UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

DESARROLLO DE UN PROGRAMA GENERADOR Y VISUALIZADOR DE OBJETOS DE TOPOLOGÍA COMBINATORIA PARA COMPUTACIÓN DISTRIBUIDA

T E S I S

QUE PARA OPTAR POR EL GRADO DE:

MAESTRO EN CIENCIAS (COMPUTACIÓN)

PRESENTA:

FAUSTO DAVID SALAZAR MORA

DIRECTOR DE TESIS:

DR. SERGIO RAJSBAUM GORODEZKY

INSTITUTO DE MATEMÁTICAS - UNAM

MÉXICO, D.F. FEBRERO 2016

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

A Dios por permitirme seguir en el camino, aprendiendo y disfrutando de todas las bendiciones que me rodean.
A mis padres María de Jesús Mora Sánchez y Ángel Salazar Cano por su amor, sabiduría, consuelo y todo el esfuerzo que han hecho para apoyarme en todas las etapas de mi vida.

A mi hermano Diego por siempre escucharme, aconsejarme y en especial por haberme animado a estudiar la maestría.

A todos mis amigos de la prepa, carrera, trabajo, maestría y clases de baile, por compartir su tiempo, energía y diversión conmigo, ayudándome a estar más animado en muchos de los momentos díficiles y tensos de los dos últimos años.

Agradecimientos

Agradezco a mi tutor, el Dr. Sergio Rajsbaum, por aceptar dirigir mi trabajo, lo cual hizo con gran calidad y profesionalismo; y en especial por haberme brindado la confianza y orientación para encontrar un tema de tesis que se adaptó más a mis intereses. Aunque díficiles, disfruté y aprendí mucho en sus clases, una de las cuales fue fundamental en la realización de este trabajo.

A mis sinodales la Maestra María Guadalupe Ibargüengoita y a los Doctores Armando Castañeda, David Flores y Luis B. Morales por el tiempo dedicado a la lectura y corrección de este trabajo.

Al Ing. Marío Rodríguez y a Humberto Del Ángel por haberme ayudado a revisar esta tesis así como también por sus sugerencias.

A todo el personal del Posgrado: Lulú, Amalia, Cecilia, Álvaro, el Dr. Jorge Ortega, todos los profesores con los que tomé clase y el personal de biblioteca por su trabajo y compromiso.

Al CONACYT por el apoyo económico que me permitió realizar estos estudios.

Y finalmente a la UNAM por haberme brindado la oportunidad de estudiar un posgrado de alta calidad y por los divertidos cursos extracurriculares que tomé.

Índice general

I Introducción **11**

1. Introducción **13**

1.1. Antecedentes 13

1.2. Planteamiento del problema 14

1.3. Contribuciones 16

1.3.1. Objetivos particulares 16

1.3.2. Áreas de conocimiento y habilidades aplicadas 17

1.4. Trabajos relacionados 18

1.5. Organización del trabajo 21

2. Marco teórico **23**

2.1. Sistema distribuido 23

2.2. Complejos simpliciales 25

2.3. Representando sistemas distribuidos con complejos simpliciales 27

2.4. Tareas 29

2.5. Modelos de cómputación distribuida 31

2.5.1. Memoria compartida 32

2.6. Complejo de protocolo 34

2.7. Conclusión 35

II Desarrollo del programa **37**

3. Especificación y análisis de requerimientos **39**

3.1. Requerimientos funcionales 39

3.2. Requerimientos no funcionales 41

3.3. Casos de uso 41

3.4. Conclusión 44

4. Diseño y arquitectura 45

4.1. Arquitectura del programa 45

4.1.1. Patrón arquitectónico 46

4.2. Diseño de las principales clases 48

4.2.1. Diseño de los procesos 51

4.2.2. Diseño de complejos simpliciales y simplejos 53

4.2.3. Diseño de los protocolos de comunicación 55

4.2.4. Diseño de objetos geométricos 58

4.3. Conclusión 59

5. Implementación 61

5.1. Inicio del programa 61

5.2. Construcción del complejo inicial 63

5.2.1. "Simplicial Complex Panel" 63

5.2.2. Secuencia de construcción del complejo inicial 70

5.3. Generación de complejos de protocolo 74

5.3.1. Implementación 75

5.3.2. Secuencia de generación de complejos de protocolo 78

5.4. Algoritmo generador de complejos de protocolo *immediate snapshot* 82

5.4.1. Historias 83

5.4.2. Generando las historias 84

5.4.3. Construcción del complejo de protocolo a partir de las historias generadas 91

5.4.4. Implementación del algoritmo generador de complejos de protocolo 93

5.5. Representación gráfica de complejos simpliciales 96

5.5.1. Actualización de las vistas mediante comandos 96

5.5.2. Transformación de un complejo simplicial a su representación geométrica 98

5.5.3. Cálculo de coordenadas 101

5.6. Información acerca de herramientas de desarrollo 105

5.7. Conclusiones 105

6. Conclusiones y trabajo futuro 107

6.1. Trabajo futuro 109

A. Generación de complejos simpliciales para otros modelos **113**

A.1. Modelo *read/write* (WR) 113

A.1.1. Generación de complejos de protocolo 116

B. Ejemplo de cómo se integra un nuevo modelo al programa **121**

B.1. Introducción 121

B.2. Prerequisitos 122

B.3. Descripción del ejemplo 122

B.3.1. Funcionamiento 123

B.3.2. Conclusión 126

C. Recomendaciones para el nuevo desarrollador **127**

Bibliografía 129

Resumen

En los últimos años el uso de técnicas y modelos de la topología combinatoria en el estudio de problemas de computación distribuida ha ayudado a obtener resultados importantes. Una de estas técnicas es la caracterización de todos los posibles estados en los que se puede encontrar un sistema distribuido como una estructura combinatoria conocida como complejo simplicial, lo cual facilita el razonamiento acerca de las propiedades del sistema. Los complejos simpliciales se pueden visualizar como construcciones de objetos geométricos tales como vértices, aristas, triángulos, poliedros, etcétera. Los sistemas distribuidos pueden tener muchas características y representarlos de forma combinatoria lo largo de varias ejecuciones puede ser complicado.

Ante este problema se plantea la creación de un programa cuyo propósito es permitir a los investigadores y estudiantes producir automáticamente visualizaciones de complejos simpliciales para representar diversos modelos de computación distribuida. Se desarrolla la primera versión de este programa, en la cual se ofrece la capacidad de generar visualizaciones de complejos simpliciales para modelos de sistemas distribuidos en los que participan a los más tres procesos que se comunican a través de una memoria compartida de lectura y escritura tipo *atomic immediate snapshot*, no se asumen fallas y los complejos producidos pueden ser cromáticos o no cromáticos.

Además de servir para fines de estudio e investigación, también se pretende que las visualizaciones producidas por este programa puedan ser usadas en publicaciones[1], por lo tanto éste ofrece herramientas para la personalización de algunas propiedades de la visualización como el color y tamaño de vértices, etiquetas, aristas y caras; y controles interactivos para rotar, realizar acercamientos y alejamientos, acomodar manualmente vértices, etcétera.

El programa se diseñó de forma modular y extensible, con el fin de que sea fácil integrar en futuras versiones nuevas características tales como la generación y visualización de complejos para otros modelos de computación distribuida.

El programa es gratuito y se puede descargar desde la siguiente dirección:

[1] De hecho, muchas de las figuras usadas en esta tesis fueron producidas por el programa.

http://www.mcc.unam.mx/~salazar.f/.

Así mismo, su código es libre y se encuentra disponible en

https://github.com/fdsmora/DCCT.

Parte I

Introducción

Capítulo 1

Introducción

1.1. Antecedentes

El crecimiento de Internet durante las últimas decadas ha traido consigo el desarrollo de nuevas tecnologías tales como la computación en la nube, en la que miles de computadores distribuidas por todo el mundo se comunican a través de Internet para ofrecer todo tipo de servicios a las corporaciones y a la población en general. En menor escala, actualmente los procesadores de las computadores personales y servidores contienen múltiples núcleos de procesamiento que se comunican a través de una memoria compartida para resolver tareas comunes.

Sea cual sea el alcance de estas tecnologías, tienen en común que están conformadas por múltiples dispositivos computacionales que se tienen que comunicar y cooperar eficientemente para resolver problemas, aun a pesar de sus diferentes capacidades de cómputo, su susceptibilidad a fallar y la fiabilidad de los medios que usan para comunicarse. Todos estos factores generan una incertidumbre que influye en la naturaleza de la computación distribuida, ya que ninguno de estos dispositivos puede saber con certeza el estado global del sistema del que forma parte. A pesar de esto, se tienen que entender y diseñar algoritmos distribuidos eficientes, correctos y tolerantes a fallas, tarea que puede volverse extremadamente compleja [27].

En los últimos años han surgido nuevas e interesantes técnicas para analizar algoritmos distribuidos. Mientras que en los sistemas secuenciales la computabilidad se entiende a través de la tesis Church-Turing, en los sistemas distribuidos, cuyos componentes pueden fallar y el cómputo requiere de coordinación entre participantes, los aspectos de computabilidad tienen una faceta distinta. Aquí también hay muchos problemas que no son computables; no obstante, estos límites a la computabilidad reflejan la dificultad de tomar decisiones frente a la imposibilidad de conocer el estado global del sistema y en realidad tienen muy poco que ver con la capaci-

dad computacional inherente de cada uno de los participantes (que son procesos secuenciales). Más aún, los argumentos de imposibilidad de computación distribuida tienen que ver con argumentos geométricos de tipo topológico, en contraste con los argumentos de imposibilidad de computación secuencial, que tienen que ver con argumentos de conteo y diagonalización [21].

La topología es una rama de las matemáticas que estudia las propiedades de los objetos que se preservan bajo deformaciones continuas como estiramientos y doblamientos pero no bajo operaciones discontinuas como romper los objetos o pegar las partes que ya estaban separadas. La topología combinatoria es la rama de la topología que se enfoca en el estudio de construcciones discretas, como por ejemplo, una esfera construida con triángulos. Este tipo de objetos se conocen como *complejos simpliciales*.

El enfoque de estudiar los fundamentos teóricos de la computación distribuida a través de topología combinatoria es una herramienta muy poderosa ya que nos ayuda a comprender las propiedades esenciales de sistemas distribuidos que aparentemente son muy diferentes. Este enfoque muestra que esencialmente la computación distribuida es una manera de estirar un objeto geométrico para hacerlo encajar en otro según lo especifique un determinado problema.

En [22] se da la primera caracterización basada en topología combinatoria de un problema de computación distribuida. En [19] se estudian a fondo los fundamentos teóricos de la concurrencia mediante topología combinatoria.

1.2. Planteamiento del problema

De manera general e intuitiva, los complejos simpliciales son construcciones de objetos llamados *simplejos* y que satisfacen ciertas propiedades. Los simplejos son objetos que se pueden ver de diferentes maneras de acuerdo a su dimensión: un simplejo de dimensión cero es un vértice; de dimensión uno es una arista; de dimensión dos es un triángulo; de dimensión tres es un tetraedro; y así sucesivamente. Tanto los simplejos como los complejos simpliciales se pueden representar de forma geométrica en un espacio euclidiano.

En computación distribuida, un complejo simplicial nos sirve para representar todos los posibles estados en los que se puede encontrar un sistema distribuido en algún momento determinado. Estos estados globales se conforman por los estados locales de cada *proceso* que forma parte del sistema distribuido. Cada estado local refleja la incertidumbre que tiene cada proceso acerca del estado global del sistema.

Los sistemas distribuidos pueden tener muchas características como lo son el tipo de comunicación (e.g., paso de mensajes o memoria compartida), cantidad y tipos de fallas que se pueden

presentar, clases de algoritmos distribuidos que ejecutan, número de veces que se comunican los procesos, etcétera. Modelos teóricos de computación distribuida tan variados a veces tienen representaciones en complejos simpliciales con estructuras muy complicadas, lo cual dificulta su visualización y estudio. Un ejemplo de esto son los complejos simpliciales que representan las ejecuciones de un algoritmo distribuido en el que participan tres procesos que se comunican mediante objetos llamados Test&Set; esto con el fin de resolver un problema conocido como (N, k)-*consensus task* [20]. En la Figura 1.1 observamos como a lo largo de diferentes rondas de comunicación entre los procesos, los complejos simpliciales se van haciendo más complicados ya que cada vez contienen más subdivisiones y agujeros.

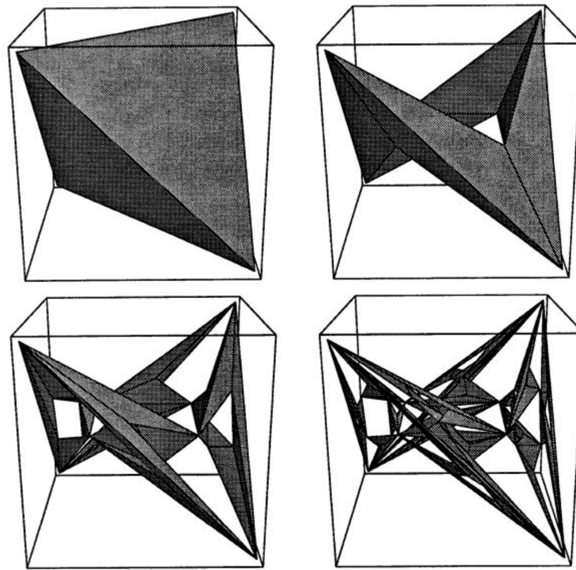

Figura 1.1: Complejos simpliciales de multiples rondas de comunicación entre procesos usando variables Test&Set. *Fuente:* [20]

El interés por estudiar y visualizar los complejos simpliciales derivados de un modelo de computación distribuida en específico surge de que las propiedades de estos determinan las tareas que se pueden resolver en ese modelo, así como también el costo computacional de resolver estas tareas [19].

Es común que los estudiantes e investigadores en el área dibujen los complejos simpliciales a mano o utilizando algún programa con capacidades de dibujo. Un claro ejemplo de esto es el hecho de que la mayor parte de las ilustraciones de complejos simpliciales en [19] se realizaron en Power Point®, lo cual fue una labor ardua, en palabras de uno de los autores.

Por otro lado existe la necesidad por parte del investigador y del estudiante de experimentar con nuevas ideas, modificando o agregando nuevas características a los modelos de sistemas dis-

tribuidos que estudian y así examinar cómo cambia la estructura de sus complejos simpliciales. Sin embargo, imaginar y dibujar un complejo simplicial puede ser tardado, díficil y propenso a errores, más aún si éste contiene muchas subdivisiones, vértices, aristas, etcétera. Aun la simple tarea de cambiar el nombre a algún proceso y reetiquetar todos los vértices correspondientes puede ser tediosa al realizarse a mano o incluso con Power Point®.

A diferencia de otras áreas de investigación científica [18], los investigadores y estudiantes que estudian el cómputo distribuido desde un enfoque topológico todavía no cuentan con un programa de visualización científica o matemática que permita analizar mediante simple observación los problemas que se estudian.

La visualización matemática ha demostrado ser una herramienta eficaz para analizar fenómenos matemáticos complejos y ha permitido obtener pistas importantes que han llevado a demostraciones rigurosas de problemas importantes [29].

1.3. Contribuciones

Se propone el diseño e implementación de un programa que permita computar y generar automáticamente visualizaciones geométricas de complejos simpliciales, los cuales representan las posibles ejecuciones que puede tener un sistema distribuido. Las visualizaciones que genera este programa deberían ser aptas para ser publicadas en artículos de investigación. La generación automática de visualizaciones también debería ayudarle al investigador o estudiante a experimentar y a comprender cómo se comportan los modelos de computación distribuida. El programa debe estar diseñado y documentado de forma tal que sea fácil extenderlo con nuevas mejoras y funcionalidades.

1.3.1. Objetivos particulares

Se pretende que la primera versión del programa tenga las siguientes características.

- Generación de visualizaciones de complejos simpliciales, tanto iniciales como de protocolo; estos últimos basados en el modelo *immediate snapshot* para uno, dos o tres procesos.

- Generación de visualizaciones de complejos simpliciales de protocolo en sus representaciones cromática y no cromática.

- Despliegue de información en texto de los complejos simpliciales generados, tal como representación del complejo simplicial en notación de conjuntos, tipo de complejo y número

vértices y caras que lo conforman.

- Controles de interacción con los complejos simpliciales generados. Estas interacciones incluyen rotaciones, acercamientos, desplazamientos y acomodo manual de vértices.
- Controles para personalizar el color y tamaño de las etiquetas, vértices, aristas y caras que conforman los complejos simpliciales generados.
- Acceso al programa mediante una página web, minimizando el esfuerzo de instalación y configuración.
- Funcionamiento del programa en Windows, Mac OS X y Linux.
- Código del programa libre y bien documentado para facilitar la implementación de nuevas funcionalidades y mejoras.

1.3.2. Áreas de conocimiento y habilidades aplicadas

Para el desarrollo de este programa se requirió tener conocimientos y habilidades en las áreas de computación distribuida, topología, algoritmos combinatorios, geometría, programación e ingeniería de software.

Para poder generar los complejos simpliciales derivados de un modelo de computación distribuida primero hay que entender la definición matemática de un complejo simplicial y cómo éste se relaciona con aquél. Después, de acuerdo a su especificación, hay que entender con detalle el modelo en términos de los mecanismos de comunicación que usan los procesos y las posibles maneras en que éste puede fallar. Una vez comprendido, se generan todos los posibles estados en los que pueden terminar los procesos después de una ejecutar una ronda de comunicación, considerando las características del modelo.

Para generarlos de forma automática, se implementaron y adaptaron algunos algoritmos combinatorios y se procedió a transformar el conjunto de posibles estados generados en un complejo simplicial, el cual se construye geométricamente en términos de vértices, aristas y caras.

Para definir, diseñar y construir el programa se llevó a cabo un proceso de definición de requierimientos, diseño de componentes e implementación de la solución, fases durante las cuales se usaron algunas herramientas de la ingeniería de software entre las cuales están la especificación de requerimientos funcionales y no funcionales; descripción de casos de uso; diagramas UML y

de secuencia; patrones de diseño de software y el uso de un sistema controlador de versiones (Git).

1.4. Trabajos relacionados

No se encontró que existan productos de software que generen visualizaciones de complejos simpliciales usados en computación distribuida, salvo algunos pequeños programas realizados por estudiantes en cursos de sistemas distribuidos. Es por esto que se decidió realizar este trabajo.

Para el desarrollo de nuestro programa se decidió buscar algún otro programa o biblioteca de software que pudiera ser fácilmente extendido para producir visualizaciones de complejos simpliciales para computación distribuida.

Durante tres meses se buscaron y probaron diversas opciones. Como primera alternativa se consideró a *Mathematica*, sin embargo se descartó ya se consideró demasiado complejo para usarlo en este trabajo.

Otra razón que incitó a buscar otras alternativas es que al inicio se contempló que el programa solamente produciría visualizaciones de complejos simpliciales que representan sistemas en los que participan sólo dos procesos. Éstos se representan geométricamente como *gráficas*, es decir, construcciones de vértices conectados por aristas. Para esto existen muchos paquetes de software libre simples y especializados en visualización de redes y teoría de gráficas. Se encontraron algunos como *Graphviz* [2], que ofrece funciones de propósito general de dibujo de gráficas; *Gephi* [1], una plataforma para la exploración y visualización de todo tipo de gráficas; *igraph*, una colección de herramientas para análisis de gráficas; y el paquete de teoría de gráficas de *SageMath* [7]. Todos estos, salvo Graphviz, se investigaron a fondo y en algunos casos se desarrollaron prototipos para probar sus características. A continuación los describimos.

- **SageMath.** Es un sistema de software libre para matemáticas. Se pueden desarrollar aplicaciones con el lenguaje de programación Python. La idea era utilizar sus capacidades de visualización para teoría de gráficas, sin embargo, no se encontró documentación acerca de como se pueden controlar las posiciones de las etiquetas y de los vértices en el espacio, por lo tanto se descartó este sistema. En la Figura 1.2 mostramos algunas visualizaciones de los complejos simpliciales que se produjeron con el prototipo que se desarrolló.

- **igraph.** [4] Es una colección de herramientas de análisis de gráficas que enfatiza eficiencia, portabilidad y facilidad de uso, además de ser software libre y poderse programar

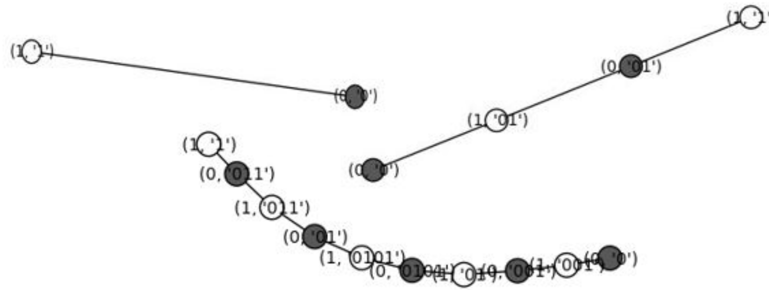

Figura 1.2: Complejo inicial y complejos de protocolo en dos rondas para dos procesos generados con SageMath. Las etiquetas muestran las vistas locales de los procesos.

con R, Python y C/C++. También se desarrolló un prototipo para probar sus características. Los complejos simpliciales que generó el prototipo se muestran en la Figura 1.3. Al contrario de SageMath, este software ofrece mayor control sobre la personalización de las visualizaciones, por ejemplo, se puede controlar las posiciones de los vértices y las etiquetas.

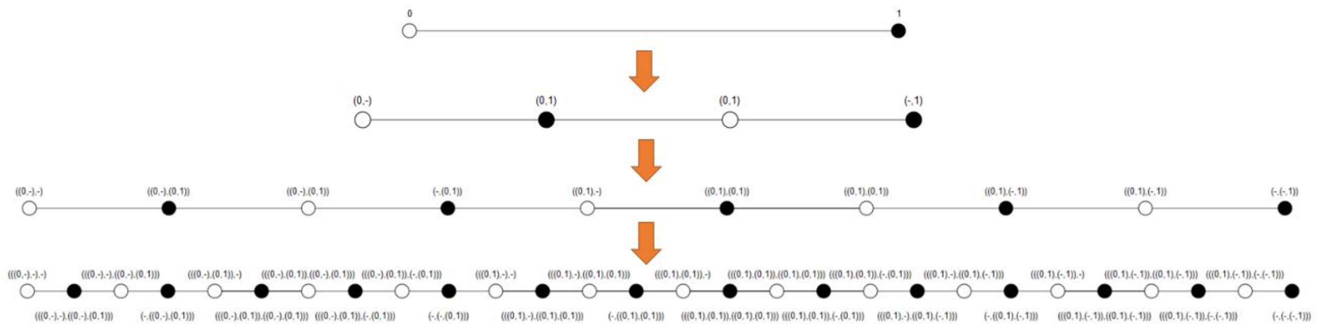

Figura 1.3: Complejo inicial y complejos de protocolo en tres rondas para dos procesos generados con igraph. Las etiquetas muestran las vistas locales de los procesos.

- **Gephi.** Es una plataforma de visualización y exploración de todo tipo de gráficas, tales como redes y sistemas complejos y jerárquicos. Es software libre, se puede programar en Java y Python, se puede extender mediante *complementos* [1], genera visualizaciones muy bellas de hasta 50,000 nodos, y provee algoritmos de dibujo de gráficas eficientes. No se desarrolló ningún prototipo para probar sus características pues se tomó la decisión de buscar otro software que permitiera manejar objetos de mayor dimensión que las gráficas.

A pesar de que no se desarrolló un prototipo para probar sus características, la mejor op-

[1] En inglés, *plugins*.

ción fue Gephi, pues fue la más completa y avanzada entre las que tres que describimos. Sin embargo, después se decidió que también el programa debería ser capaz de generar complejos simpliciales con tres procesos, los cuales se representan como triángulos. Entonces se efectuó una nueva búsqueda de bibliotecas de software que pudieran producir visualizaciones de polígonos y poliedros. Se encontraron los siguientes programas.

- **Polymake.** [17] Es una herramienta para estudiar la combinatoria y la geometría de politopos convexos y poliedros, además de ser capaz de trabajar con complejos simpliciales, matroides, y demás objetos geométricos; permite computar propiedades geométricas, combinatorias, cierres convexos, etcétera; y generar visualizaciones en 3D. Es un programa muy completo y además permite estudiar el aspecto combinatorio de complejos simpliciales. Sin embargo se consideró muy complejo para fines de este trabajo.

- **JavaView.** [5] Gracias a la investigación que se realizó del programa Polymake, se encontró que una de sus versiones pasadas usa esta biblioteca parar producir visualizaciones en 3D. Los programas que se desarrollan usando esta biblioteca se pueden implementar como *applets* de Java, por lo cual se pueden incrustar fácilmente en páginas web. Como permite la visualización y manipulación de caras para construir poliedros, es posible crear complejos simpliciales para tres o más procesos. No cuenta con mucha documentación o ejemplos. Se había decidido usar esta biblioteca para el desarrollo del programa, sin embargo poco después se encontró que la versión más reciente del programa Polymake utiliza otra biblioteca más moderna, que es la que a continuación describimos.

- **jReality.** [6] Es un paquete de visualización matemática en 3D, interactivo, de código abierto, desarrollado en Java; permite extenderse mediante el desarrollo de complementos, cuenta con un foro activo de usuarios y un amplio conjunto de tutoriales y ejemplos. La calidad de las visualizaciones que genera es superior a las de JavaView ya que usa una biblioteca basada en OpenGL llamada *jogl* [2]. Las aplicaciones desarrolladas no se distribuyen mediante *applets*, sino con una tecnología más reciente llamada Java Web Start [3].

[2] `http://jogamp.org/jogl/www/`

[3] Esta permite a los usuarios iniciar una aplicación para la plataforma Java directamente desde Internet usando un navegador web. Al contrario de los *applets*, estas aplicaciones no se ejecutan incrustadas en el navegador del usuario, sino independientemente en su propia ventana. Para iniciar una aplicación de este tipo, el usuario descarga de Internet un archivo JNLP (que es un archivo basado en XML), el cual al ser ejecutado, descarga y ejecuta la aplicación.

- **Gratrix WebGL Uniform Polyhedra.** [**3**] Es un programa que permite examinar e interactuar con una colección de diferentes tipos de poliedros; las visualizaciones son en 3D y de muy alta calidad y está desarrollada en JavaScript y WebGL [4] y se carga directamente en el navegador del usuario. Su código también está disponible para ser estudiado y modificado [5].

La elección final fue jReality.[6]. A pesar de que no se ejecuta como un *applet* inscrustado en el navegador web (lo cual puede resultarle menos amigable y de más díficil acceso a los usuarios que no esten acostumbrados a la tecnología Java Web Start), fue más importante la mayor cantidad disponible de documentación y recursos de ayuda en caso de que se tuvieran problemas durante el desarrollo, además de la calidad superior de las visualizaciones que produce.

1.5. Organización del trabajo

La parte I comprende el presente capítulo y el Capítulo 2, "Marco teórico".

En el Capítulo 2 hacemos una revisión de los conceptos y definiciones de computación distribuida a través de topología combinatoria relacionados con las funcionalidades que ofrece esta versión del programa.

La parte II comprende el resto de los capítulos, en los que se discute el desarrollo del programa.

En el Capítulo 3, "Especificación y análisis de requerimientos", damos los requerimientos funcionales y no funcionales que esta versión del programa satisface y también los casos de uso que describen las formas en las que se puede interactuar con el programa.

En el Capítulo 4, "Diseño y arquitectura", discutimos la arquitectura y el diseño de los componentes principales del programa, así como las razones por las cuales éste logra ser extensible y modular.

En el Capítulo 5, "Implementación", describimos cómo implementamos los componentes descritos en el capítulo anterior y también los principales problemas que se tuvieron que resolver

[4]Es una API de JavaScript que sirve para dibujar gráficos 3D y 2D interactivos y es compatible con los navegadores web más populares, sin requerir el uso de complementos.

[5]Se puede ver el código de la aplicación al abrir el código fuente de la página en cualquier navegador.

[6]A pesar de esta elección, considero que quizás la mejor alternativa hubiese sido el programa Gratrix Uniform Polyhedra, ya que se ejecuta directamente en el navegador del usuario y no presenta problemas de bloqueos por permisos de seguridad ni alto consumo de memoria, los cuales se descubrieron ya avanzado el desarrollo de nuestro programa; desafortunadamente esta opción se descubrió cuando el programa ya estaba casi terminado. Sugiero considerar esta opción si se decide en el futuro crear una nueva versión del programa desde cero.

durante esta fase.

En el Capítulo 6, "Conclusiones y trabajo futuro" discutimos las ideas surgidas a partir de la experiencia de haber desarrollado este trabajo, las cuales consideramos serán valiosas para el desarrollo de futuras versiones del programa.

En el Apéndice A describimos cómo se logró extender el programa con la capacidad de generar visualizaciones de complejos de protocolo basadas el modelo de computación distribuida conocido como *read/write* (WR) [9].

En el Apéndice B se describé un breve ejemplo acerca de cómo se puede extender el programa fácilmente con nuevos modelos de computación distribuida.

Finalmente, en el Apéndice C se da una serie de recomendaciones para personas que deseen continuar extendiendo y mejorando este programa.

Capítulo 2

Marco teórico

Para comprender el propósito y las funcionalidades del programa, es necesario primero hacer una revisión de los conceptos de computación distribuida y topología combinatoria que éste reproduce. Es importante hacer esta revisión para entender mejor su construcción, ya que muchos elementos de su estructura, (e.g., clases, métodos, variables, etc.) modelan varios de los conceptos y operaciones mencionados en este capítulo.

Los conceptos y definiciones presentados en este capítulo son tomados de [19] y [21].

2.1. Sistema distribuido

Un sistema distribuido es una colección de $n+1$ *procesos* que se denotan por $p_0, ...p_n$; y un medio de comunicación que puede ser una memoria compartida o una red mediante la cual los procesos se envían mensajes. Un *proceso* es una entidad secuencial, la cual se modela formalmente como una máquina de estados.

Cada proceso p_i se identifica por un *id* [1] y también posee una *vista local* v_i que representa el estado actual del sistema y el estado interno del proceso. Cada proceso tiene incertidumbre acerca de las vistas de los demás procesos, por ejemplo, p_i no sabe si algún otro proceso ya ha recibido un mensaje o si un valor escrito en memoria compartida ya fue leído por otro proceso. Un ejemplo de esto es el complejo de la Figura 2.1, el cual representa el estado inicial de un sistema distribuido en el que participan tres procesos: p_0 (verde), p_1 (blanco) y p_2 (rojo). Las etiquetas en cada vértice muestran las vistas de cada proceso, en donde cada uno solamente conoce su *id* pero no conoce el de los otros dos, por lo tanto se utiliza el simbolo “-” para representar esta incertidumbre.

[1] En este trabajo asumimos que para cualquier proceso p_i, $id = i$.

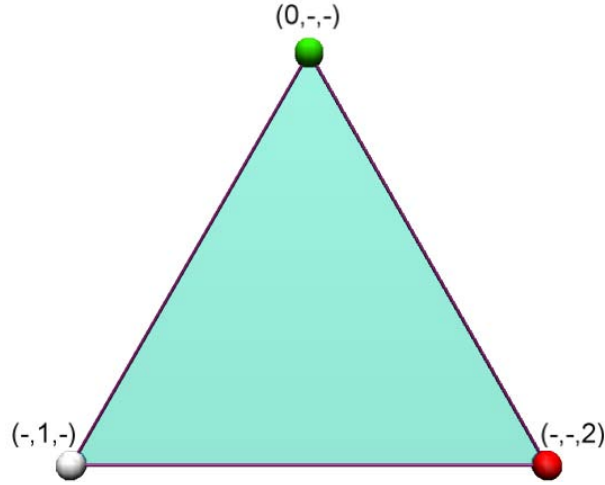

Figura 2.1: Complejo simplicial inicial que muestra la incertidumbre que tiene cada proceso acerca del conocimiento de los demás.

Durante este trabajo en algunas ocasiones usarémos la notación ${}_rv_i$ para denotar la vista del proceso p_i al final de una r-ésima ronda de comunicación entre los procesos del sistema. Cuando no sea necesario resaltar el estado de una vista al final de alguna ronda de comunicación, simplemente nos referiremos a ésta como v_i. El valor de la vista ${}_rv_i$ de un proceso p_i al final de una r-ésima ronda de comunicación está dado por la siguiente relación:

$$
{}_rv_i = \begin{cases} i & \text{si } r = 0 \\ [w_0, w_1, ..., w_n] & \text{si } r > 0 \end{cases}
$$

$r = 0$ denota el estado inicial del sistema antes de que se ejecute la primera ronda de comunicación e i es igual al *id* del proceso. A partir de la primera ronda de comunicación, es decir, cuando $r > 0$, la vista es un arreglo[2] de $n + 1$ entradas, en donde:

$$
w_j = \begin{cases} {}_{r-1}v_j & \text{Si la vista } v_j \text{ de } p_j \text{ fue comunicada en la ronda } r-1. \\ - & \text{Si la vista del proceso } p_j \text{ es desconocida.} \end{cases}
$$

Como ejemplo, en el complejo simplicial de la ronda 1 mostrado en la Figura 2.4 se muestran las vistas de los procesos después de comunicarse una vez mediante el protocolo *immediate snapshot*.

Cada proceso ejecuta un *protocolo* finito, el cual es un algoritmo distribuido que define cómo y cuando se comunican los procesos. Un protocolo puede ser *libre de espera*: cada proceso sin fallas eventualmente termina de ejecutarse. También puede ser de *información completa*: cada

[2] En algunas ocasiones usaremos llaves o paréntesis en lugar de corchetes para representar el arreglo.

proceso comunica su vista entera actual en cada mensaje. La ejecución del protocolo hace que las vistas locales *evolucionen*, es decir, los procesos van adquiriendo un conocimiento cada vez más completo del estado global del sistema.

Cada proceso comienza con una vista inicial y ejecuta instrucciones definidas en el protocolo hasta que *falla*, lo que significa que se detiene y no ejecuta instrucciones adicionales; o bien, se detiene porque ya completó el protocolo. Las fallas pueden ser por *paro*: hacen que el proceso se detenga y no ejecute más pasos. Si un proceso falla por paro antes de ejectuar algún paso, decimos que no participa en la ejecución.

El objetivo de un sistema distribuido es resolver una *tarea*. Una tarea en computación distribuida es lo análogo a una función en computación secuencial y es su unidad básica (Figura 2.2). Una vez que los procesos sin falla terminan de ejecutar el protocolo, deben de elegir un valor de salida.

La especificación de la tarea determina qué valores de salida están permitidos para qué estados iniciales de los procesos. Por supuesto que cada proceso inicialmente no conoce el estado inicial de los demás, es por esto tienen que ejecutar el protocolo para comunicarse y así intentar reducir la incertidumbre acerca del estado global del sistema. Con esta información cada vez más completa, los procesos pueden resolver la tarea. Sin embargo, la incertidumbre es el principal problema en un sistema distribuido: si los procesos comunicarán siempre todo lo que saben, la comunicación fuera instantánea y sin fallas, estos terminarían con vistas locales idénticas y resolverían trivialmente la tarea. Pero usulamente este no es el caso en los sistemas distribuidos reales.

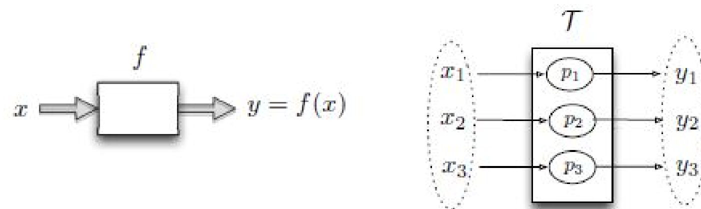

Figura 2.2: Función f vs. tarea T para tres procesos p_1, p_2, p_3. *Fuente:* [21]

2.2. Complejos simpliciales

Un sistema distribuido puede contener un enorme y complejo conjunto de posibles ejecuciones. Bajo un enfoque de topología combinatoria, estas posibles ejecuciones se pueden describir separandolas en piezas discretas conocidas como *simplejos*. La estructura de esta descompo-

sición, esto es, como encajan los simplejos, está dada por una estructura llamada *complejo*. Como su nombre lo indica, un complejo puede ser muy complicado y para entender las propiedades subyacentes más esenciales se utilizan las herramientas conceptuales de la topología combinatoria.

Hay tres formas distintas de entender los complejos simpliciales: combinatoria, geométrica y topológica. Para fines de este trabajo solo es necesario revisar algunos conceptos del aspecto combinatorio.

Definición 1. Dado un conjunto S y una familia A de subconjuntos de S, decimos que A es un *complejo simplicial abstracto* en S si las siguientes condiciones se satisfacen:

(1) Si $X \in A$ y $Y \subseteq X$, entonces $Y \in A$;

(2) $\{v\} \in A$ para toda $v \in S$.

Un elemento de S se conoce como *vértice*, y un elemento de A se conoce como *simplejo*. El conjunto de vértices en A se denota por $V(A)$. Se dice que un simplejo $\sigma \in A$ tiene *dimensión* $|\sigma| - 1$. En particular, los vértices son simplejos de dimensión 0. Un simplejo de dimensión n es algunas veces llamado $n-$simplejo. Geométricamente, un simplejo de dimensión uno se puede ver como una arista, de dimensión dos como un triángulo y de dimensión tres como un tetraedro. Un complejo simplicial que solo contiene vértices y aristas es una *gráfica*. Por brevedad, algunas veces decimos *complejo* en lugar de *complejo simplicial*.

Usamos letras minúsculas para referirnos a los vértices (*x, y, z,...*), minúsculas del alfabeto griego para denotar simplejos (σ, τ,...) y mayúsculas caligráficas para denotar complejos simpliciales ($\mathcal{A}$, $\mathcal{B}$,...).

Un simplejo τ es una *cara* de σ si $\tau \subseteq \sigma$, y es una *cara propia* si $\tau \subset \sigma$. Si τ es de dimensión d, entonces τ es una $d-$cara de σ. Claramente, las $0-$caras de σ y los vértices de σ son los mismos objetos, entonces, para $v \in S$, podríamos escribir $\{v\} \subseteq \sigma$ o $v \in \sigma$, dependiendo qué aspecto de la relación entre v y σ deseamos enfatizar. Un simplejo σ en un complejo $\mathcal{A}$ es una *faceta* si no es una cara propia de ningún otro simplejo en $\mathcal{A}$. La dimensión de un complejo $\mathcal{A}$ es la máxima dimensión de cualquiera de sus facetas. Un complejo es *puro* si todas sus facetas tienen la misma dimensión. Un complejo $\mathcal{B}$ es un subcomplejo de $\mathcal{A}$ si todo simplejo de $\mathcal{B}$ es también un simplejo de $\mathcal{A}$. Una *subdivisión* de un complejo $\mathcal{A}$ se construye “dividiendo” los simplejos de $\mathcal{A}$ en simplejos más pequeños (Figura 2.3).

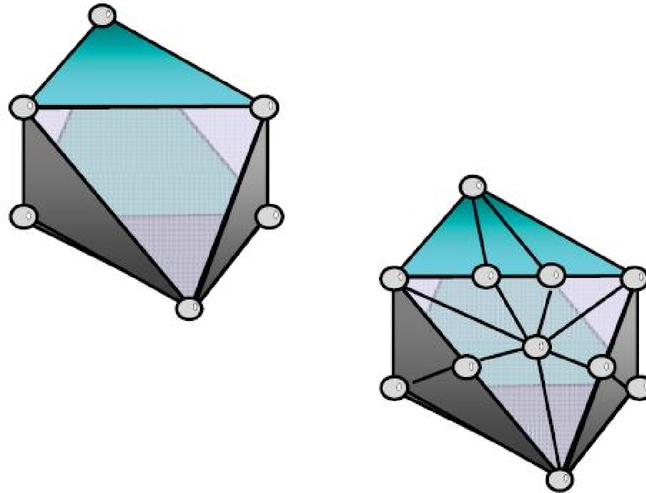

Figura 2.3: Vista geométrica de un complejo simplicial y su subdivisión. *Fuente:* [21]

2.3. Representando sistemas distribuidos con complejos simpliciales

Consideremos una ejecución de un sistema distribuido que contiene $n+1$ procesos, en la que cada proceso p_i tiene una vista local v_i. Caracterizamos el estado de cada proceso como un vértice definido por el par $x_i = (p_i, v_i)$, en el cual $nombre(x_i) = p_i$ y $vista(x_i) = v_i$. Si sólo los procesos $p_0, ..., p_n$ participan, entonces el estado del sistema está caracterizado por el conjunto de pares asociados con los procesos participantes: $\sigma = \{(p_0, x_0), ..., (p_n, x_n)\}$. Este conjunto de pares es un *simplejo*. Denotamos como $nombres(\sigma)$ el conjunto de procesos en σ y $vistas(\sigma)$ el conjunto de sus vistas. Notemos que si σ es un simplejo que representa el estado del sistema en una ejecución, entonces también lo es $\sigma' \subseteq \sigma$ ya que σ' describe el estado en una ejecución en la que quizás participan menos procesos. El conjunto de todos los simplejos que representan todas las posibles asociaciones de vistas con procesos en una ejecución de un sistema distribuido forman un *complejo simplicial.* En computación distribuida, cada vértice x_i en un simplejo se caracteriza por un par distinto (p_i, v_i). Cada proceso p_i es asociado con un color distinto, el cual se usa para colorear su correspondiente vértice. Es por esto que decimos que un un complejo formado de tales simplejos está *coloreado* con esos nombres de procesos o bien, es *cromático*, como por ejemplo, los complejos de la Figura 2.4.

Por el contrario, un complejo *no coloreado*, *no cromático* o *anónimo* es aquel cuyos vértices solo se caracterizan por las vistas v_i de los procesos, sin importar a que proceso pertenece cada

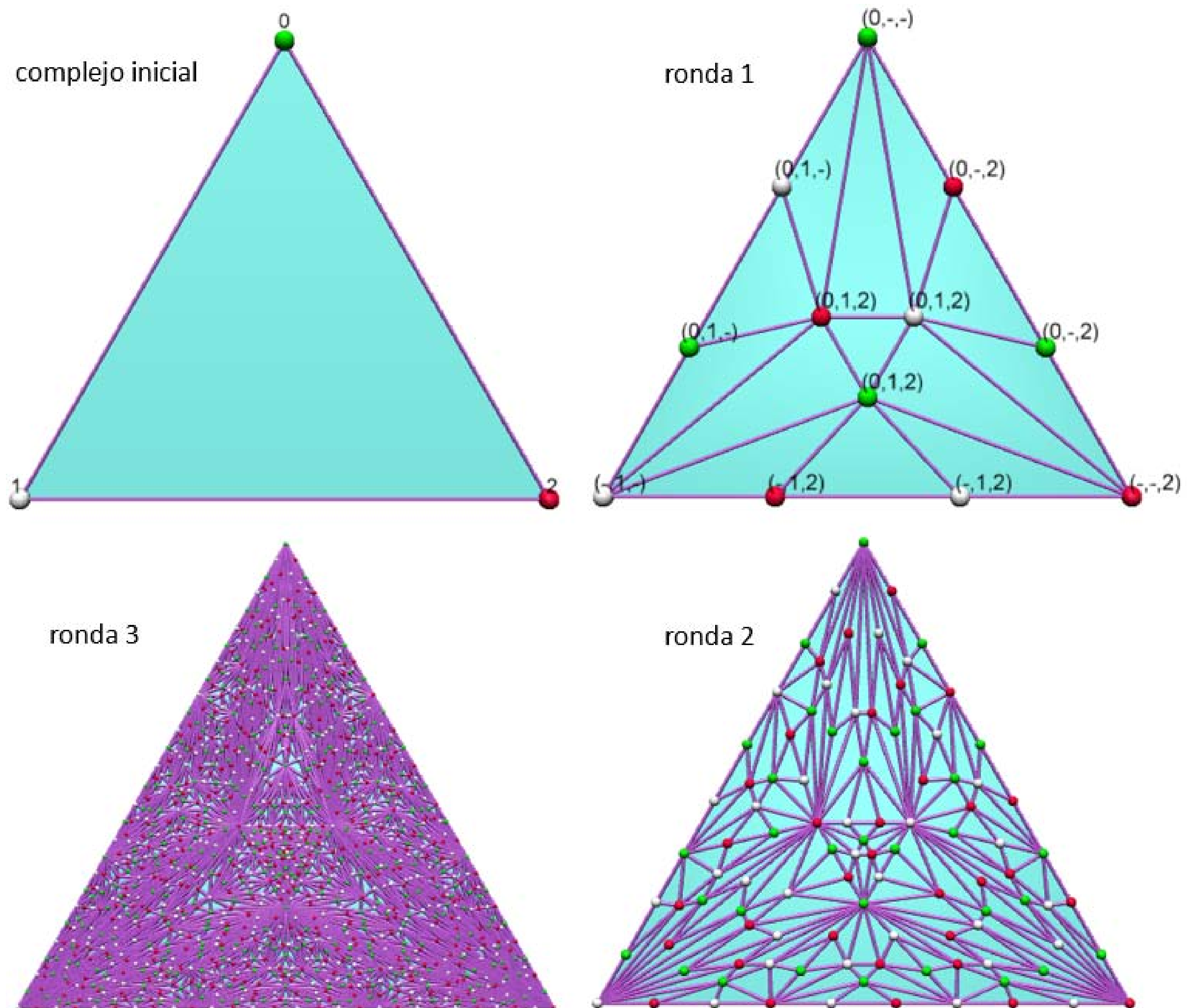

Figura 2.4: Complejo inicial y complejos de protocolo inducidos a lo largo de la ejecución de tres rondas de comunicación del protocolo *immediate snapshot* entre tres procesos: p_0 (verde), p_1 (blanco) y p_2 (rojo). Cada vértice se etiqueta con la vista obtenida por cada proceso al final de la ronda de comunicación (las etiquetas en los demás complejos se omiten por claridad en las imágenes).

vista. Como se prescinde de la identidad de cada proceso, todos los vértices se pintan del mismo color, o bien, no tienen color, por ejemplo, los complejos de la Figura 2.5.

2.4. Tareas

Formalmente, una *tarea*[3] para $n+1$ procesos es una tripleta $(\mathcal{I}, \mathcal{O}, \Delta)$ en donde $\mathcal{I}$ es un complejo de dimensión n que define las posibles vistas iniciales que tiene cada proceso al comienzo de la ejecución del sistema: si $\{(p_{i_0}, v_{i_0}), ..., (p_{i_k}, v_{i_k})\}$ es un simplejo de $\mathcal{I}$, entonces es posible que una ejecución comience teniendo cada p_{i_j} con una vista inicial v_{i_j}, para $0 \leq j \leq k$; $\mathcal{O}$ es un complejo puro de dimensión n que define las posibles opciones de los valores de salida: Si $\{(p_{i_0}, w_{i_0}), ..., (p_{i_k}, w_{i_k})\}$ es un simplejo de $\mathcal{O}$, entonces es posible que una ejecución termine con cada p_{i_j} habiendo elegido el valor de salida w_{i_j}, para $0 \leq j \leq k$; finalmente Δ es un mapeo que le asigna a cada simplejo de entrada σ en $\mathcal{I}$ un subcomplejo $\Delta(\sigma) \subseteq \mathcal{O}$, con la siguiente interpretación: si el sistema comienza en un estado $\sigma \in \mathcal{I}$, entonces toda ejecución debe terminar en algún estado $\tau \in \Delta(\sigma)$.

El mapeo Δ satisface ciertas propiedades formales: todo proceso que termina de ejecutarse debio de haber iniciado, por lo tanto Δ debe *preservar colores*, esto es, para cada $\tau \in \Delta(\sigma)$, $nombres(\tau) \subseteq nombres(\sigma)$. Consideremos una ejecución en la cual los procesos en $\sigma \in \mathcal{I}$ participan, pero un subconjunto de procesos $\sigma' \subset \sigma$ terminan antes de que el resto inicie. Los procesos que terminaron después deben de poder elegir valores compatibles con los valores ya elegidos por los procesos que terminaron primero. Esto se expresa mediante el siguiente requerimiento de *mapeo portador*:

$$\text{Si } \sigma' \subseteq \sigma \text{ entonces } \Delta(\sigma') \subseteq \Delta(\sigma).$$

Un protocolo *resuelve* una tarea $(\mathcal{I}, \mathcal{O}, \Delta)$ si para cada simplejo $\sigma \in \mathcal{I}$ y en cada ejecución del protocolo iniciando desde σ, cada proceso sin falla elige un valor de salida y cada simplejo definido por estas elecciones está en $\Delta(\sigma)$.

Ejemplo de tarea: *consenso binario*

La tarea del *consenso* representa uno de los problemas fundamentales en computación distribuida [14]. Consiste en lo siguiente: dos o más procesos tienen como entrada un valor tomado

[3]Aunque en esta versión de programa no se involucran tareas, incluimos la presente sección para hacer al lector a comprender mejor el uso de complejos simpliciales en el estudio de sistemas distribuidos.

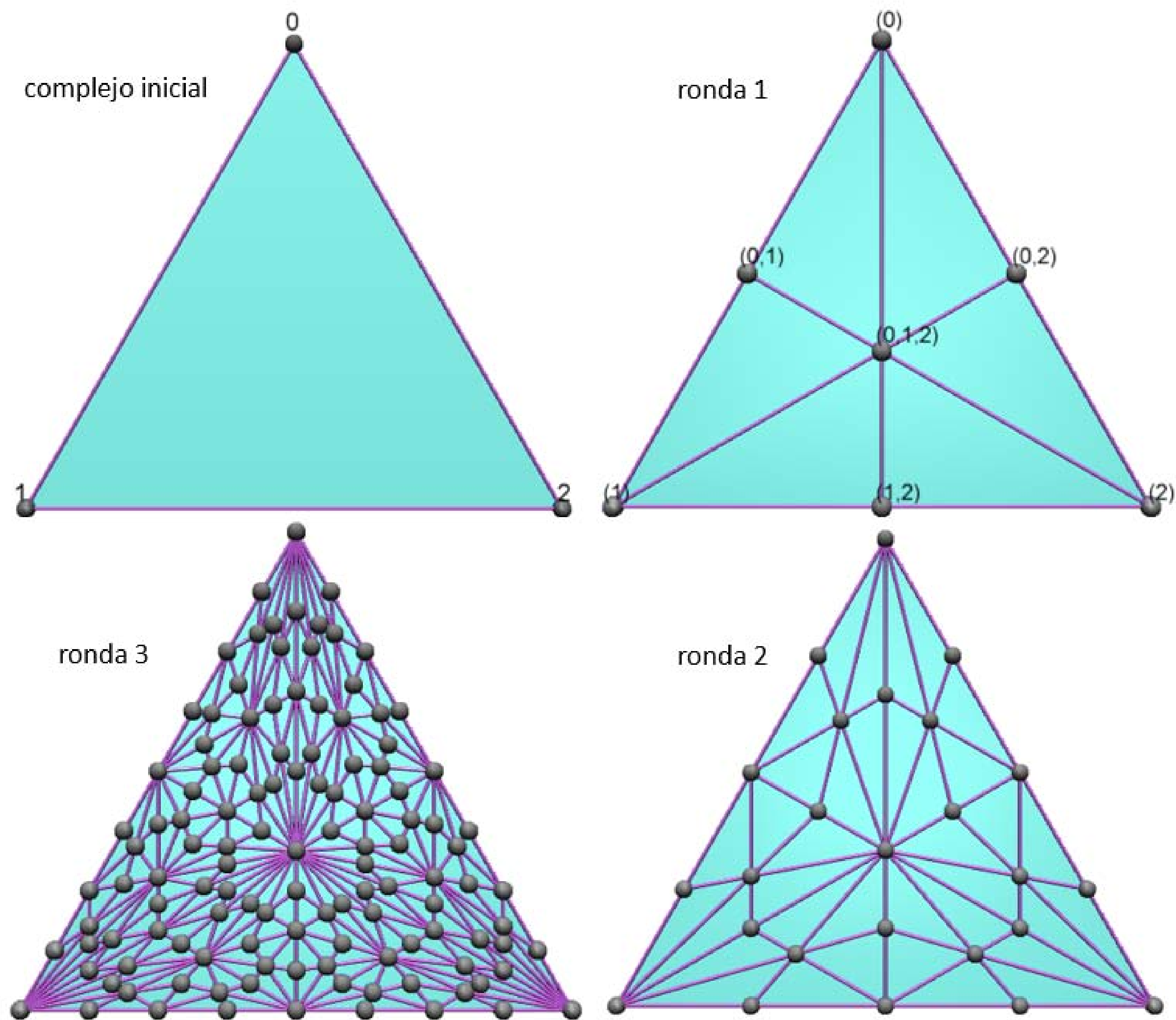

Figura 2.5: Representación no cromática de los complejos de la Figura 2.4. Los procesos no tienen identidad, solo se distinguen por su vista. En las etiquetas del complejo de al ronda 1 se omite el símbolo '-' que se usa para denotar que un proceso no escribió su vista, pues mientras que en un complejo cromático cada entrada de la memoria esta asociada al *id* de proceso, en esta representación los *id*s de procesos no se toman en cuenta (las etiquetas en los demás complejos se omiten por claridad en las imagenes).

de un conjunto de $n+1$ elementos y todos deben de decidir un único valor de tal forma que se cumplan las siguientes propiedades:

1. *Consistencia*. Todos los procesos deciden el mismo valor.

2. *Validez*. El valor decidido es el de la entrada de alguno de los procesos.

Una versión simple de la tarea de consenso es el *consenso binario*, en la cual participan solo dos procesos. En la Figura 2.6 podemos ver dos diferentes versiones de la especificación de la tarea del consenso binario con complejos simpliciales. En el lado izquierdo de la figura podemos observar que el complejo de entrada $\mathcal{I}$ consiste en un único simplejo $\{(p_0, v_0), (p_1, v_1)\}$, en donde p_0 es representado en el vértice de color blanco, p_1 es el vértice de color negro y el simplejo es representado por los vértices y la arista. En este ejemplo el proceso p_0 inicia con la vista $v_0 = 0$, mostrado al interior del vértice. De igual forma para el proceso p_1. El hecho de que ambos vértices estén conectados por una arista denota el posible escenario en el sistema distribuido en el cual estos procesos inician la ejecución con estas vistas; por el contrario, el complejo inicial en la parte derecha de la figura denota otra versión de la tarea en la cual se muestran todas las posibles asignaciones de las vistas iniciales v_0, v_1 a los procesos p_0, p_1. Los complejos de salida de la figura representan las configuraciones válidas que especifica la tarea del consenso: los procesos terminan la ejecución decidiendo ambos 0 o ambos 1, pero no existe ninguna configuración en la cual los dos decidan valores distintos. En la figura también se muestra mediante las flechas punteadas cómo el mapeo portador Δ le asigna a cada simplejo del complejo de entrada un subcomplejo del complejo de salida, siendo su interpretación operacional la siguiente: si, por ejemplo, el proceso p_1 falla antes de comunicarle su valor propuesto a p_0, entonces p_0 decide su valor v_0, es decir, $\Delta(\{(p_0, v_0)\}) = \{(p_0, w_0)\}$, donde $\{(p_0, w_0)\} \in \mathcal{O}$ y $w_0 = 0$. De la misma forma para p_1.

2.5. Modelos de cómputación distribuida

Las tareas especifican los problemas que se quiere resolver, pero los modelos de computación distribuida nos dicen cómo se pueden resolver. Un modelo de computación distribuida es una abstracción de alto nivel para simplificar el diseño y la verificación de los algoritmos concurrentes. Un modelo típicamente especifica cómo se comunican los procesos, cómo se calendarizan y cómo pueden fallar.

Principalmente se estudian dos modelos de comunicación entre procesos: memoria compartida, en el cual los procesos se comunican escribiendo y leyendo una memoria compartida; y paso

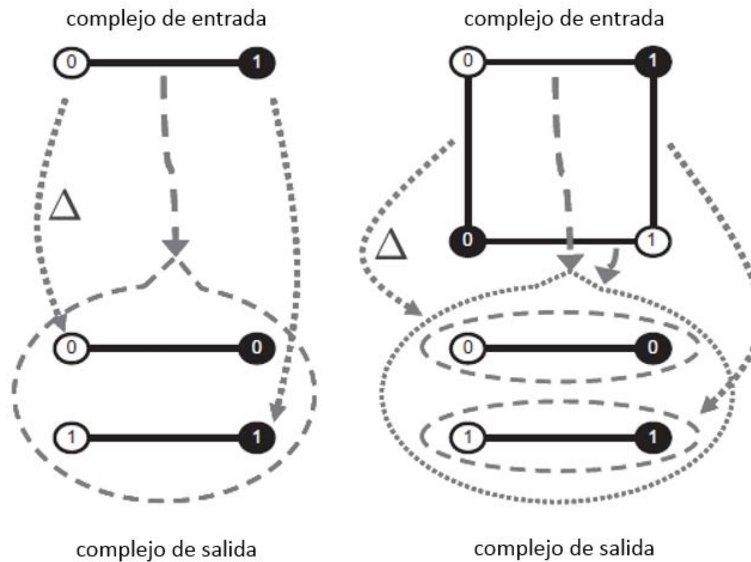

Figura 2.6: Dos versiones distintas de la especificación de la tarea del consenso binario vista mediante complejos simpliciales. *Fuente:* [19]

de mensajes, en el cual los procesos se comunican enviándose mensajes a través de canales de comunicación.

Esta primera versión del programa solamente permite la generación y visualización de complejos simpliciales para el modelo en el cual dos o tres procesos se comunican por memoria compartida tipo *atomic immediate snapshot*, asumiendo la ausencia de fallas, tanto para la versión cromática como la no cromática.

2.5.1. Memoria compartida

Una memoria compartida puede verse como un arreglo de registros. Un registro puede consistir en un solo bit, en un número constante de bits o en un número arbitrario. Un registro que puede ser escrito por un solo proceso pero leido por varios procesos se conoce como *SWMR* [4]. Si la variable puede ser escrita por varios procesos se conoce como *MWMR* [5]. Con estos tipos de registros es posible construir modelos de memoria tipo *atomic snapshot* [8] (copia atómica) y *atomic immediate snapshot* [11] (copia atómica inmediata), en los cuales los procesos pueden leer una cantidad de registros contiguos en un solo paso. Estos modelos no son muy realistas en el sentido de que los procesadores reales no proveen de este tipo de operaciones, sin embargo, son muy útiles para obtener resultados de resolución de tareas, por ejemplo, al simplificar cotas inferiores: una tarea imposible de resolver utilizando *atomic snapshot* es también imposible de

[4]Single Writer/Multiple Reader
[5]Multiple Writer/Multiple Reader

resolver utilizando lecturas y escrituras de registros individuales.

Atomic Snapshot

Una memoria tipo *atomic snapshot* es una memoria en donde cada proceso nombrado p_i tiene asignado un registro con índice i, en el cual puede escribir; y a su vez, la lectura consiste en una operación indivisible la cual le devuelve a cada proceso una copia de la memoria entera de forma instantánea. Al final del desarrollo de este programa se logró implementar la generación de complejos simpliciales para éste modelo. En la Figura 2.7 se muestran el complejo de protocolo cromático de la primera ronda para tres procesos.

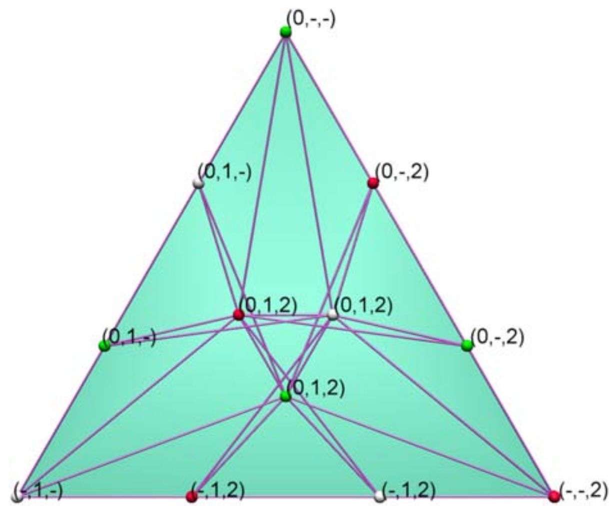

Figura 2.7: Complejo de protocolo cromático de la primera ronda para tres procesos, basado en el protocolo *atomic snapshot*.

Atomic Immediate Snapshot

Este modelo de memoria es una extensión del modelo *atomic snapshot*, pero además de retornar de forma instantánea una copia entera de la memoria, garantiza que esta operación se realiza *inmediatamente* después de terminarse de ejecutar la operación de escritura. En otras palabras, podemos ver las operaciones de escribir y *snapshot* juntas como una única operación atómica.

En la Figura 2.8 mostramos un ejemplo de las tres posibles ejecuciones (a), (b) y (c) que pueden suceder cuando dos procesos p_0 y p_1 que se comunican de forma asíncrona una sola vez mediante este modelo de memoria. Supongamos que cada proceso inicia con su vista inicial igual a su nombre. Las columnas de las tablas representan los registros de la memoria, denotados como

`mem[i]`. Cada proceso p_i escribe su valor de entrada en el registro `mem[i]` que le corresponde, en diferentes instantes de tiempo. En cada renglón se muestra qué operación se ejecuta sobre los registros. Cada renglón, salvo el último, representa un determinado instante de tiempo. En el último renglón de cada tabla se muestran las vistas de cada proceso al final de la ejecución. Así, por ejemplo, en el primer renglón de la ejecución (a), write$_0$(mem[0]) indica que p_0 escribe su valor de entrada 0 en el registro `mem[0]`. Inmediatamente después de escribir efectua una operación de lectura tipo *immediate snapshot* denotada por snap$_0$(mem[0]). Para ilustrar que la lectura es instantánea se ve que la operación se repite en todas las columnas en el mismo renglón, indicando que se leen todos los registros de la memoria al mismo tiempo. Al terminar p_0 de ejecutar su operación de lectura, el proceso p_1 aún no ha escrito aún su valor de entrada en su registro, por lo tanto al final de la ejecución, p_0 termina con la vista $(0, -)$. En el último renglón de la tablas mostramos las vistas finales de los procesos al final de las ejecuciones.

(a)

Tiempo	mem[0]	mem[1]
↓	write$_0$(mem[0])	
	snap$_0$(mem[0])	snap$_0$(mem[1])
		write$_1$(mem[1])
	snap$_1$(mem[0])	snap$_1$(mem[1])
	vista(p$_0$)=(0,-)	**vista(p$_1$)=(0,1)**

(b)

mem[0]	mem [1]
	write$_1$(mem[1])
snap$_1$(mem[0])	snap$_1$(mem[1])
write$_0$(mem[0])	
snap$_0$(mem[0])	snap$_0$(mem[1])
vista(p$_0$)=(0,1)	**vista(p$_1$)=(-,1)**

(c)

Tiempo	mem[0]	mem[1]
↓	write$_0$(mem[0])	write$_1$(mem[1])
	snap$_0$(mem[0]), snap$_1$(mem[0])	snap$_0$(mem[1]), snap$_1$(mem[1])
	vista(p$_0$)=(0,1)	**vista(p$_1$)=(0,1)**

Figura 2.8: Posibles escenarios de ejecución en los que participan dos procesos, p_0 y p_1, comunicándose a través de una memoria compartida tipo *immediate snapshot*.

2.6. Complejo de protocolo

Sea $(\mathcal{I}, \mathcal{O}, \Delta)$ una tarea. Consideremos una ejecución del sistema distribuido en la cual los procesos intercambian información mediante algún mecanismo de comunicación. Al final de esta ejecución cada proceso tiene posiblemente una nueva vista local. Un conjunto de vértices $\{(p_0, v_0), ..., (p_n, v_n)\}$ es un simplejo si existe alguna ejecución en la cual p_0 tiene la vista v_0 al final de la ejecución, p_1 la vista v_1, y así sucesivamente. El complejo simplicial compuesto por

estos simplejos es llamado *complejo de protocolo*. A lo largo de esta tesis, y en el código del programa, nos referiremos a la ejecución en la que los procesos intercambian sus vistas mediante el mecanismo de comunicación como *ronda de comunicación* o *ronda de ejecución*. También nos referiremos al mecanismo de comunicación como *protocolo*.

Sea $\mathcal{P}$ un complejo de protocolo y Ξ un mapeo portador de $\mathcal{I}$ a $\mathcal{P}$. Ξ se conoce como *mapeo portador de ejecución*, el cual lleva cada simplejo $\sigma \in \mathcal{I}$ a un subcomplejo $\Xi(\sigma) \subseteq \mathcal{P}$. Ξ lleva a cada vértice x_i a la ejecución aislada en la cual p_i termina de ejecutar el protocolo sin haberse comunicado con los demás procesos; también lleva a cada simplejo $\sigma = \{(p_0, v_0), ..., (p_n, v_n)\}$ al subcomplejo de ejecuciones en donde p_0 termina con la vista v_0, p_1 con la vista v_1, etcétera.

El complejo de protocolo $\mathcal{P}$ se relaciona con el complejo de salida $\mathcal{O}$ mediante un *mapeo de decisión* δ que manda cada vértice $x_i = (p_i, v_i)$ en $\mathcal{P}$ a un vértice $x'_i = (p_i, w_i)$, en donde $nombre(x_i) = nombre(x'_i)$. Operacionalmente este mapeo debe de entenderse de la siguiente manera: si existe una ejecución del protocolo en la cual p_i termina con la vista v_i y después elige como salida al valor w_i, entonces $x_i \in \mathcal{P}$, $x'_i \in \mathcal{O}$ y $\delta(x_i) = x'_i$. También δ mapea simplejos a simplejos, ya que cualquier conjunto de vistas mutuamente compatibles se relaciona con un conjunto de valores de decisión mutuamente compatibles.

Retomando el ejemplo de la Figura 2.8, en la Figura 2.9 mostramos como los diferentes complejos de protocolo que se generan a lo largo de tres rondas de comunicación. En la ronda cero el complejo de protocolo consiste en el complejo inicial $\mathcal{I}$. El vértice asociado con p_0 se muestra en color blanco y el de p_1 en color negro. La etiqueta junto a cada vértice denota la vista del proceso al terminar la ronda de ejecución. Vemos que en al final de la ronda uno se induce un complejo de protocolo compuesto por tres 1-simplejos (aristas) conectados. Cada simplejo corresponde a cada posible ejecución mostrada en la Figura 2.8. De igual manera, por cada simplejo contenido en el complejo de la ronda uno, en la ronda dos se generan tres simplejos conectados. Este mismo patrón se repite en cada ronda de ejecución. También podemos observar como cada proceso comunica su vista entera en cada ronda, ya que el protocolo es de información completa. Con este ejemplo podemos observar la estructura recursiva y conexa que tienen los complejo de protocolo inducidos por el protocolo *immediate snapshot* [21].

2.7. Conclusión

En este capítulo revisamos algunos de los conceptos fundamentales de la teoría de computación distribuida a través de la topología combinatoria. Revisamos los conceptos fundamentales de sistemas distribuidos: procesos, mecanismos de comunicación, vistas locales, tipos de proto-

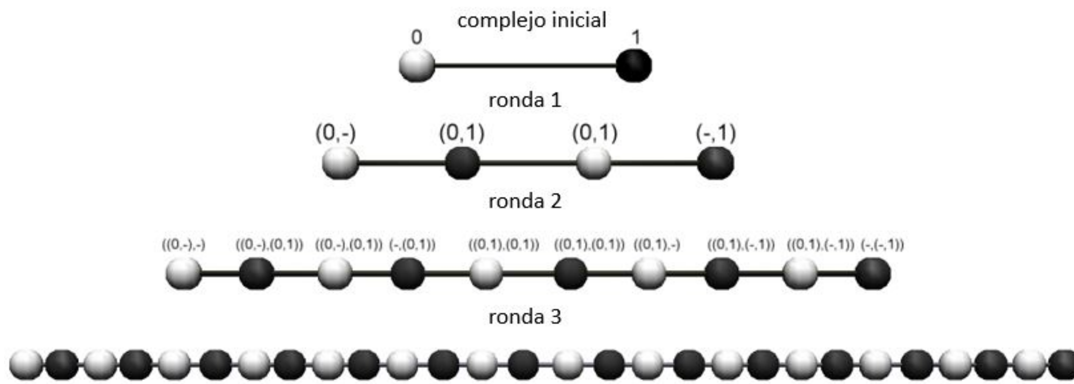

Figura 2.9: Complejos de protocolo generados a lo largo de dos rondas de comunicación entre dos procesos p_0 y p_1, bajo el modelo del ejemplo de la Figura 2.8

colos, fallas y tareas. También revisamos la definición de complejo simplicial y cómo es que estos se utilizan para modelar el comportamiento de un sistema distribuido y para especificar tareas. Finalmente de manera breve revisamos el modelo de memoria *immediate snapshot* que fue el primero que se implementó para producir complejos de protocolo en el programa y también revisamos el concepto de complejo de protocolo.

Parte II

Desarrollo del programa

Capítulo 3

Especificación y análisis de requerimientos

Los requerimientos de un sistema de software son descripciones de lo que debe de hacer el sistema, los servicios que proporciona y las restricciones en su operación. Los requerimientos son el reflejo de la necesidad por parte de los usuarios de un sistema que cumpla un determinado propósito [30].

3.1. Requerimientos funcionales

Los requerimientos funcionales para un programa describen los servicios específicos que el programa ofrecerá y como reaccionará el programa ante ciertas entradas [26]. A continuación damos los requerimientos funcionales de nuestro programa.

1. El programa deberá tener una interfaz de usuario que consista en lo siguientes controles.

    1.1. Complejo simplicial inicial $\mathcal{I}$. Se especifica mediante los siguientes controles:

    - Número de procesos en el simplejo. En esta versión del programa los complejos iniciales solo van a contener un simplejo, por lo tanto se debe ofrecer un control que permita introducir si este simplejo contiene uno, dos o tres procesos.
    - Por cada proceso se le solicita al usuario introducir el nombre del proceso y el color con el cual se pintará su vértice. Los nombres deben ser únicos y deben de consistir solamente en un carácter. Si no es así, el programa debe mostrar un mensaje de error después de haber presionado el botón especificado en el requerimiento 1.2..

1.2. Un botón que al ser presionado haga que el programa genere el complejo inicial de acuerdo a los datos introducidos y muestre su visualización.

1.3. Controles para introducir la información del modelo de computación distribuida que determina la forma del complejo de protocolo que se va a generar. Cuando se haya generado y mostrado el complejo inicial, ocultar los controles especificados en el requerimiento 1.1. y mostrar los controles que a continuación se describen.

1.3.1. Un control que muestre los protocolos disponibles y que permita seleccionar aquel bajo el cual se va a generar el complejo de protocolo.

1.3.2. Dependiendo del protocolo seleccionado, mostrar controles que permitan especifiar los parámetros específicos ese protocolo, por ejemplo, si se seleccionó memoria compartida, mostrar controles que permitan especificar si es tipo *atomic snapshot*, *immediate snapshot*, protocolo *read/write*, etcétera [1].

1.3.3. Un botón que al ser presionado haga que el programa genere el complejo de protocolo de acuerdo a los parámetros introducidos y muestre su visualización.

1.3.4. Un botón que al ser presionado el programa genere el complejo de protocolo de la siguiente ronda de comunicación y muestre su visualización. Este botón solo se mostrará si ya se haya generado al menos un complejo de protocolo.

1.3.5. Siempre que este desplegada en pantalla la visualización de un complejo de protocolo, proveer de controles para cambiar entre su representación cromática y no cromática.

1.3.6. Un botón para permitir mostrar nuevamente los controles especificados en el requerimiento 1.1. y ocultar los controles especificados en el requerimiento 1.3., esto con el fin de permitir al usuario especificar un nuevo complejo inicial.

1.4. Un espacio para desplegar la visualización de los complejos simpliciales generados. Este espacio deberá de permitir al usuario realizar las siguientes acciones.

1.4.1. Rotar el complejo simplicial.

1.4.2. Permitir hacer acercamientos y alejamientos del complejo simplicial.

1.4.3. Reacomodar manualmente los vértices en el espacio.

1.5. Una consola que muestre la siguiente información del complejo cuya visualización se muestra.

[1] Aunque en está versión del programa solo se permitirá elegir *immediate snapshot*, dejar el programa listo para permitir integrar fácilmente el soporte a otros modelos

1.5.1. El complejo simplicial en notación de conjuntos.

1.5.2. Número de vértices y caras que contiene el complejo simplicial.

3.2. Requerimientos no funcionales

Son restricciones a los servicios y funciones ofrecidas por el programa. Incluyen restricciones de tiempo de respuesta y procesamiento, propiedades de calidad y apego a estándares. Este tipo de requerimientos se aplican al el programa como un todo, en lugar de a servicios o funciones individuales [30].

Especificamos los requerimientos no funcionales en forma de características que debe de tener el programa.

1. Facilidad de acceso e instalación.

2. Extensibilidad. El diseño del programa debe ser de tal forma que sea fácil extenderlo con nuevas funcionalidades tales como generación de visualizaciones para otros modelos de computación distribuida (paso de mensajes, memoria iterada, modelos con tolerancia a fallas, etcétera) y mecanismos para personalizar la visualización, e.g., cambiar el color de los vérticas, aristas y caras. También debe ser fácil reemplazar las bibliotecas de visualización en caso de que se necesite hacer en el futuro.

3. Producción de visualizaciones claras, fáciles de entender y aptas para su uso en publicaciones.

3.3. Casos de uso

En la Figura 3.1 se muestra el diagrama de los siguientes casos de uso.

1. **Generar complejo inicial**

   - **Descripción.** Genera un complejo inicial de acuerdo a los parámetros introducidos por el usuario y muestra su visualización.
   - **Actores.** Usuario.
   - **Precondiciones.** Ninguna.
   - **Flujo normal de eventos.**

   1.1. El usuario inicia el programa.

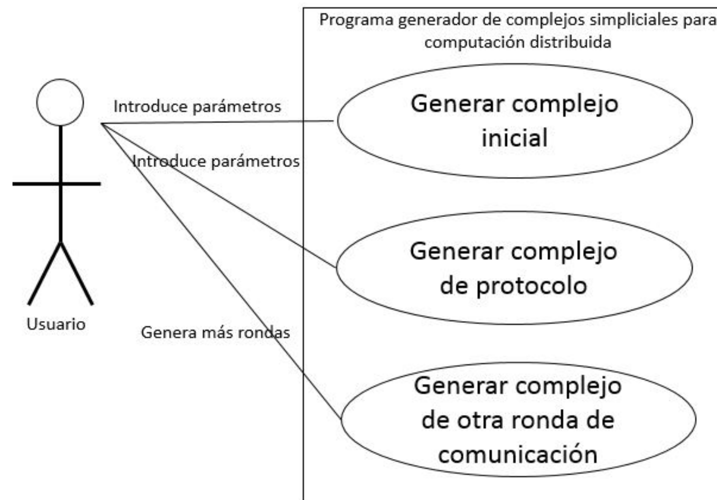

Figura 3.1: Diagrama de los casos de uso del programa.

1.2. Se despliega la interfaz de usuario del programa.

1.3. El usuario selecciona el número de procesos que contiene el simplejo que conforma el complejo inicial.

1.4. El programa habilita los controles para introducir los nombres de cada proceso.

1.5. El usuario introduce los nombres y colores de cada proceso.

1.6. El usuario presiona el botón para visualizar el complejo inicial (requerimiento 1.2.).

1.7. El programa valida que se hayan introducido todos los nombres de los procesos, que estos sean únicos y consistan en un solo carácter.

- Si falla la validación, se despliega un mensaje de error indicando que condición no se cumplió.
- Si no falla la validación, se muestra la visualización de el complejo inicial, se muestra en la consola la información del complejo, se ocultan los controles para introducir el complejo inicial y se habilitan los controles para introducir los datos del modelo (requerimiento 1.3.).

- **Flujo alternativo de eventos.**

1.1. El usuario realiza los pasos del flujo de eventos en el caso de uso 2 o en el caso de uso 3.

1.2. El usuario presiona el botón para generar un nuevo complejo inicial (requerimiento 1.3.6.).

1.3. Se repite la secuencia del flujo normal de eventos a partir del segundo evento.

- **Postcondiciones.** El programa internamente mantiene la información del complejo inicial que se utilizará como base para generar el complejo de protocolo de la primera ronda de comunicación.

2. **Generar complejo de protocolo**

- **Descripción.** Genera un complejo de protocolo con base en el complejo inicial generado y los parámetros del modelo de computación distribuida introducidos por el usuario y muestra su visualización.
- **Actores.** Usuario.
- **Precondiciones.** Un complejo inicial debe haber sido generado.
- **Flujo normal de eventos.**

2.1. El usuario realiza los pasos del flujo de eventos en el caso de uso 1.

2.2. El usuario selecciona el modelo de computación distribuido.

2.3. El sistema muestra los controles para introducir los parámetros especificos del modelo seleccionado.

2.4. El usuario introduce los parámetros y presiona el botón para generar y mostrar el complejo de protocolo (requerimiento 1.3.3.).

2.5. El programa despliega la visualización del complejo de protocolo generado y muestra en la consola la información del complejo de protocolo. También oculta los controles para introducir los parámetros del modelo (requerimiento 1.3.) y solo deja habilitado el botón para generar un el complejo de protocolo de la siguiente ronda de comunicación (requerimiento 1.3.4.) y el botón para especificar un nuevo complejo inicial (requerimiento 1.3.6.).

- **Postcondiciones.** El programa internamente mantiene la información del complejo de protocolo generado y el cual se utilizará como base para generar el complejo de protocolo de la siguiente ronda de comunicación.

3. **Generar complejo de protocolo de otra ronda de comunicación**

- **Descripción.** Genera un complejo de protocolo partiendo del complejo de protocolo generado en la última ronda de comunicación.
- **Actores.** Usuario.

- **Precondiciones.** El complejo de protocolo de la anterior ronda de comunicación debe haber sido generado.
- **Flujo normal de eventos.**

3.1. El usuario realiza los pasos del flujo de eventos en el caso de uso 2.

3.2. El usuario presiona el botón que hace que el programa genere el complejo de protocolo de una nueva ronda de comunicación (requerimiento 1.3.4.).

3.3. El programa despliega la visualización del complejo de protocolo generado y muestra en la consola la información del complejo de protocolo. Los controles para introducir los parámetros del modelo (requerimiento 1.3.) y los controles para introducir un nuevo complejo inicial (requerimiento 1.1.) se mantienen ocultos y continuan habilitados el botón del requerimiento 1.3.4. y el botón del requerimiento 1.3.6..

- **Postcondiciones.** El programa internamente mantiene la información del complejo de protocolo producido en esta ronda y el cual se utilizará como base para generar el complejo de protocolo de la siguiente ronda de comunicación.

3.4. Conclusión

Con base en los requerimientos funcionales y los casos de uso descritos en este capítulo, podemos concluir que esta versión del programa permite generar y mostrar visualizaciones de complejos iniciales y complejos de protocolo para varias rondas de comunicación para varios modelos de computación distribuida con a lo más tres procesos y los complejos podrán ser cromáticos o no cromáticos. Por otro lado los requerimientos no funcionales dados nos comunican que el programa debe de ser fácil de usar y entender y también debe de ser fácil de modificar y extender en el futuro.

Capítulo 4

Diseño y arquitectura

En el presente capítulo discutimos la arquitectura del programa y los aspectos de su diseño que satisfacen los requerimientos. También damos algunos diagramas que ilustran su arquitectura y diseño.

El lenguaje de programación con el que se escribió el programa es Java, el cual es un lenguaje orientado a objetos, por lo tanto en este capítulo (y en el Capítulo 5 también) usamos términos tales como *clase*, *objeto*, *método*, *herencia*, *interfaz*, etcétera.

4.1. Arquitectura del programa

Una *arquitectura de software* es una descripción de los subsistemas y componentes de un sistema de software y las relaciones entre ellos. Los subsistemas y componentes son típicamente especificados en diferentes vistas para mostrar las propiedades funcionales y no funcionales reelevantes de un sistema de software. La arquitectura de un sistema de software es un artefacto que resulta de la actividad de diseñar el software [13].

Un *componente* es una parte encapsulada de un sistema de software. Un componente tiene una interfaz. Los componentes sirven como bloques de construcción para la estructura de un sistema. Desde una perspectiva de programación, los componentes se pueden ver como módulos, clases, objetos o un conjunto de funciones relacionadas [13].

La arquitectura de un sistema de software es importante ya que afecta sus propiedades no funcionales tales como robustez, desempeño y mantenibilidad [12]. Mientras que los componentes individuales de un sistema implementan los requerimientos funcionales, la arquitectura del sistema es la principal fuerza que influye en la implementación de los requerimientos no funcionales[30].

Aunque esta no es propiamente la arquitectura exacta del programa, éste se podría ver conformado por dos principales componentes. El primer componente, al cual llamaremos *generador*, consiste en el conjunto de algoritmos y estructuras de datos que se encargan de computar y manipular representaciones de complejos simpliciales. El segundo componente, al cual llamaremos *visualizador*, es el que se encarga de producir y mostrar las visualizaciones de los complejos computados por el generador. Éste también provee la interfaz de usuario mediante la cual se introducen los datos de entrada en el programa.

En la sección 4.1.1 se muestra la arquitectura del programa, en donde descomponemos estos dos componentes en subcomponentes con responsabilidades más específicas, y los organizamos de acuerdo a un patrón de arquitectura de software conocido como MVC.

4.1.1. Patrón arquitectónico

Para lograr mayor flexibilidad en el diseño conviene estructurar los componentes identificados en la sección anterior en un patrón arquitectónico de software.

Un patrón arquitectónico expresa un esquema de organización de la estructura fundamental de un sistema de software. Provee un conjunto de subsistemas predefinidos, especifica sus responsabilidades e include reglas y guías para organizar las relaciones. El uso de patrones arquitectónicos nos ayuda a lograr la implementación de las propiedades no funcionales del programa [13].

El programa es un sistema interáctivo en el que la interfaz de usuario (visualizador) y el núcleo funcional (generador) están claramente separados. Más aún, la interfaz de usuario podría cambiar en el futuro, de acuerdo al requerimiento no funcional 2.. Esto indica que el patrón arquitectónico más adecuado para continuar estructurando el programa es el patrón *Modelo-Vista-Controlador* (MVC) [13].

El patrón Modelo-Vista-Controlador, como su nombre lo indica, divide una aplicación interactiva en tres componentes. El modelo contiene la funcionalidad principal y los datos del programa. Las vistas muestran información al usuario. El controlador (o controladores, dependiendo de la implementación del programa) maneja las entradas del usuario. La vistas y los controladores comprenden la interfaz de usuario. Un mecanismo de propagación de cambios asegura la consistencia entre la interfaz de usuario y el modelo [13]. Este patrón es muy popular y es ampliamente usado en el diseño de aplicaciones web y de escritorio. En la Figura 4.1 mostramos las tarjetas CRC[1] que describen los componentes que implementan el patrón MVC

[1] Las tarjetas Class-Responsability-Colaboration nos ayudan a especificar los objetos y componentes de una

en la arquitectura del programa.

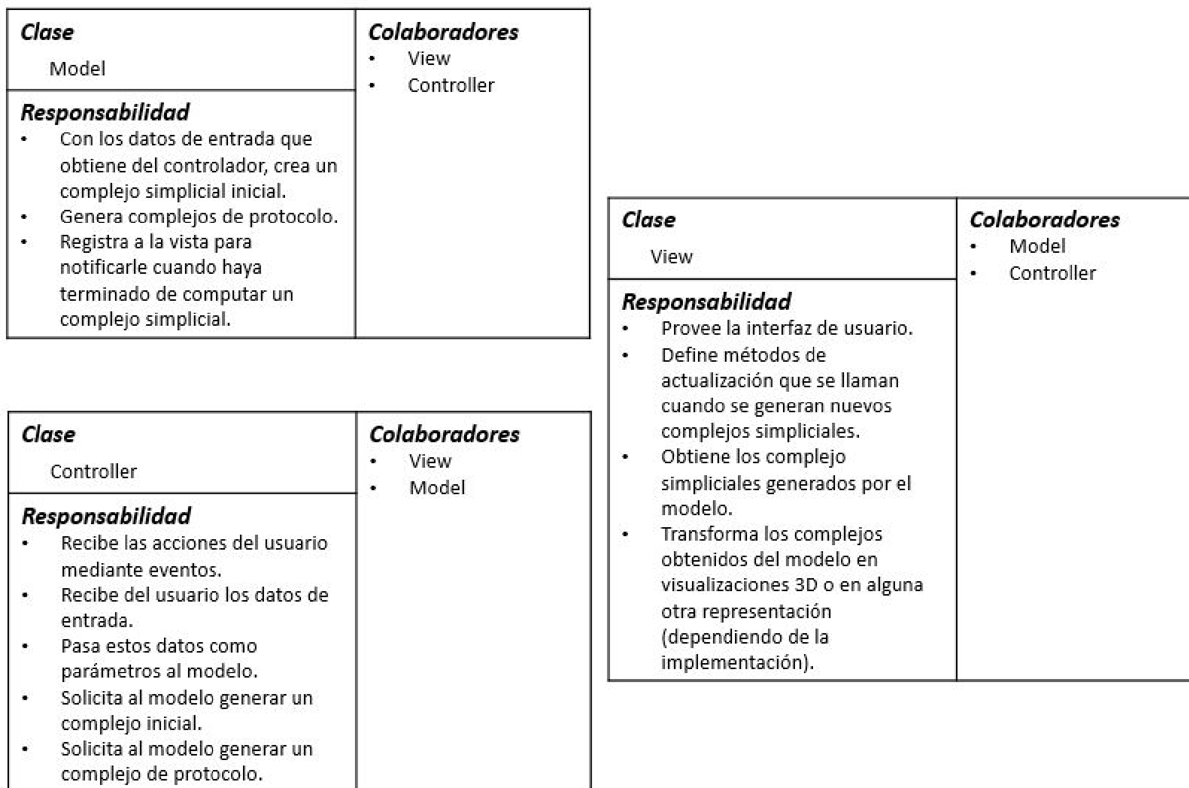

Figura 4.1: Tarjetas CRC que describen los componentes de la arquitectura del programa.

Refinando la arquitectura

En el programa el modelo se implementa como una clase llamada `Model`. Para el caso del controlador este no es el caso, ya que debido a los requerimientos del programa, durante la implementación se encontró conveniente codificar éste como un grupo de clases interrelacionadas que discuteremos con más detalle en el Capítulo 5. Por ahora, para hacer esta discusión más simple, supondremos que este componente es una clase individual la cual llamaremos `Controller`.

Para implementar la vista se definió una interfaz `View`, la cual solo ofrece declaraciones de métodos sin implementación. Las clases `jRealityView` y `SCOutputConsole` implementan los métodos de la interfaz `View`. La primera se encarga de producir visualizaciones de complejos simpliciales, manejando los detalles particulares de la biblioteca jReality. La segunda es la consola que definimos en el requerimiento 1.5..

aplicación de manera informal, especialmente en las primeras fases del desarollo de software [13].

La ventaja de definir la vista como una interfaz es que no acoplamos los demás componentes del programa a jReality, facilitando así su reemplazo por alguna otra biblioteca si así se requiriese en el futuro. De esta forma aplicamos el principio de diseño orientado a objetos que dice: *Programe hacia una interfaz, no hacia una implementación* [16]. Es así como logramos extensibilidad y facilitamos futuras modificaciones al programa.

El diagrama de clases simplificado se muestra en la Figura 4.2. En cada clase e interfaz se muestran los métodos que implementan las responsabilidades descritas en las tarjetas de la Figura 4.1. Notamemos que la clase `Controller` no tiene ninguna referencia hacia la interfaz `View`, al contrario de lo que se esperaría en una arquitectura MVC. Esto es debido a que vimos que apegarse estrictamente a esta arquitectura incrementaba innecesariamente la complejidad del código del programa, así que se decidió introducir una variación en el diseño. En el Capítulo 5 se explican con mayor detalle los elementos que justifican esta decisión.

También se encontró que en todo momento durante la ejecución del programa solo es necesario tener un único objeto que represente al modelo. Por lo tanto se aplicó el patrón de diseño *Singleton*, [16] el cual asegura que solo se puede crear una vez una instancia de la clase en donde se implementa y también proveé un punto de acceso global a ésta. Esto es justamente lo que se necesitó para el diseño de la clase `Model`; y como el programa puede tener solamente una interfaz de usuario, entonces también se aplica este patrón en la clase `jRealityView`.

4.2. Diseño de las principales clases

En esta sección describimos cómo se diseñaron las clases que representan los principales objetos en el programa: `Process`, `Simplex`, `SimplicialComplex`, `CommunicationProtocol`, `ImmediateSnapshot`, `GeometricComplex`, `Face` y `Vertex`.

Funcionamiento del generador

Para facilitar la comprensión del diseño de las principales clases en el programa conviene dar al lector una idea general acerca de cómo el programa genera complejos simpliciales.

Recordemos que la responsabilidad del generador, representado por la clase `Model`, es crear complejos iniciales y computar complejos simpliciales de protocolo para las rondas de comunicación que solicite el usuario, partiendo del complejo inicial y del protocolo de comunicación, los cuales fueron provistos por éste. Una vez computado el complejo de protocolo, este notifica a las vistas llamando a sus métodos `update()` y éstas puede entonces invocar el método

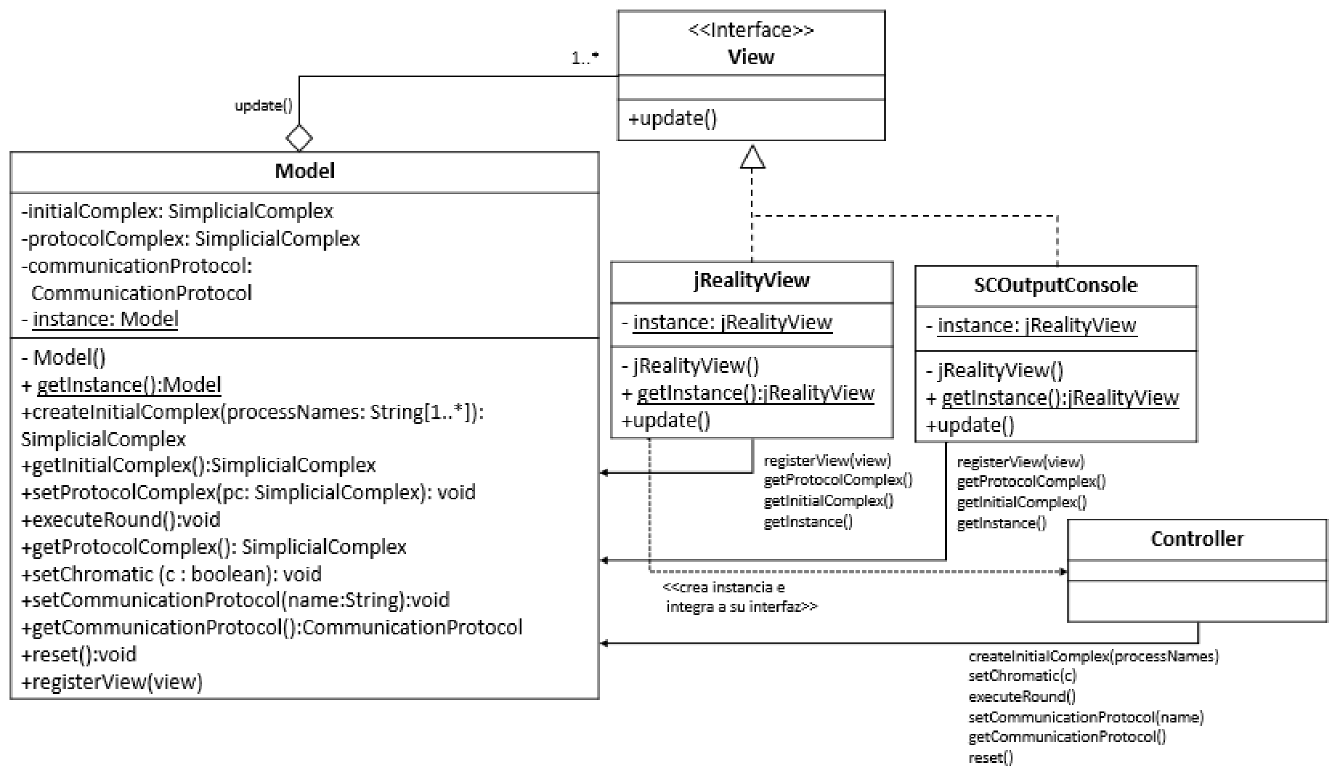

Figura 4.2: Diagrama UML de clases simplificado de la arquitectura MVC del programa. En el código del programa las clases definen también otros métodos, pero por simplicidad estos se omiten en el diagrama.

`getProtocolComplex()` del modelo parar obtener el objeto que representa el complejo de protocolo que acaba de ser generado. Las vistas pueden transformar a este objeto en una visualización del complejo de protocolo o mostrar información textual acerca de éste. Los detalles acerca de cómo se transforma el objeto en una visualización se discutirán en el Capítulo 5.

A grandes rasgos describimos la idea de funcionamiento del método `executeRound()` para computar complejos de protocolo, independientemente del protocolo que utilicen los procesos para comunicarse (en el Capítulo 5 describimos con detalle cómo se implementó este funcionamiento para el modelo de memoria compartida *immediate snapshot*): primero se obtiene el complejo simplicial base a partir del cual se va a producir el complejo de protocolo, este complejo de entrada puede ser el complejo inicial o el complejo de protocolo de la ronda anterior; así mismo se obtiene el objeto que representa el protocolo de comunicación y dependiendo de este objeto el programa selecciona un algoritmo especializado que computa el complejo de protocolo de la ronda actual. Este algoritmo genera todos los posibles escenarios en los que los procesos se pueden comunicar en una ronda a través del protocolo de comunicación dado. Nos referirémos a este algoritmo como *generador de escenarios*.

Despues de la ejecución del generador de escenarios, otro procedimiento recibe el conjunto de escenarios generados y crea copias de cada proceso (las cuales también incluyen sus vistas) en cada simplejo del complejo de entrada. Se hace que cada copia comunique su vista y obtenga las vistas de los demás en el orden especificado por cada escenario, por supuesto, mediante el protocolo de comunicación dado. Al final de este procedimiento cada copia de proceso obtiene nuevas vistas; con estos procesos se arman nuevos simplejos y con estos a su vez se integra el complejo de protocolo de esta ronda de comunicación.

`Model` notifica a las vistas llamando a sus métodos `update()` para indicarles que el compejo de protocolo ya ha sido producido. En este método las vistas obtienen el complejo de protocolo llamando el método `getProtocolComplex()` y cada una representará a su manera el complejo de protocolo, ya sea en forma de una visualización en 3D o en texto.

De la anterior descripción observamos que el comportamiento de los procesos y la generación de escenarios depende del protocolo de comunicación seleccionado. Los escenarios cuando el protocolo es memoria *immediate snapshot* van a ser diferentes de aquellos producidos cuando el protocolo es paso de mensaje con tolerancia de t fallas, por ejemplo.

Por otra parte, si el protocolo es memoria compartida, los procesos pueden leer la memoria de diferentes formas, por ejemplo, atómicamente mediante operaciones *snapshot* o leer registro por registro de forma asíncrona. Si el protocolo fuera paso de mensajes, los procesos necesitarían

efectuar operaciones para enviar y recibir mensajes.

4.2.1. Diseño de los procesos

Como vimos en la sección 2.1, cada proceso p_i se puede caracterizar por un par (i, v_i), en donde i corresponde al *id* del proceso y v_i a su vista. Por lo tanto en el programa cada proceso se representa como una clase `Process` que posee los atributos `id` y `view`. La idea es que si el sistema distribuido consta de n procesos, cada proceso sea identificado por un entero $id \in \{0...n-1\}$.

También se asigna a cada proceso un atributo `name`. Conceptualmente el *id* de un proceso y este atributo representan lo mismo, sin embargo, en el código se hace esta distinción para permitir flexibilidad en la forma en que los usuarios nombran los procesos, ya que se ha visto que en algunas publicaciones los procesos son nombrados de diferentes maneras, por ejemplo: `p`, `q`, `r`,...; `a`, `b`, `c`,...; etcétera. Este atributo solamente funciona como un *alias* para el proceso, pero el que es realmente importante para la generación de complejos simpliciales es el atributo `id`. También, si el usuario no especifica los valores de `name` para cada proceso (mediante los controles en la interfaz de usuario) a estos por defecto se les asigna el valor `id` de cada proceso.

Para simular la comunicación entre procesos, como primer intento se trató de definir métodos de comunicación en la clase `Process`. Estos métodos podían ser para leer o escribir memoria compartida, o bien, envío y recepción de mensajes.

Para llevar esta idea a cabo, se obtuvo el diseño mostrado en la Figura 4.3. La clase `Process` es una clase abstracta que define los atributos esenciales de un proceso y la subclase `ISProcess` agrega los métodos para permitir la comunicación en el contexto de un modelo *immediate snapshot*. Para otros modelos se tendrían que agregar subclases similares a ésta.

Desafortunamente se tuvieron problemas en la implementación de este diseño, ya que de acuerdo al paso 1.7. del caso de uso 1, el complejo inicial se debe construir antes de especificar el protocolo de comunicación que se usará para generar los complejos de protocolo. Por lo tanto, en este punto del flujo de programa no se puede saber que subclase de la clase `Process` se usará ni tampoco se pueden usar instancias de la clase `Process` ya que es abstracta.

Como alternativa se pensó en aplicar el patrón de diseño conocido como *Estrategia* [16], el cual define una familia de algoritmos, encapsula cada uno de estos y los hace intercambiables. En este caso el algoritmo es el grupo de métodos de comunicación, el cual varía de acuerdo al modelo de computación distribuida que el usuario escoge.

Sin embargo, se observó que el único contexto en donde los procesos deben de comunicarse es durante las rondas de comunicación que son simuladas por el generador, por lo tanto, se

consideró más simple que las clases especializadas que representan los diferentes protocolos de comunicación provean los métodos para que los procesos se comuniquen sus vistas. La desventaja de este diseño es que puede ser un poco confuso el hecho de los procesos no son los que se comunican activamente, sino que es la clase de comunicación la que provee los métodos. Pero esto no afecta el funcionamiento del programa y simplifica más el código. En la Figura 4.4 mostramos el diagrama UML que muestra el diseño final de la clase `Process` y en la clase `ImmediateSnapshot` mostrada en la Figura 5.10 se observa cómo ésta ofrece los métodos de comunicación `write()` y `snapshot()`.

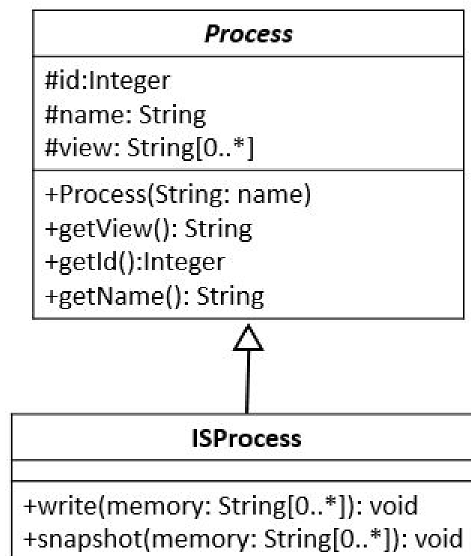

Figura 4.3: Diagrama UML simplificado que muestra como se concibió incialmente el diseño de los procesos.

La clase `Process` tiene un atributo llamado `chromatic`, el cual denota si el proceso va a formar parte de un simplejo en su representación cromática o no cromática. La representación cromática o no cromática de un proceso solamente afecta la manera en que se representa la vista del proceso (en particular, afecta la forma en que el método `getView()` devuelve la representación de la vista del proceso). Si el proceso tiene `chromatic=true`, entonces el valor devuelto por su método `getView()` tendrá el formato descrito en la sección 2.1. Si el proceso tiene `chromatic=false`, el formato es casi igual, excepto que no se utiliza el símbolo "`-`" ni ningún otro para denotar que la vista de los demás procesos es desconocida. Esto es porque en la representación cromática los procesos tienen asignado un registro identificado con su `id` para comunicar su vista (como en la memoria compartida, la i-ésima entrada esta reservada para que el proceso con $id = i$ escriba su vista y si esta es desconocida se usa el símbolo ya mencionado),

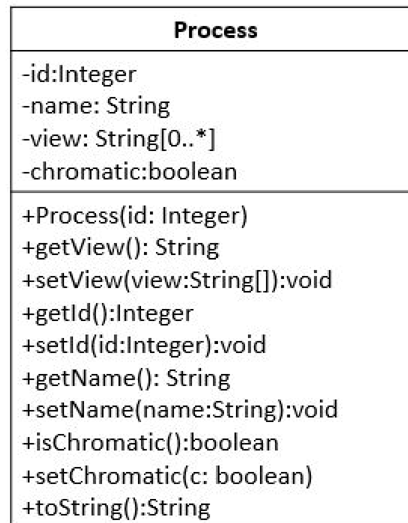

Figura 4.4: Diagrama UML simplificado que muestra el diseño final de la clase `Process`. Por simplicidad, algunos métodos implementados en el código del programa se omiten en este diagrama.

pero en el caso de la representación no cromática la noción de `id` se omite: los procesos solo se identifican con el valor de sus vistas, por lo tanto en este caso simplemente omitimos cualquier símbolo que indique cualquier noción asociada al `id` de un proceso (una comparación de las vistas cromáticas y no cromáticas de los procesos la podemos ver en las Figuras 2.4 y 2.5, en donde se muestran los mismo complejos simpliciales pero en su representación cromática y no cromática, respectivamente).

Finalmente el método `toString()` devuelve la representación en texto del proceso en el formato (i, v_i), donde i es el `id` del proceso y v_i es su vista.

Esta clase se encuentra ubicada en el paquete `unam.dcct.topology`.

4.2.2. Diseño de complejos simpliciales y simplejos

Clase `Simplex`

En la Figura 4.5 mostramos el diseño de las clases que representan a los complejos simpliciales y a los simplejos. Estas clases también se encuentran ubicadas en el paquete `unam.dcct.topology`.

La clase `Simplex` posee un atributo llamado `parent`, el cual hace refencia al simplejo a partir del cual éste fue producido durante una ronda de comunicación [2]. Hacer que cada simplejo

[2] A manera de ilustración, considerese un simplejo de dimensión 2 (gráficamente, un triángulo). Durante una ronda de comunicación del protocolo *immediate snapshot* éste se subdivide en varios "subtriánglos". El simplejo padre de cada uno de estos es el representado por el triángulo que se subdividió.

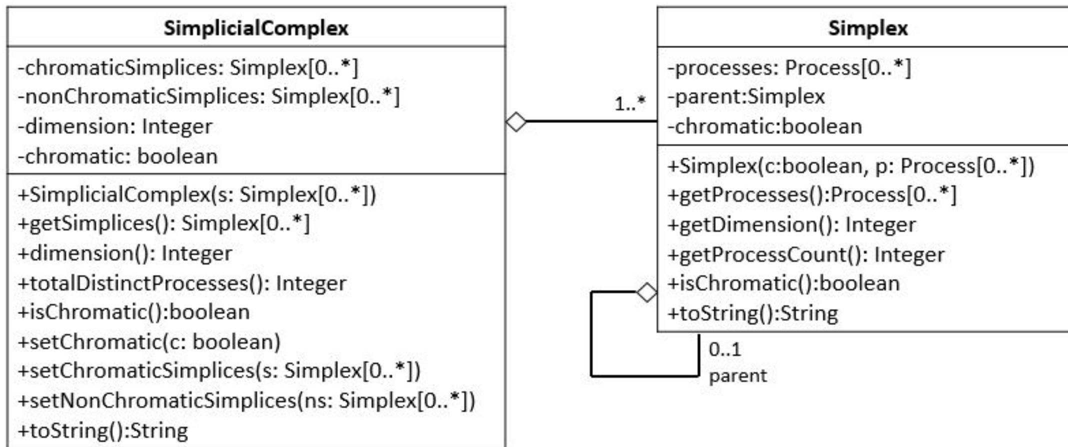

Figura 4.5: Diagrama UML de clases que muestra las clases que representan al complejo simplicial, al simplejo y sus relaciones.

mantenga una referencia a su simplejo "padre" resultó útil durante el proceso de transformación a una representación gráfica. Esto lo revisaremos con más detalle en el Capítulo 5.

La clase `Simplex` también puede ser representada de forma cromática y no cromática (requerimiento 1.3.5.). En el programa un simplejo cromático es aquel cuyos procesos son cromáticos, de igual manera para un simplejo no cromático. Un simplejo cromático en comparación con su representación no cromática no solamente difiere en la "cromaticidad" de sus procesos, sino también en su dimensión. Por ejemplo, en la Figura 2.4 consideremos el simplejo que forma parte del complejo cromático de la ronda 1 representado por el triángulo cuyos vértices tienen las etiquetas `(0,1,-),(0,1,-) y (0,1,2)`. La representación no cromática de este simplejo es la arista del complejo de la ronda 1 en la Figura 2.5 cuyos vértices tienen las etiquetas `(0,1) y (0,1,2)`. Podemos ver que el mismo simplejo cuando es visto cromáticamente su dimensión es 2, mientras que cuando es visto no cromáticamente su dimensión es 1. Esto es porque en la representación no cromática los procesos únicamente se identifican por los valores de sus vistas, por lo tanto el vértice con la etiqueta `(0,1)` representa cualquier número de procesos que tengan esa vista, por lo tanto éste representa a los procesos con las vistas (0,1,-) en la versión cromática.

Para facilitar la implementación de la clase `Simplex`, se decidió que si se desea representar un simplejo en la representación cromática y no cromática, se tienen que crear dos instancias diferentes de la clase `Simplex` usando el método constructor `Simplex(c:boolean, p:Process[0..*])`, en el cual se especifica si el simplejo será cromático o no con el parámetro `c` y también se pide la lista de procesos que conformarán el simplejo. Una vez construida

la instancia del simplejo, ésta no puede cambiar su representación cromática, es por esto que no se ofrece un método `setChromatic` como en el caso de la clase `Process`.

El método `toString()` devuelve una representación en notación de conjuntos del simplejo. Para construirla, se usan los métodos `toString()` de los procesos contenidos.

Clase `SimplicialComplex`

El diseño de la clase `SimplicialComplex` también se vio influído por el requerimiento 1.3.5. Para el caso de un complejo simplicial, solo es necesario tener una instancia de la clase `SimplicialComplex` para representarlo, y podemos cambiar su representación cromática y no cromática en tiempo de ejecución simplemente usando el método `setChromatic()`. El valor del atributo `chromatic` afecta el valor que devuelve el método `getSimplices()`, devolviendo la lista `chromaticSimplices` o `nonChromaticSimplices` según sea el caso. Es fácil intuir que `chromaticSimplices` contiene solo simplejos con `chromatic=true` y la lista `nonChromaticSimplices` contiene las versiones no cromáticas de estos mismos simplejos.

Por defecto, al momento de su creación, una instancia de la clase `SimplicialComplex` es cromática. Para obtener la versión no cromática de este complejo, es necesario dotar a esta instancia de una lista de simplejos no cromáticos que corresponden a las versiones de los simplejos cromáticos provistas en el método constructor. Para esto se ofrece el método `setNonChromaticSimplices`. Por simetría, también se ofrece el método `setChromaticSimplices` [3]. El método `toString()` devuelve una representación en notación de conjuntos del complejo simplicial, la cual se construye usando los métodos `toString()` de los simplejos contenidos.

4.2.3. Diseño de los protocolos de comunicación

En la Figura 4.6 mostramos el diagrama de clases que ilustra la estructura del generador de complejos de protocolo. Estas clases se encuentran repartidas en los paquetes `unam.dcct.model` y `unam.dcct.model.immediatesnapshot`.

Los protocolos de comunicación se representan por la clase abstracta `CommunicationProtocol`. Ésta define un método `executeRound(baseComplex:SimplicialComplex)`, el cual genera un complejo de protocolo con base en `baseComplex`. Este parámetro es un complejo inicial si es que se va a generar el complejo de protocolo de la primera ronda; o es el complejo de protocolo de la

[3] Considero que esta solución no es muy elegante y "ensucia" un poco la interfaz de la clase `SimplicialComplex`, pues los clientes de ésta solo deberían de proveer la lista de simplejos en el constructor sin preocuparse si son cromáticos o no. Esto se discutirá con mayor detalle en la sección 5.4.4

última ronda ejecutada, si es que se va a generar un complejo para una nueva ronda. A grandes rasgos, este método primero genera todos los posibles escenarios de ejecución de una ronda del comunicación (cada escenario representado por la interfaz Scenario), después toma cada conjunto de procesos contenidos en cada simplejo en baseComplex y ejecuta una simulación de ejecución de cada escenario en la que participan los procesos; en otras palabras, hace que estos se comuniquen en el orden especificado por cada escenario. Como resultado, se obtienen nuevos procesos con nuevas vistas y finalmente estos se usan para construir el nuevo complejo de protocolo, el cual es devuelto por este método.

El conjunto de escenarios se obtiene llamando al método createScenarioGenerator(), el cual devuelve un objeto que implementa la interfaz Iterable<Scenario>. La idea es permitir recorrer sobre el conjunto de escenarios y simular la ejecución de cada uno mediante una llamada al método execute(), el cual recibe los procesos originales contenidos en un simplejo en baseComplex y devuelve el conjunto de procesos con nuevas vistas. Las interfaces Iterable e Iterator son provistas por la biblioteca de clases de Java y representan una aplicación del patrón de diseño *Iterador* [16].

Como vemos en el diagrama, el método createScenarioGenerator es declarado como abstracto. Esto quiere decir que las clases que deriven de CommunicationProtocol son las responsables de dar la implementación concreta de este método.

Conceptualmente la cantidad y caracteristicas del conjunto de escenarios de una ronda de comunicación de un protocolo están sujetas a las características y especificaciones de éste. Por lo tanto se debe dar una clase especializada por cada uno, las cuales también deben extender la clase abstracta CommunicationProtocol; así mismo se deben proveer clases que implementen el comportamiento definido en las interfaces Iterable<Scenario>, Iterator<Scenario> y Scenario. Este diseño está basado en el patrón *método fábrica* [16] y el beneficio resultante por su aplicación en nuestro programa es la fácil integración de la generación de complejos a partir de nuevos protocolos. Por ejemplo, en el diagrama tenemos la clase ImmediateSnapshot, la cual provee su propia manera de generar sus posibles escenarios al implementar el método createScenarioGenerator(), para lo cual también se tuvieron que crear las clases ISScenarioGenerator, ImmediateSnapshotIterator y ImmediateSnapshotScenario, las cuales implementan las interfaces ya descritas.

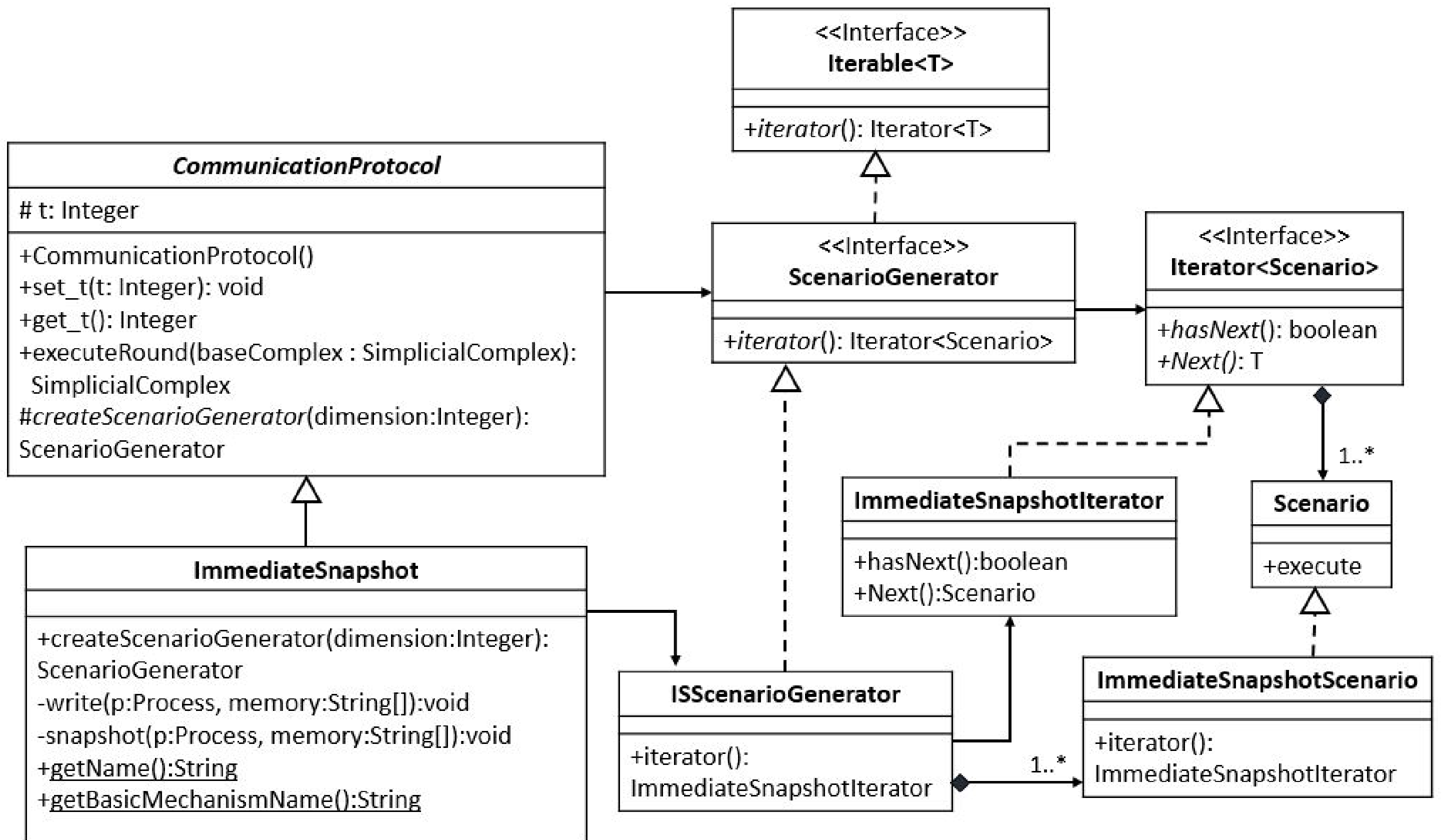

Figura 4.6: Diagrama UML que muestra las clases que representan los protocolos de comunicación implementados en el programa.

4.2.4. Diseño de objetos geométricos

Las clases que a continuación se describen representan las visualizaciones geométricas de las clases `SimplicialComplex`, `Simplex` y `Process`, las cuales son `GeometricComplex`, `Face` y `Vertex`, respectivamente. El diagrama de la Figura 4.7 muestra la relación entre todas estas clases, las cuales residen en el paquete `unam.dcct.view.geometry`.

En un principio se pensó en dotar a las clases `SimplicialComplex`, `Simplex` y `Process` con las capacidades para que se pudieran dibujar a sí mismas, es decir, que incluyeran métodos similares a los que provee la interfaz `Geometry` que vemos en el diagrama. Sin embargo, esto complicaba mucho su diseño, además de que es contrario al *principio de única responsabilidad* [28] en diseño orientado a objetos. Para evitar que cada una de estas clases tuviera dos responsabilidades (la de mantener la información relacionada con sistemas distribuidos y la de generar sus representaciones geométricas), la responsabilidad de las representaciones geométricas se le asignó a las clases discutidas en esta sección.

La interfaz `Geometry` representa a cualquier objeto que puede ser representado geométricamente. Está provee operaciones que permiten dibujar un objeto geometrico por alguna biblioteca de visualización. Gracias a esta interfaz, la clase `jRealityView` es capaz de dibujar complejos simpliciales o simplejos indistintamente, ya que sus representaciones geométricas `GeometricComplex` y `Face` la implementan.

Las instancias de `GeometricComplex` se construyen recibiendo como parámetros instancias de la clase `SimplicialComplex`, cuyos simplejos se utilizan para crear las caras de la representación geométrica del complejo, es decir, las instancias de la clase `Face`; a su vez, cada una de éstas construye los vértices que representan a cada proceso contenido en el simplejo.

Estas clases también poseen el atributo `chromatic` ya que necesitan representar gráficamente los complejos y simplejos cromáticos y no cromáticos.

Para permitir la representación cromática y no cromática de un complejo simplicial, la clase `GeometricComplex` mantiene internamente dos listas: `chromaticFaces` y `nonChromaticFaces`.

Las caras también mantienen la misma relación padre-hijo que tienen los simplejos: cada instancia de la clase `Face` posee un atributo `parent` que almacena una referencia a su cara padre. Esta relación permite calcular las coordenadas de los vértices del complejo simplicial. Esto se explica con mayor detalle en la sección 5.5.3.

Los objetos `ChromaticBehaviour` y `NonChromaticBehaviour` se encargan de calcular las coordenadas de los vértices de una cara y de asignarles colores a los mismos. Se usa uno u otro dependiendo del valor del atributo `chromatic` de la cara. Estos objetos son representados me-

diante la interfaz `ChromaticityBehaviour`. Este diseño es una aplicación del patrón *Estrategia* [16].

4.3. Conclusión

En el presente capítulo diseñamos la arquitectura del programa basándonos en el patrón arquitectónico Modelo-Vista-Controlador. Identificamos los componentes `View` y `CommunicationProtocol` como aquellos más suceptibles a ser modificados en el futuro y para ello producimos diseños débilmente acoplados y extensibles. Con esto satisfacemos el requerimiento no funcional 2.. Finalmente describimos el diseño de las clases que representan los principales objetos involucrados en el programa. A manera de resumen a continuación los listamos.

1. Relacionados con el aspecto de topología combinatoria de computación distribuida: `SimplicialComplex`, `Simplex` y `Process`. En el programa están en el paquete `unam.dcct.topology`.

2. Relacionados con la generación de complejos de protocolo: `CommunicationProtocol`, `ImmediateSnapshot`, `Scenario` e `ImmediateSnapshotScenario`, entre otros. Estos residen en los paquetes `unam.dcct.model` y `unam.dcct.model.immediatesnapshot`.

3. Relacionados con la representación gráfica de los objetos de topología combinatoria: `Geometry`, `GeometricComplex`, `Face` y `Vertex`. Estos residen en el paquete `unam.dcct.view.geometry`.

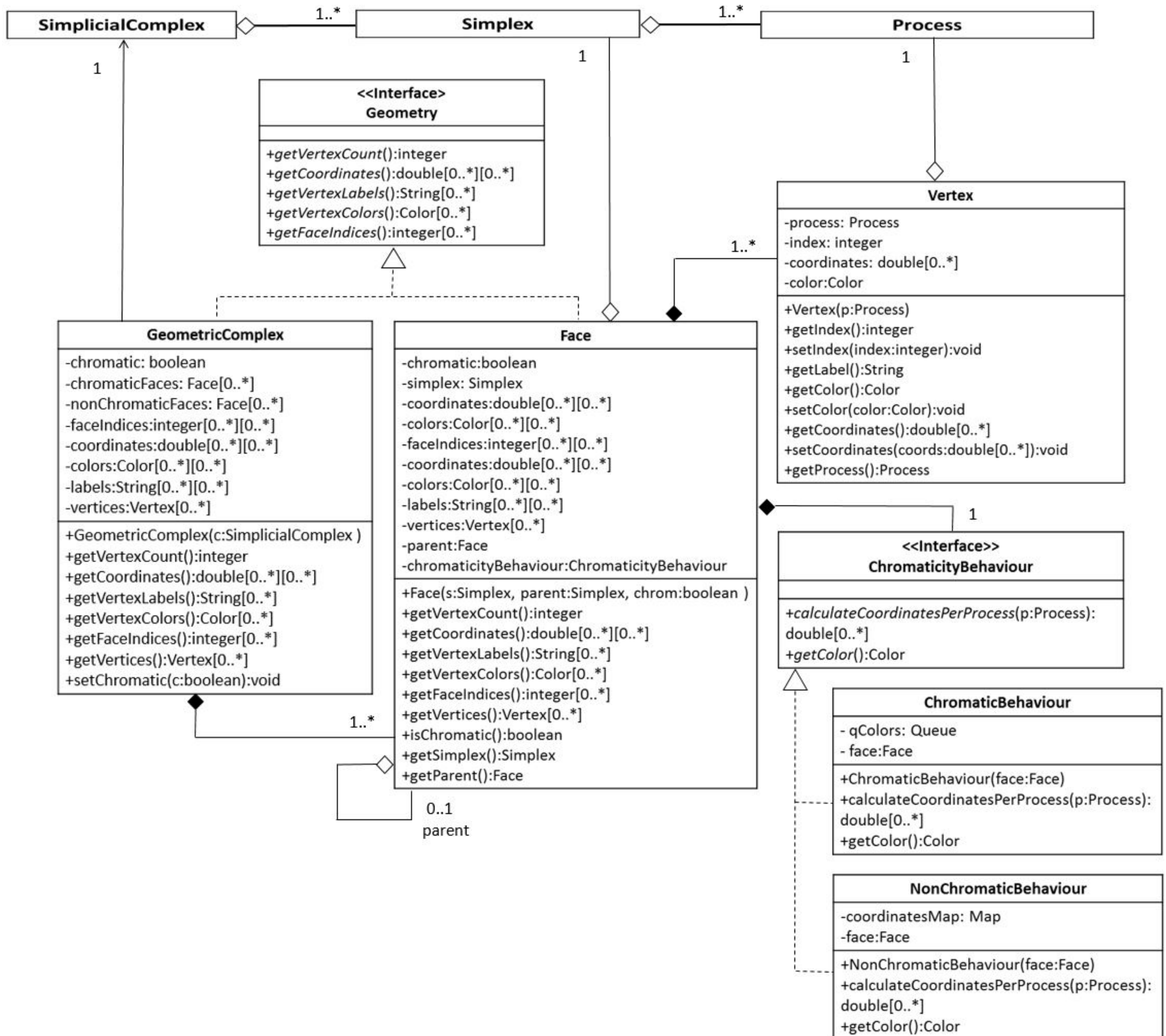

Figura 4.7: Diagrama UML que muestra las clases que representan las representaciones geometricas de los objetos que maneja el programa.

Capítulo 5

Implementación

En este capítulo describimos los principales problemas que se tuvieron al implementar los requerimientos descritos en el Capítulo 3 y también discutimos sus soluciones, apegandonos al diseño y la arquitectura descritos en el Capítulo 4.

Es importante resaltar que las descripciones textuales y en diagramas dadas son simplificadas y resumidas respecto a cómo es en realidad el código del programa: éstas solo ilustran lo esencial, pues sería complicado y no aportaría mucho describir con exactitud todos los detalles de programación. Para eso el lector puede referirse directamente al código del programa, el cual se procuró que estuviese escrito de forma clara, además de estar documentado y disponible al público. De hecho este capítulo (y el anterior) puede servir como una guía para su mejor comprensión.

5.1. Inicio del programa

El enfoque que tomamos para explicar la implementación de los mecanismos que hacen que el programa inicie es explicando primero la perspectiva dinámica, es decir, explicamos la secuencia de interacciones entre objetos y clases que se da en tiempo de ejecución; despúes describimos la perspectiva estática: explicamos la estructura y las responsabilidades de cada una de las clases.

En la Figura 5.1 mostramos el diagrama de secuencia[1] que muestra la interacción entre las principales clases y objetos involucrados en el momento en que el usuario ejecuta el programa.

La clase `DCCT_App`[2], que yace en el paquete `unam.dcct.main`, es el punto de partida en donde se inicia la ejecución del programa. Esta invoca el método estático `getInstance()` de la clase

[1]Diagramas generados en `www.websequencediagrams.com`

[2] *Distributed Computing through Combinatorial Topology Application*

`jRealityView` para obtener una referencia a su única instancia, ya que esta clase esta diseñada de acuerdo al patrón *Singleton*. Como es la primera vez que se ejecuta el programa, entonces internamente invoca a su método constructor para crear esta instancia, la cual a su vez intenta obtener la única instancia de la clase `Model`. Con esta instancia `jRealityView` puede registrarse en el modelo a través del método `registerView(view:View)` que éste ofrece, de tal forma que así se podrá actualizar esta vista cuando cambie el estado del modelo. Este mecanismo está basado en el patrón de diseño *Observador* [16].

Después se invoca el método `start()` de la clase `jRealityView`, el cual invoca al método `configViewer()` que crea instancias de las clases `SceneGraphComponent` y `JRViewer`, que son parte de la biblioteca jReality.

`JRViewer` es el visualizador ofrecido por jReality y provee la interfaz gráfica que se muestra al usuario, la cual incluye el área en la que se despliegan los gráficos generados y también permite integrar controles de usuario personalizados mediante un mecanismo de complementos.

`SceneGraphComponent` es una estructura de datos que representa el objeto 3D que se despliega en la visualización y ofrece métodos para poder configurarlo añadiendo efectos tales como colores, iluminación, interacción, etcétera. [3]. Este es el objeto que produce las visualizaciones de nuestro programa.

Después se obtiene una instancia de la clase `SimplicialComplexPanel`, la cual representa un panel que agrupa el conjunto de controles de usuario que desarrollamos. Este objeto se añade en forma de plugin al visualizador `JRViewer` a través de una llamada al método `registerPlugin(scPanel)`. El mismo procedimiento se hace con la consola, representada por la clase `SCOutputConsole`; y también con un panel llamado `ContentAppearance` (provisto por jReality) que permite personalizar las propiedades de color, tamaño y transparencia de los vérties, etiquetas y caras de los objetos visualizados. La consola `SCOutputConsole` también implementa la interfaz `View`, por lo tanto también se registra en la clase `Model` cuando es creada (al igual que la clase `jRealityView`) para ser notificada cuando ésta actualize su estado.

Indicamos que el objeto de tipo `SceneGraphComponent` (identificado por la variable `sgc`) va a ser el contenido principal desplegado en la visualización mediante la llamada al método `setContent(sgc)`; y finalmente se invoca al método `startup()` de la clase `JRViewer` para que se inicie el visualizador de jReality, mostrando la pantalla inicial del programa al usuario (Figura 5.2).

[3]Para más información, se puede consultar la documentación de la API de jReality en `http://www3.math.tu-berlin.de/jreality/api/`

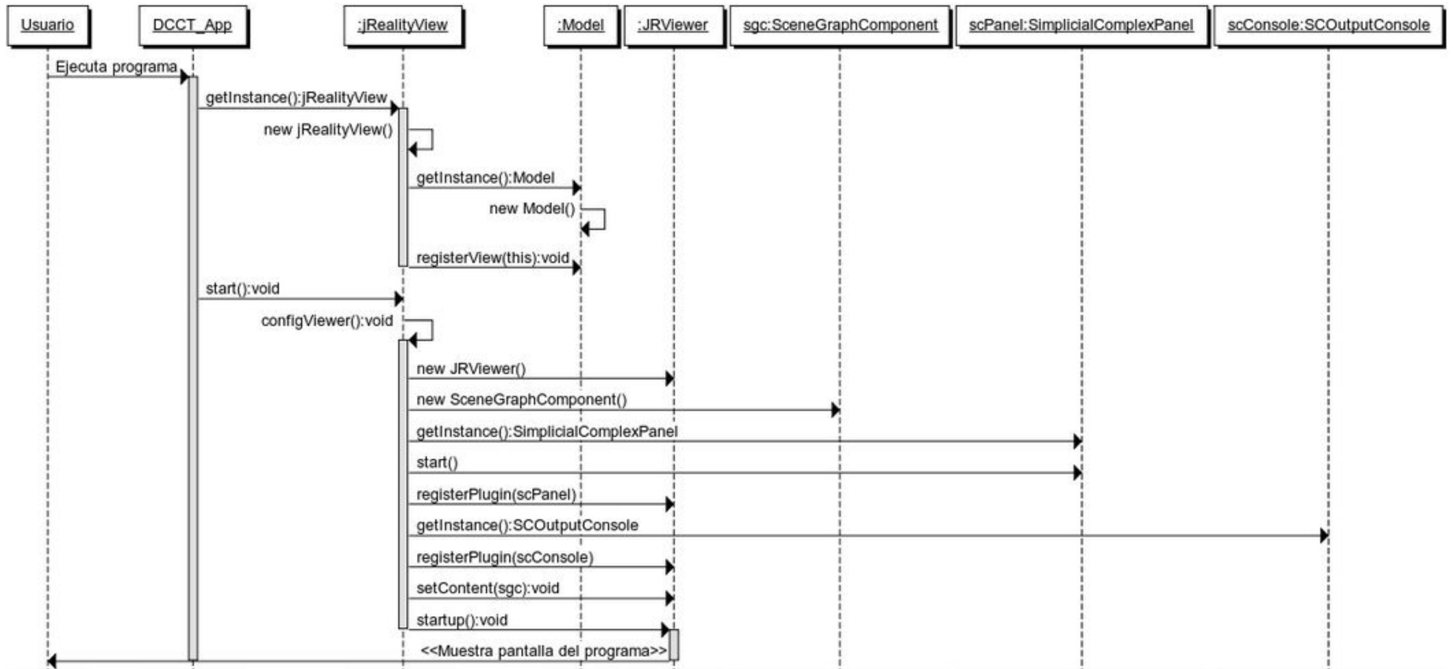

Figura 5.1: Diagrama de secuencia simplificado que muestra la interacción entre los principales objetos y clases involucrados en el proceso de iniciar el programa.

5.2. Construcción del complejo inicial

Ahora describimos la secuencia de interacciones que se llevan a cabo para hacer que el usuario pueda generar y desplegar una visualización de un complejo simplicial inicial, tal como se describe en el caso de uso 1.7.. Pero para comprender mejor este proceso es conveniente primero describir el panel “Simplicial Complex Panel”.

5.2.1. “Simplicial Complex Panel”

En este panel se agrupan los controles que permiten al usuario manejar la generación y visualización de complejos simpliciales. Mediante este control de usuario implementamos los requerimientos funcionales comprendidos entre el 1.1. y el 1.3..

Para llevar a cabo la implementación de estos requerimientos resultó conveniente pensar en ellos como una secuencia de pasos para llevar a cabo el proceso de generar complejos simpliciales, por lo tanto los controles se implementaron como un “asistente” o, como comúnmente se conoce en inglés, un *wizard*, en donde cada paso muestra un grupo distinto de controles que solicitan al usuario introducir cierta información. Una vez introducida la información, el programa realiza ciertas acciones concernientes al paso actual, como por ejemplo validar que los datos introducidos

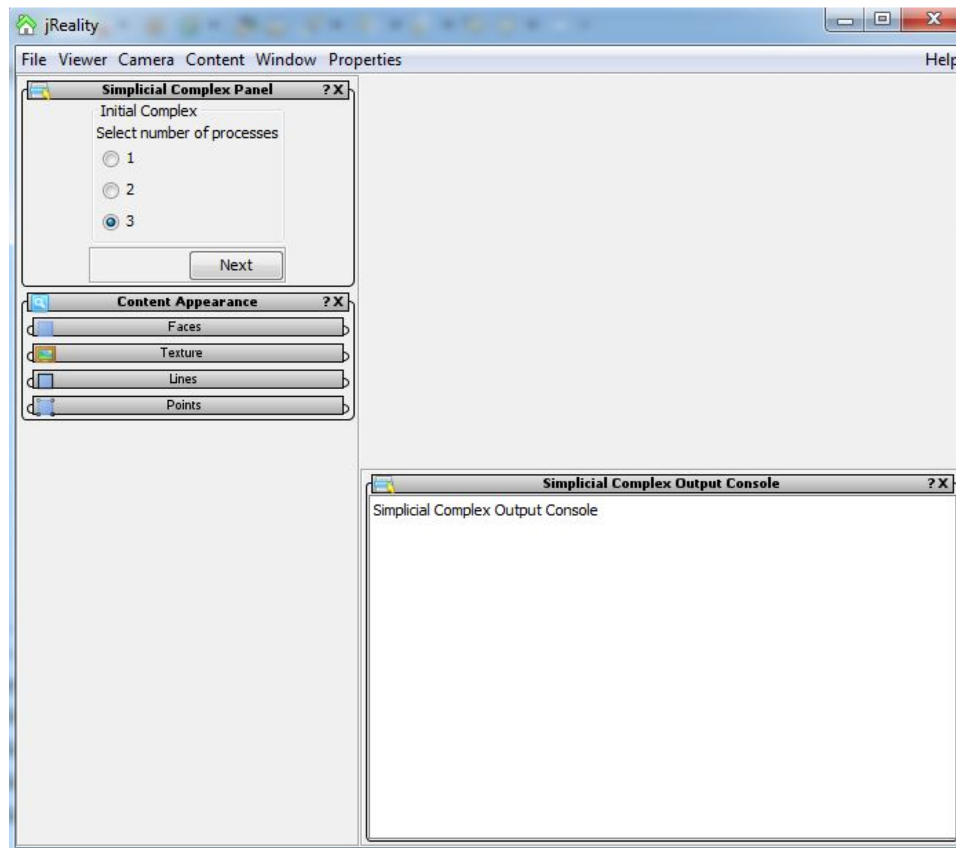

Figura 5.2: Pantalla del programa al iniciar. El panel "Simplicial Complex Panel" contiene los controles para controlar la generación de complejos simpliciales. El panel "Content Appearance" permite personalizar el objeto desplegado en la visualización, y el panel "Simplicial Complex Output Console" muestra información en texto del complejo simplicial mostrado en pantalla.

son correctos, o efectuar algún procesamiento, para después mostrar los controles del siguiente paso.

Por ejemplo, en nuestro programa los controles descritos en el requerimiento 1.1. (Especificar número de procesos en el simplejo) se agrupan en el primer paso, cuyos controles se observan en la Figura 5.2. Una vez que el usuario selecciona el número de procesos, el botón "Next" se habilita, indicando que el usuario puede pasar al siguiente paso.

En el segundo paso (Figura 5.3) se le pide al usuario seleccionar el color con el que se pintará el vértice que representa a cada proceso en el complejo simplicial y también que introduzca el nombre con el que se identificará cada proceso. Los nombres y colores que se muestran en la Figura son los que el programa selecciona por defecto. Una vez introducidos los datos, el usuario puede presionar el botón "Generate", el cual valida que los datos son correctos (requerimiento 1.1.) y después se genera el complejo simplicial inicial con los datos previamente proporcionados. También se muestra un botón "Back" para permitir al usuario corregir datos introducidos en el paso anterior.

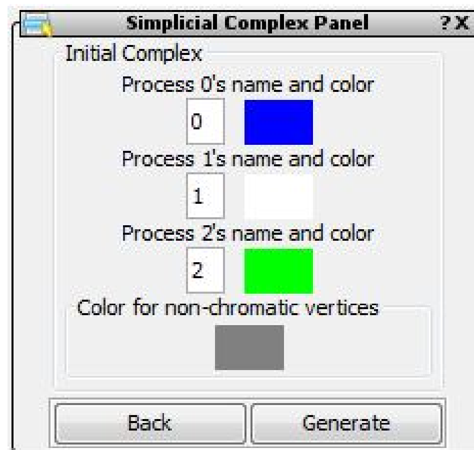

Figura 5.3: Segundo paso para generar el complejo simplicial inicial. Se solicita al usuario elegir el color con el que se pintará cada vértices que representa a cada proceso y también se pide introducir el nombre de cada proceso. Los valores mostrados en la imagen son los que el programa da por defecto.

Implementación

El código del programa que hace funcionar este panel esta escrito en la clase `SimplicialComplexPanel`, en la clase `Step` y en sus subclases. Estas clases yacen en el paquete `unam.dcct.view.UI`.

Clase `SimplicialComplexPanel`

La clase `SimplicialComplexPanel` representa el panel principal que contiene todos los controles de usuario mostrados en cada paso. En la Figura 5.4 mostramos su diagrama de clases y un esquema del diseño visual del panel, así como la relación de cada atributo de la clase con cada componente visual.

Esta clase deriva de la clase `JPanel`, la cual es parte de la biblioteca de componentes de interfaz de usuario Swing[4] de Java. Esta clase representa un contenedor genérico que sirve para agrupar otros controles. Nuestra clase contiene dos paneles: `pContent` y `pButtons`. El primero sirve como un área en donde se insertan los controles del paso actual, representado por `currentStep`. El segundo sirve para colocar los botones de navegación “Next” y “Back”.

El método `actionPerformed()` se encarga de manejar los eventos generados cuando el usuario presiona los botones “Next” y “Back”, es decir, cambiar al siguiente paso o regresar al paso anterior.

Como solamente puede existir un solo panel de este tipo aplicamos el patrón *Singleton* en el diseño de esta clase, por lo tanto el constructor de esta clase se marca como privado, se incluye un atributo estático privado `instance` que contiene la única instancia de esta clase y se implementa un método estático `getInstance()` que devuelve el valor de `instance`.

Clase `Step`

La clase abstracta `Step` (Figura 5.5a) representa las propiedades y operaciones comunes a todos los pasos. La idea es que cada clase que representa un paso en concreto herede las propiedades y operaciones de `Step`.

El atributo `scPanel` es una referencia al objeto `SimplicialComplexPanel`, de esta forma cuando cada paso es visitado este puede insertar sus controles en el panel `pContent` y puede manipular los botones del panel `pButtons`. También tiene una referencia a la única instancia de la clase `Model`, ya que cada paso puede manipular el modelo si así lo requiere.

El atributo `lbTitle` representa el título que se muestra en la parte superior del panel `pContent` y puede ser cambiado por cada paso.

Los métodos `goBack()` y `validateAndExecute()` sirven para llevar a cabo la navegación entre pasos. Estos son invocados por el método `actionPerformed()` de la clase `SimplicialComplexPanel` cuando el usuario presiona los botones “Back” y “Next”, respectivamente.

[4] Para más información consultar `http://docs.oracle.com/javase/tutorial/uiswing/start/about.html`

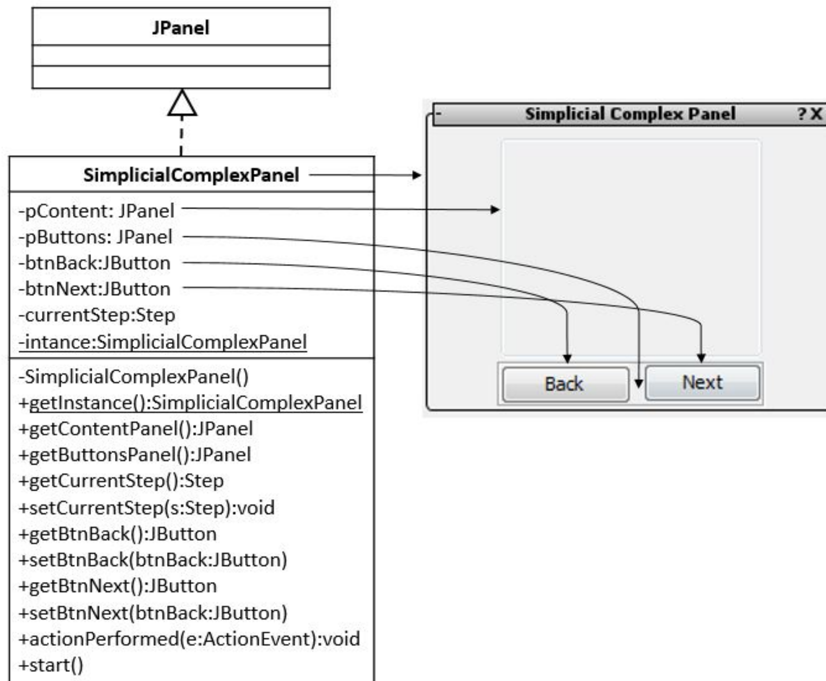

Figura 5.4: Diagrama de clase de `SimplicialComplexPanel` y su representación visual.

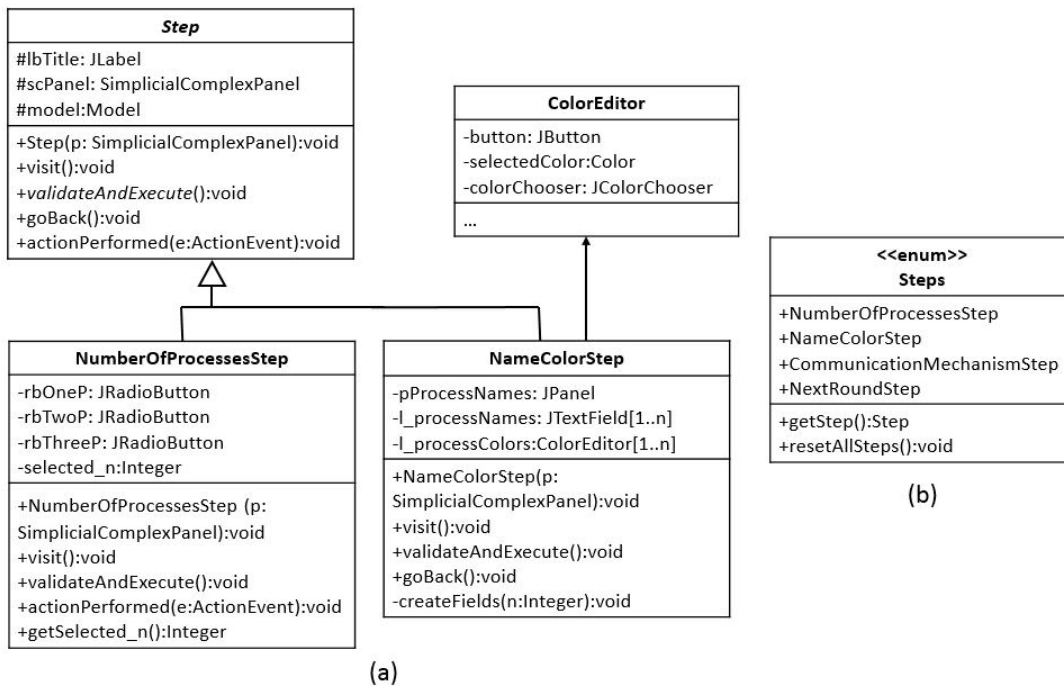

Figura 5.5: (a) Diagrama de clase de `Step` y las subclases `NumberOfProcessesStep` y `NameColorStep`. (b) Enumeración `Steps` que mantiene las referencias a las instancias de todos los pasos.

El método `validateAndExecute()` es definido en la clase `Step` como un método abstracto, es decir, las clases que heredan de `Step` son forzadas a implementar este método, ya que cada paso debe definir las acciones que se deben ejecutar una vez que el usuario introduce los datos pedidos y presiona el botón "Next". De igual forma, en este punto es donde cada paso puede implementar lógica de validación de datos. También en este método cada paso puede decidir si puede trasladar al usuario al siguiente paso, y decidir cual ese siguiente paso. Como ejemplo en el Listado 5.1 mostramos el código del método `ValidateAndExecute()` del paso `NumberOfProcessesStep`.

Listado 5.1: `NumberOfProcessesStep.validateAndExecute()`

```
1  public void validateAndExecute(){
2    Step next = Steps.NameColorStep.getStep();
3    scPanel.setCurrentStep(next);
4    next.visit();
5  }
```

En este paso este método no efectua ningún procesamiento más que solo llevar al usuario al siguiente paso. El siguiente paso es denotado como el valor `NameColorStep` de un tipo `enum`[5] llamado `Steps`. Así como este paso, las referencias a los demás pasos estan almacenadas como valores de `Steps` (Figura 5.5b). La motivación de esto es para soportar la navegación entre los pasos del asistente y la preservación del estado de los controles de cada paso a lo largo de la navegación. Por ejemplo, si el usuario elige "dos procesos" en el paso `NumberOfProcessesStep`, avanza al siguiente paso, y decide regresar al paso anterior, el control para elegir el número de pasos debe de mostrar la última elección hecha por el usuario, en este caso "dos procesos".

En un principio se pensó en implementar cada paso como un *singleton* y que cada instancia de cada paso fuera almacenada en un diccionario que iba a ser accedido como un atributo estático de la clase `Step`. Así, cuando un paso requiriese mandar al usuario al siguiente o al anterior paso, este iba a obtener la referencia del paso a través del diccionario, para después visitarlo. Después se vió que implementar cada paso como un *singleton* no era apropiado ya que también se tiene la necesidad de reiniciar todo el asistente cuando el usuario deseara iniciar de nuevo el proceso de crear un complejo inicial, por lo tanto nuevas instancias de todos los pasos tendrían que ser creadas, lo cual es contrario a la intención del patrón *singleton* la cual es mantener una sola y única instancia a lo largo de la ejecución del programa. De igual forma,

[5]En el lenguaje Java un tipo `enum` es un tipo especial de dato que le permite a una variable tener un conjunto predefinido de valores constantes. La variable debe se igual a uno de los valores que han sido predefinidos para esta. Para más información consultar `https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html`

vimos que resultó más adecuado asignar la responsabilidad de mantener las instancias de cada paso al tipo `enum Steps` que a un diccionario.

`Steps` también ofrece el método estático `resetAllSteps()`, que crea nuevas instancias de cada paso (para reestablecer el estado inicial de todos los controles), lo cual permite el reinicio del asistente. Este método se invoca cuando el usuario presiona el botón “Start over” que ofrecen otros pasos que más adelante verémos. Esto satisface el requerimiento 1.3.6..

Cuando se navega de un paso hacia otro decimos que se “visita” el paso de destino, lo cual significa que los controles del paso actual que se muestran en el panel principal se reemplazan por los controles del nuevo paso. Esto se realiza invocando el método `visit()` del nuevo paso.

El método `goBack()` permite a los pasos mandar al usuario al paso anterior. Este método no se define abstracto ya que algunos pasos no tienen un paso predecesor, como es el caso del primer paso, por lo tanto no se requiere que todos los pasos lo implementen.

Clase `NumberOfProcessesStep`

Este es el primer paso del asistente. Su función es mostrar los controles que permiten al usuario elegir el número de procesos que contendrá el complejo simplicial (requerimiento 1.1.). Esta clase hereda todas las propiedades y métodos de la clase `Step`.

Los atributos `rbOneP`, `rbTwoP` y `rbThreeP` representan los botones de opción que permiten elegir al usuario entre uno, dos o tres procesos. Por defecto, el valor seleccionado de procesos es 3. Cada vez que el usuario elige un valor, el método `actionPerformed()` guarda el valor elegido en el atributo privado `selected_n`, y este se puede obtener por otros pasos mediante el método `getSelected_n()`. El método `validateAndExecute()` solamente lleva al usuario al siguiente paso, `NameColorStep`.

Clase `NameColorStep`

Es el segundo paso del asistente. Con base en la opción seleccionada en el paso anterior (la cual obtiene mediante el método `getSelected_n()`), muestra el número de campos de texto y selectores de colores que le permiten al usuario especificar el nombre de cada proceso y el color con que se pintará su correspondiente vértice (requerimiento 1.1.).

El método `createFields()` es el que se encarga de crear los campos de texto y los selectores de colores. Se necesitó personalizar un poco los selectores de colores para las necesidades de este programa, por lo que se creó una clase interna `ColorEditor`.

El método `validateAndExecute()` implementado por esta clase se ejecuta cuando el usuario

presiona el botón “Next”. Este lleva a cabo la validación especificada en el requerimiento 1.1., y después obtiene una referencia al modelo mediante su método `getInstance()` para solicitarle que construya el complejo inicial mediante una llamada a su método `createInitialComplex()`. Finalmente se lleva al usuario al siguiente paso, `CommunicationModelStep`, el cual se describe más adelante.

“Simplicial Complex Panel” como vista/controlador

En la Sección 4.1.1 se mencionó que el componente que fungiría como controlador en la arquitectura basada en el patrón MVC del programa sería implementado como un grupo de clases interrelacionadas y que el diseño no se apegaría estrictamente a la arquitectura MVC. Este grupo de clases interrelacionadas en efecto son las clases `SimplicialComplexPanel` y las clases `Step` y sus subclases. Todas estas clases conforman una parte del componente “vista” en la arquitectura del programa (la consola `SCOutputConsole` y la pantalla `JRViewer` de jReality conforman el resto), ya que cada una proporciona diferentes controles de usuario.

Cada una de estas clases funge al mismo tiempo como vista y controlador. No se proporcionan clases separadas que implementen las responsabilidades de un controlador, que son interpretar las entradas del usuario e invocar métodos del modelo. Si decidieramos apegarnos estrictamente al patrón MVC tendríamos que crear dos clases por cada paso, una que fungiera como controlador y otra como vista. Esto supondría un incremento en el número de clases y en la complejidad del código y se considera que esto no vale la pena ya que cada paso tiene un diseño simple y una función bien definida.

Flujo de inicio del panel `SimplicialComplexPanel`

Habiendo descrito las clases que conforman el asistente “Simplicial Complex Panel”, en la Figura 5.6 mostramos el diagrama de secuencia que ilustra cómo se inicia este panel cuando es creado por primera vez por la clase `jRealityView`.

5.2.2. Secuencia de construcción del complejo inicial

Cuando se ha mostrado la interfaz del programa al usuario (lo cual ha sido descrito en los diagramas de secuencia anteriores), el usuario lleva a cabo las acciones descritas en el caso de uso 1.7.. El programa ejecuta la secuencia de acciones mostrada en la Figura 5.7.

En el primer paso del asistente “Simplicial Complex Panel”, el usuario selecciona el número de procesos que conformarán el sistema distribuido. Este evento es manejado por el método

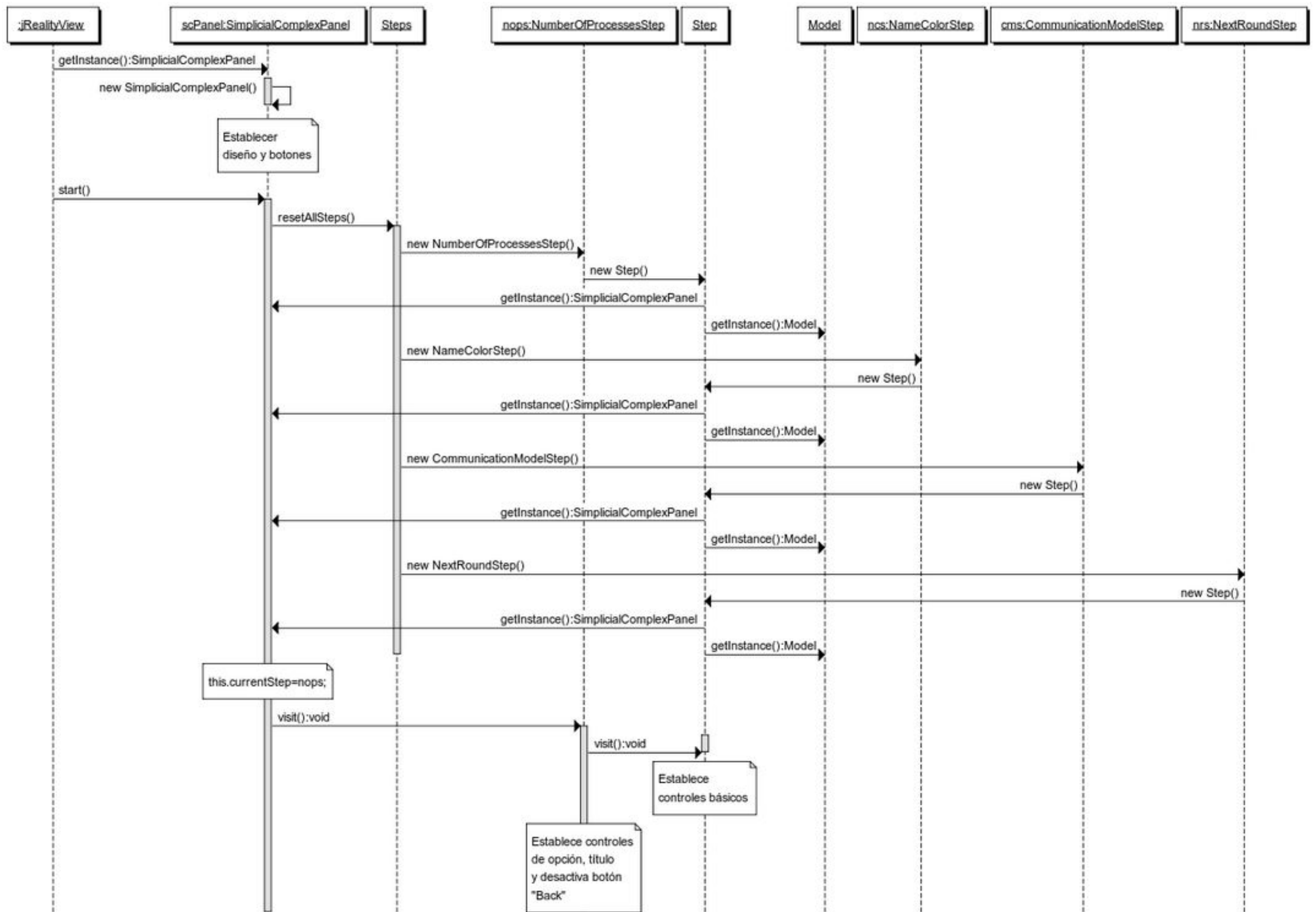

Figura 5.6: Diagrama de secuencia que ilustra como se inicia el panel “Simplicial Complex Panel”.

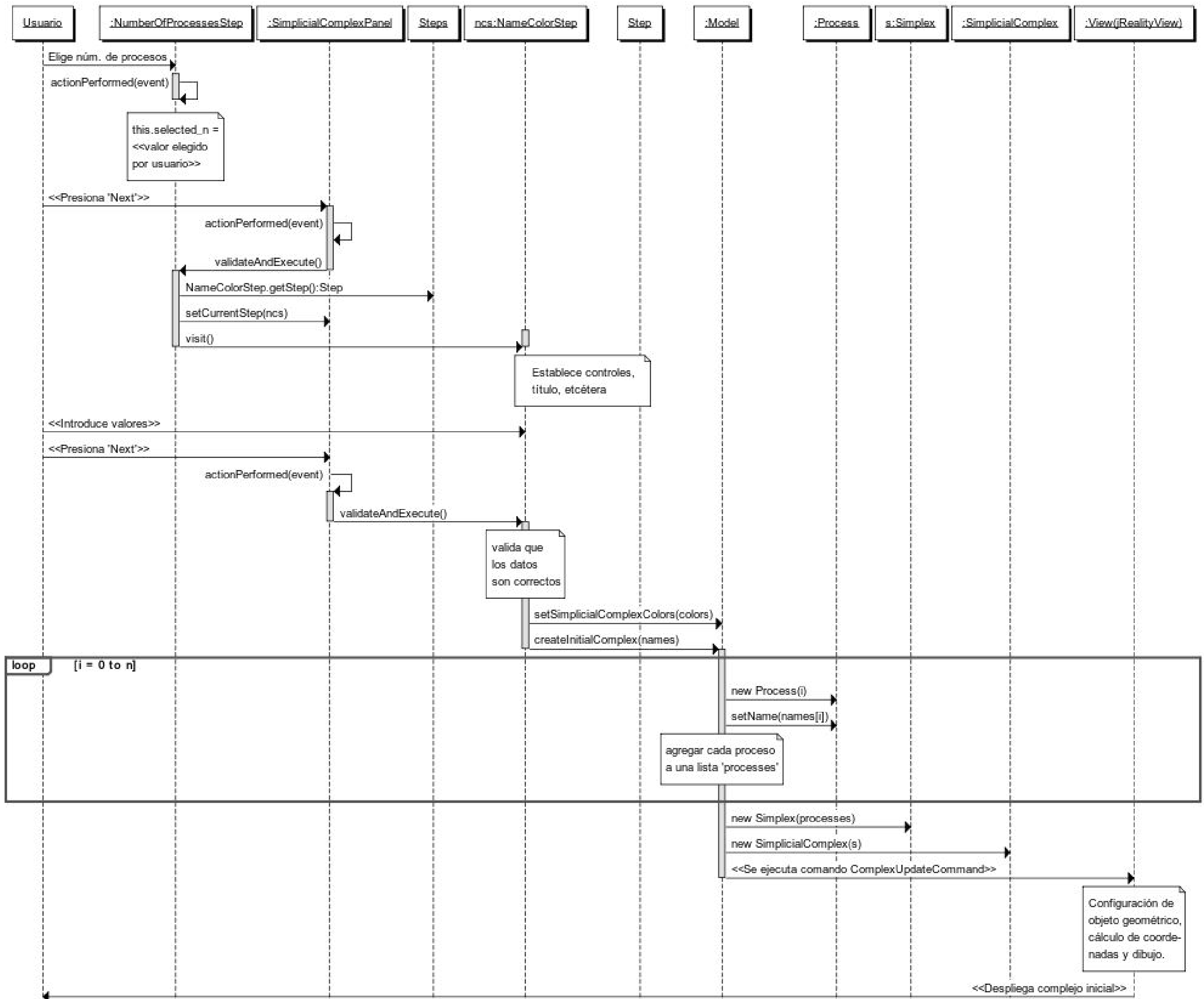

Figura 5.7: Diagrama de secuencia de la construcción de un complejo inicial.

`actionPerformed()`, el cual establece el valor del atributo `selected_n` con el número de procesos seleccionados por el usuario.

Cuando el usuario presiona el botón "Next", el evento se maneja por el método `actionPerformed()` de la clase `SimplicialComplexPanel`. En este caso se ejecuta el método `ValidateAndExecute()` del paso actual, `NumberOfProcessesStep`, lo cual hace que se visite siguiente paso (Listado 5.1); en seguida se muestran los botones del paso `NameColorStep`.

Dependiendo del valor de `selected_n`, se muestra el número correspondiente de controles para capturar nombre y color de cada proceso. Al presionar el botón "Next", se maneja el evento correspondiente y el método `validateAndExecute()` valida los datos introducidos; si son correctos, se obtiene la referencia al modelo mediante una llamada al método `Model.getInstance()` (omitimos esto en el diagrama por simplicidad) y se invoca su método `setSimplicialComplexColors(colors)`, donde `colors` son los colores elegidos por el usuario; despúes se invoca el método
`createInitialComplex(names)`, donde `names` es la lista de los nombres de procesos dados por el usuario. En este método se ejecuta un ciclo donde en cada iteración se crea un nuevo proceso.

Cada proceso se inicializa con el valor de su atributo `id` y a cada uno se le asigna uno de los nombres dados por el usuario. Los nuevos procesos se agrupan en una lista, la cual se pasa como parámetro de inicialización al construir una nueva instancia de la clase `Simplex`, la cual a su vez se usa para inicializar la instancia de la clase `SimplicialComplex`, que es el complejo inicial[6].

Finalmente se notifica a las vistas registradas en el modelo que se ha construido un complejo inicial. Esto se hace mediante un mecanismo basado en el patrón de diseño *Comando* [16], el cual se implementó para evitar tener una larga lista de sentencias condicionales `if`, lo cual haría menos claro el código. Usando las funciones ofrecidas por la biblioteca jReality, la vista `jRealityView` transforma este complejo inicial en un objeto geométrico apto para desplegarse en pantalla . El código de la clase `SCOutputConsole` obtiene la representación en notación de conjuntos de este complejo, así como algunos datos adicionales y los despliega en la consola (este procedimiento lo omitimos en el diagrama para evitar saturarlo).

[6]De esto notamos que los complejos iniciales se componen de un solo simplejo. En futuras versiones del programa se podría extender este procedimiento para soportar complejos iniciales con más de un simplejo.

5.3. Generación de complejos de protocolo

Una vez construido el complejo inicial, el siguiente paso en el asistente es aquel en el que se muestran controles para que el usuario seleccione el protocolo de comunicación bajo el cual se van a generar complejos de protocolo a lo largo de diferentes simulaciones de rondas de comunicación entre los procesos que conforman el sistema distribuido. En la Figura 5.8 mostramos una captura de pantalla de este paso en el asistente. En esta parte del programa se implementa el requerimiento 1.3..

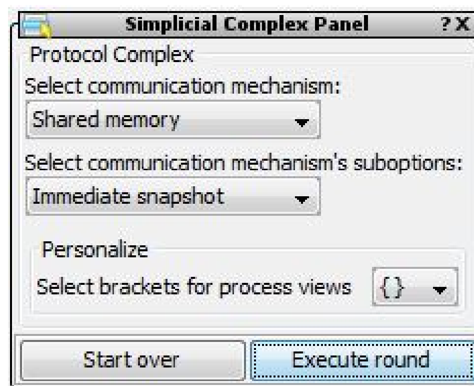

Figura 5.8: Tercer paso en el asistente, en el cual el usuario puede seleccionar el modelo de computación distribuida bajo el cual se van a generar los diferentes complejos de protocolo a lo largo de varias rondas de comunicación. En la primera lista se puede elegir el protocolo que usarán los procesos para comunicarse y en la segunda lista la manera particular en la que esta comunicación se llevará a cabo. En el panel "Personalize" el usuario puede elegir con qué tipo de paréntesis se mostrarán agrupadas las vistas de los procesos en las etiquetas de los vértices en la visualización.

Al presionarse el botón "Execute round", el programa genera el complejo de protocolo de la primera ronda de comunicación con base en los datos proporcionados y después en el asistente se muestra el siguiente y último paso (Figura 5.9).

El propósito de este último paso es permitir al usuario alternar entre la representación cromática y no cromática del complejo de protocolo que actualmente se muestra y también generar otros complejos de protocolo ejecutando más rondas de comunicación. También se le permite regresar al paso anterior al presionar el botón "Change model" para poder cambiar parámetros del modelo de computación distribuida, y reiniciar el asistente para comenzar a construir nuevamente un complejo inicial, al presionar el botón "Start over". En este paso se

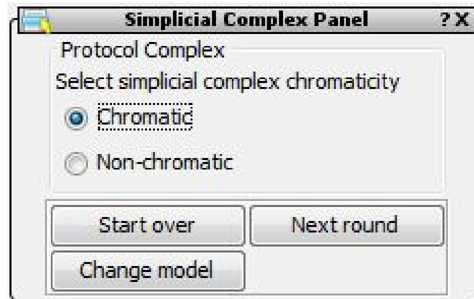

Figura 5.9: Último paso en el asistente, en el cual el usuario puede generar un nuevo complejo de protocolo ejecutando una nueva ronda de comunicación y también puede cambiar la representación del complejo de protocolo actual entre cromática y no cromática.

implementan los requerimientos 1.3.4., 1.3.5. y 1.3.6..

El número de rondas de comunicación que se permite ejecutar es limitado, actualmente se permiten a lo más ejecutar tres rondas. Esto es porque a partir de tres rondas el consumo de memoria del programa aumenta y el desempeño del mismo comienza a degradarse, esto debido a la naturaleza recursiva de la subdivisión de los complejos simpliciales. Por ejemplo, en el modelo *immediate snapshot* para tres procesos, cada triángulo que compone el complejo de protocolo que se forma al subdividir el complejo de la ronda anterior, en la siguiente ronda se subdivide recursivamente siguiendo el mismo patrón de subdivisión de las rondas anteriores, lo cual resulta en un incremento exponencial del número de vértices, caras, etiquetas, y demás información que el programa debe computar y mantener en memoria, impactando su desempeño. Es por esto que también se muestra un mensaje al usuario advirtiendole acerca de esta situación cuando intenta generar el complejo de protocolo para la tercera ronda.

5.3.1. Implementación

Las clases en donde se implementan estos pasos son `CommunicationProtocolStep` y `NextRoundStep`. Estas clases también derivan de la clase base `Step`. En la Figura 5.10 mostramos su diseño.

`CommunicationProtocolStep`

Representa el paso del asistente que permite especificar el modelo de computación distribuida que servirá para generar complejos de protocolo. Consta de dos listas de selección[7]: `cbProtocols`, la cual permite seleccionar el protocolo de comunicación con el que se comuni-

[7] Estos controles también se conocen como *combo box*

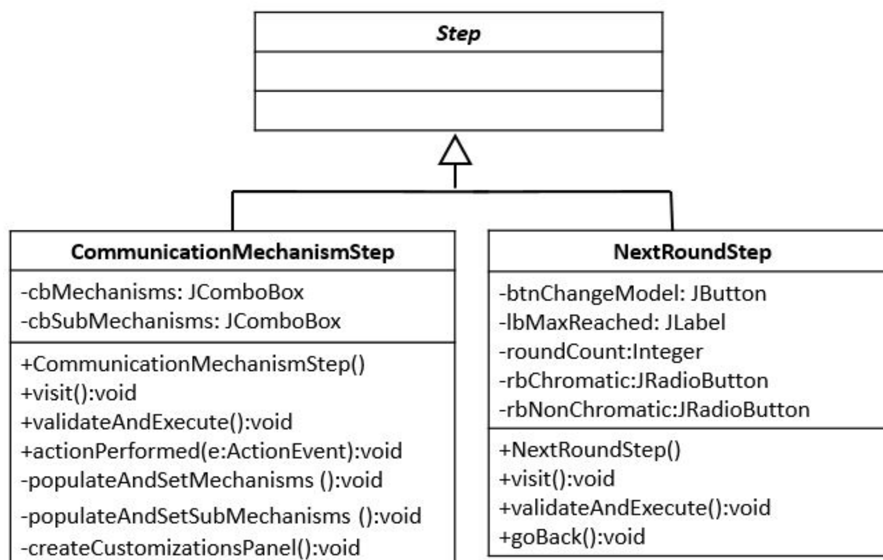

Figura 5.10: Diagrama de clase que muestra las clases `CommunicationProtocolStep` y `NextRoundStep`.

can los procesos; y `cbSubMechanisms`, la cual permite elegir características más específicas del protocolo seleccionado en la primera lista, las cuales llamaremos *submecanismos*.

Los métodos privados `populateAndSetProtocols()` y `populateAndSetSubMechanisms()` se encargan de llenar con valores estas listas de selección. Los valores que se usan para llenar estas listas se obtienen de un diccionario llamado `availableCommunicationProtocols`, el cual almacena como llaves los nombres de los protocolos y como valor por cada llave una lista que contiene los nombres de los submecanismos correspondientes a cada protocolo. Las llaves se usan para poblar la lista de selección `cbProtocols` y los valores asignados a cada llave la lista `populateSubMechanisms`. En el programa este diccionario se implementa como un tipo `Map<K,V>`. [8] Este diccionario puede ser accedido desde cualquier parte del programa ya que está declarado como una variable pública y estática en una clase llamada `Constants`[9], que reside en el paquete `unam.dcct.misc`. La detección de los protocolos de comunicación implementados en el programa y el llenado del diccionario `availableCommunicationProtocols` se realiza en el método estático `getCommunicationProtocolInfo()` de la clase `Constants`. En este método se

[8]El tipo `Map<K,V>` del lenguaje de programación Java es una estructura de datos que mapea llaves tipo `K` a valores tipo `V`. Un mapa no puede contener llaves duplicadas.

[9]El propósito de esta clase es proveer de un punto de acceso global a la mayoría de los valores constantes que se usan a lo largo de todo el programa. En su mayoría estos valores son cadenas de texto o constantes númericas. Esto con el fin de reutilizar las constantes cuando se necesiten y así evitar que se tengan que declarar una y otra vez, reduciendo así la posibilidad de introducir errores en el código.

utiliza una biblioteca llamada `Reflections`[10], la cual permite examinar la estructura del programa en tiempo de ejecución. En concreto la usamos para encontrar todas las clases existentes en el programa que deriven de la clase `CommunicationProtocol`. Cuando estas se encuentran, se invocan sus métodos `getBasicProtocolName()` y `getName()`, los cuales se usan para llenar el diccionario.

El método `createCustomizationsPanel()` se encarga de crear un panel con el título "Personalize", cuya intención es mostrar controles para que el usuario pueda personalizar la forma en que se muestra la representación gráfica del complejo de protocolo. Actualmente solo se pueden elegir el tipo de paréntesis con los que se agruparán las vistas de los procesos en las etiquetas de los vértices en la visualización.

Cuando el usuario presiona el botón "Execute round", el método `validateAndExecute()` solicita al modelo generar el complejo de protocolo de la primera ronda de comunicación, esto lo hace invocando el método `executeRound()` de la clase `Model`. Después de esto se dirige al usuario al siguiente paso, `NextRoundStep`.

El método `goBack()` se ejecuta cuando el usuario presiona el botón "Start over" (Requerimiento 1.3.6.). Este método reinicia el asistente, primero invocando al método `resetAllSteps()` de la enumeración `Steps` que se había discutido, y se lleva al usuario al primero paso del asistente, `NumberOfProcessesStep`.

Los botones "Execute round" y "Start over" en realidad son los mismos botones "Next" y "Back" de los pasos anteriores: cuando se carga este paso al ejecutarse su método `visit()`, simplemente se cambian las leyendas de estos botones.

`NextRoundStep`

Esta clase representa el último paso en el asistente. Contiene los controles que permiten cambiar entre la representación cromática y no cromática del complejo de protocolo que actualmente se esta mostrando en pantalla; y también permite generar el complejo de protocolo de la siguiente ronda de comunicación.

El cambio de representación del complejo de protocolo se hace mediante los botones de opción etiquetados "Chromatic" y "Non-chromatic", los cuales son representados por los atributos `rbChromatic` y `rbNonChromatic`. Este cambio de representación se maneja en el método `actionPerformed()`, en el cual se cambia la representación del complejo mediante una llamada al método `setChromatic()` de la clase `Model`.

[10]Para más información consultar `https://github.com/ronmamo/reflections`

La generación del complejo de protocolo de la siguiente ronda de comunicación se lleva a cabo en el método `validateAndExecute()`, el cual invoca el método `executeRound()` de la clase `Model`. El número máximo de rondas permitidas está representado por el atributo `MAX_ALLOWED_ROUNDS` de la clase `Constants`. En este método también se encuentran las instrucciones para desplegar el mensaje de advertencia que mencionamos al inicio de esta sección.

En este paso también se le ofrece al usuario la opción de reinicar el asistente mediante el botón “Start over”, y de también cambiar el protocolo de comunicación mediante un nuevo botón llamado “Change model”, el cual es agregado al panel `pButtons` de la clase `SimplicialComplexPanel` al ejecutarse el método `visit()`.

5.3.2. Secuencia de generación de complejos de protocolo

Complejo de la primera ronda

En la Figura 5.11 mostramos el diagrama de secuencia de acciones que ejecuta el programa para generar el complejo de protocolo de la primera ronda de comunicación. Esto corresponde al caso de uso 2..

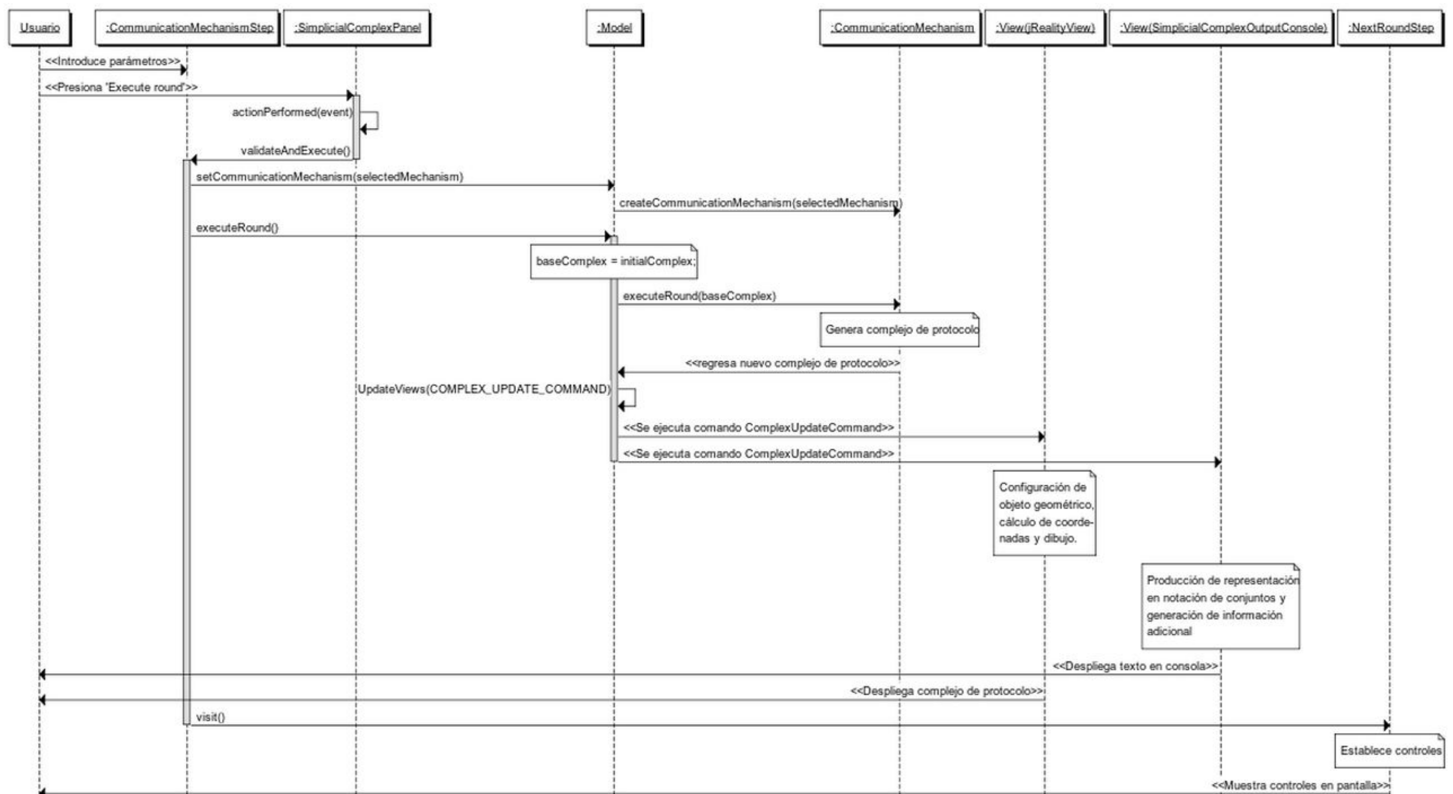

Figura 5.11: Diagrama de secuencia de generación del complejo de protocolo para la primera ronda de comunicación.

Una vez cargados los controles del paso `CommunicationProtocolStep` el usuario selecciona los valores que se piden y presiona el botón "Execute round". Como este botón es el mismo botón "Next" pero con otra leyenda, este evento es manejado por el método `actionPerformed()` de la clase `SimplicialComplexPanel`, la cual a su vez invoca el método `validateAndExecute()` de la clase `CommunicationProtocolStep`.

Como es la primera vez que este método se llama aún no se ha llegado al límite de rondas de ejecución permitidas, por tanto, primero se invoca el método `setCommunicationProtocol(selectedProtocol)`, de la clase `Model`, al cual le pasamos el nombre del protocolo seleccionado por el usuario. En este método se invoca el método estático `createCommunicationProtocol(selectedProtocol)` de la clase `CommunicationProtocol`. Este método es una aplicación del patrón de diseño *método fábrica* [16], la cual crea la instancia de la clase correspondiente al nombre del protocolo de comunicación que se esta pasando como parámetro.

Después se invoca el método `executeRound()` de la clase `Model`. Este método a su vez invoca al método `executeRound(baseComplex)` de la clase `CommunicationProtocol`. El objeto `baseComplex` es el complejo simplicial a partir de cual se generará el complejo de protocolo de la siguiente ronda. El complejo de la primera ronda se genera a partir del complejo inicial, por lo tanto tenemos que `baseComplex` es el complejo inicial. Este método regresa el nuevo complejo de protocolo de la nueva ronda de comunicación.

Lo siguiente es notificar a las vistas registradas en el modelo acerca de este evento mediante el comando `COMPLEX_UPDATE_COMMAND`. La vista `jRealityView` nuevamente transforma este complejo en una objeto geométrico y lo muestra en pantalla; y la vista `SimplicialComplexConsole` produce información en texto del complejo de protocolo, la cual muestra en la consola. Finalmente se cargan en el asistente los controles del siguiente paso, `NextRoundStep`.

Complejos de protocolo de más rondas

En la Figura 5.12 mostramos el diagrama de secuencia de acciones que ejecuta el programa para generar complejos de protocolo para más de una ronda de comunicación. Esto corresponde al caso de uso 3..

Supongamos que el usuario se encuentra en el paso `NextRoundStep` y presiona el botón "Next round". Este evento se maneja por el método `actionPerformed()` ya que es también en realidad el botón "Next" pero con otra leyenda. Entonces se invoca el método `validateAndExecute()`, en donde primero se verifica si esta es la tercera ronda para mandar el mensaje de adverten-

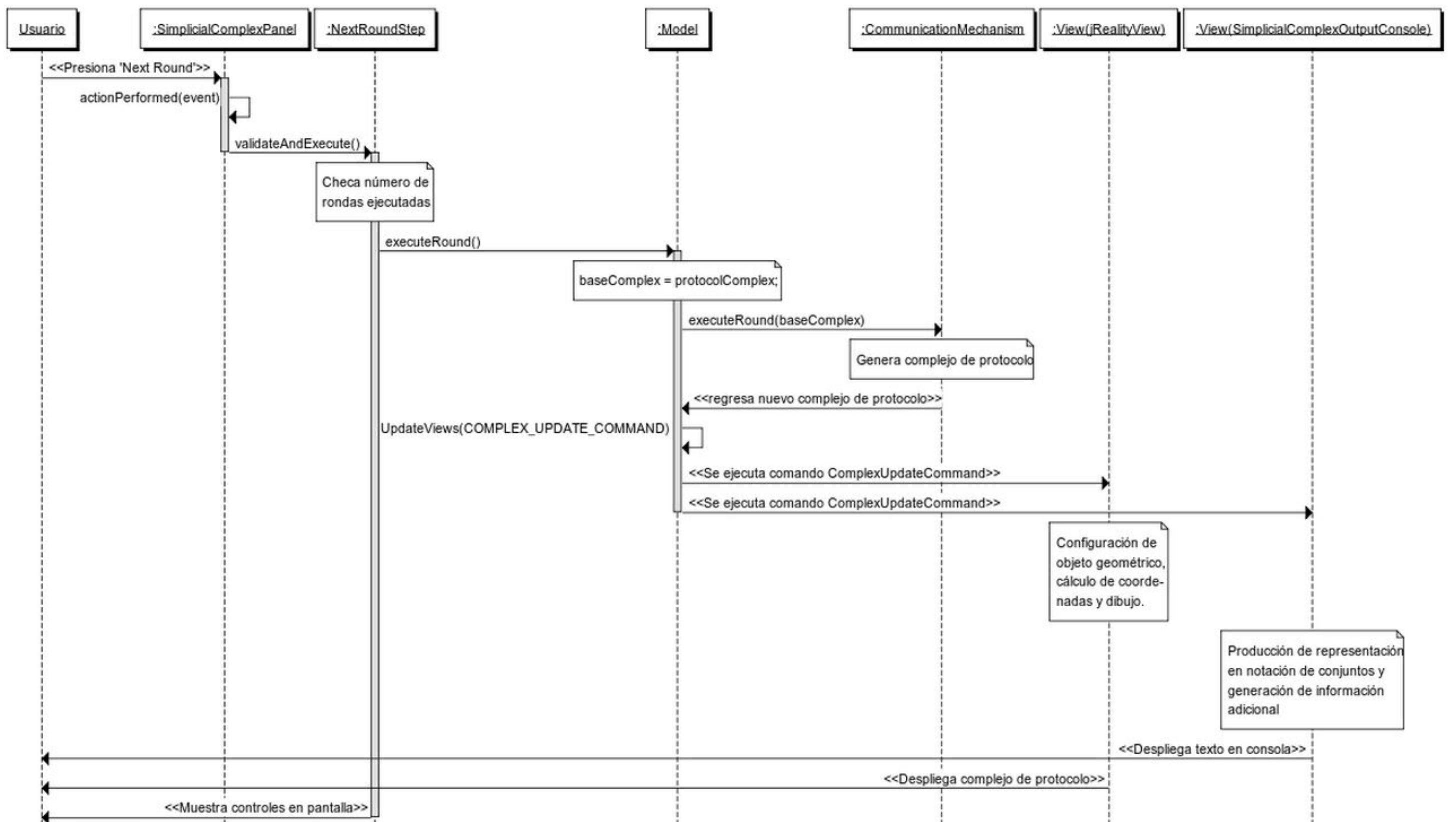

Figura 5.12: Diagrama de secuencia de generación del complejo de protocolo para más de una ronda de comunicación

cia que ya habíamos mencionado. Entonces se invoca el método `executeRound()` de la clase `Model`. En éste se establece como valor de la variable `baseComplex` al complejo de protocolo generado en la última ronda, ya a partir de este se generará el nuevo complejo de protocolo. Se genera entonces el complejo de protocolo de esta nueva ronda mediante una llamada al método `executeRound(baseComplex)` de la clase `CommunicationProtocol` y se le notifica a las vistas acerca de esto mediante el comando `COMPLEX_UPDATE_COMMAND`. Las vistas manejan esta actualización de la misma forma que ya hemos descrito y se muestran en pantalla los resultados.

Cambio de representación del complejo entre cromática y no cromática

En la Figura 5.13 mostramos las acciones que realiza el programa para cambiar la representación de un complejo de protocolo de cromática a no cromática.

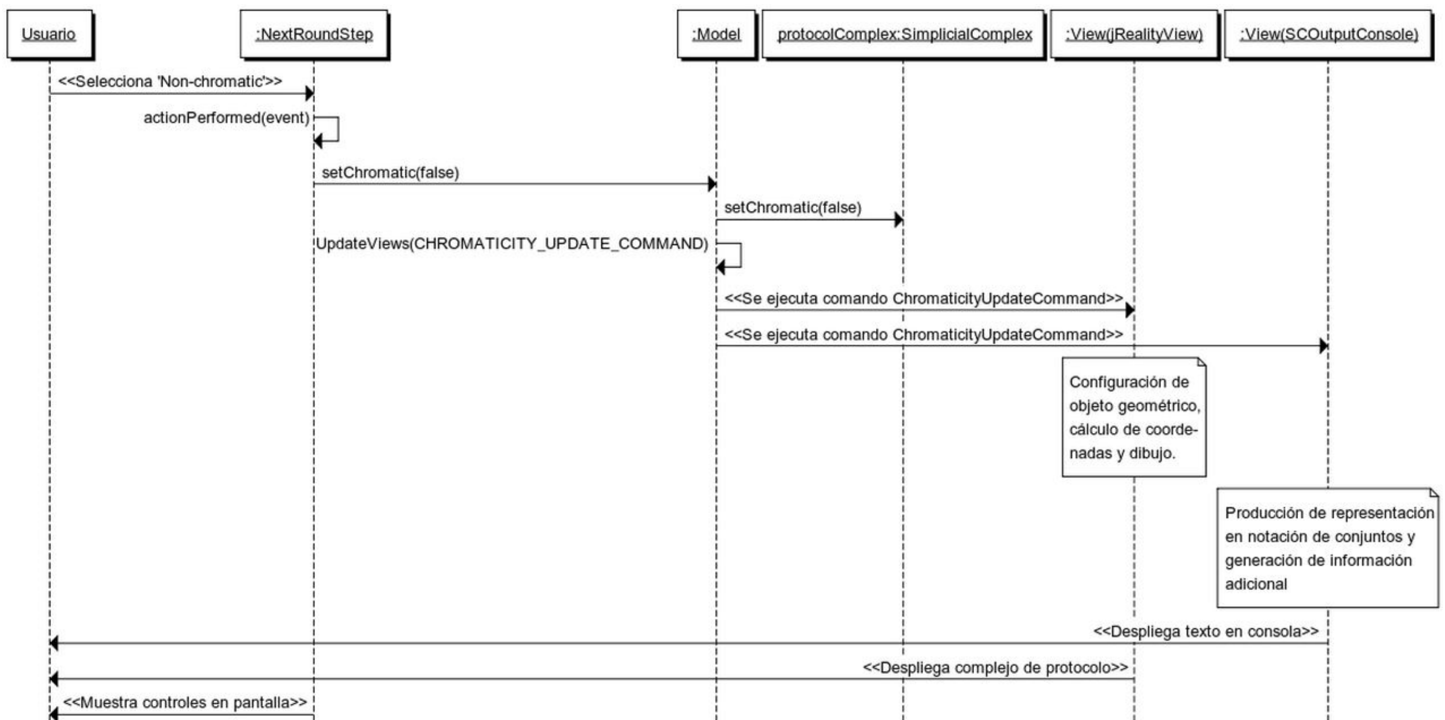

Figura 5.13: Diagrama de secuencia de cambio de representación cromática a no cromática

Cuando el usuario hace click en el botón de opción etiquetado como "Non-chromatic", este evento se maneja por el método `actionPerformed()` de la misma clase `NextRoundStep`. Aquí simplemente se invoca al método `setChromatic()` de la clase `Model`, donde `chromatic` es la opción que el usuario eligió, en este caso sería `false`. En este método simplemente se cambia el estado del último complejo de protocolo generado (que es el que actualmente se muestra

en pantalla) mediante una llamada al método `setChromatic()` y después se le notifica a las vistas de este cambio mediante el comando `CHROMATICITY_UPDATE_COMMAND`. En el caso de la vista `jRealityView` ésta toma el objeto geométrico que ya había construido y que actualmente se muestra en pantalla e invoca su método `setChromatic()`, lo cual hace que internamente se re-computen los datos de vértices, colores, coordenadas y etiquetas de tal forma que representen la representación no cromática del complejo de protocolo actual y éste se vuelve a dibujar. En el caso de la consola, simplemente se actualiza la información en texto de tal forma que ésta corresponda al mismo complejo pero no cromático. Por ejemplo, la representación en notación de conjuntos del complejo no cromático es diferente de la del cromático, ya que en este último caso no tenemos un par $(id, view)$, sino solamente $(view)$.

5.4. Algoritmo generador de complejos de protocolo *immediate snapshot*

Como hemos visto, un complejo de protocolo se genera a partir de un complejo inicial o de un complejo de protocolo de la ronda de comunicación anterior.

Un complejo de protocolo contiene todos los posibles estados en los que terminan los procesos del sistema después de ejecutar una ronda del protocolo. Estos estados consisten en los valores de las vistas de los procesos.

El complejo de protocolo obtenido después de la ronda de comunicación del protocolo *immediate snapshot* es correcto si los valores de las vistas v_i de los procesos p_i que lo conforman son el resultado de todos los posibles escenarios de ejecución y satisfacen las siguientes condiciones [11].

Condiciones del protocolo *immediate snapshot*.

1. *Autocontención*: $i \in v_i$

2. *Atomic snapshot*: Para todo i, j, se cumple solo una de estas condiciones: $v_i \subseteq v_j$ o $v_j \subseteq v_i$.

3. *Inmediatez*: Para toda i, j, si $i \in v_j$, entonces $v_i \subseteq v_j$.

La estrategía seguida para generar complejos de protocolo bajo este modelo es primero obtener una codificación de *todos* los escenarios de ejecución válidos, conocidos como *historias*, que pueden darse al ejecutar una ronda de comunicación de este protocolo. Con base en estos escenarios podemos contruir los procesos y simplejos que conforman el complejo de protocolo.

5.4.1. Historias

Un escenario de ejecución se puede expresar como una *historia*, la cual sirve para modelar la ejecución de un sistema concurrente. Una *historia* es una secuencia finita de eventos de *invocación* y *respuesta* [23]. Una *llamada a método* en una historia H es un par que consiste en una invocación y la siguiente respuesta que le corresponde en H.

Para un proceso p_i que se comunica mediante memoria compartida *immediate snapshot* las llamadas a métodos que hace serían $write(mem[i])$ y $snap(mem[*])$. Para fines de este trabajo asumimos que las invocaciones a estos dos métodos tienen respuestas inmediatas.

Un ejemplo de las historias de las ejecuciones de la Figura 2.8 serían las siguientes:

(a) $p_0.write(mem[0])$, $p_0.snap(mem[*])$, $p_1.write(mem[1])$, $p_1.snap(mem[*])$

(b) $p_1.write(mem[1])$, $p_1.snap(mem[*])$, $p_0.write(mem[0])$, $p_0.snap(mem[*])$

(c) $p_0.write(mem[0])$, $p_1.write(mem[1])$, $p_0.snap(mem[*])$, $p_1.snap(mem[*])$

En las ejecuciones (a) y (b) observamos que las llamadas a métodos *write* son seguidas inmediatamente por llamadas *snap*, lo cual refleja la propiedad de inmediatez del modelo *immediate snapshot*. Para la ejecución (c) tenemos que éste no es el caso, sin embargo se hace así para denotar que las llamadas a métodos *write* y *snap* de ambos procesos son concurrentes, en cuyo caso ambos procesos terminarán con vistas idénticas.

Para obtener las vistas de los procesos del complejo de protocolo que queremos construir basta con ejecutar simulaciones de comunicación entre los procesos basadas en estas historias. Cada historia determina el orden en que cada proceso escribe y lee la memoria compartida.

Lo primero que se hizo, entonces, fue obtener una codificación de todas las posibles historias que se pueden dar en una ronda de ejecución de este protocolo, las cuales posteriormente se usan para simular la comunicación y obtener las nuevas vistas; con estas vistas se crean nuevos procesos, nuevos simplejos y al final, el complejo de protocolo.

Se implementó entonces un algoritmo que genera la codificación de todas las historias para $n < 10$ procesos que ejecutan el protocolo *immediate snapshot*. Esta restricción es más que suficiente ya que nuestro programa solo requiere trabajar con complejos de a lo más tres procesos, y es poco probable que en el futuro se requiera que trabaje con complejos de diez o más procesos.

5.4.2. Generando las historias

El algoritmo que se implementó para generar las codificaciones de todas las historias está basado en la observación de que nuestra codificación de una historia válida del protocolo *immediate snapshot* es equivalente a una *partición ordenada* de un conjunto de *id*s de los procesos del sistema, es decir, del conjunto de enteros $\{0, 1, ..., n-1\}$.

Una partición de un conjunto S es un conjunto de subconjuntos no vacíos y disjuntos de S tal que su unión es S. Llamamos *bloque* a cada subconjunto de una partición. Una partición, junto con un orden total en sus bloques se denomina *partición ordenada*[31]. Por ejemplo, $[\{3\}, \{1, 2\}, \{0\}]$, $[\{1, 2\}, \{3\}, \{0\}]$ y $[\{0\}, \{1, 2\}, \{3\}]$ son algunas de las particiones ordenadas del conjunto $\{0, 1, 2, 3\}$.

Formalmente esta equivalencia se expresa mediante el siguiente teorema.

Teorema 1. Una partición ordenada del conjunto $\{0, 1, ..., n-1\}$ equivale a una historia de ejecución válida de una ronda de ejecución del modelo *immediate snapshot*.

Demostración. Para demostrar este teorema se da un algoritmo que con base en una partición ordenada del conjunto $\{0, 1, ..., n-1\}$ obtiene un conjunto de procesos con vistas que satisfacen las condiciones 5.4.

Algoritmo simulador de una ronda de ejecución del protocolo immediate snapshot

1. Considerese una partición ordenada H del conjunto $\{0, 1, ..., n-1\}$ que contiene k bloques, con $1 \leq k \leq n$; y también consideresé un arreglo M con n entradas que representa una memoria compartida.

2. Por cada bloque $b \in H$:

   2.1. Por cada $i \in b$:

      2.1.1. Hacemos que cada proceso p_i tal que $id = i$ escriba en la $i-$ésima entrada de M el contenido de su vista v_i.

   2.2. Nuevamente, por cada $i \in b$:

      2.2.1. A cada proceso p_i tal que $id = i$ le asignamos como nuevo contenido de su vista v_i una copia entera de la memoria M, simulando así la operación *snapshot*.

Cumplimiento de las condiciones 5.4

1. *Autocontención*: Al inicio de toda ejecución del protocolo, para cada proceso p_i tenemos que $v_i = \{i\}$ (Sección 2.1). Durante la primera ronda de comunicación este valor es escrito en M. Cuando los procesos p_i obtienen una copia de M, actualizando así el valor de sus respectivas vistas v_i, el valor i continua contenido en v_i ya que previamente había sido escrito en la memoria. Este comportamiento se repite a lo largo de la ejecución del protocolo, por lo tanto la condición se satisface.

2. *Atomic snapshot*: Primero verifiquemos el caso $v_i = v_j$: Sin pérdida de generalidad considérense un par de procesos p_i, p_j tales que $i, j \in b$, donde $b \in H$. En el paso 2.1.1 del algoritmo ambos procesos escriben sus vistas v_i, v_j en M. Después, en el paso 2.2.1 ambos procesos obtienen copias completas de la memoria M, con las cuales se actualizan sus respectivas vistas v_i, v_j. Como ambos procesos ya había escrito sus vistas completas en M, al final de la ejecución de este paso tenemos que la vista completa de p_i, v_i, fue leída por p_j y la vista completa de p_j, v_j fue leída por p_i, por lo tanto $v_i = v_j$. Ahora verifiquemos el caso $v_i \subset v_j$. Sin pérdida de generalidad considérense un par de procesos p_i, p_j tales que $i \in b$ y $j \in b'$, donde $b, b' \in H$, $b \neq b'$ y b está ordenado antes que b' en H. Al terminarse de ejecutar la iteración del paso 2 del algoritmo para b, tenemos que el procesos p_i escribió su vista en M y efectuó una operación de *snapshot* sobre M. Después, en la iteración del paso 2 que se ejecuta para b', el proceso p_j escribe su vista en M y efectua una operación de *snapshot* sobre M. Como p_i ya había escrito y leído la memoria en la iteración anterior, su vista v_i es leída por p_j. Sin embargo, cuando p_i leyó la memoria, p_j no había escrito su vista ya que pertenece a un bloque que está ordenado después de b y cuya iteración del paso 2 aún no había sido ejecutada, por lo tanto tenemos que $v_i \subset v_j$.

3. *Inmediatez.* Sin pérdida de generalidad, supongamos que $i \in v_j$ al final de la ejecución de alguna iteración del paso 2.2.1 y que i fue procesado antes que j en los ciclos 2.1 y 2.2. De esto se sigue que p_i obtuvo una copia entera de M antes que p_j, es decir, actualizó el valor de su vista v_i antes que p_j. Notemos que una vez que se comienza a ejecutar el ciclo 2.2, el contenido de la memoria M no puede cambiar, ya que el algoritmo es secuencial y en el ciclo 2.2 no hay instrucciones que modifiquen M. Por lo tanto, p_j obtiene una copia de M idéntica a la que obtuvo p_i, entonces $v_i = v_j$. El caso $v_i \subset v_j$ se dá cuando j pertenece a un bloque diferente al que pertenece i y que está ordenado después. Por la naturaleza secuencial de este algoritmo, cuando p_j lee la memoria, la vista v_i ya había sido completamente establecida en alguna iteración anterior del ciclo 2. Es posible que algún

otro proceso, p_k, modificara la memoria después de p_i y antes que p_j, sin embargo, esto no afecta el conjunto de valores que p_i en su momento leyó de M, por lo tanto $v_i \subseteq v_j$.

■

Habiendo establecido que podemos obtener vistas de procesos válidas a partir de particiones ordenadas del conjunto de *id*s de procesos $\{0, 1, ..., n-1\}$, para obtener *todas* las posibles historias de ejecución de una ronda del protocolo *immediate snapshot* necesitamos un algoritmo que genere todas las particiones ordenadas del conjunto.

Para generar todas las particiones ordenadas de conjuntos implementamos dos algoritmos. El primero genera las todas las particiones de conjuntos sin importar el orden y el segundo obtiene todas las permutaciones de los bloques de estas particiones. Al final el programa produce todas las particiones ordenadas codificadas de la siguiente forma: Cada bloque es un grupo *id*s de procesos y se delimitan por el carácter "|". En la Figura 5.14 mostramos algunos ejemplos de las codificaciones de particiones que produce el programa para $n = 2$, $n = 3$ y $n = 4$.

La implementación del algoritmo que genera las particiones ordenadas de conjuntos es el método `generate` de la clase `PartitionGenerator` que esta ubicada en el paquete `unam.dcct.model.ImmediateSnapshot`. Mostramos el código de esta implementación en el Listado 5.2. Este método invoca a los métodos `generateAllRGF` y `generateOrderedPartitions`, los cuales explicamos en las siguientes seccióñes.

Listado 5.2: Método `generate` .

```
1 public static String generate(int n){
2       List<int[]> allRGF = generateAllRGF();
3       String allPartitions = generateOrderedPartitions(allRGF);
4       return allPartitions;
5    }
```

Algoritmo generador de particiones de conjuntos

Para la generación de particiones de conjuntos se implementó el algoritmo explicado en [32], el cual genera todas las *funciones restringidas de crecimiento* (*Restricted Growth Functions* o RGF), las cuales son una forma de codificar particiones de conjuntos.

Formalmente, una función restringida de crecimiento de un conjunto $\{0, 1, ..., n-1\}$ es un vector $(v_1, v_2, ..., v_n)$ que satisface $v_1 = 1$ y $v_i \leq max\{v_1, ..., v_{i-1}\}$. En otras palabras, cada

```
n=2     n=3                        n=4

0|1     0|1|2      0|1|2|3    1|3|0|2    2|0 1 3    0 2|1 3
1|0     0|2|1      0|1|3|2    1|3|2|0    3|0|1|2    0 3|1|2
0 1     0|1 2      0|1|2 3    1|3|0 2    3|0|2|1    0 3|2|1
        1|0|2      0|2|1|3    1|0 2|3    3|0|1 2    0 3|1 2
        1|2|0      0|2|3|1    1|0 3|2    3|1|0|2    1 2|0|3
        1|0 2      0|2|1 3    1|2 3|0    3|1|2|0    1 2|3|0
        2|0|1      0|3|1|2    1|0 2 3    3|1|0 2    1 2|0 3
        2|1|0      0|3|2|1    2|0|1|3    3|2|0|1    1 3|0|2
        2|0 1      0|3|1 2    2|0|3|1    3|2|1|0    1 3|2|0
        0 1|2      0|1 2|3    2|0|1 3    3|2|0 1    1 3|0 2
        0 2|1      0|1 3|2    2|1|0|3    3|0 1|2    2 3|0|1
        1 2|0      0|2 3|1    2|1|3|0    3|0 2|1    2 3|1|0
        0 1 2      0|1 2 3    2|1|0 3    3|1 2|0    2 3|0 1
                   1|0|2|3    2|3|0|1    3|0 1 2    0 1 2|3
                   1|0|3|2    2|3|1|0    0 1|2|3    0 1 3|2
                   1|0|2 3    2|3|0 1    0 1|3|2    0 2 3|1
                   1|2|0|3    2|0 1|3    0 1|2 3    1 2 3|0
                   1|2|3|0    2|0 3|1    0 2|1|3    0 1 2 3
                   1|2|0 3    2|1 3|0    0 2|3|1
```

Figura 5.14: Particiones ordenadas de conjuntos $\{0, .., n-1\}$ que representan escenarios del protocolo *immediate snapshot* que produce el algoritmo generador para $n = 2$, $n = 3$ y $n = 4$. Cada bloque se delimita con el carácter "|".

entrada v_i en el vector denota que el elemento i esta ubicado en el v_i-ésimo bloque contenido en una partición. Por ejemplo, las funciones restringidas de crecimiento de las particiones `0|2|1`, `02|1` y `012`, son `132`, `131` y `111` respectivamente, considerando que los bloques comienzan a numerarse desde 1. En el Listado 5.3 mostramos el código Java del método `generateAllRGF`, que la implementación que se hizo del algoritmo.

El algoritmo genera todas las RGF en orden lexicográfico. El arreglo de enteros `V` representa la RGF que se genera en cada iteración del ciclo de la linea 10 y cada elemento se inicializa con el número uno (linea 5), ya que la primera partición contiene un solo bloque que contiene a todos los elementos. Cada entrada en `V` siempre se va a incrementar en uno. El ciclo de la linea 10 encuentra qué entradas se pueden incrementar en uno de tal forma que se mantengan las condiciones de crecimiento restringido. Para esto se utiliza un arreglo auxiliar `M` que almacena el máximo legal y que en todo momento satisface la condición $M[i] = max\{V[0], ..., V[i]\} + 1$.

Listado 5.3: Algoritmo `generateAllRGF` .

```
1  List<int[]> generateAllRGF(int n){
2       List<int[]> allRGF = new ArrayList<int[]>();
3       int[] M = new int[n];
4       int[] V = new int[n];
```

```
5       for (int i=0; i<n; i++){
6          V[i]=1;
7          M[i]=2;
8       }
9       int j = 0;
10      while (true){
11         allRGF.add(V.clone()); // agrega una copia de V
12         j=n-1;
13         while (V[j]==M[j]) { j--; }
14         if (j>0){
15            V[j]++;
16            for (int i=j+1; i<n; i++){
17               V[i]=1;
18               if (V[j]==M[j])
19                  M[i]=M[j]+1;
20               else
21                  M[i]=M[j];
22            }
23         }else
24            return allRGF;
25      }
26   }
```

Análisis de la complejidad. Antes de analizar la complejidad del algoritmo es necesario exponer algunas definiciones, las cuales se tomaron de [25].

Sea $S(n)$ el conjunto de particiones de un conjunto $\{0, ..., n-1\}$. Para enteros positivos n y m, sea $S(n,m)$ el conjunto de particiones de $\{0, ..., n-1\}$ que contienen exactamente m bloques no vacíos. El *número de Bell*, $B(n)$ se define como $B(n) = |S(n)|$ y el *número de Stirling del segundo tipo*, $S(n,m)$, se define como $S(n,m) = |S(n,m)|$. De estas definiciones vemos que.

$$B(n) = \sum_{m=1}^{n} S(n,m)$$

Los primeros números de Bell son: $B(1) = 1$, $B(2) = 2$, $B(3) = 5$, $B(4) = 15$, $B(5) = 52$, ... De acuerdo a [10], los números de Bell están acotados de la siguiente forma:

$$B(n) < \left(\frac{0{,}792n}{ln(n+1)} \right)^n$$

Por lo tanto su crecimiento es exponencial.

Cada iteración del ciclo de la línea 10 se ejecuta en tiempo lineal: El ciclo de la línea 13 a lo más ejecuta n iteraciones, al igual que el ciclo de la línea 16. Cada iteración del ciclo de la línea 10 produce una nueva partición codificada en una RGF (línea 11), y como el total de particiones de un conjunto $\{0, ..., n-1\}$ es igual a $B(n)$, este ciclo se ejecuta $B(n)$ veces, por lo tanto la complejidad de este algoritmo es $O(B(n))$, la cual es exponencial.

Algoritmo generador de particiones ordenadas

El siguiente paso es generar todas las particiones ordenadas a partir de las codificaciones de particiones producidas por el algoritmo anterior. En el Listado 5.4 mostramos el código del método `generateOrderedPartitions` en donde se implementa este algoritmo.

Las instrucciones de las líneas 2 a 10 transforman las funciones de crecimiento restringido generadas por el algoritmo anterior en particiones de conjuntos. Cada partición, almacenada en la variable `partition`, la representamos como una lista de objetos tipo `StringBuilder` [11] en donde cada uno representa un bloque. En las líneas 4 y 5 se crea una lista con `k` entradas que representa a la partición. En la línea 6 se inicializan los bloques `StringBuilder` contenidos en la lista `partition`. En el ciclo de la línea 7 agregamos cada elemento `i` del conjunto al bloque al cual está asignado, es decir, al bloque en la `rgf[i]`-ésima posición.

Para obtener todas las particiones ordenadas de `partition` se invoca al método `generatePartitionPermutations` (Listado 5.5) en la línea 10.

El método inicia con la creación de un objeto `PermutationGenerator`. Ésta es una clase en donde se implementó el algoritmo generador de permutaciones *Johnson-Trotter* [32]. Este algoritmo es simple y eficiente ya que genera cada permutación en $O(1)$.

En cada iteración del ciclo de la línea 3 se obtiene una permutación `p`. Usamos cada elemento `i` en `p` para acomodar el `i`-ésimo bloque de `partition` dentro de la nueva partición ordenada `orderedPartition`. Finalmente el método `add` agrega `orderedPartition` al objeto `allPartitions` que contiene la representación en texto de todas las particiones ordenadas.

De vuelta al método `generateOrderedPartitions`, en la línea 13 se convierte el objeto `allPartitions` en una representación en texto como las que se mostraron en la Figura 5.14.

Listado 5.4: Algoritmo `generateOrderedPartitions` .

```
1 String generateOrderedPartitions(List<int[]> allRGF){
2     StringBuilder allPartitions = new StringBuilder();
```

[11] Los objetos `StringBuilder` de Java representan cadenas de caracteres, que al contrario de los objetos `String`, sus caracteres y tamaño pueden ser modificados en tiempo de ejecución.

```
3      for (int[] rgf : allRGF){
4        int k = getMax(rgf);
5        List<StringBuilder> partition = new ArrayList<StringBuilder>(k);
6        initPartition(k, partition);
7        for (int i=0; i< n; i++){
8          StringBuilder block = partition.get(rgf[i]);
9          block.append(Integer.toString(i));
10       }
11       generatePartitionPermutations(partition, allPartitions, k);
12     }
13     return allPartitions.toString();
14   }
```

Listado 5.5: Algoritmo `generatePartitionPermutations` .

```
1  void generatePartitionPermutations(List<StringBuilder> partition, StringBuilder
       allPartitions, int k){
2      PermutationGenerator permutations = new PermutationGenerator(k);
3      for (List<Integer> p : permutations){
4        List<StringBuilder> orderedPartition=new ArrayList<StringBuilder>(k);
5        for (int i : p){
6          orderedPartition.add(partition.get(i));
7        }
8        add(orderedPartition, allPartitions);
9      }
10   }
```

Análisis de la complejidad. Como ya revisamos, el número de Stirling del segundo tipo, $S(n,m)$ cuenta el conjunto de particiones de un conjunto de cardinalidad n que contienene exactamente m bloques no vacíos. Para contar el número de particiones ordenadas basadas en un conjunto de cardinalidad m (como lo es una partición con m bloques), vemos que los $m!$ elementos se pueden ordenar de $m!$ diferentes maneras. Por lo tanto, podemos expresar el número de particiones ordenadas, $A(n)$, en términos de los número de Stirling del segundo tipo de la siguiente manera:

$$A(n) = \sum_{m=1}^{n} m!S(n,m)$$

Alternativamente, el número de particiones ordenadas se puede expresar con la siguiente

recurrencia.

$$A(n) = \begin{cases} 1 & \text{si } n = 1 \\ \sum\limits_{i=1}^{n} \binom{n}{i} A(n-i) & \text{si } n > 1 \end{cases}$$

Esta fórmula la interpretamos de la siguiente manera: Contamos cuantas formas se tienen de construir un bloque con i elementos, para cada $i = 1..n$. Esto corresponde a la parte $\sum\limits_{i=1}^{n} \binom{n}{i}$ de la fórmula. Habiendo contado todos los posibles bloques con i elementos, por cada uno de estos ahora hay que contar cuantos posibles bloques de diferentes tamaños se pueden construir con los $n - i$ elementos restantes. Esto corresponde al factor $A(n - i)$. En cuanto al caso base, es fácil ver que si $n = 1$, solo se puede construir un solo bloque que contiene al único elemento del conjunto.

Para cada $n = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...$, tenemos que $T(n) = 1, 1, 3, 13, 75, 541, 4683,$ $47293, 545835, 7087261, ...$ respectivamente. Estos valores corresponden a la secuencia de enteros conocida como *números ordenados de Bell* [24]. Como vemos, cada uno de estos números cuenta el número de $(n-1)-$simplejos [12] en que se subdivide cada uno de los $(n-1)-$simplejos de un complejo en cada ronda de comunicación, bajo el protocolo *immediate snapshot*.

Entonces la complejidad del algoritmo `generateOrderedPartitions` es $O(A(n))$, es decir, es igual a la secuencia de los números ordenados de Bell por cada valor de n, por lo tanto la complejidad de generar todos los escenarios de ejecución del una ronda del protocolo *immediate snapshot* es exponencial.

5.4.3. Construcción del complejo de protocolo a partir de las historias generadas

Habiendo generado el conjunto de posibles historias correspondientes a la ejecución de rondas del protocolo *immediate snapshot*, lo siguiente es construir a partir de este conjunto el nuevo complejo de protocolo.

Esto se realiza por medio del siguiente algoritmo.

[12] Estos números también cuentan el número de caras de los permutoedrones [33], que son politopos (generalización a cualquier dimensión de un polígono bidimiensional o un poliedro tridimensional) cuyos vértices se forman permutando sus coordenadas. Al parecer los permutoedrones son objetos similares a los complejos simpliciales que se estudian en computación distribuida.

Algoritmo generador de complejos de protocolo

1. Tomamos cada simplejo σ del complejo de protocolo de la ronda anterior (o cada simplejo σ del complejo inicial, si estamos construyendo el complejo de protocolo de la primera ronda). Por cada σ:

   1.1. Ejecutamos una versión ligeramente modificada del algoritmo simulador de ronda de ejecución de *immediate snapshot* que dimos en la demostración del Teorema 1. Tal modificación consiste en los siguientes cambios en los pasos 2.1.1 y 2.2.1: En lugar de hacer que los procesos p_i escriban y lean la memoria M, creamos copias exactas de estos procesos, las cuales denominaremos p'_i, y hacemos que estas copias lleven a cabo la escritura y lectura de la memoria M.

   1.2. Con el conjunto de copias de procesos p'_i obtenidos en cada simulación de historia H creamos un nuevo simplejo σ'.

2. Con el conjunto de nuevos simplejos σ' creamos el complejo de protocolo $\mathcal{P}$ que corresponde a esta ronda de ejecución.

Análisis de la complejidad.

Teorema 2. La complejidad del algoritmo generador de complejos de protocolo es $A(n)^r$, donde r es el número de rondas de ejecución del protocolo *immediate snapshot*.

Demostración. Caso base. Para ejecutar la primera ronda de comunicación del protocolo *immediate snapshot* partimos del complejo inicial, el cual consiste en un solo simplejo, por lo tanto el paso 1.1 se ejecuta una sola vez. Este paso genera $A(n)$ historias de ejecución, cada una de las cuales se convierte en un nuevo simplejo σ' (paso 1.2). Por lo tanto, el complejo de protocolo de la primera ronda que se construye en el paso 2 consiste en $A(n)$ simplejos.

Hipótesis de inducción. Supongamos el complejo de protocolo P_{r-1} construido al final de la ejecución de la $r-1$-ésima ronda de comunicación consiste en $A(n)^{r-1}$ simplejos.

Paso inductivo. En la ejecución de la r-ésima ronda se ejecuta el paso 1.1 por cada $\sigma \in P_{r-1}$. Como $|P_{r-1}| = A(n)^{r-1}$ y por cada simplejo con n procesos se generan $A(n)$ historias de ejecución las cuales a su vez derivan cada una en un nuevo simplejo σ', tenemos que
$|P_r| = A(n)^{r-1}A(n) = A(n)^r$ ■

De esto podemos ver que, por ejemplo, el complejo de protocolo de tres procesos generado en la primera ronda contiene $A(3)^1 = 13$ simplejos, el de la segunda $A(3)^2 = 169$ y el de la

tercera $A(3)^3 = 2197$. Es por este crecimiento exponencial en el número de simplejos generados por ronda de comunicación afecta el desempeño del programa, por lo que a lo más se permiten ejecutar tres rondas.

5.4.4. Implementación del algoritmo generador de complejos de protocolo

Es importante mencionar que esta implementación se vió muy influída por el requerimiento 1.3.5., en el cual se le permite al usuario cambiar la representación cromática y no cromática del complejo de protocolo mostrado en pantalla.

Inicialmente el programa generaba el complejo de protocolo solamente en su representación cromática. La idea era que cuando el usuario deseara ver la representación no cromática de este, el programa transformaría en ese momento el complejo cromático en uno no cromático. Esta idea resultó díficil de implementar, especialmente con el complejo de protocolo de la segunda o tercera ronda de comunicación. Así que se decidió mejor implementar la idea de "precomputar" ambas representaciones del complejo, esto es, ejecutar dos veces el procedimiento para generar el complejo de protocolo desde la primera ronda: una vez sería para generar la versión cromática y la otra para generar la versión no cromática. Con esto se mantienen en memoria los datos de las dos representaciones del complejo de protocolo, aunque por defecto el programa le muestra inicialmente al usuario la representación cromática del complejo. Cuando el usuario cambia a la versión no cromática en el paso del asistente "Next round", el programa simplemente obtiene los datos almacenados de esta representación en la memoria y los convierte en su representación geométrica y la muestra en pantalla. Las desventajas de este enfoque es que el programa siempre mantiene los datos del complejo de protocolo no cromático en memoria, aún cuando el usuario no esté interesado en ver esta representación; y también que el código de los algoritmos de generación de complejos de protocolo se hizó un poco más complicado.

En la Figura 5.15 mostramos el diagrama de secuencia que ilustra cómo el programa ejecuta el algoritmo generador de complejos de protocolo. [13] La ejecución del algoritmo inicia con una llamada al método `executeRound(baseComplex)`, en donde lo primero que se hace es activar la representación cromática de `baseComplex`, para asegurarse de que primero se genere el complejo de protocolo cromático. Después se llama al método `generateOrCompleteNewComplex(baseComplex)`. Este método genera el complejo de protocolo desde cero o bien "completa" uno que ya ha sido generado. Por completar nos referimos a que primero se crea desde cero el complejo de protocolo de forma cromática (es por eso que se estableció `bc.setChromatic(true)`). Para generar la ver-

[13]Para entenderlo mejor es útil revisar los diagramas de clases de las Figuras 4.3, 4.5 y 5.10.

sión no-cromática se vuelve a llamar a este método, pero en ésta segunda llamada el complejo de protocolo ya fue creado por la primera llamada, por lo tanto el resultado es que se "completa" el complejo, es decir, se le asignan los nuevos simplejos no cromáticos generados, tal como se puede ver en la última secuencia de acciones (hasta abajo) en el diagrama.

Regresando a la explicación de la primera llamada a este método, primero se obtiene la lista de simplejos de `baseComplex` el cual devolverá su lista interna `chromaticSimplices` por ser actualmente cromático. Por cada simplejo `s` [14] en esta lista se solicita que se cree el objeto que da acceso a todos los escenarios de ejecución mediante una llamada al método `createScenarioGenerator(s.dimension())` el cual, como vemos, requiere como parámetro la dimensión del simplejo actual `s` para generar estos escenarios. Como se mencionó en la sección 4.2.3, la implementación de este método abstracto la da la clase que representa el protocolo de comunicación con base en el cual se está generando el complejo de protocolo. En este caso es implementado por `ImmediateSnapshot`. Este método genera los escenarios obtieniendo todas las particiones ordenadas del conjunto de enteros $\{0, ..., dimension + 1\}$, donde $dimension + 1$ es igual al total de procesos contenidos en el simplejo. Como se vió en la sección anterior, las particiones ordenadas son equivalentes a los escenarios de ejecución de una ronda del protocolo *immediate snapshot*, pero otros modelos implementados en el futuro podrían usar otros métodos para generar sus posibles escenarios, pero para que se integren en el programa deben de encapsular cada escenario en un objeto de tipo `Scenario`.

También es importante mencionar que en el programa no se generan todos los escenarios una y otra vez por cada iteración del ciclo que estamos describiendo, ya que se implementó un arreglo que funciona como un *cache*, el cual mantiene en memoria el conjunto de escenarios generados por cada una de distintas dimensiones de simplejos. Éste devuelve el conjunto de escenarios que se requiera, evitando tener que volver a recomputarlos una y otra vez[15].

Después de esto se crea el objeto `scnGen` de tipo `Iterable<Scenario>`[16] que permite iterar sobre cada uno de los escenarios. A continuación se ejecuta una simulación de comunicación de cada escenario `scn` entre los procesos del simplejo `s`, denotados por `originalProcesses`. La simulación se hace con una llamada al método `execute` de la interfaz `Scenario`, que en realidad

[14] Esto corresponde al paso 1 del algoritmo generador de complejos de protocolo que describimos en la sección anterior.

[15] Por brevedad omitimos estas acciones en el diagrama.

[16] Este objeto corresponde a la clase `ISScenarioGenerator` que se muestra en el diagrama de la Figura 5.10, sin embargo, por brevedad y simpleza, en la implementación se optó por implementar esta interfaz como una *clase anónima*.

es implementada por la clase `ImmediateSnapshotScenario`. Ésta a su vez ejecuta la simulación con una llamada al método `simulateCommunication(originalProcesses, sharedMemory, order)`, donde `sharedMemory` es un arreglo que simula ser una memoria compartida, y `order` es un arreglo de índices creado a partir del propio escenario que indica el orden en que los grupos de procesos escriben y leen `sharedMemory`. La implementación de este método se muestra en el Listado 5.6, el cual es la implementación del algoritmo dado en la prueba del Teorema 1, la cual también incluye las modificaciones descritas en el paso 1.1 del algoritmo generador de complejos de protocolo.

Como mencionamos en la Sección 4.2.1, el protocolo de comunicación es quién da las primitivas de comunicación, en este caso, la clase `ImmediateSnapshot` ofrece los métodos `write(Process p,String[] sharedMemory)` y `snapshot(Process p,String[] sharedMemory)`, aunque esto se omite en el diagrama.

Listado 5.6: Método `simulateCommunication` .

```
1  List<Process> simulateCommunication(List<Process> processes, String[] memory, int[]
       order) {
2      List<Process> newProcesses = new ArrayList<Process>(order.length);
3      for(int i=0;i<order.length;i++){
4          Process p = (Process)processes.get(order[i]).clone();
5          write(p, memory);
6          newProcesses.add(p);
7      }
8      for (Process p: newProcesses){
9          snapshot(p, memory);
10     }
11     return newProcesses;
12  }
```

Con los nuevos procesos contenidos en la lista `newProcesses` se crean los nuevos simplejos `ns` que formarán parte del complejo de protocolo, pero antes se les asocia una referencia al simplejo del cual procedieron, `s`, mediante la llamada al método `setParent(s)`.

Conforme se van generando cada grupo de simplejos derivados de `s`, los cuales se denotan como `newSimplices`, estos se van acumulando en una lista llamada `allNewSimplices`, la cual agrupa a todos los simplejos que conformarán el nuevo complejo de protocolo.

Finalmente, para construir la representación no cromática del complejo de protocolo se ejecu-

ta otra vez este algoritmo para generar los simplejos no cromáticos y así completar `newComplex`. Esto corresponde a la secuencia de acciones mostradas al final del diagrama.

5.5. Representación gráfica de complejos simpliciales

En esta sección discutimos cómo el programa transforma los complejos simpliciales en representaciones geométricas.

Recordemos que la clase `Model` notifica a las vistas registradas acerca de algún cambio en su estado, de tal forma que estas puedan actualizar la interfaz de usuario para reflejar tales cambios. Esto se hace mediante los mecanismos implementados que estan basados en los patrones de diseño *observador* y *comando*. En particular, nos enfocaremos en discutir cómo la vista `jRealityView` construye objetos geométricos para representar complejos iniciales o de protocolo y cambiar sus representaciones cromáticas y no cromáticas.

5.5.1. Actualización de las vistas mediante comandos

Los cambios de estado en la clase `Model` que son notificados a las vistas son tres:

1. Cuando el usuario crea un complejo inicial o genera un complejo de protocolo de una ronda de comunicación.

2. Cuando el usuario cambia la representación de cromaticidad del complejo de protocolo que actualmente se muestra en pantalla.

3. Cuando el usuario reinicia el asistente para volver a crear desde el inicio un complejo inicial.

Estas acciones se manejan en los métodos `createInitialComplex()`, `executeRound()` y `reset()` de la clase `Model`. Estos métodos notifican a las vistas registradas en `Model` acerca de estos cambios, para que ellas efectúen los cambios correspondientes en la interfaz de usuario. En el diagrama de la arquitectura MVC del programa mostrado en la Figura 4.2 vemos que estas notificaciones se hacen mediante una llamada a un método `update()`, sin embargo, al momento de realizar la implementación, este método consistía en una larga secuencia de instrucciones `if` que manejaban cada estos casos, en donde cada caso a su vez manejaba diferentes subcasos y procesamiento. Tener secuencias condicionales tan largas hacen que el código sea díficil de modificar y extender [16]. Por lo tanto se decidió encapsular cada una de estas acciones en objetos conocidos como *comandos* [16]. Se implementó un comando genérico `Command`, del cual

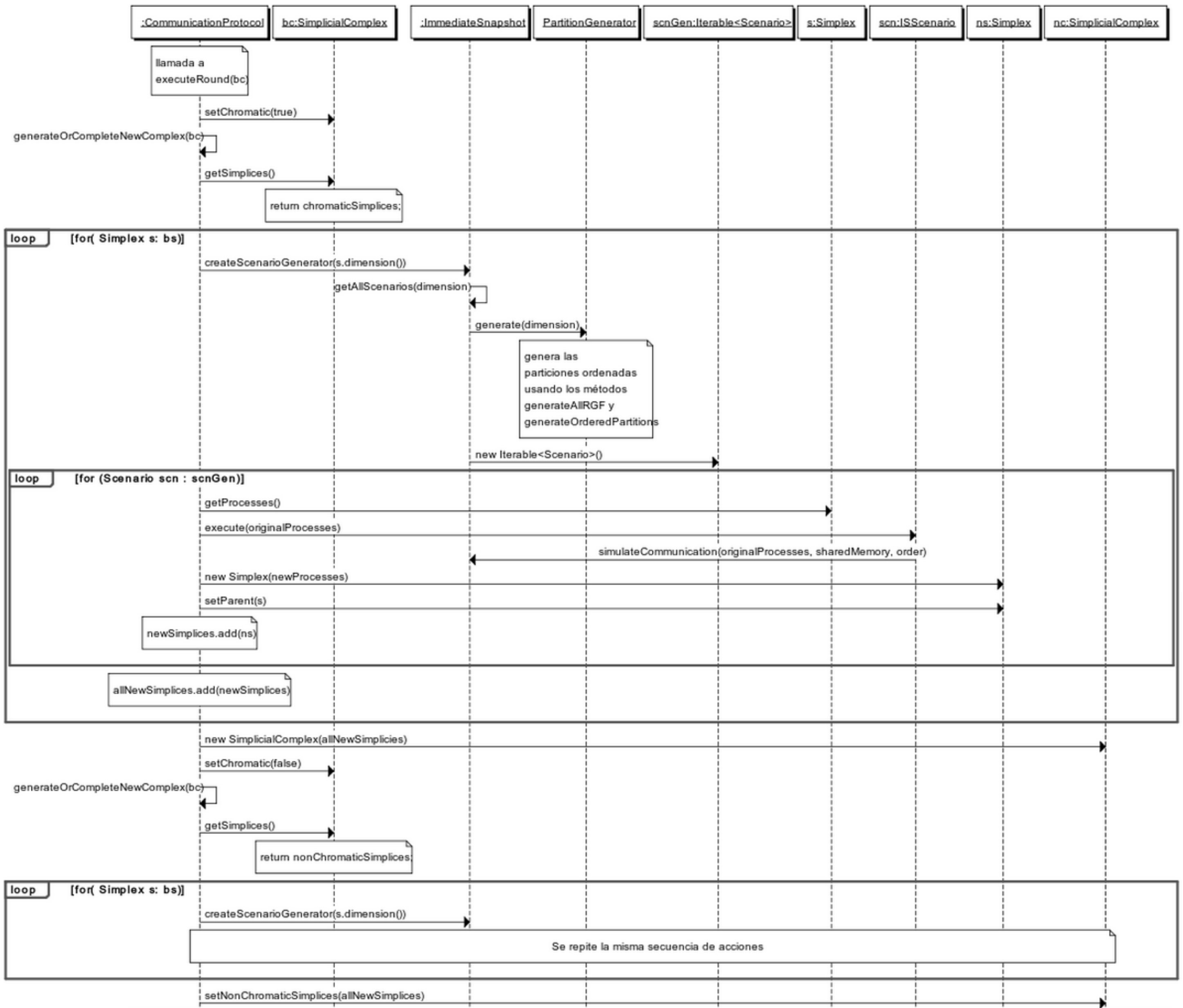

Figura 5.15: Diagrama de secuencia que ilustra la comunicación entre los diferentes objetos y clases para generar un complejo de protocolo. Por brevedad se abrevian algunos nombres de variables: `bc` denota `baseComplex`, `bs` denota `baseSimplicies`, `s` denota `Simplex`, `ISScenario` denota `ImmediateSnapshotScenario`, `ns` denota `newSimplex` y `nc` denota `newComplex`.

derivan los tres comandos respectivos que encapsulan estas acciones: `ComplexUpdateCommand`, `ChromaticityUpdateCommand` y `ResetViewCommand`. Estas clases están ubicadas en el paquete `unam.dcct.view.commands`. Así, en lugar de tener el método `update()` en la interfaz `View`, como se ve en la Figura 4.2, se implementaron métodos que manejan los diferentes cambios de estado de la clase `Model`. Estos nuevos métodos en la interfaz `View` son: `displayComplex()`, `updateChromaticity()` y `reset()`. Estos son llamados por los comandos correspondientes, y a su vez por los comandos son creados por el método privado `updateViews()` en la clase `Model`. Las clases `jRealityView` y `SimplicialComplexPanel` dan la implementación de estos métodos.

5.5.2. Transformación de un complejo simplicial a su representación geométrica

Como hemos visto, la clase `jRealityView` es la que se encarga de mostrar en pantalla las representaciones geométricas de complejos simpliciales. Como ejemplo, en el Listado 5.7 mostramos el código del método `displayComplex` de la clase `jRealityView`, el cual transforma un complejo simplicial en su representación geométrica.

Listado 5.7: Método `displayComplex` .

```
1  ...
2  private Geometry geometricObject;
3  ...
4  public void displayComplex() {
5       Model m = Model.getInstance();
6       // Checa si el complejo es inicial o de protocolo
7       SimplicialComplex protocolComplex = m.getProtocolComplex();
8       if (protocolComplex==null){
9          geometricObject = new GeometricComplex(m.getInitialComplex());
10      }else
11      {
12         geometricObject = new GeometricComplex(protocolComplex);
13      }
14      updateView();
15 }
```

En el código vemos que el atributo `geometricObject` almacena la representación geométrica del complejo simplicial, `GeometricComplex`, pero esta también podría almacenar una instancia de `Face`, lo cual permite que el programa pueda dibujar por separado representaciones de

simplejos. Actualmente en la interfaz de usuario no se soporta esta funcionalidad, pero se diseñó así el programa para que pueda ser fácilmente implementada en el futuro si así se requiriese.

El método `updateView` se encarga de dibujar el objeto geométrico en pantalla usando los métodos de la biblioteca jReality, extrayendo del objeto `geometricObject` toda la información de vértices, coordenadas, etiquetas, colores y caras a través de los métodos de la interfaz `Geometry` (Figura 4.7). Para conocer la implementación de este método el lector puede consultar el código del programa.

Para entender con detalle cómo se construye el objeto `GeometricComplex` mostramos el diagrama de secuencia de la Figura 5.16.

El método constructor de la clase `GeometricComplex` recibe una instancia del complejo simplicial, representado por la variable `sc`, a partir del cual se va a crear su representación geométrica. Primero se cambia la representación geométrica de este complejo a cromática para primero construir las caras en su representación cromática. Esto se hace mediante el método `buildFaces()`. En este método primero se obtiene la lista de simplejos que componen el complejo simplicial `sc` para después construir una cara por cada simplejo `s`. Para construir cada cara primero se obtiene el simplejo padre del simplejo actual ya que el método constructor de la clase `Face` lo requiere y después éste se invoca, pasándole también la variable `chromatic`, que en este caso almacena el valor `true`. Lo primero que se hace es asignar los valores de estos parámetros a los atributos correspondientes. Cuando `parentFace` es nulo significa que el complejo simplicial actual es inicial, por lo tanto la cara actual no tendrá ningún valor en su atributo `parent`. Como mencionamos en el Capítulo 4, y como más adelante veremos, establecer una relación padre-hijo entre caras nos ayudará a realizar el cálculo de las coordenadas de los vértices para dibujar complejos de protocolo.

Después se invoca el método `setVertices()`, el cual va a construir los vértices que componen la cara a partir de los procesos contenidos en `simplex`. Al terminar de construir los vértices, se crea el objeto encargado de asignar los colores a los vértices y calcular sus coordenadas. Como se mencionó en la sección 4.2.4, estas acciones dependen de la representación cromática o no cromática de la cara: este objeto puede ser de tipo `ChromaticBehaviour` o `NonChromaticBehaviour`. En el diagrama solo mostramos el primer caso.

Después se llama el método `setAttributes()`, el cual va a establecer el color, las etiquetas y las coordenadas de los vértices, así como también los "indices de caras"[17]. Los colores de

[17]Por indices de caras nos referimos a una manera de especificar las caras de un complejo simplicial en términos de sus vértices. Se construye de la siguiente manera: A cada vértice le asignamos un único "indice", el cual es un entero del conjunto $\{0, .., n-1\}$, donde n es el total de vértices contenidos en la cara. Con este conjunto construimos

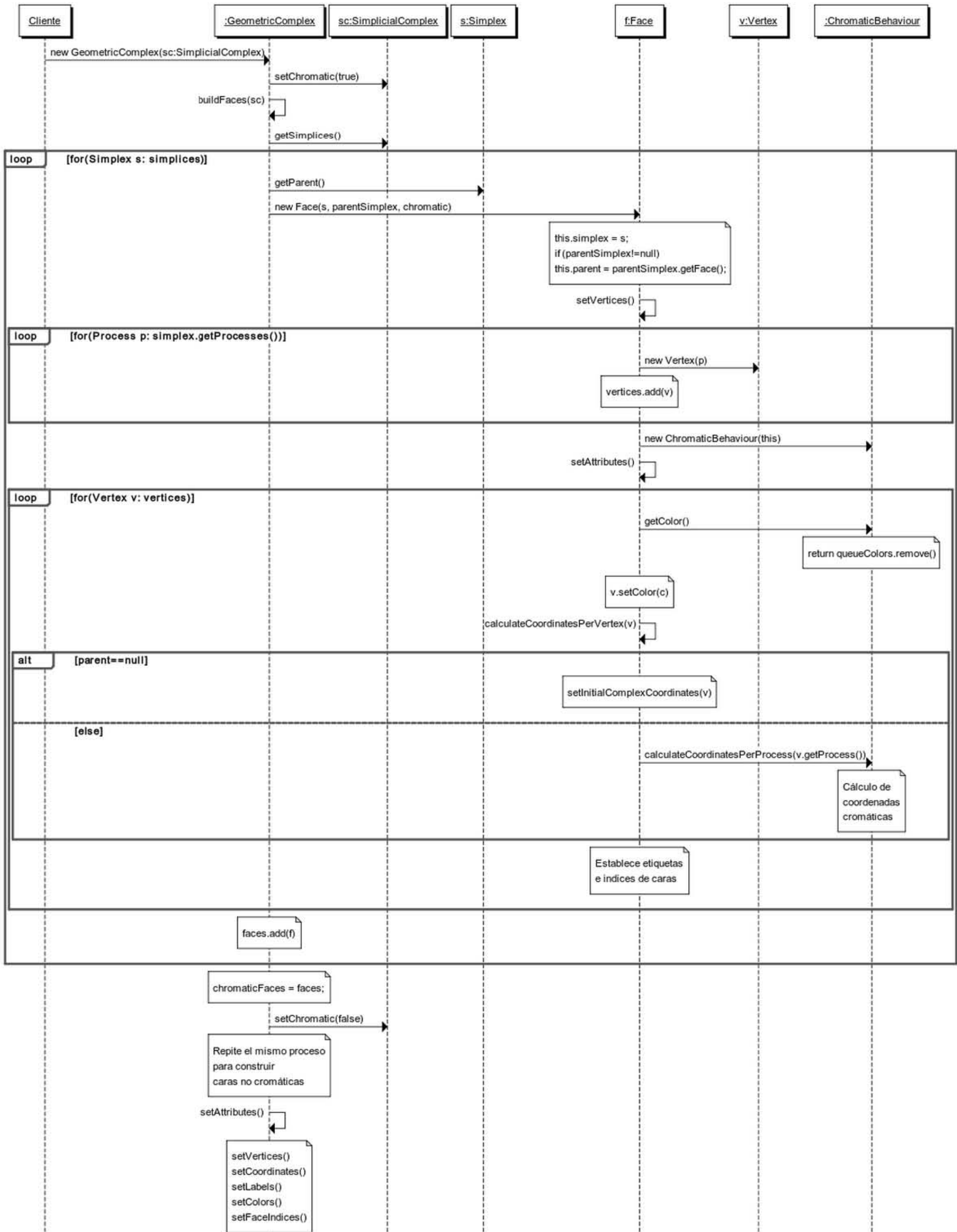

Figura 5.16: Construcción de la representación geométrica de un complejo simplicial.

los vértices, en el caso cromático, se obtienen usando cola llamada `queueColors`. Esto se hace para garantizar que no haya vértices adyacentes con el mismo color. En el caso no cromático, simplemente se usa el mismo color para todos los vértices.

Después se calculan las coordenadas por vértice. Cuando la cara actual tiene `parent=null` se asigna a cada vértices coordenadas preestablecidas por el programa, ya que esta cara corresponde a un complejo inicial. En el caso contrario, las coordenadas se tienen que calcular con un procedimiento más elaborado.

Cada cara se agrega a una lista `faces`, la cual una vez completada se asigna a la lista `chromaticFaces`. Después se cambia a la representación no cromática del complejo simplicial `sc` y se repite (casi) la misma secuencia de llamadas a métodos y procesamiento para obtener las representaciones no cromáticas de las caras del complejo geométrico no cromático.

Finalmente se invoca el método `setAttributes()` de la clase `GeometricComplex` para establecer los vértices y sus coordenadas, etiquetas y colores así como también los indices de caras del complejo. Como los vértices y sus atributos ya habían sido establecidos por cada cara, en general estos métodos simplemente obtienen estos valores de las caras. El lector puede refirse al código del programa para conocer los detalles.

5.5.3. Cálculo de coordenadas

A los vértices que conforman un complejo inicial se les asignan coordenadas preestablecidas y las coordenadas de los vértices en los complejos de protocolo generados en cada $r-$ésima ronda de comunicación se calculan con base en el valor de cada vista ${}_rv_i$ correspondiente a cada proceso p_i que conforma el complejo.

Denotamos las coordenadas de un vértice que representa a un proceso en el sistema distribuido como una función $coords : vistas \to \mathbb{R}^3$, donde $vistas$ es el conjunto de vistas de procesos en el sistema distribuido.

Primero explicamos el procedimiento para cálcular las coordenadas de los vértices de complejos no cromáticos por ser el más simple y después el cromático. Las fórmulas mostradas están basadas en las que se discuten en [19].

---

una conjunto de sus subconjuntos de cardinalidad k. Cada subconjunto indica los k vértices que componen cada cara, por lo tanto el número de subconjuntos es el número de caras del complejo. Por ejemplo, los indices de caras serían $\{\{0,1,2\}\}$ para un triágulo (ya que solo contiene una cara), $\{\{0,1,2\},\{1,2,3\},\{0,1,3\},\{0,2,3\}\}$ para un tetraedro y $\{\{0,1,2\},\{0,1,3\},\{1,3,4\},\{1,5,6\},\{0,1,6\}\}$ para el complejo de la Figura 2.5, ya que consideramos como caras a los triángulos que lo conforman. Estos "indices de caras" son requeridos para dibujar complejos simpliciales por algunas aplicaciones de visualización matemática tales como jReality.

Coordenadas de complejo no cromático

Las coordenadas de un vértice que representa a un proceso p_i están dadas por la siguiente fórmula.

$$coords({}_r v_i) = \begin{cases} default(i) & \text{si } r = 0 \\ (0,0,0) & \text{si } {}_r v_i \text{ es desconocida} \\ \frac{1}{n}\sum\limits_{j=0}^{n-1} coords({}_{r-1} v_j) & \text{si } r > 0 \end{cases}$$

Está fórmula índica que las coordenadas de los vértices del complejo de protocolo de la ronda r se calculan con base en los vértices del complejo de la ronda anterior $r-1$. $default(i)$ denota las coordenadas preestablecidas que se le asignan a los vértices que corresponden a cada proceso p_i del complejo inicial. Estas podrían ser cualquier valor arbitrario pero en el programa se les asignan valores tales que el complejo se vea bien proporcionado.

Esta fórmula en realidad nos dá las coordenadas de los vértices que constituyen la *subdivisión baricéntrica* del complejo de protocolo de la ronda anterior (o del complejo inicial, si la ronda actual es la primera). Esto se puede apreciar en el complejo de protocolo de la Figura 2.5.

Coordenadas de complejo cromático

La siguiente fórmula da las coordenadas de los vértices que constitutyen la *subdivisión cromática* [19] del complejo de protocolo de la ronda anterior o del complejo inicial. La subdivisión cromática es similar a la baricéntrica, con la diferencia de que en lugar de tener un vértice en el baricentro del complejo tendrémos a los más n vértices separados a una distancia d del baricentro que se calcula usando un factor $0 < \epsilon < 1$.

$$coords({}_r v_i) = \begin{cases} default(i) & \text{si } r = 0 \\ (0,0,0) & \text{si } {}_r v_i \text{ es desconocida} \\ coords({}_{r-1} v_i) & \text{si } |{}_r v_i| = 1 \\ \frac{1-\epsilon}{n} coords({}_{r-1} v_i) + \sum\limits_{j \neq i}^{n-1} \frac{1+\epsilon/(n-1)}{n} coords({}_{r-1} v_j) & \text{si } r > 0 \end{cases}$$

Al igual que con las coordenadas no cromáticas, $default(i)$ denota las coordenadas preestablecidas que se le asignan a los vértices del complejo inicial. Cuando tenemos $|{}_r v_i| = 1$ quiere decir que el proceso p_i solo "se vió" a sí mismo durante la ronda de comunicación: no se enteró de las vistas de los demás procesos posiblemente por alguna falla o porque estos se tardarón en comunicar sus vistas, por lo tanto se toman las mismas coordenadas que tenía el vértice del

proceso en el complejo de la ronda anterior. En el último caso las coordenadas se calculan con la distancia desde el baricentro que se mencionó al principio. En la Figura 2.4 podemos ver la subdivisión cromática del complejo inicial de tres procesos donde las posiciones de los vértices se calcularon con esta fórmula.

Implementación

Como vimos en la sección anterior, el cálculo de coordenadas se lleva a cabo por la clase `Face`, la cual a su vez delega esta tarea a las clases `ChromaticBehaviour` o `NonChromaticBehaviour`, dependiendo de si el complejo es cromático o no cromático.

Cada instancia de `Face` tiene un atributo `parent` que referencia a la cara del complejo de protocolo de la ronda anterior. Esto sirve para obtener las coordenadas de los vértices de la cara "padre" para calcular los de la cara actual, tal como se ve en las fórmulas.

En el caso cromático, cada vértice de la cara "padre" se referencia por el `id` del proceso al cual corresponde.

En el Listado 5.8 se muestra el código del método en donde se implementa el cálculo de coordenadas del complejo cromático. Este método es un miembro de la clase `ChromaticBehaviour`.

Listado 5.8: Método `calculateCoordinatesPerProcess` .

```
1  public double[] calculateCoordinatesPerProcess(Process p) {
2        String[] processView = p.getViewArray();
3        int n = p.getViewElementsCount();
4        int pid = p.getId();
5        /* Si el proceso solo se vio durante la ronda de comunicacion, usa las
             coordenadas del vertice que lo represento en el complejo de la ronda
             anterior. */
6        if (n == 1)
7           return parent.getCoordinates()[pid];
8        final float EPSILON = Constants.EPSILON_DEFAULT ;
9        double smallFactor = (1-EPSILON)/n;
10       double bigFactor = (1+(EPSILON/(n-1)))/n;
11       double[] res = {0.0,0.0,0.0};
12       for (int i = 0; i<processView.length; i++){
13          if (i==pid)
14             res = LinearAlgebraHelper.vectorSum(
15                   LinearAlgebraHelper.scalarVectorMultiply(smallFactor,parent.getCoordinates()[pid])
16                   ,res);
```

```
17        else if (processView[i]!=null){
18          double[] coords = parent.getCoordinates()[i];
19          res = LinearAlgebraHelper.vectorSum(
               LinearAlgebraHelper.scalarVectorMultiply(bigFactor, coords),res);
20        }
21      } return res;
22    }
```

Para el cálculo de coordenadas de complejos no cromáticos no se puede referenciar a los vértices de la cara “padre” por el `id` de proceso porque como hemos visto, en esta representación no existe esa noción, por lo tanto los vértices son identificados por los valores de las vistas de cada proceso. Por lo tanto en la clase `NonChromaticBehaviour` se implementó un atributo `coordinatesMap` de tipo `Map<String, double[]>` que usa como llave el valor de la vista del proceso y como valor las coordenadas del proceso. En el Listado 5.9 mostramos el código del método que implementa el cálculo de las coordenadas no cromáticas.

Listado 5.9: Método `calculateCoordinatesPerProcess` .

```
1  public double[] calculateCoordinatesPerProcess(Process p) throws ClassCastException{
2    String[] processView = p.getViewArray();
3    int n = p.getViewElementsCount();
4    NonChromaticBehaviour parentNcBehaviour = null;
5    try {
6      parentNcBehaviour = (NonChromaticBehaviour)parent.chromaticityBehaviour;
7    } catch (ClassCastException e) {
8      throw new ClassCastException("The parent's face chromaticity behaviour must be
           non-chromatic "); }
9    double factor = 1.0/n;
10   double[] res = {0.0,0.0,0.0};
11   double[] pCoords;
12   for (int i = 0; i<processView.length; i++){
13     if (processView[i]!=null){
14       pCoords = parentNcBehaviour.getCoordinates(processView[i]);
15       res = LinearAlgebraHelper.vectorSum(
16            LinearAlgebraHelper.scalarVectorMultiply(factor, pCoords),res);
17     }
18   } return res;
19 }
```

5.6. Información acerca de herramientas de desarrollo

El programa se desarrolló en un sistema operativo Windows 7. El entorno de desarrollo integrado utilizado fue Eclipse Mars, versión (4.5.0). Se utilizó el kit de desarrollo de Java (JDK) versión 7. Se usó Maven para el control de dependencias. El sistema de control de versiones usado es Git, y el código del proyecto está alojado en GitHub (`https://github.com/fdsmora/DCCT`).

5.7. Conclusiones

En este capítulo revisamos cómo se implementaron las principales funcionalidades del programa descritas en el Capítulo 2. No discutimos cómo se implementaron algunas características como la consola `SCOutputConsole` o cómo se guardan las preferencias del usuario al cerrar el programa[18], pero esto se puede consultar fácilmente en el código del programa.

[18] Éste código está en el método `configViewer()` de la clase `jRealityView`

Capítulo 6

Conclusiones y trabajo futuro

El programa desarrollado en esta tesis esencialmente ilustra algunas de las ideas fundamentales del estudio de computación distribuida a través de la topología combinatoria: modelar todos los posibles estados globales en los que se puede encontrar un sistema distribuido conformado por n procesos secuenciales como un *complejo simplicial*; y mostrar cómo éste se transforma de acuerdo a las características del protocolo que ejecutan los procesos con el fin de resolver una tarea.

Las propiedades topológicas inducidas en los complejos simpliciales son de interés para el investigador ya que determinan el tipo de problemas que un sistema distribuido puede resolver así como el costo computacional requerido. El programa desarrollado facilita la observación de estos objetos.

Para concebir e implementar este proyecto fue primeramente necesario comprender la relación que existe entre la topología y la computación distribuida: entender los conceptos básicos de topología tales como complejo simplicial, simplejo y mapeo portador; y comprender los modelos de computación distribuida en términos de procesos, vistas locales, protocolos, mecanismos de comunicación, tipos de fallas y tareas.

Por otro lado, para lograr la construcción del programa se siguió un proceso de ingeniería de software. No se planeó formalmente un modelo de proceso de software a seguir al inicio del proyecto, sin embargo se podría decir que al final se siguió un modelo de desarrollo incremental [30] pues el sistema se desarrolló en incrementos, en donde cada uno era para probar una nueva idea o añadir una nueva funcionalidad y muchas veces se tuvieron que hacer cambios en el diseño a pesar de que ya se habían comenzado a implementar funcionalidades, tal como se discutió en las secciones 1.1. y 5.4.4, por ejemplo. También se podría decir que se siguió parte del proceso de ingeniería de software orientada a componentes [30] ya que se efectuó una búsqueda, análisis

y evaluación de componentes para la generación de visualizaciones en 3D, tal como se describió en la sección 1.4.

Se realizaron las principales actividades para el desarrollo de un producto de software: especificación, diseño e implementación y validación [30]. Mostramos un breve resumen de lo que se realizó durante estas actividades:

1. *Especificación.* Se especificaron los requerimientos funcionales y no funcionales, así como la descripción de los casos de uso.

2. *Diseño.* Se identificaron los componentes principales que conforman el programa. Se estructuró la arquitectura de acuerdo al popular patrón conocido como Modelo-Vista-Controlador y se diseñaron las clases que representan los principales objetos con los que el programa trabaja: procesos, simplejos, complejos simpliciales, protocolos de comunicación, escenarios de ejecución, vértices, caras, entre otros. Se usarón diagramas de clases UML para describir estos diseños.

   Para facilitar la extensión del programa con nuevas funcionalidades se documentó extensamente el código y en su diseño y construcción se aplicaron los siguientes patrones de diseño de software propuestos en [16]: *Singleton*, *Estrategia*, *Iterador*, *Método fábrica*, *Observador* y *Comando*.

3. *Implementación.* El programa se escribió en el lenguaje de programación Java y se utilizó una biblioteca de visualización matemática llamada jReality para generar las visualizaciones 3D de complejos simpliciales, la cual también permite que el programa pueda ser distribuido a través de una página web. También se implementaron controles de usuario para permitir construir complejos iniciales con a lo más tres procesos, generar complejos de protocolo basados en el modelo *immediate snapshot*, visualizar su representación cromática y no cromática; y personalizar las visualizaciones. Las funcionalidades de personalización y la calidad de las visualizaciones producidas por esta biblioteca hacen que éstas sean adecuadas para usarse en publicaciones.

   Para generar los complejos de protocolo se encontró una manera de generar todos los posibles escenarios de ejecución del protocolo *immediate snapshot* mediante la implementación de algoritmos combinatorios que generan las particiones ordenadas de un conjunto. Con estos escenarios se construyen los complejos de protocolo y eventualmente sus representaciones gráficas, cuyas coordenadas de vértices son calculadas mediante fórmulas basadas

en el cálculo de coordenadas de los vértices de la subdivisión baricéntrica de un complejo simplicial.

Debido a la complejidad de los procedimientos implementados para generar los complejos simpliciales en el programa, se usaron diagramas de secuencia para visualizar y entender mejor cómo se comportan dinámicamente e interactuan en tiempo de ejecución los objetos implementados en el programa.

4. *Validación.* Para validar que el programa cumpliera con los requerimientos funcionales y no funcionales principalmente se realizaron pruebas manuales durante su desarrollo. No se requirió realizar otro tipo de pruebas más sofisticadas ya que el sistema no es muy grande y no consume grandes cantidades de datos.

   Se aplicó también la *trazabilidad de requerimientos* [26], que es la propiedad que tiene cada requerimiento de ser claramente identificado y relacionado con otros. En el Capítulo 3 vemos cómo se relacionan unos requerimientos con otros en sus descripciones; en el Capítulo 4 en cada descripción del diseño se mencionó qué requerimiento se estaba buscando cumplir; y en el Capítulo 5, en algunos casos se mencionó que requerimientos se satisfacían al realizar determinada implementación, por ejemplo, al implementar la interfaz de usuario.

6.1. Trabajo futuro

Como trabajo futuro se sugiere implementar las siguientes funcionalidades y mejoras.

- *Generación de visualizaciones de complejos de protocolo para otros modelos tales como paso de mensajes, iterated snapshot, test and set, etcétera.* Por cada nuevo modelo solamente es necesario crear una clase que lo represente, la cual debe de extender clase abstracta `CommunicationProtocol` y proveer su propia lógica para generar todos los posibles escenarios de ejecución de las rondas de comunicación. El programa automáticamente producirá las visualizaciones de los complejos simpliciales resultantes (Ver los apéndices A y B).

  También se podría intentar desarrollar un algoritmo genérico para generar complejos de protocolo para *cualquier* modelo de computación distribuida, al cual solamente se le tendrían que suplir como parámetros las características del modelo en cuestión, tales como el mecanismo de comunicación, el número de fallas que se toleran, si las lecturas son de tipo *snapshot* o iteradas, etcétera. Adicionalmente se podría desarrollar una nueva interfaz de

usuario que permita introducir el código o pseudocódigo del algoritmo distribuido que ejecutarán los procesos y con base en este producir los complejos de protocolo correspondientes.

- *Construcción de complejos de entrada con más de un simplejo.* Un ejemplo de un complejo simplicial de entrada con varios simplejos se vió en el ejemplo de la tarea del consenso binario (Figura 2.6).

- *Mejoras en el acomodo de vértices y caras.* Como vimos en la sección 5.5.3, dependiendo de los valores de vistas de los procesos de los complejos de protocolo se calculan las coordenadas de los vértices de sus complejos simpliciales. A veces los vértices se enciman mucho al tener coordenadas similares y cuando hay muchos de éstos es díficil ver la estructura entera del complejo simplicial. Para ayudar a resolver este problema se permite al usuario arrastrar con el ratón cada vértice y así acomodar y "desenmarañar" los complejos con muchos vértices y caras encimadas (requerimiento1.4.3.). Sin embargo, cuando hay muchos vértices esto puede ser muy tardado. Para hacer la interacción más amigable, en este caso se podría implementar la funcionalidad de también permitir arrastrar caras y también implementar algún algoritmo que automáticamente separe y acomodé vértices y caras, minimizando el traslape entre éstos[1].

- *Visualización de complejos simpliciales para cuatro procesos.* Para lograrlo, en el código del programa se tendrían que proveer cuatro procesos como parámetros para la construcción de instancias de la clase `Simplex`. Para lograr su representación gráfica en forma de tetraedros se tendría que implementar un tratamiento especial para estos simplejos al momento de transformarlos en caras, el cual consistiría en construir una cara por cada subconjunto con tres vértices.

- *Visualización de simplejos por separado.* Esta funcionalidad consistiría en permitir al usuario seleccionar y examinar de forma aislada algún simplejo que forme parte de un complejo simplicial desplegado en pantalla. En la sección 5.5.2 se discutió un poco acerca de cómo se podría hacer esto.

- *Reducir el consumo de memoria del programa en tiempo de ejecución.* El programa tiene el problema de que su consumo de memoria RAM aumenta considerablemente (alrededor

[1]Estos algoritmos se estudian en el área de investigación conocida como *dibujo de gráficas*. Para más información se pueden consultar los libros descritos en la página `http://www.graphdrawing.org/books.html`. También los programas Gephi[1] y Graphviz[2] ofrecen implementaciones de algunos de estos algoritmos.

de 700 MB) cuando el complejo simplicial visualizado posee más de mil vértices. Esto sucede, por ejemplo, cuando se genera el complejo de protocolo de la tercera ronda del modelo *immediate snapshot* cromático para tres procesos. Se requiere realizar un análisis para determinar si esto es por causa de la biblioteca jReality o por el diseño de nuestro programa.

Como sugerencia para solucionar este problema se podría aplicar el patrón de diseño *Flyweight* [16] el cual se usa para resolver este tipo de problemas. Éste se aplicaría en el proceso de creación de vértices.

También se podría introducir paralelismo y programación multihilo para hacer más eficiente y rápido este programa.

- *Reimplementación del programa usando el software Gratrix WebGL Uniform Polyhedra para la generación de visualizaciones 3D.* En la sección 1.4 se sugirió usar este software si un día se decide reconstruir el programa desde cero. Las ventajas son que el acceso es mediante una interfaz web que no requiere permisos de seguridad y que aparentemente el consumo de memoria es bajo.

Como actividad inicial para comenzar la mejora y extensión de este programa revisar el apéndice C.

Apéndice A

Generación de complejos simpliciales para otros modelos

Una vez desarrolladas las funcionalidades básicas del programa descritas en esta tesis se continuó extendiendolo con la capacidad de generar complejos simpliciales de protocolo para dos nuevos modelos de computación distribuida, también basados en una memoria compartida: El modelo *read/write* (WR)[9] y el modelo *Atomic Snapshot* (AS)[19]. En éste apéndice solo describimos el modelo WR[1].

En las Figuras A.1 y A.2 se muestran los complejos de protocolo generados por el programa para el protocolo WR para 3 procesos en las versiones cromática y no cromática respectivamente.

A.1. Modelo *read/write* (WR)

Este modelo consiste en $n+1$ procesos denotas por los números $[n] = \{0, 1, ..., n\}$. Los procesos se comunican mediante una memoria compartida `mem`, la cual consisten en $n+1$ registros *SWMR* [2]. Cada proceso accede a la memoria invocando las operaciones atómicas `write`(x) o `read`(j), $0 \leq j \leq n$. En su primera operación, el proceso i escribe su entrada en `mem[i]`, después lee cada uno de los $n+1$ registros en un orden arbitrario. Esta secuencia de operaciones, consistente en una escritura seguida de todas las lecturas de todos los registros se abrevia `WScan(i)`[9].

Las operaciones tipo *snapshot* no son implementadas por los procesadores reales. En realidad

[1] El modelo AS fue más simple de implementar. Su código se puede consultar en las clases que están en el paquete `unam.dcct.model.nonimmediatesnapshot`.

[2] Single Writer/Multiple Reader

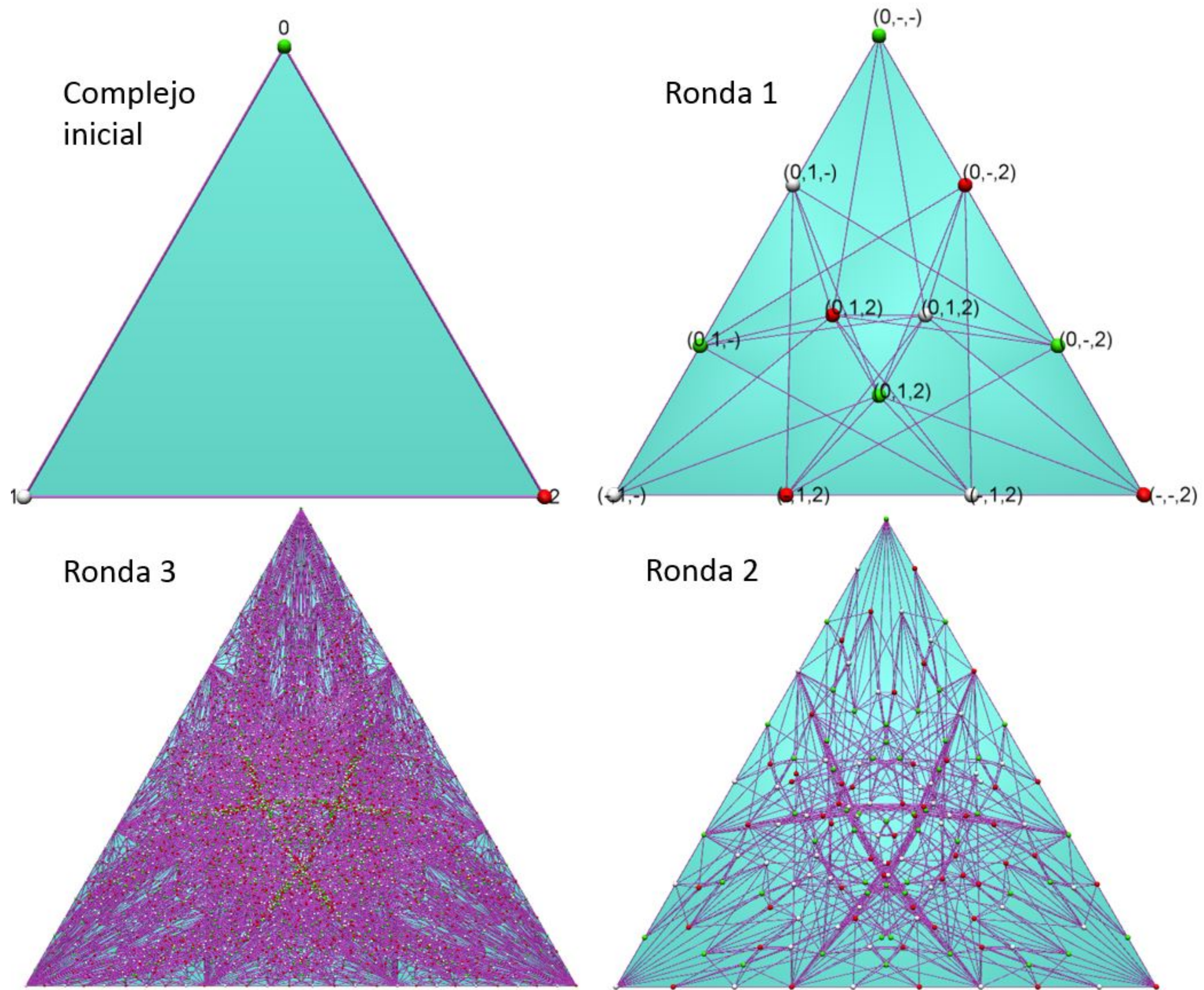

Figura A.1: Complejo inicial y complejos de protocolo producidos a lo largo de la ejecución de tres rondas de comunicación del protocolo *read/write* (WR) entre tres procesos: p_0 (verde), p_1 (blanco) y p_2 (rojo). Cada vértice se etiqueta con la vista obtenida por cada proceso al final de la ronda de comunicación (las etiquetas en los demás complejos se omiten por claridad en las imagenes).

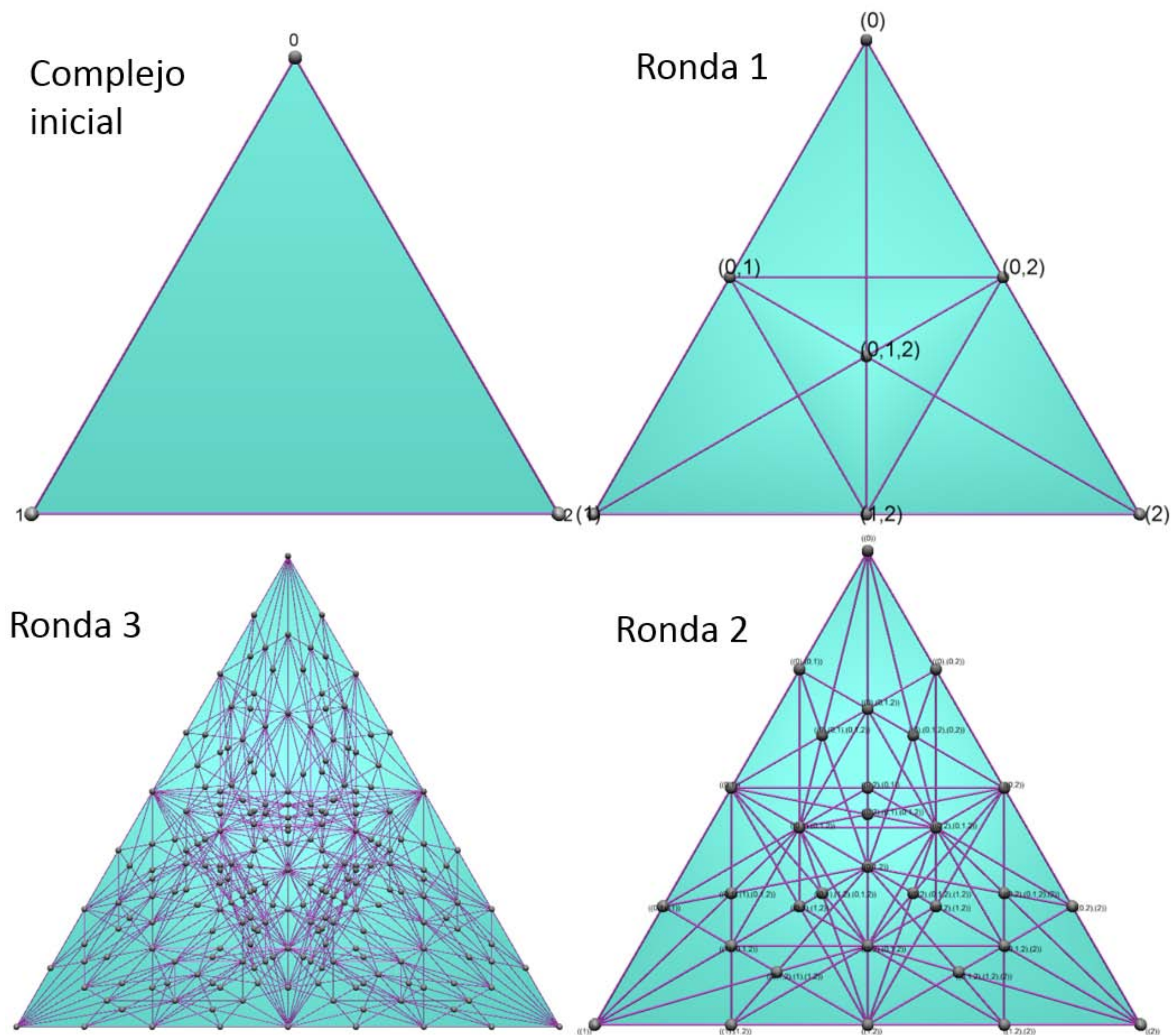

Figura A.2: Representación no cromática de los complejos de la Figura A.1.

estas son implementadas como operaciones que escriben y leen localidades de memoria compartidas. Los protocolos *read/write* son computacionalmente equivalentes a los protocolos tipo *snapshot* [9].

A.1.1. Generación de complejos de protocolo

Como se vió en la sección 5.4, para que el programa genere un complejo de protocolo debe de primero generar todos los posibles escenarios de ejecución de una ronda de comunicación del protocolo. Para la generación de los escenarios de ejecución del protocolo WR se adoptó un enfoque diferente al de los escenarios del protocolo *immediate snapshot*.

Mientras que para generar todas las posibles vistas de los procesos después de una ronda del protocolo *immediate snapshot* el programa simula las escrituras y lecturas de todos los procesos basándose en cada posible escenario de ejecución, para el protocolo WR se buscó generar directamente las vistas de los procesos sin necesidad de efectuar la simulación de los escenarios de ejecución.

Como se vió en la sección 5.4, las vistas de los procesos p_i después de una ronda del protocolo *immediate snapshot* satisfacen ciertas condiciones; de igual manera, se observó que las vistas v_i de los procesos p_i después de una ronda del protocolo WR solo satisfacen las siguientes condiciones:

1. *Autocontención*: $i \in v_i$

2. *Lectura*: Para todo i, j, si $i \notin v_j$ entonces $j \in v_i$

   El algoritmo para generar todas las posibles vistas que corresponden a ejecuciones válidas del protocolo WR consistió entonces en generar, para cada $i \in [n]$ todos los subconjuntos de $[n]$ que satisfagan estas condiciones. Una vez teniendo estos subconjuntos es fácil convertirlos a vistas de procesos.

Algoritmo

Cada simplejo generado se representa como un a matriz $M[i][j]$, $0 \leq i \leq j \leq n$ tal que $M[i][j] = 1$ si en la ejecución de una ronda del protocolo p_i leyó la vista del proceso p_j y $M[i][j] = 0$ si no la leyó. El algoritmo consiste entonces en generar todas las posibles combinaciones de 0 y 1 en M tal que se satisfagan las condiciones antes descritas. Cada combinación de 0 y 1 en la matriz representa las vistas válidas de los $n + 1$ procesos que corresponden a una ronda de ejecución del protocolo WR, en otras palabras, representa un simplejo del complejo de protocolo.

En el listado A.1 mostramos el código de los métodos que implementan el algoritmo. Esté código está en la clase `WR_ScenarioGenerator` que se ubica en el paquete `unam.dcct.model.WR`.

El método `generateMatrices(n)` genera las matrices que representan cada simplejo válido del protocolo. Primero genera las $\binom{n}{n-1}$-combinaciones del conjunto $[n]$ y por cada `combination` llena la matriz `M` en su totalidad con 1 y después ejecuta el método `setNextValue(M,0,0,combination)`.

El método `setNextValue(M,0,0,combination)` establece en 0 cada posición `M[i][j]`, donde `i=combination[current]`. Las líneas 14 y 15 implementan las condiciones que deben de cumplir subconjuntos que representan las vistas. Como la matriz inicialmente esta llena de 1, entonces el algoritmo comienza estableciendo 0 en la posición actual (línea 16). Las líneas 19 y 24 establecen recursivamente el valor correspondiente en la siguiente posición. El método `produce(M)` en la línea 22 simplemente indica que la matriz en su estado actual representa una combinación de 1 y 0 que representan un simplejo válido. Por brevedad omitimos su implementación. Finalmente en las líneas 26 y 27 se incrementa el valor en la posición actual para encontrar nuevas combinaciones de 1 y 0. Como vemos esté algoritmo es similar a un *backtracking*[3].

Listado A.1: Código del algoritmo que genera los simplejos del protocolo WR

```
1   protected void generateMatrices(int n) {
2     int[][] M = new int[n][n];
3     List<List<Integer>> combinations = generateCombinations(n, n-1);
4     for (List<Integer> combination : combinations ){
5       fillWithOnes(M);
6       setNextValue(M, 0, 0, combination);
7     }
8   }
9
10  private void setNextValue(int[][] M, int current, int j, List<Integer>
        combination) {
11    int n = M.length;
12    int i = combination.get(current);
13    while(true){
14      if (j>i ||
15            (j<i && M[j][i]==1))
16        M[i][j]=0;
17      while (true){
18        if (j+1 < n)
```

[3] https://en.wikipedia.org/wiki/Backtracking

```
19                 setNextValue(M, current, j+1, combination);
20              else {
21                 if (current+1==combination.size())
22                    produce(M);
23                 else
24                    setNextValue(M, current+1, 0, combination);
25              }
26              if (M[i][j]<1)
27                 ++M[i][j];
28              else return;
29           }
30        }
31     }
```

Implementación

Para implementar éste modelo en el programa se creo la clase `WriteRead` en el paquete `unam.dcct.model.WR`. Para integrar este modelo en el programa esta clase debe de heredar la clase abstracta `CommunicationProtocol` e implementar los método requeridos. También se deben implementar clases que implementen las interfaces `Scenario` e `Iterator<Scenario>` e `Iterable<Scenario>`. Esto se ve en el diagrama de la Figura A.3. Este mecanismo para integrar nuevos modelos se discutió en la sección 4.2.3.

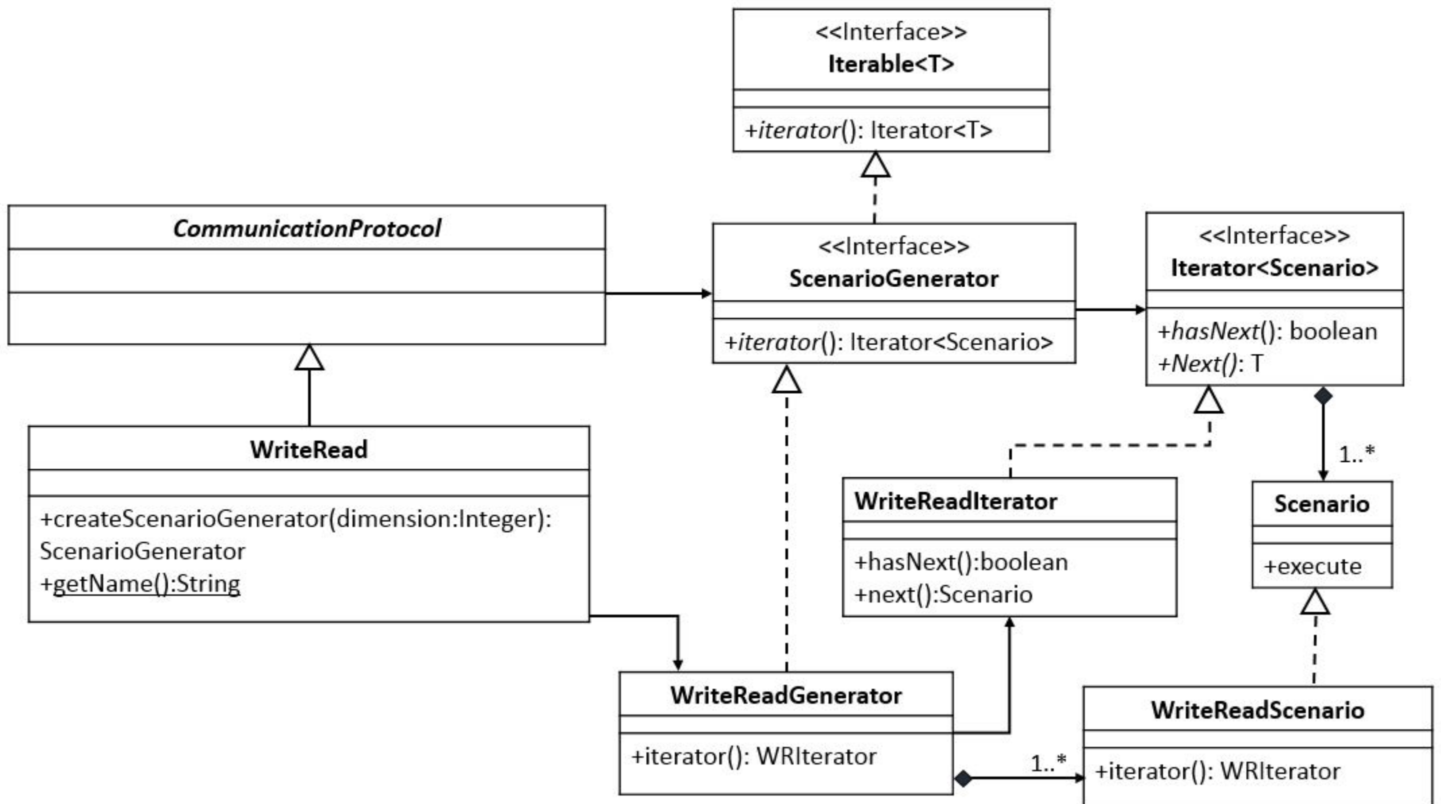

Figura A.3: Diagrama UML que muestra cómo se integran al programa las clases que implementan el modelo WR. Nótese que este diagrama es la versión de la Figura 4.6 para el protocolo WR.

Apéndice B

Ejemplo de cómo se integra un nuevo modelo al programa

B.1. Introducción

En este apéndice se describe cómo se puede hacer para que el programa genere complejos de protocolo para un nuevo modelo de computación distribuida (protocolo). Este ejemplo consiste en un modelo de prueba que genera el complejo de la Figura B.1. [1]

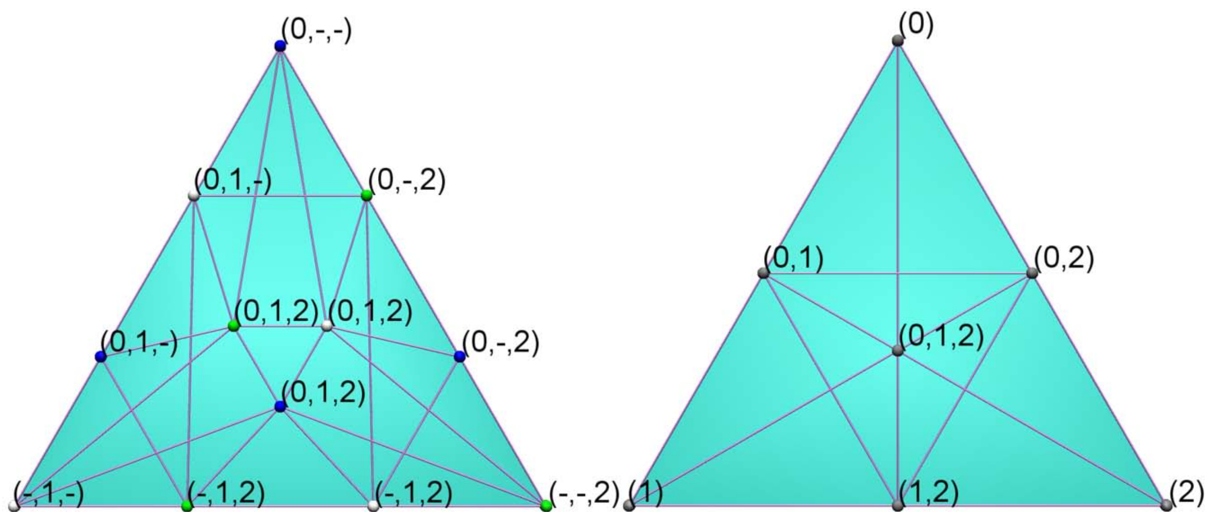

Figura B.1: Complejo de protocolo cromático y representación no cromática para tres procesos generado por una ronda de ejecución de un protocolo de prueba.

[1] Este complejo, aunque está basado en el protocolo *immediate snapshot*, no corresponde a ningún modelo real, solamente se creó para ilustrar el procedimiento de cómo un usuario puede generar complejos de protocolo basados en su propio modelo.

B.2. Prerequisitos

Para entender mejor esta guía es deseable estar familiarizado con los siguientes temas:

1. Programación en Java. En particular los siguientes temas:
    1.1. Clases anónimas [2].
    1.2. Genéricos [3].
2. Programación orientada a objetos.
3. Patrones de diseño de software: *Iterador*, *Método fábrica*, *Comando* (se pueden consultar en [16].

También es recomendable haber leído esta tesis, en especial los capítulos 4 y 5.

B.3. Descripción del ejemplo

Como se mencionó en la sección 4.2.3, el desarrollador que desee implementar un nuevo modelo debe proveer su propia clase que herede de la clase `CommunicationProtocol`. A su vez, debe proveer clases que implementen los métodos definidos en las interfaces `Iterable<Scenario>`, `Iterator<Scenario>` y `Scenario`.

En este ejemplo la clase `TestProtocol` hereda de la clase `CommunicationProtocol`; la clase `TestProtocolScenario` implementa la interfaz `Scenario`; y la clase `TestProtocolIterator` implementa la interfaz `Iterator<Scenario>`. Para el caso de la interfaz `Iterable<Scenario>` se define una clase anónima en el método `createScenarioGenerator()` de la clase `TestProtocol` (Listado B.1, línea 5).

En la Figura B.2 se muestra el diagrama de clases que ilustra esta implementación. Éstas clases se encuentran en el paquete `unam.dcct.model.tutorial`.

Listado B.1: Implementación de la interfaz Iterable<Scenario>como una clase anónima en el método `createScenarioGenerator()` de la clase `TestProtocol`

```
1  @Override
2  protected Iterable<Scenario> createScenarioGenerator(int dimension) {
3     final List<int[][]> allScenarios = getAllScenarios(dimension);
```

[2]`https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html`
[3]`https://docs.oracle.com/javase/tutorial/java/generics/`

```
4
5      return new Iterable<Scenario>(){
6         @Override
7         public Iterator<Scenario> iterator() {
8            return new TestProtocolIterator(allScenarios);
9         }
10     };
11  }
```

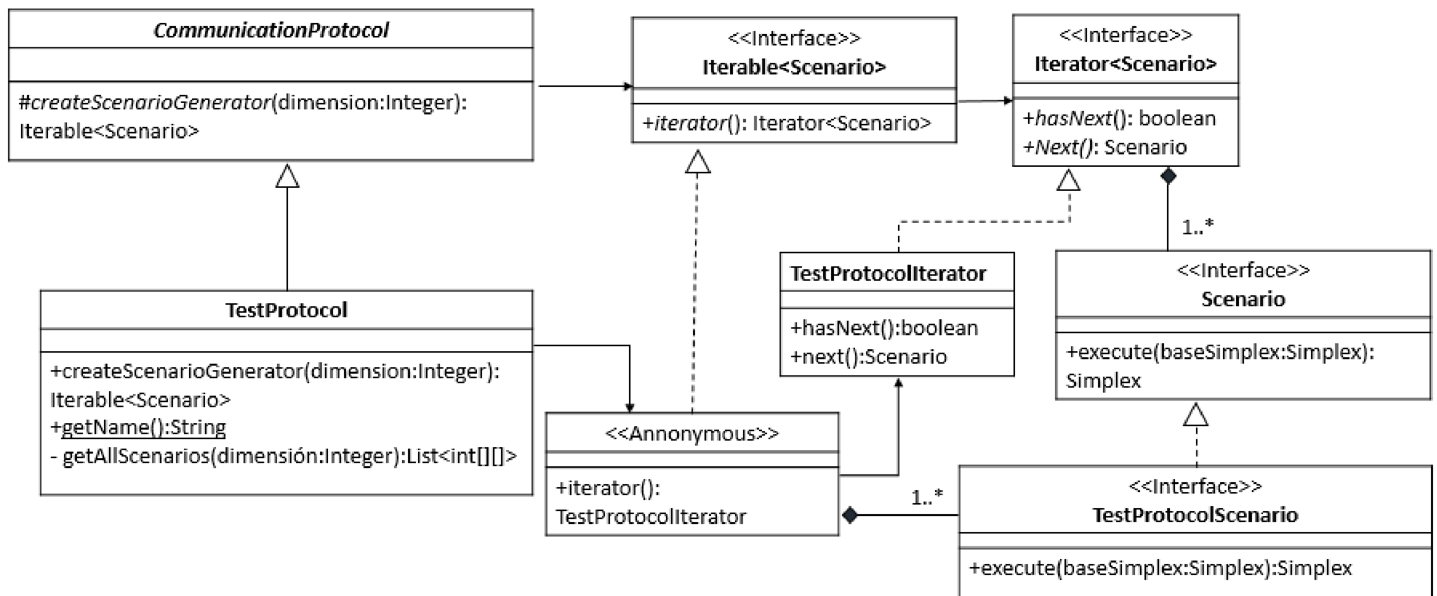

Figura B.2: Diagrama UML que muestra cómo se integran al programa las clases que implementan el protocolo de prueba. Éste diagrama es similar al de las Figuras A.3 y 4.6.

B.3.1. Funcionamiento

Cuando el usuario genera un complejo de protocolo al presionar el botón “Next round”, sucede la secuencia de acciones descritas en la sección 5.3.2. En el caso de este protocolo de prueba, eventualmente se ejecutará el método `createScenarioGenerator(dimension)` de la clase `TestProtocol`. Primero se obtienen todos los posibles escenarios de una ronda de ejecución del protocolo (Listado B.1, línea 3) al invocar el método `getAllScenarios(dimension)`. Este método primero intenta obtener la lista de escenarios correspondientes a `dimension` (que es la dimensión del simplejo para el cual se están generando los escenarios) de la lista `allScenariosPerDimension`.

En caso de que no exista tal lista, quiere decir que es la primera vez que se van a generar estos escenarios, por lo tanto se solicita al método `generate(dimension)` de la clase `TestProtocolScenarioGenerator` que los genere, y al hacerlo estos se almacenan en la entrada correspondiente a `dimension` en la lista `allScenariosPerDimension`. Cuando se ha generado la lista de escenarios se crea el objeto de la clase anónima `Iterable<Scenario>` (el cual permitirá recorrer y ejecutar cada escenario para así lograr crear el complejo de protocolo de la ronda actual (Listado B.1, línea 5).

Codificación de escenarios

En este ejemplo cada escenario de ejecución se representa cómo una matriz de ceros y unos que códifica cómo quedarón las vistas de los procesos al final de la ronda de ejecución del protocolo, tal cómo se hizo en la sección A.1.1.

Estas matrices se declaran como atributos de la clase `TestProtocolScenarioGenerator`. (Listado B.2). De hecho, este método manual de especificar los escenarios de ejecución de una ronda de algún nuevo protocolo es una manera muy fácil y rápida probarlo: simplemente el desarrollador tiene que declarar todas las matrices de ceros y unos que correspondan la las vistas de procesos después de la ejecución de cada escenario[4]. El otro método para generar los escenarios de ejecución es de forma automática mediante los algoritmos que se han descrito en esta tesis.

Listado B.2: Declaración de las matrices que representan las vistas de los procesos después de ejecutar una ronda del protocolo de prueba. Este código se encuentra en la clase `TestProtocolScenarioGenerator`

```
1 // Three processes (dimension 2) scenarios
2   private static int[][][] mat2 = new int[][][] {
3     // Dummy scenarios
4     {{1,0,0},{1,1,0},{1,0,1}},
5     {{1,1,0},{0,1,0},{0,1,1}},
6     {{1,0,1},{0,1,1},{0,0,1}},
7     {{1,1,0},{1,1,0},{0,1,1}},
8     {{1,0,1},{0,1,1},{1,0,1}},
```

[4] **ADVERTENCIA: Si el desarrollador desea seguir este método, tiene que declarar los escenarios para todas las dimensiones de simplejos soportadas por el programa, desde la 0 hasta la n (tal como se hace en el ejemplo), de lo contrario el programa podría fallar al intentar generar un complejo de protocolo.**

```
9       // Immediate Snapshot execution scenarios
10      {{1,1,1},{0,1,0},{1,1,1}},
11      {{1,1,1},{1,1,1},{0,0,1}},
12      {{1,1,0},{1,1,0},{1,1,1}},
13      {{1,0,1},{1,1,1},{1,0,1}},
14      {{1,1,1},{0,1,1},{0,1,1}},
15      {{1,0,0},{1,1,0},{1,1,1}},
16      {{1,0,1},{1,1,1},{1,0,1}},
17      {{1,1,0},{0,1,0},{1,1,1}},
18      {{1,1,1},{0,1,0},{0,1,1}},
19      {{1,1,1},{0,1,1},{0,0,1}},
20      {{1,0,1},{1,1,1},{0,0,1}},
21      {{1,0,0},{1,1,1},{1,0,1}},
22      {{1,0,0},{1,1,1},{1,1,1}},
23      {{1,1,1},{1,1,1},{1,1,1}}
24    };
25    // Two processes (dimension 1) scenarios
26    private static int[][][] mat1 = new int[][][] {
27      {{1,0},{1,1}},
28      {{1,1},{1,1}},
29      {{1,1},{0,1}}
30    };
31    // One process (dimension 0) scenario
32    private static int[][][] mat0 = new int[][][] {
33      {{1}}
34    };
```

TestProtocolIterator

Como vemos, en el Listado B.1, línea 8, se crea una instancia de esta clase. Esta clase permite iterar sobre los escenarios de ejecución del protocolo de prueba.

TestProtocolScenario

Esta clase encapsula una matriz de ceros y unos y al invocar su método `execute(baseSimplex)` crea un nuevo simplejo cuyos procesos tienen las vistas definidas por dicha matriz.

B.3.2. Conclusión

Unas vez implementadas todas las clases descritas en este apéndice, el programa automáticamente incorpora el nuevo modelo y adquiere la capacidad de generar los complejos de protocolo para éste. Para más detalles acerca de cómo funciona este ejemplo consultar el código del programa.

Apéndice C

Recomendaciones para el nuevo desarrollador

Para el desarrollador que desee continuar extendiendo y mejorando este programa, ademas de leer completamente esta tesis (poniendo especial atención en la sección de “Trabajo futuro” y el apéndice B), se recomienda tener conocimientos y habilidades en lo siguiente:

- Fundamentos de la teoría de computación distribuida mediante topología combinatoria, lo cual se puede estudiar en los primero tres capítulos de [19].
- Programación en Java nivel intermedio o avanzado.
- Patrones de diseño de software. El libro clásico es [16] y una introducción muy buena y amigable es [15].
- jReality. Para esto se recomienda estudiar los tutoriales en `http://www3.math.tu-berlin.de/jreality/mediawiki/index.php/Developer_Tutorial`.

Derivado de la experiencia obtenida en este trabajo, también se hacen las siguentes recomendaciones:

- Usar Maven para el manejo de dependencias.
- Usar Git como controlador de versiones.
- Realizar pruebas unitarias durante el desarrollo.
- Automatizar tareas repetitivas (por ejemplo, subir las nuevas versiones al servidor) mediante el desarrollo de *scripts*. Quizás trabajar en un entorno Linux podría ayudar en esto.

Bibliografía

[1] Gephi - The Open Graph Viz Platform. http://gephi.github.io/.

[2] Graphviz - Graph Visualization software. http:/http://www.graphviz.org/.

[3] Gratrix.net - WebGL Uniform Polyhedra. http://gratrix.net/polyhedra/webgl/poly.html.

[4] iGraph - The network analysis package. http://igraph.org/.

[5] JavaView - Interactive 3D Geometry and Visualization. http://www.javaview.de/.

[6] jReality - A Java library for real-time interactive 3D graphics and audio. http://www3.math.tu-berlin.de/jreality/.

[7] Sagemath - Open Source Mathematical Software System. http://www.sagemath.org/.

[8] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, Septiembre 1993.

[9] F. Benavides and S. Rajsbaum. The read/write protocol complex is collapsible. In *Latin American Theoretical Informatics Symposium*, 2016.

[10] D. Berend and T. Tassa. Improved bounds on bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 2010.

[11] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 41–51, Nueva York, NY, USA, 1993. ACM.

[12] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley Publishing Co., Nueva York, NY, USA, 2000.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., Nueva York, NY, USA, 1996.

[14] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Abril 1985.

[15] E. Freeman, E. Robson, B. Bates, and K. Sierra. *Head First Design Patterns*. O'Reilly Media, 2004.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] E. Gawrilow and M. Joswig. Polymake: An approach to modular software design in computational geometry. In *Proceedings of the Seventeenth Annual Symposium on Computational Geometry*, SCG '01, pages 222–231, Nueva York, NY, USA, 2001. ACM.

[18] A. Hanson. Scientific visualization: Advances and challenges, book review. *Computational Science and Engineering, IEEE*, 2(4):87, 1995.

[19] M. Herlihy, D. Kozlov, and S. Rajsbaum. *Distributed Computing through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., 2014.

[20] M. Herlihy and S. Rajsbaum. Set consensus using arbitrary objects (preliminary version). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 324–333, Nueva York, NY, USA, 1994. ACM.

[21] M. Herlihy, S. Rajsbaum, and M. Raynal. Computability in distributed computing: A tutorial. *SIGACT News*, 43(3):88–110, August 2012.

[22] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, Noviembre 1999.

[23] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[24] A. Knopfmacher and M. Mays. A survey of factorization counting functions. *International Journal of Number Theory*, 1(04):563–581, 2005.

[25] D. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search.* ACM, Nueva York, NY, USA, 1999.

[26] P. Laplante. *Requirements Engineering for Software and Systems.* Taylor & Francis Group, 2009.

[27] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., 1996.

[28] R.C. Martin. *Agile Software Development, Principles, Patterns, and Practices.* Pearson, 2012.

[29] K. Polthier. Visualizing mathematics—online. In *Mathematics and Art*, pages 29–42. Springer, 2002.

[30] I. Sommerville. *Software Engineering: (Update) (9th Edition) (International Computer Science).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2011.

[31] R. Stanley. *Enumerative Combinatorics*, volume 2. Cambridge University Press, 2001.

[32] D. Stanton and D. White. *Constructive Combinatorics.* Springer-Verlag New York, Inc., Nueva York, NY, USA, 1986.

[33] G. Ziegler. *Lectures on polytopes*, volume 152. Springer Science & Business Media, 1995.