9
Obtenido 2	5	9	8	7	11	10	7	8	6	5	9	6	4	7	6

Tabla A.21: Muestras de simulación de selección de rutas en sumidero 1 y 2. De autoría propia.

Esperado 1	18	16	18	17	17	17	16	18	17	18	16	17	18	16	17
Esperado 2	18	18	16	16	17	16	16	17	17	18	17	17	17	17	17
Obtenido 1	15	18	18	18	17	17	18	18	17	16	18	17	18	16	17
Obtenido 2	18	17	18	17	16	18	16	16	18	18	16	16	16	18	17

Tabla A.22: Muestras de simulación de elementos en paralelo en sumidero 1 y 2. De autoría propia.

	Esperado	Obtenido
Media	14,00	13,87
Varianza	44,00	34,70
Observaciones	15	15
Diferencia hipotética de las medias	0	
Grados de libertad	28	
Estadístico t	0,06	
Valor crítico de t (dos colas)	2,05	

Tabla A.23: Prueba de medias para muestras no apareadas de simulación de red sencilla. De autoría propia.

	Esperado 1	Obtenido 1
Media	37,00	36,73
Varianza	182,14	131,92
Observaciones	15	15
Diferencia hipotética de las medias	0	
Grados de libertad	27	
Estadístico t	0,06	
Valor crítico de t (dos colas)	2,05	

Tabla A.24: Prueba de medias para muestras no apareadas de simulación de rutas booleanas. De autoría propia.

	Esperado 1	Obtenido 1
Media	7,67	7,80
Varianza	3,24	3,89
Observaciones	15	15
Diferencia hipotética de las medias	0	
Grados de libertad	28	
Estadístico t	-0,19	
Valor crítico de t (dos colas)	2,05	

Tabla A.25: Prueba de medias para muestras no apareadas de sumidero 1 durante simulación de selección de rutas. De autoría propia.

	Esperado 2	Obtenido 2
Media	7,33	7,20
Varianza	3,24	3,89
Observaciones	15	15
Diferencia hipotética de las medias	0	
Grados de libertad	28	
Estadístico t	0,19	
Valor crítico de t (dos colas)	2,05	

Tabla A.26: Prueba de medias para muestras no apareadas de sumidero 2 durante simulación de selección de rutas. De autoría propia.

	Esperado 1	Obtenido 1
Media	17,07	17,20
Varianza	0,64	0,89
Observaciones	15	15
Diferencia hipotética de las medias	0	
Grados de libertad	27	
Estadístico t	-0,42	
Valor crítico de t (dos colas)	2,05	

Tabla A.27: Prueba de medias para muestras no apareadas de sumidero 1 durante simulación de elementos en paralelo. De autoría propia.

	Esperado 2	Obtenido 2
Media	16,93	17,00
Varianza	0,50	0,86
Observaciones	15	15
Diferencia hipotética de las medias	0	
Grados de libertad	26	
Estadístico t	-0,22	
Valor crítico de t (dos colas)	2,06	

Tabla A.28: Prueba de medias para muestras no apareadas de sumidero 2 durante simulación de elementos en paralelo. De autoría propia.

Como en todos las simulaciones el estadístico t está entre los valores positivo y negativo del valor crítico de t para dos colas, no se rechaza la hipótesis que plantea la nula diferencia entre los resultados de Simio y el simulador desarrollado, así que el simulador puede que esté dando resultados de manera correcta.

A.2.8. Evaluación y validación de la investigación

A.2.8.1. Comportamiento e inversión de control

Existe inversión de control en varios de los mecanismos implementados en el *framework*, entre los que destacan:

- Los modelos llaman a la implementación de los métodos que extienden los usuarios para llevar a cabo las simulaciones.

- El sistema dinámico provee un comportamiento predefinido para la ejecución de transiciones y salidas, llegando incluso a establecer reglas para su uso.
- El software ejecuta las simulaciones a partir de los elementos inyectados definidos por los usuarios.

Para una lista más extensa, puede consultarse las enumeraciones de motivos expuestas en las evaluaciones anteriores.

A.2.8.2. Cerrado a la modificación

El simulador desarrollado posee la implementación y es capaz de implementar componentes de tipo:

- Fuente.
- Sumidero.
- Servidor.
- Ruta.

Por lo tanto, es posible usar el *framework* para la elaboración de cualquier simulador de eventos discretos, sin necesidad de modificar la base de código existente.

A.2.8.3. Seguimiento de buenas prácticas de Ingeniería de Software

A continuación se muestran algunos datos de la versión previa al lanzamiento:

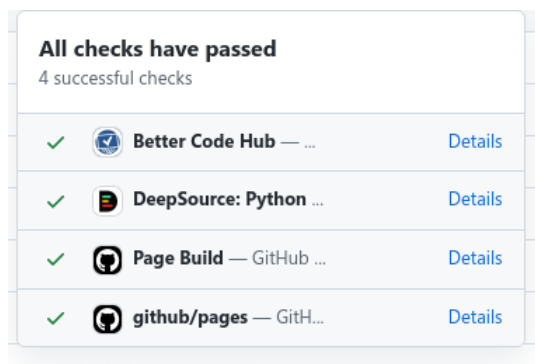


Figura A.3: Checks de la *release*. De autoría propia.

La versión previa al lanzamiento pasó todas las pruebas de Better Code Hub y Deep Source como muestra la Figura A.3.

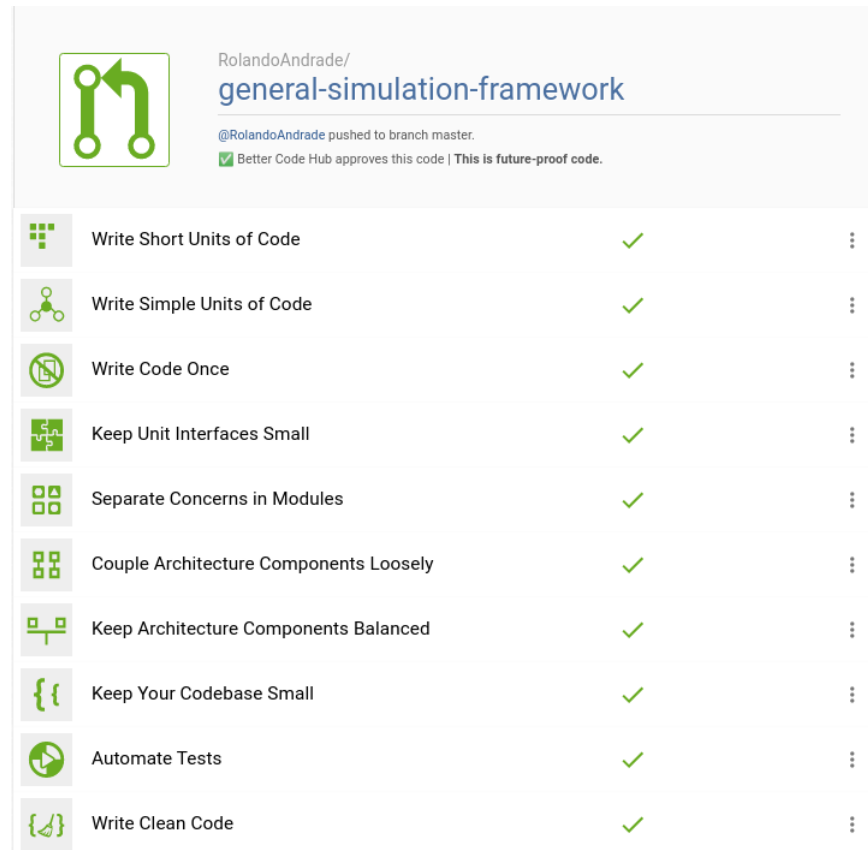


Figura A.4: Análisis del software de Better Code Hub. De autoría propia.

Better Code Hub arrojó un 10 sobre 10 (Figura A.4) en la calificación de la calidad de código, tomando en cuenta las métricas que mide. Ante esta evaluación, la herramienta indicó que "Better Code Hub aprueba este código".

La evaluación ha sido alterada a lo largo del proyecto para aprobar la métrica de "balance entre los módulos", pues existía un desbalance en la cantidad de código entre los módulos que representaban al dominio. Al ser un detalle conocido y justificado por el diseño aplicado, la evaluación fue modificada para aprobarlo.

Por otro lado, una métrica al borde de ser rechazada fue "mantener las unidades de interfaces reducidas", que evalúa la cantidad de parámetros de los métodos, ya que más de 23 métodos poseen más de 2 parámetros, sin embargo, todos están dentro del margen de tolerancia de la herramienta.

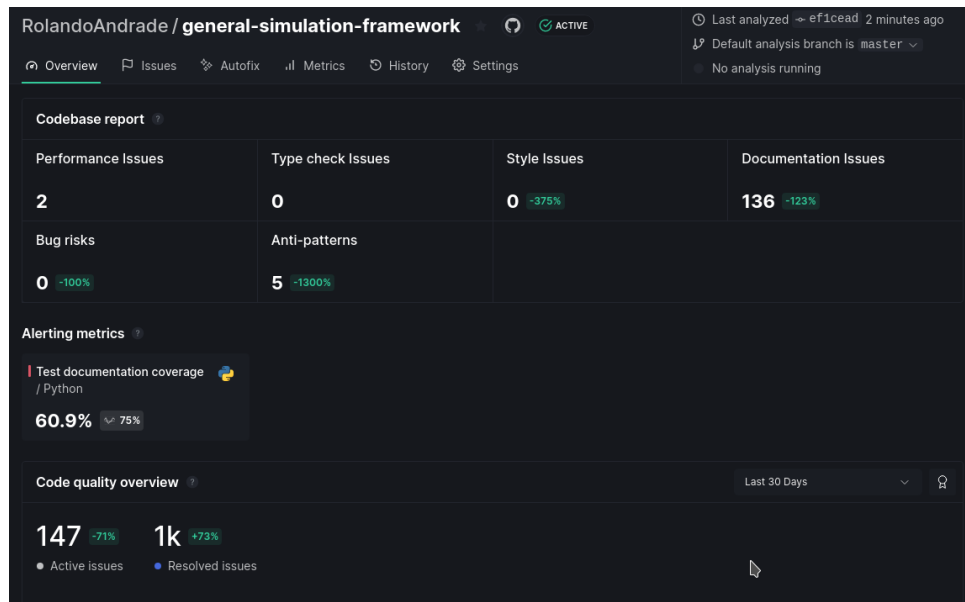


Figura A.5: Análisis del software de DeepSource. De autoría propia.

Por su parte, DeepSource arrojó varias métricas interesantes:

- 136 errores de documentación: surgen por la falta de documentación en la mayoría de los métodos sobrescritos en subclases y reglas de estilo de los comentarios para la documentación. Se decidió ignorar los resultados porque no se encontró ningún caso de un método o clase, no documentado de algún modo.
- 5 antipatrones: fueron hechos con el fin de mejorar la legibilidad del código, como la inclusión de sentencias *if* anidadas en vez de usar operadores lógicos. Se corrigieron automáticamente los antipatrones hasta que quedaron cinco (5) que no se consideraron como tal, por lo que se ignoraron.
- 1 riesgo de seguridad: para el uso de expresiones introducidas por el usuario se usa una función de Python llamada *eval*, que es capaz de ejecutar código malicioso. DeepSource da como alternativa el uso de *ast.literal_eval*, por lo que se deja esta observación como una recomendación.
- Métricas alarmantes: solo el 60.9 % de las pruebas se encuentran documentadas. Se ignoró la métrica, ya que la mayoría de las pruebas que no están documentadas son pruebas unitarias sobre métodos ya documentados. No obstante, se toma la recomendación para futuras versiones.

A.2.9. Validación de resultados

Los datos, métricas y análisis de medias pueden ser consultados en la Sección A.2.7.2, donde no se encontró evidencia estadística suficiente para desestimar la idea de que los resultados del simulador desarrollado y los resultados del simulador de pruebas son iguales, por lo que los resultados del simulador, y por tanto del *framework* son correctos.

Anexo B

Juego de la vida

B.1. Definición

El “Juego de la vida” es un autómata celular. Los autómatas celulares son un tipo de modelo matemático para un sistema dinámico que evoluciona en pasos discretos, que se compone celdas sobre las cual se van a producir ciertas interacciones [44].

B.2. Leyes genéticas de Conway

B.2.1. Supervivencia

Si cada célula tiene 2 o 3 vecinos vivos y está viva, entonces sobrevive a la siguiente generación [44].

B.2.2. Fallecimiento

Si una célula tiene menos de dos vecinos vivos fallece por soledad, si tiene más de tres vecinos vivos, fallece por superpoblación [44].

B.2.3. Nacimiento

Si una celda vacía pasa a tener exactamente 3 células vivas alrededor su estado futuro en el siguiente turno será el de célula viva (nacimiento de nuevo individuo) [44].

Anexo C

Nuevos paradigmas y futuro de la investigación

Los avances tecnológicos y la inclusión de la Ingeniería de Software en el área de la simulación, ha significado cambios de paradigma sobre los *frameworks* a desarrollar a futuro con el fin de corregir los errores de sistemas anteriores y aprovechar las capacidades tecnológicas de los últimos años.

C.1. Implementación de los *frameworks*

A lo largo de la historia se fueron creando varios lenguajes de modelado y diseño de simulaciones como PSM++ o SIMSCRIPT que con el tiempo fueron remplazados por lenguajes de programación orientados objetos como C++ y Java. Tener las mismas estructuras de datos requeridas y poder reutilizar los componentes por medio de la inclusión de librerías, además de poseer las características de un lenguaje de propósito general, permitió que a finales de los 90 se realizara este traslado [6].

Con el impacto de los datos en la actualidad, se espera que en los próximos años las herramientas de simulación estén directamente integradas a herramientas de hojas de cálculo y lenguajes de programación estadísticos como R y MATLAB a fin de agilizar el proceso de obtención, manipulación, procesamiento y pronóstico de los datos [6].

Por otra parte, se empezó a ver un retraso notable en los simuladores de código

abierto respecto a la infraestructura existente, pues no se adaptan en tiempos, precisión y rendimiento al estado del arte de la infraestructura, pudiendo quedar obsoletos en los próximos años [45].

El futuro de los diseños de los frameworks estarán basados en los mecanismos de inyección por módulos y complementos, lo que permitirá ensamblar simuladores con poco esfuerzo [45].

Finalmente, para la implementación de herramientas completas de simulación, se espera la conjunción de lenguajes de modelado de simulación como Modelica, con lenguajes de programación de propósito general como C++, a través de compiladores híbridos, lo que facilitará tanto los procesos de simulación como los procesos de corrección por parte de los usuarios finales [6].

C.2. Computación paralela

Los investigadores en los próximos años se enfocarán en la realización de herramientas de código abierto para la implementación de software de simulación desacoplados de la infraestructura y que permitan correr simulaciones en mononúcleo, multinúcleo y multihilo [45].

Con cada generación de computadoras, se prevé el crecimiento de la complejidad computacional de los modelos mientras que los tiempos para su ejecución cada vez se vuelven más razonables. Con la popularización de los mecanismos de cómputo en red y sistemas distribuidos, cada vez más surge la necesidad de adaptar las herramientas a los procesos de computación paralela [6].

El paradigma usado actualmente, solo explota los eventos que ocurren simultáneamente para procesarlos en paralelo, sin embargo, estadísticamente hablando son muy raros, por lo que este mecanismo de paralelismo es poco efectivo [6].

La nueva generación de ambientes de desarrollo para la implementación de herramientas de simulación, pese a no poder paralelizar los procesos de trayectorias por las relaciones de causalidad, promete ser más eficiente al paralelizar los procesos

de cambio de estado, al realizar el cómputo de las variables que se alteran de manera paralela, disminuyendo los tiempos de cálculo numérico [6].

No obstante, algunos investigadores han querido optimizar aún más los tiempos a costa de la precisión de los resultados, proponiendo modelos conservativos y optimistas, caracterizados por ser predecibles bajo cierto margen de error, lo que reduciría el número de iteraciones de las operaciones, además de permitir el cómputo paralelo de muchos estados en distintos tiempos al existir independencia de la causalidad [6]. Estos modelos estadísticos, también quieren ser llevados a microarquitecturas para generar trazas sintéticas en tiempos razonables, lo que implicará un cambio drástico en los usos de la simulación en tiempo real [45].

El principal problema que afrontan estas propuestas, y que se está buscando resolver, es la trazabilidad de los datos, puesto que los estados al usar una estructura de datos almacenada en memoria y ser reversibles, pueden generar inconsistencias que no han logrado tener soluciones prácticas [6].

Ante el estancamiento de las velocidades secuenciales en un solo procesador, el límite de velocidad para la mayoría de las simulaciones actuales ya está determinado, por lo que es previsible que el siguiente paso sea hacia la resolución de los problemas de paralelismo [6].

C.3. Combinación de modelos de simulación

Desde los inicios de la simulación, se ha discutido sobre la mejor forma de simular. El debate entre simulación orientada por eventos, procesos y actividades se ha extendido hasta los días de hoy, llegando a existir herramientas que cubren cada uno de los paradigmas [2].

Pese al orden establecido, con los estudios matemáticos en el área de los últimos años, ha crecido el número de modelos analíticos para representar los sistemas dinámicos como Redes de Petri y Autómatas Celulares Asíncronos [6].

Todos los modelos propuestos presentan redescubrimientos y similitudes que están bajo investigación, con el fin de conocer las relaciones entre los mismos para saber el

significado de sus transformaciones y combinaciones.

C.4. Inteligencia Artificial

Desde finales del siglo pasado se han investigado mecanismos y *frameworks* para la integración de la simulación con la inteligencia artificial.

Los primeros intentos de fusión fueron propuestas de utilizar la inteligencia artificial para optimizar el procesamiento de operaciones y el modelado de sistemas [46].

No obstante, pese a realizarse muchas investigaciones al respecto desde esos primeros artículos, a día de hoy no existen herramientas de código abierto que integren las ideas, por lo que es posible que con el nuevo auge de la inteligencia artificial, el área pueda volver a considerarse para la simulación asistida.

A pesar del estancamiento en el área de apoyo a la simulación por inteligencia artificial, se han propuesto y desarrollado *frameworks* de inteligencia artificial que permiten realizar pronósticos a través de simulaciones, lo que promete ser una alternativa a una mejor toma de decisiones [47].

Finalmente, se ha visto el potencial de los entornos virtuales simulados para el entrenamiento de modelos de inteligencia artificial, por lo que se han diseñado ambientes de desarrollo como OpenAI Gym para el entrenamiento de agentes por medio de aprendizaje reforzado, así que se espera que en los próximos años siga habiendo innovaciones en el área [48].

Anexo D

Uso del *framework*

A continuación se presenta una traducción y resumen de los ejemplos de uso del *framework* extraídos del Jupyter Notebook oficial del proyecto [49]:

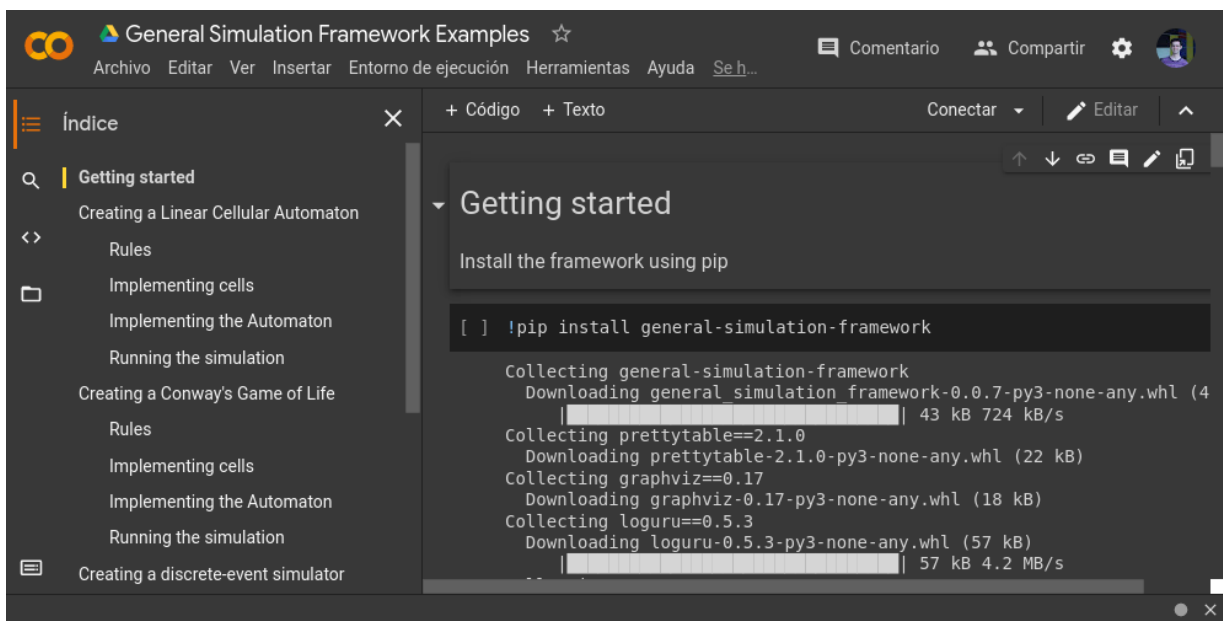


Figura D.1: Imagen del cuaderno en Colaboratory. De autoría propia.

Los mismos pueden ser ejecutados y visualizados en la fuente de referencia, sin necesidad de realizar la instalación de algún software aparte del navegador web.

D.1. Instalación

Para instalar los paquetes necesarios del *framework* verifique tener los requisitos de Python establecidos en las limitaciones y corra el siguiente comando:

```
1 \ $ pip install general-simulation-framework
```

Código D.1: Instalación del *framework*. Extraído de [49].

D.2. Creación un autómatas celular lineal

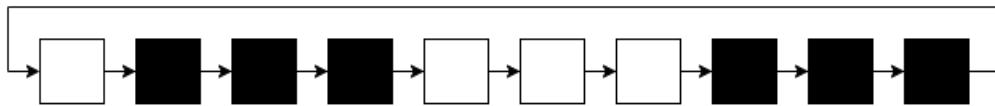


Figura D.2: Autómata celular lineal. Extraído de [49].

Un autómata celular es un modelo discreto de computación estudiado en la teoría de los autómatas.

Consiste en una rejilla regular de celdas, cada una de las cuales se encuentra en uno de un número finito de estados, como por ejemplo encendido y apagado. La rejilla puede tener cualquier número finito de dimensiones. Para cada celda, se define un conjunto de celdas llamado su vecindad en relación con la celda especificada. Se selecciona un estado inicial (tiempo $t = 0$) asignando un estado para cada celda. Se crea una nueva generación (avanzando t en 1), según alguna regla fija (generalmente, una función matemática) que determina el nuevo estado de cada celda en función del estado actual de la celda y de los estados de las celdas de su vecindad. Normalmente, la regla de actualización del estado de las células es la misma para cada célula y no cambia con el tiempo.

D.2.1. Reglas

Para este ejemplo, la regla para actualizar el estado de las celdas es: para cada celda del autómata, la celda tomará el estado de su celda vecina izquierda. Con esta

regla, las celdas se moverán un paso hacia la derecha en cada generación.

D.2.2. Implementación de las celdas

Para implementar las celdas, puede extender la clase *DiscreteTimeModel*, y definir los métodos abstractos protegidos.

```

1 ...
2 class Cell(DiscreteTimeModel):
3     """Célula del automáta celular lineal
4
5     Tiene un estado vivo o muerto. Cuando recibe una entrada, cambia su estado a esa
6     ↪ entrada. Su salida es su mismo estado.
7
8     Attributes:
9         __symbol (str): Símbolo que representa la celda cuando es mostrado en consola.
10    """
11    __symbol: str
12
13    def __init__(self, dynamic_system: DiscreteEventDynamicSystem, state: bool, symbol
14    ↪ : str = None):
15        """
16        Args:
17            dynamic_system (DiscreteEventDynamicSystem): Atómata al que pertenece.
18            state (bool): Estado que determina si está viva (True) o muerta (False).
19            symbol (str): Símbolo que representa la celda cuando es mostrado en consola.
20        """
21        super().__init__(dynamic_system, state=state)
22        self.__symbol = symbol or "\u2665"
23
24    def __state_transition(self, state: bool, inputs: Dict[str, bool]) -> bool:
25        """
26        Recibe una entrada y cambia el estado de la celda.
27
28        Args:

```

```

26         state (bool): Estado actual de la celda.
27         inputs: Un diccionario cuya clave es la celda fuente de entrada y el valor su
    ↪ estado.
28
29     Returns:
30         Nuevo estado de la celda.
31     """
32     next_state: bool = list(inputs.values())[0]
33     return next_state
34
35     def __output_function(self, state: bool) -> bool:
36         """
37         Retorna el estado actual de la celda.
38         """
39         return state
40
41     def __str__(self):
42         """Muestra la celda con el símbolo establecido."""
43         is_alive = self.get_state()
44         if is_alive:
45             return self.__symbol
46         else:
47             return "-"

```

Código D.2: Implementación de las células. Extraído de [49].

La clase *Cell*, debe recibir el *DiscreteEventDynamicSystem* al que pertenece el modelo. También se incluye el estado de la celda como un booleano y un símbolo que representa a las celdas cuando son impresas.

Cuando se ejecuta una generación, el framework obtendrá las salidas de cada celda definida por *_output_function*, y las inyectará en el siguiente modelo mediante *_state_transition*. El método de transición de estado, recibe un diccionario con el modelo de entrada de origen y su estado, y devuelve el nuevo estado que tomará la

celda.

D.2.3. Implementación del autómeta

El autómeta es un sistema dinámico, específicamente un sistema dinámico de eventos discretos, por lo que extiende la clase *DiscreteEventDynamicSystem*:

```

1 ...
2 class LinearAutomaton(DiscreteEventDynamicSystem):
3     """Implementación del autómeta celular.
4
5     Posee un grupo de celdas conectadas entre sí. La celda de salida de cada celda es
6     ↪ equivalente a su vecina derecha.
7
8     Atributos:
9         __cells (List[Cell]): Grupo de celdas del autómeta.
10    """
11    __cells: List[Cell]
12
13    def __init__(self, cells: int = 5, random_seed: int = 42):
14        """
15        Args:
16            cells (int): Número de celdas del autómeta.
17            random_seed (int): Semilla para la generación aleatoria de estados.
18        """
19        super().__init__()
20        seed(random_seed)
21        self.__create_cells(cells)
22        self.__create_relations(cells)
23
24    def __create_cells(self, cells: int):
25        """Añade celdas al autómeta
26
27        Args:
28            cells (int): Número de celdas del autómeta.

```

```

27     """
28     self.__cells = []
29     for i in range(cells):
30         is_alive = random() < 0.5
31         self.__cells.append(Cell(self, is_alive))
32
33     def __create_relations(self, cells: int):
34         """Crea las conexiones entre las celdas izquierda y derecha.
35         Args:
36             cells (int): Número de celdas del autómata.
37         """
38         for i in range(cells):
39             self.__cells[i-1].add(self.__cells[i])
40
41     def __str__(self):
42         """Cambia el formato con el que es mostrado"""
43         s = ""
44         for cell in self.__cells:
45             s += str(cell)
46         return s

```

Código D.3: Implementación del sistema dinámico. Extraído de [49].

LinearAutomaton recibe el número de celdas que tendrá y una semilla aleatoria para determinar el estado aleatorio inicial de las celdas.

Primero crea las celdas, estableciendo el estado viva o muerta con una probabilidad de 0,5. La instancia de *DiscreteEventDynamicSystem* requerida por la celda, la provee el mismo autómata, inyectándose a sí mismo.

Luego, el autómata conecta los modelos, estableciendo como salida de la $celda_{i-1}$, la $celda_i$.

D.2.4. Corrida de la simulación

Use la clase *DiscreteEventExperiment* para simular un experimento.

```

1 ...
2 linear_automaton = LinearAutomaton(cells=10)
3 experiment = DiscreteEventExperiment(linear_automaton)
4 print(linear_automaton)
5 experiment.simulation_control.start(stop_time=5)
6 experiment.simulation_control.wait()
7 print(linear_automaton)

```

Código D.4: Simulación de cinco generaciones. Extraído de [49].

El código crea el experimento con el autómata lineal como sistema dinámico. Luego, lo simula durante 5 generaciones.

Como la simulación se ejecuta en un hilo diferente, debe esperar a que termine con *experiment.simulation_control.wait()* para ver los resultados finales de la simulación.

D.3. Creación del Juego de la Vida de Conway

Ya existe un apartado en la investigación (anexo B) dedicado a la definición y reglas del autómata, así que se procede con su implementación.

D.3.1. Implementación de las celdas

La base de código es exactamente la misma que el ejemplo anterior, con la única diferencia de que ahora se poseen hasta ocho entradas provenientes de vecinos. Por cada vecino se añade uno al contador si está vivo. Luego aplica las reglas de supervivencia.

```

1 ...
2 class Cell(DiscreteTimeModel):
3     ...
4     def __state_transition(self, state: bool, inputs: Dict[str, bool]) -> bool:
5         """
6         Recibe un conjunto de entradas y cambia el estado de la celda.

```

```

7     Args:
8         state (bool): Estado actual de la celda
9         inputs: Diccionario cuya clave es el vecino que emite la entrada y el valor es la
    ↪ salida de ese vecino.
10    Returns:
11        El nuevo estado de la celda.
12    """
13    values = inputs.values()
14    count_alive = 0
15    for is_alive in values:
16        if is_alive:
17            count_alive = count_alive + 1
18    return (not state and count_alive == 3) or (state and 2 <= count_alive <= 3)
19    ...

```

Código D.5: Resumen de implementación de la celda. Extraído de [49].

D.3.2. Implementación del autómeta

No existe diferencia alguna con el uso del *framework*, solo variando en cómo se crean la celdas y sus relaciones.

```

1    ...
2    class Board(DiscreteEventDynamicSystem):
3        """Juego de la vida
4
5        Tiene un conjunto de celdas conectadas entre sí. La salida de una celda va hacia todos
    ↪ sus vecinos.
6
7        Attributes:
8            __cells (List[List[Cell]]): Grupo de celdas del tablero.
9        """
10    __cells: List[List[Cell]]

```

```

11
12 def __init__(self, width: int, height: int, random_seed: int = 42):
13     super().__init__()
14     seed(random_seed)
15     self._create_cells(width, height)
16     self._define_relations(width, height)
17
18 def _create_cells(self, width: int, height: int):
19     """Crea la celdas"""
20     self._cells = []
21     for i in range(height):
22         row = []
23         for j in range(width):
24             row.append(Cell(self, random() < 0.5))
25         self._cells.append(row)
26
27 def _define_relations(self, width: int, height: int):
28     """Crea las conexiones entre las celdas.
29     Args:
30         width (int): Número de columnas
31         height (int): Número de filas
32     """
33     for i in range(height):
34         for j in range(width):
35             for x in range(max(0, i - 1), min(i + 2, height)):
36                 for y in range(max(0, j - 1), min(j + 2, width)):
37                     if x != i or y != j:
38                         self._cells[i][j].add(self._cells[x][y])
39
40 def __str__(self):
41     """Cambia como se muestra el tablero"""
42     s = ""
43     for row in self._cells:

```

```

44     for cell in row:
45         s += str(cell)
46         s += "\n"
47     return s
48
49     def get_output(self) -> DynamicSystemOutput:
50         """Imprime el tablero cada generación."""
51         print(self)
52         return super().get_output()

```

Código D.6: Implementación del autómata. Extraído de [49].

D.3.3. Corrida de la simulación

Cree el experimento y asigne el sistema dinámico.

```

1 ...
2 board = Board(width=10, height=10)
3 experiment = DiscreteEventExperiment(board)
4 print(board)
5 experiment.simulation_control.start(stop_time=10)
6 experiment.simulation_control.wait()
7 print(board)

```

Código D.7: Simulación de 10 generaciones. Extraído de [49].

D.4. Creación de un simulador de eventos discretos

Imagine una fábrica a la que llega un determinado recurso, un determinado servidor lo procesa y lo envía a otra estación y sale de la fábrica.

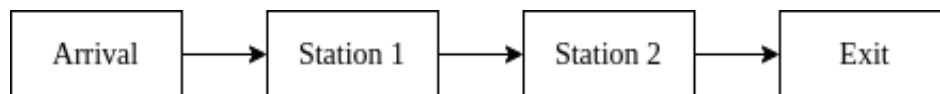


Figura D.3: Sistema de dos estaciones. Extraído de [49].

D.4.1. Reglas

- Las llegadas pueden variar tanto por cantidad de piezas como por tiempo entre esas llegadas.
- Las estaciones procesan los componentes, de uno en uno.
 - Cuando una pieza entra a la estación, actualiza el tiempo de procesamiento, y añade el elemento a procesar.
 - Cuando se completa el tiempo de procesamiento, extrae la pieza de los elementos a procesar y comienza a procesar otra si es posible.
- La salida cuenta las piezas totales procesadas.

D.4.2. Creación de las estaciones

Las estaciones son objetos de tipo *DiscreteEventModel*, por lo que puede extender la clase e implementar los métodos abstractos para crear estaciones.

```

1 ...
2 class Station(DiscreteEventModel):
3     """Estación de un simulador
4
5     Procesa las entradas que recibe. Su estado posee un número de piezas que actualmente
6     ↪ se encuentran dentro de la estación y también lleva el tiempo faltante para finalizar
7     ↪ de procesar una pieza.
8
9     Atributos:
10     __processing_time(Expression): tiempo para procesar una pieza.
11
12     """
13     __processing_time: Expression
14
15     def __init__(self, dynamic_system: DiscreteEventDynamicSystem, processing_time:
16         ↪ Expression):
17         """

```

```

14     Args:
15         dynamic_system (DiscreteEventDynamicSystem): fábrica a la que pertenece la
        ↪ estación.
16         processing_time (Expression): tiempo para procesar una pieza.
17     """
18     super().__init__(dynamic_system, state={
19         'parts': 0,
20         'remaining_time': -1
21     })
22     self._processing_time = processing_time
23
24     def _internal_state_transition_function(self, state: StationState) -> StationState:
25         """Remueve una pieza de las que hace falta procesar y programa un evento para
        ↪ procesar una pieza nueva.
26         """
27         state["parts"] = max(state["parts"] - 1, 0)
28         self.schedule(self.get_time())
29         return state
30
31     def _external_state_transition_function(self, state: StationState, inputs: Dict[str, int],
32                                           event_time: Time) -> StationState:
33         """Añade una pieza para procesar.
34         """
35         values = inputs.values()
36         state["remaining_time"] = state["remaining_time"] - event_time
37         for number_of_parts in values:
38             if state["parts"] > 0:
39                 state["parts"] = state["parts"] + number_of_parts
40             elif state["parts"] == 0:
41                 state["parts"] = number_of_parts
42                 self.schedule(self.get_time())
43         return state
44

```

```

45 def __time_advance_function(self, state: StationState) -> Time:
46     """Obtiene el tiempo para procesar la próxima pieza.
47     """
48     if state["parts"] < 1:
49         state["remaining_time"] = Time(-1)
50     else:
51         state["remaining_time"] = Time(self.__processing_time.evaluate())
52     return state["remaining_time"]
53
54 def __output_function(self, state: StationState) -> int:
55     """Retorna el número de piezas procesadas.
56     """
57     if state["parts"] > 0:
58         return 1
59     return 0
60
61 def __str__(self):
62     return self.get_id()

```

Código D.8: Implementación de la estación. Extraído de [49].

Las estaciones procesan las piezas durante un tiempo de procesamiento. Cuando el tiempo termina, se emite un evento autónomo, por lo que el *framework* ejecuta la función *__internal_state_transition_function*, donde la estación elimina una pieza del procesamiento, y programa un evento para procesar una nueva. Cuando una pieza viene de otra estación, el *framework* ejecuta la función *__external_state_transition_function* donde la estación añade nuevas piezas para procesar.

El tiempo es *Time(-1)* si no hay piezas para procesar, y evalúa una expresión en el caso de que sí hayan piezas. Una expresión es un tipo de dato del *framework* que permite incluir casi cualquier valor. También puede programar eventos en un tiempo infinito *Time("inf")* para lograr el mismo resultado.

D.4.3. Creación del generador de piezas

El generador de piezas crea partes dado un tiempo entre llegadas y la cantidad de piezas a producir.

```

1 ...
2 class Generator(DiscreteEventModel):
3     """Generador de piezas
4
5     Crea partes dado un tiempo entre llegadas y la cantidad de piezas a producir.
6
7     Attributes:
8         __interarrival_time (Expression): tiempo entre las llegadas de las piezas.
9     """
10    __interarrival_time: Expression
11
12    def __init__(self, dynamic_system: DiscreteEventDynamicSystem, piezas:
13        ↪ GeneratorState,
14            interarrival_time: Expression):
15
16        """Args:
17            dynamic_system(DiscreteEventDynamicSystem): fábrica a la que pertenece la
18            ↪ estación.
19            piezas(GeneratorState): número de piezas a generar infinitamente o lista de nú
20            ↪ meros de piezas a generar por llegadas finitas.
21            interarrival_time(Expression): tiempo entre las llegadas de las piezas.
22        """
23        super().__init__(dynamic_system, state=piezas)
24        self.schedule(Time(0))
25        self.__interarrival_time = interarrival_time
26
27    def _internal_state_transition_function(
28        self, state: GeneratorState) -> GeneratorState:
29
30        """Genera una pieza."""

```

```

26     if isinstance(state, list):
27         state.pop(0)
28         self.schedule(self.get_time())
29     return state
30
31 def _time_advance_function(self, state: GeneratorState) -> Time:
32     """Calcula el tiempo de creación de la próxima pieza."""
33     if isinstance(state, list):
34         return self._interarrival_time.evaluate() if len(state) > 0 else Time(-1)
35     else:
36         return self._interarrival_time.evaluate()
37
38 def _output_function(self, state: GeneratorState):
39     """Obtiene la pieza creada"""
40     if isinstance(state, list):
41         return state[0]
42     else:
43         return state.evaluate()
44
45 def __str__(self):
46     return "Generator"

```

Código D.9: Implementación del generador. Extraído de [49].

El estado del generador puede ser una lista de enteros o una expresión. Si es una lista, el estado representa el número de piezas por llegada para cada llegada, de modo que $pieces_0$ equivale al número de partes en la primera llegada, $pieces_1$ a la segunda llegada, y así sucesivamente. Por otro lado, si se trata de una expresión, se genera el número de partes que da la expresión durante las infinitas llegadas.

Si el estado es una lista, la salida de un evento debe ser el primer elemento de la lista, a su vez, si es una expresión, evaluará la expresión.

Si la lista de partes está vacía, el generador deja de programar eventos autónomos.

D.4.4. Creación del sumidero

```

1 ...
2 class Exit(DiscreteEventModel):
3     """Salida de piezas
4
5     Sumidero de la fábrica. Acá arriban todas las piezas procesadas en la fábrica.
6     """
7
8     def __init__(self, dynamic_system: DiscreteEventDynamicSystem):
9         super().__init__(dynamic_system, state=0)
10
11     def _external_state_transition_function(self, state: int, inputs: Dict[str, int],
12                                           event_time: Time) -> int:
13         """Almacena las partes"""
14         return state + sum(inputs.values())
15
16     def _time_advance_function(self, state: ModelState) -> Time:
17         """Previene ejecutar un evento autónomo."""
18         return Time(-1)
19
20     def _output_function(self, state: ModelState) -> int:
21         """Retorna el número de piezas que se han procesado."""
22         return state
23
24     def __str__(self):
25         return "Exit"

```

Código D.10: Implementación del sumidero. Extraído de [49].

D.4.5. Creación de la fábrica

La fábrica es el sistema dinámico donde todas las estaciones procesan las piezas.

```

1 ...
2 class FactorySystem(DiscreteEventDynamicSystem):
3     """Fábrica de pizzas
4
5     Es el sistema dinámico donde todas las estaciones procesan las piezas.
6
7     Atributos:
8         generator (Generator): generador de piezas de la fábrica.
9         exit (Exit): dalidas de la fábrica,
10    """
11    generator: Generator
12    exit: Exit
13
14    def __init__(self, piezas: GeneratorState, interarrival_time: Expression):
15        """Args:
16            piezas(GeneratorState): número de piezas a crear por llegada
17            interarrival_time(Expression): tiempo entre llegadas.
18        """
19        super().__init__()
20        self.generator = Generator(self, piezas, interarrival_time)
21        self.exit = Exit(self)

```

Código D.11: Implementación de la fábrica. Extraído de [49].

Crea el generador y la salida. Las estaciones tendrán que ser definidas externamente, por lo que es posible definir múltiples tipos de fábricas a través de la conexión entre sus estaciones.

D.4.6. Diseño y simulación de la red

A continuación se muestra una simulación sencilla en la que una parte llega en el momento $t = 0$ y dos partes en $t = 1$, con tiempos de procesamiento de uno y dos segundos.

```

1 ...
2 factory = FactorySystem([1, 2], Value(1)) # 1 y 2 piezas por llegada, cada segundo
3 station_1 = Station(factory, Value(1)) # 1 segundo para procesar
4 station_2 = Station(factory, Value(2)) # 2 segundos para procesar
5
6 # construye la red
7 factory.generator.add(station_1) # estación 1 como salida de entrada de fábrica
8 station_1.add(station_2) # estación 2 como salida de estación 1
9 station_2.add(factory.exit) # salida de fábrica como salida de estación 2
10
11 # simula 10 segundos
12 experiment = DiscreteEventExperiment(factory)
13 experiment.simulation_control.start(stop_time=10)
14 experiment.simulation_control.wait()
15
16 print(factory.exit.get_output())
17 print(experiment.simulation_report.generate_report())

```

Código D.12: Diseño y simulación de la red. Extraído de [49].

D.4.6.1. Añadir expresiones

Puede usar las expresiones predefinidas por el *framework*.

```

1 ...
2 from gsf.core.mathematics.distributions import PoissonDistribution,
   ↪ ExponentialDistribution, TriangularDistribution
3 factory = FactorySystem(PoissonDistribution(5), ExponentialDistribution(0.5))
4 station_1 = Station(factory, TriangularDistribution(1, 2, 5))
5 station_2 = Station(factory, TriangularDistribution(1, 4, 5))
6 ...

```

Código D.13: Uso de expresiones. Extraído de [49].

Anexo E

Imágenes de la página de referencia

A continuación se presentan algunas imágenes de la página de referencia generada:

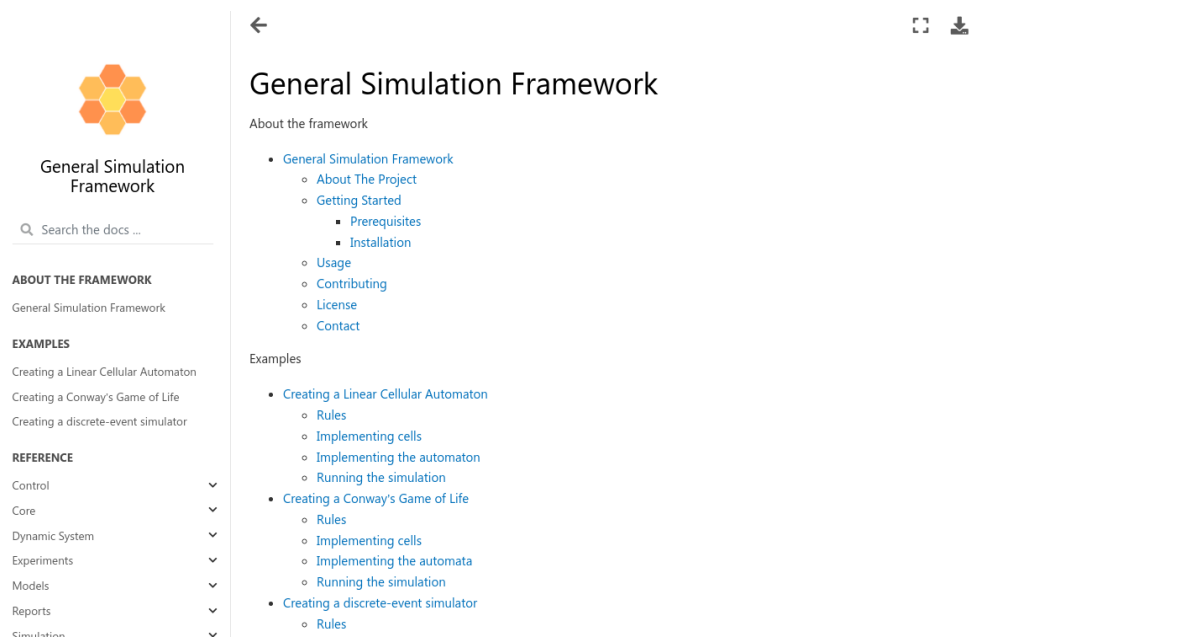
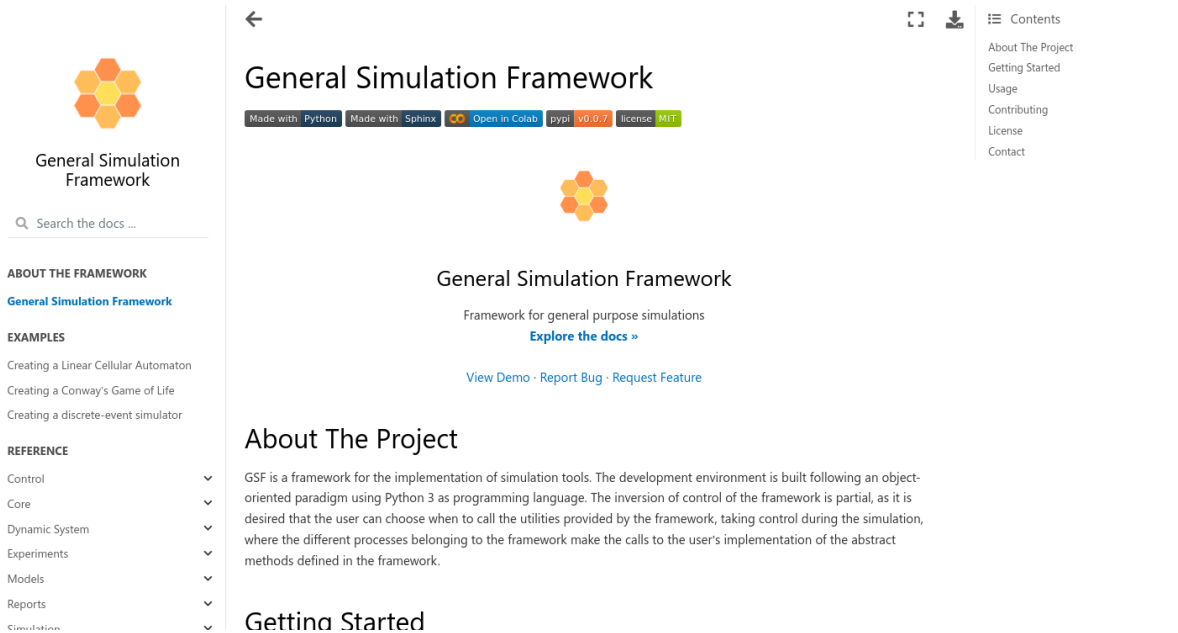


Figura E.1: Página de inicio. De autoría propia.



General Simulation Framework

Search the docs ...

ABOUT THE FRAMEWORK

General Simulation Framework

EXAMPLES

Creating a Linear Cellular Automaton

Creating a Conway's Game of Life

Creating a discrete-event simulator

REFERENCE

Control

Core

Dynamic System

Experiments

Models

Reports

Simulation

General Simulation Framework

Framework for general purpose simulations

[Explore the docs »](#)

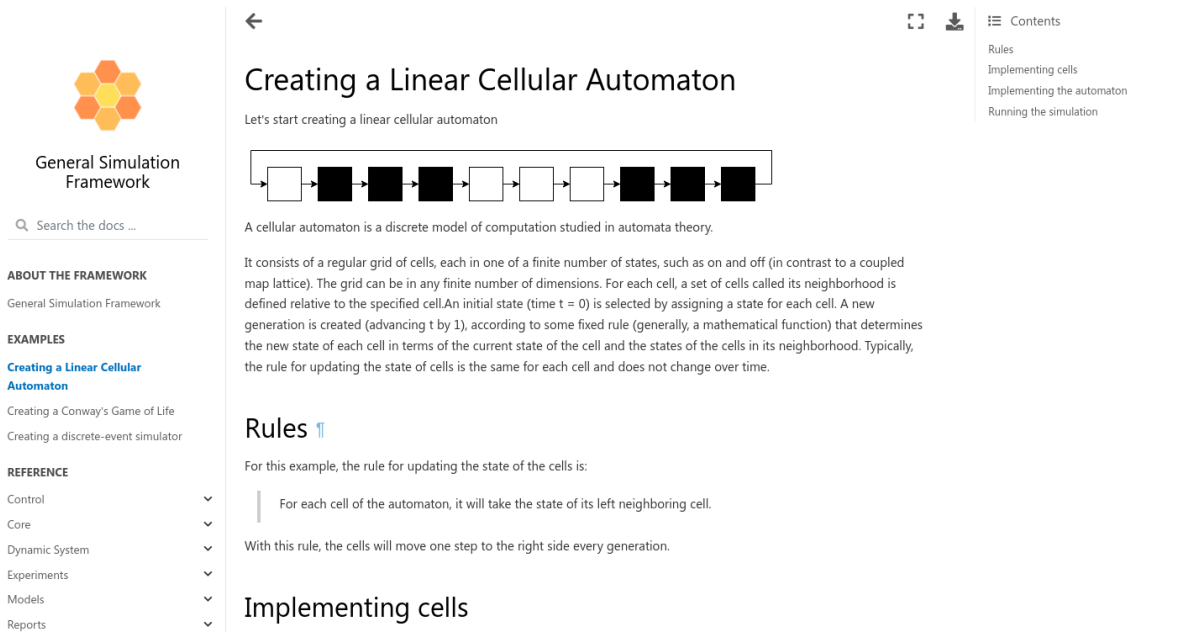
[View Demo](#) · [Report Bug](#) · [Request Feature](#)

About The Project

GSF is a framework for the implementation of simulation tools. The development environment is built following an object-oriented paradigm using Python 3 as programming language. The inversion of control of the framework is partial, as it is desired that the user can choose when to call the utilities provided by the framework, taking control during the simulation, where the different processes belonging to the framework make the calls to the user's implementation of the abstract methods defined in the framework.

Getting Started

Figura E.2: Página de información de proyecto. De autoría propia.



General Simulation Framework

Search the docs ...

ABOUT THE FRAMEWORK

General Simulation Framework

EXAMPLES

[Creating a Linear Cellular Automaton](#)

Creating a Conway's Game of Life

Creating a discrete-event simulator

REFERENCE

Control

Core

Dynamic System

Experiments

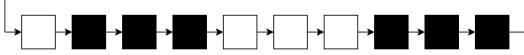
Models

Reports

Simulation

Creating a Linear Cellular Automaton

Let's start creating a linear cellular automaton



A cellular automaton is a discrete model of computation studied in automata theory.

It consists of a regular grid of cells, each in one of a finite number of states, such as on and off (in contrast to a coupled map lattice). The grid can be in any finite number of dimensions. For each cell, a set of cells called its neighborhood is defined relative to the specified cell. An initial state (time $t = 0$) is selected by assigning a state for each cell. A new generation is created (advancing t by 1), according to some fixed rule (generally, a mathematical function) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood. Typically, the rule for updating the state of cells is the same for each cell and does not change over time.

Rules ¶

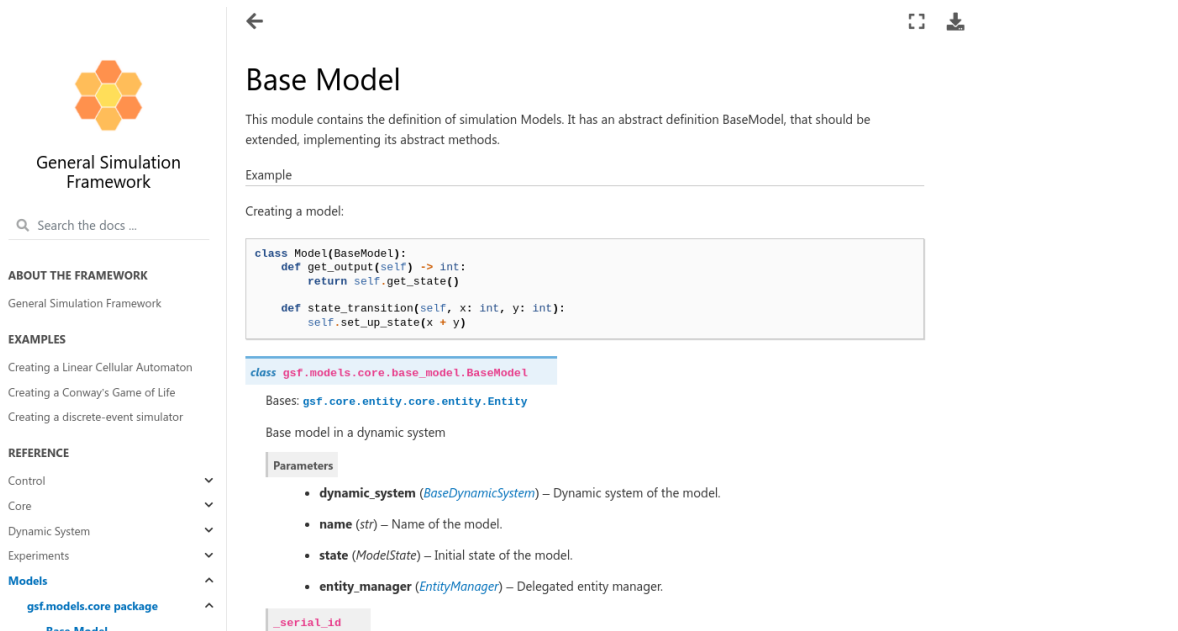
For this example, the rule for updating the state of the cells is:

For each cell of the automaton, it will take the state of its left neighboring cell.

With this rule, the cells will move one step to the right side every generation.

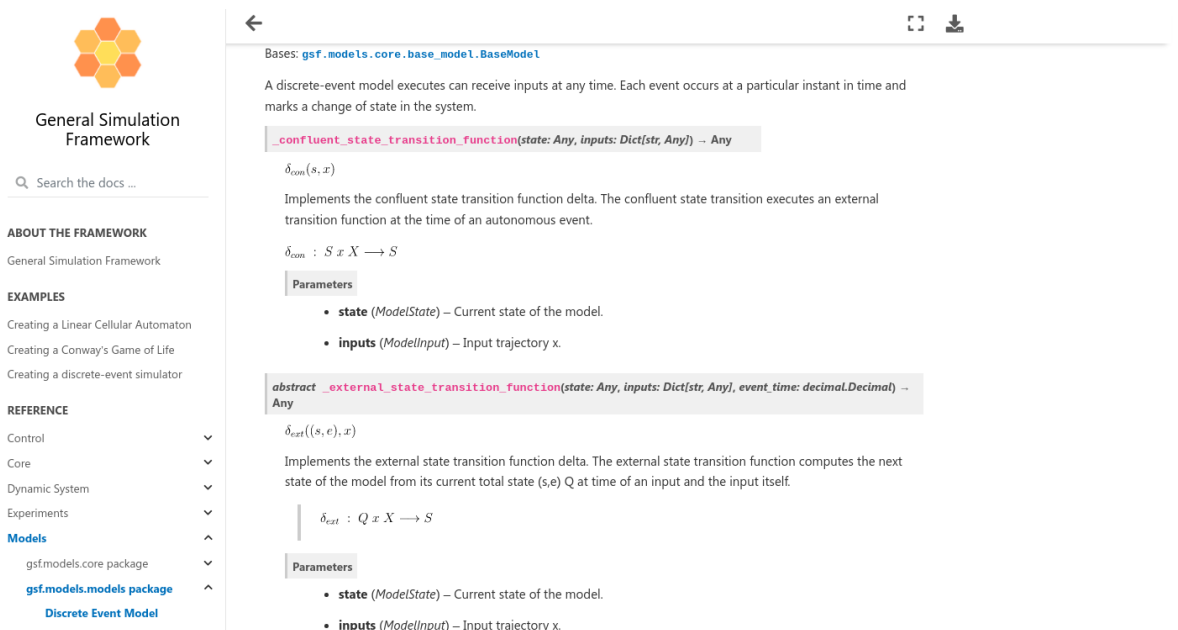
Implementing cells

Figura E.3: Página de ejemplo de implementación. De autoría propia.



The screenshot shows the documentation for the 'Base Model' module. On the left is a sidebar with the 'General Simulation Framework' logo and a search bar. The main content area is titled 'Base Model' and includes a description: 'This module contains the definition of simulation Models. It has an abstract definition BaseModel, that should be extended, implementing its abstract methods.' Below this is an 'Example' section showing a Python class definition for 'Model' that inherits from 'BaseModel'. The class has methods 'get_output' and 'state_transition'. A 'Parameters' section lists attributes like 'dynamic_system', 'name', 'state', and 'entity_manager'. The sidebar on the left has sections for 'ABOUT THE FRAMEWORK', 'EXAMPLES', and 'REFERENCE', with 'Models' expanded to show 'gsf.models.core package' and 'Base Model'.

Figura E.4: Página de referencia de un módulo. De autoría propia.



The screenshot shows the documentation for mathematical formulas. The sidebar is similar to the previous one, but the 'Models' section is expanded to show 'gsf.models.models package' and 'Discrete Event Model'. The main content area is titled 'Bases: gsf.models.core.base_model.BaseModel'. It describes a discrete-event model and provides two mathematical formulas. The first formula is for the confluent state transition function $\delta_{con}(s, x)$, which implements the confluent state transition function delta. The second formula is for the external state transition function $\delta_{ext}((s, e), x)$, which implements the external state transition function delta. Both formulas include a 'Parameters' section listing 'state' and 'inputs'.

Figura E.5: Fórmulas matemáticas de referencia. De autoría propia.

Anexo F

Imágenes del simulador

A continuación se presentan algunas imágenes del simulador desarrollado:

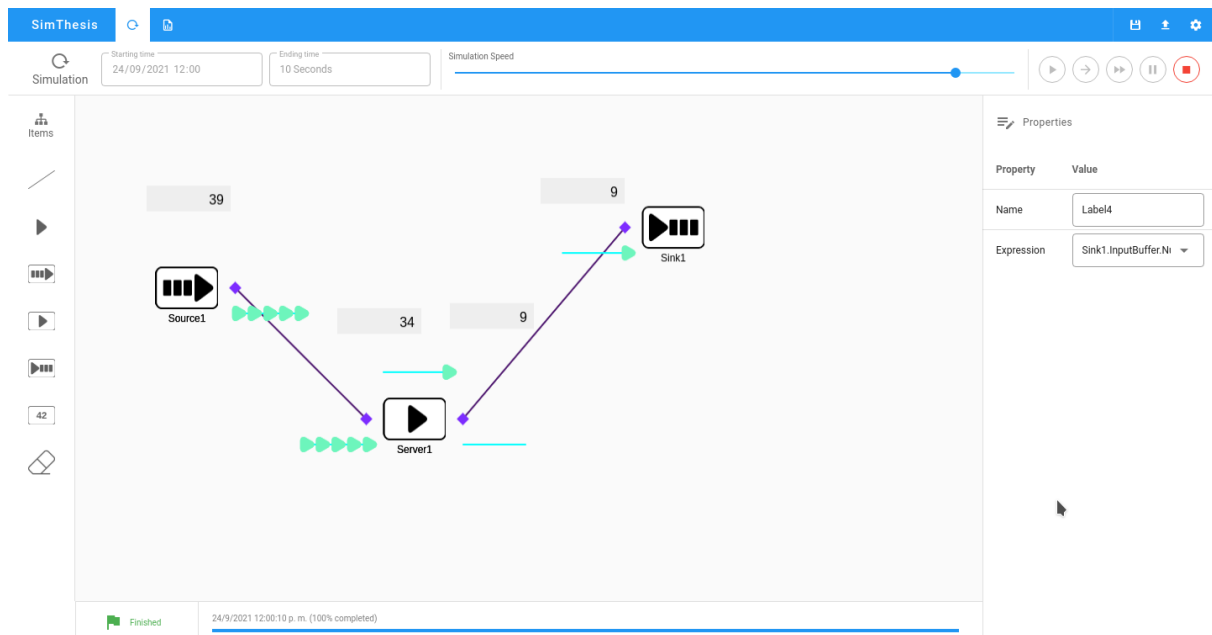


Figura F.1: Simulador luego de correr una simulación. De autoría propia.

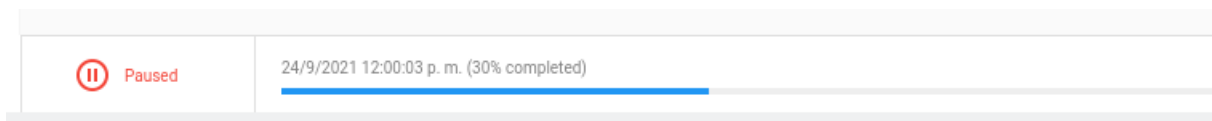


Figura F.2: Barra de estado de una simulación. De autoría propia.

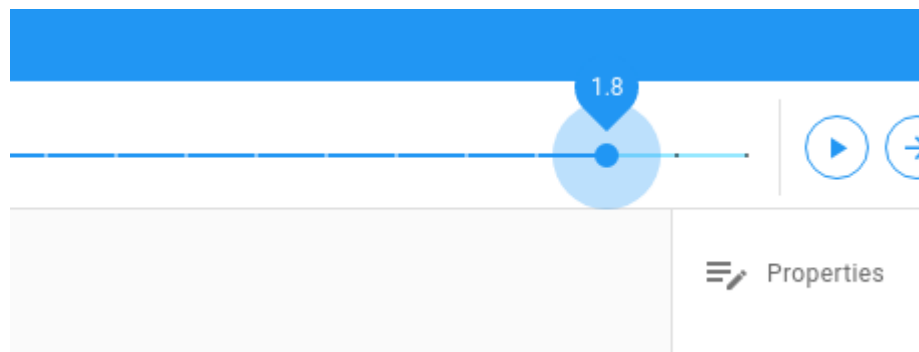


Figura F.3: Ajuste de velocidad de simulación. De autoría propia.

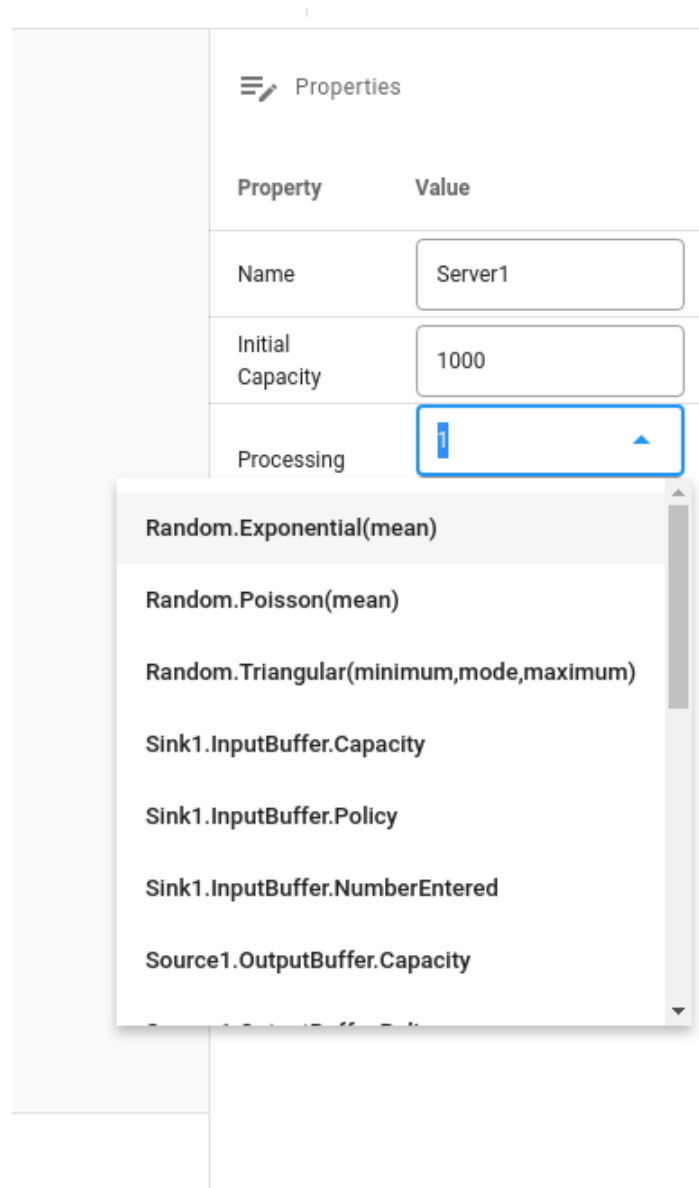


Figura F.4: Inspector de propiedades y expresiones. De autoría propia.

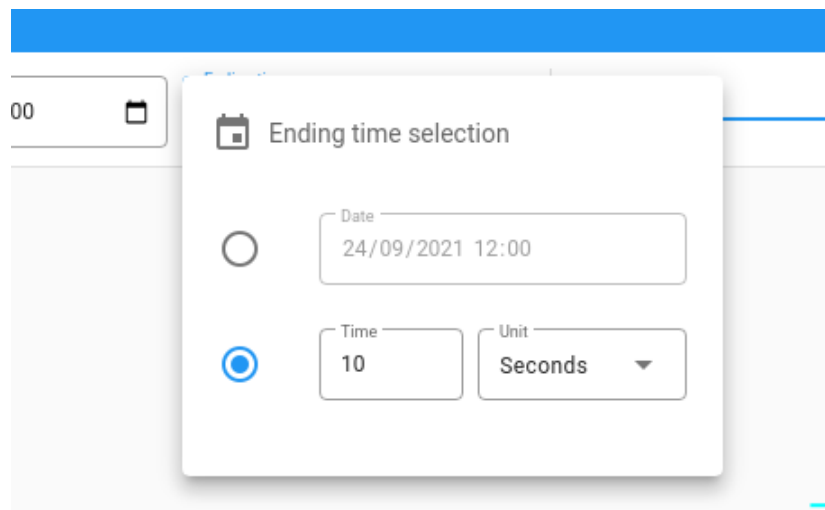


Figura F.5: Selector de tiempo de simulación. De autoría propia.



Figura F.6: Arrastre de componentes de simulación. De autoría propia.

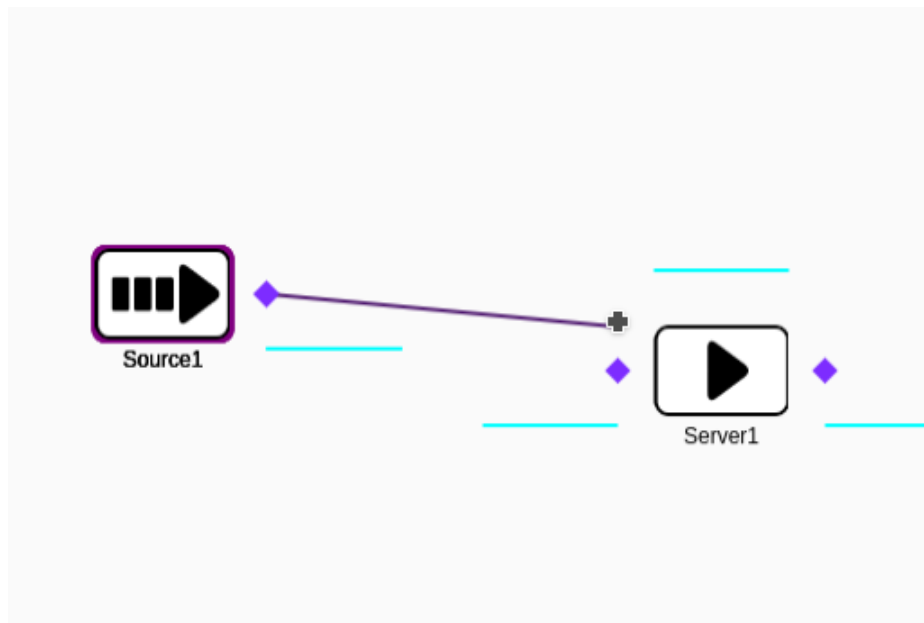


Figura F.7: Creación de rutas. De autoría propia.

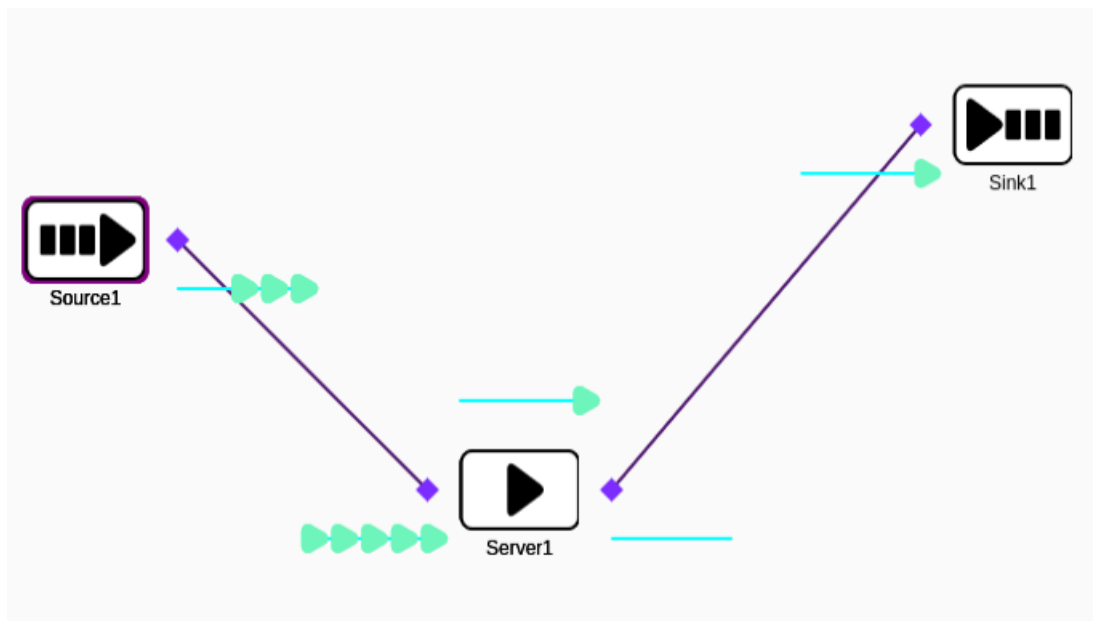


Figura F.8: Entidades en *buffer* en tiempo real. De autoría propia.

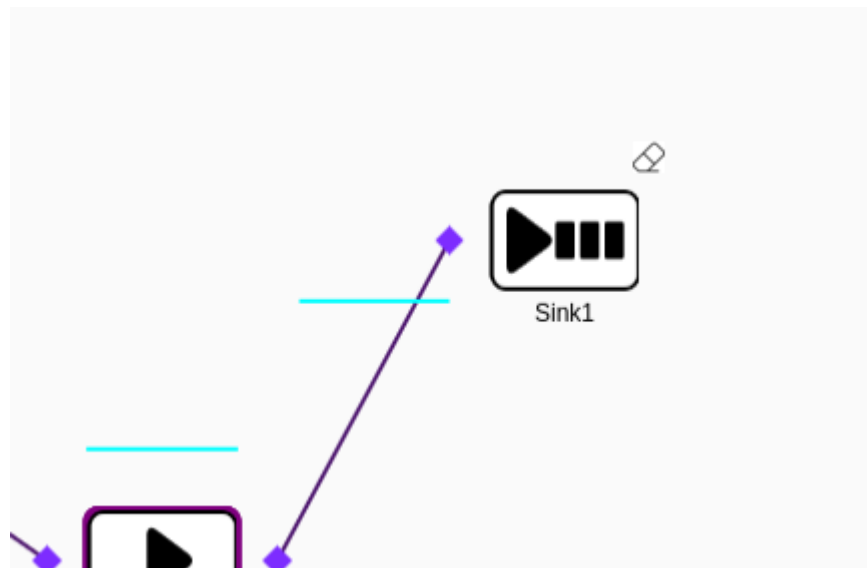


Figura F.9: Eliminación de componentes. De autoría propia.

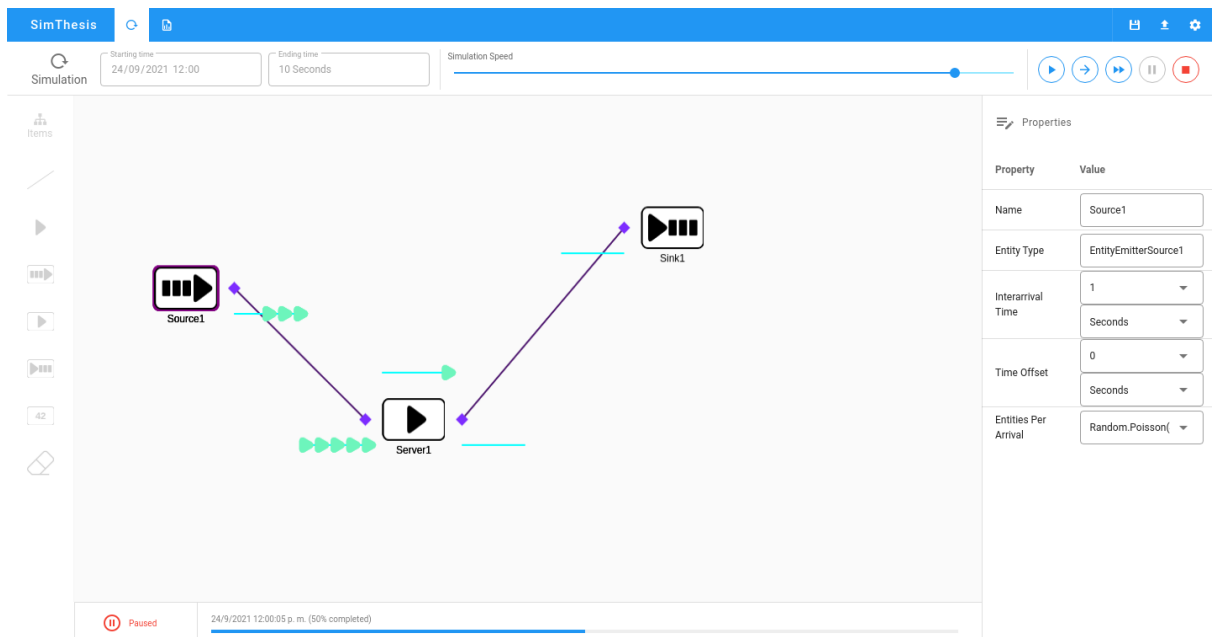


Figura F.10: Simulación paso por paso. De autoría propia.

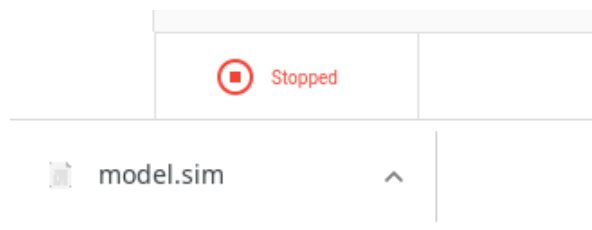


Figura F.11: Guardar el modelo diseñado. De autoría propia.

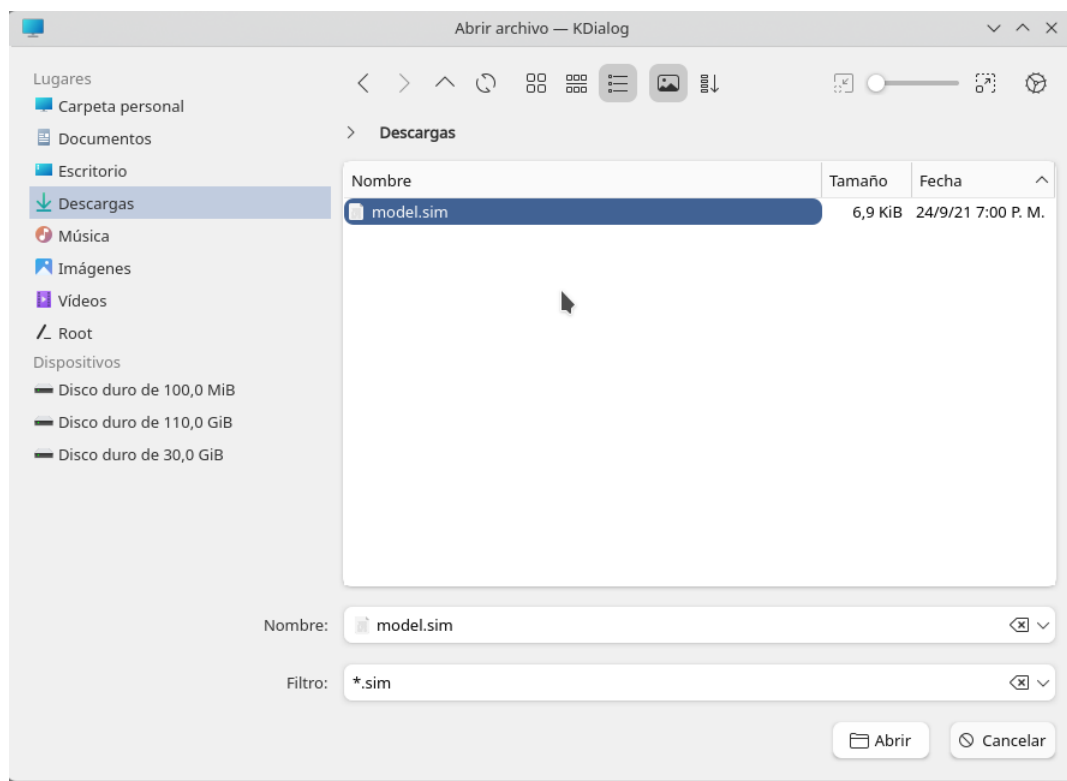


Figura F.12: Importar el modelo guardado. De autoría propia.

SimThesis					
Object Type	Object Name	Data Source	Data Item	Statistic	Value
Sink	Sink1	Sink1.InputBuffer	Entered	Total	9.000000
			NumberInStation	Maximum	1.000000
				Minimum	1.000000
				Average	1.000000
				Total	9.000000
Source	Source1	Source1.OutputBuffer	Entered	Total	41.000000
			NumberInStation	Maximum	11.000000
				Minimum	1.000000
				Average	3.727273
				Total	41.000000

Figura F.13: Estadísticas de simulación. De autoría propia.