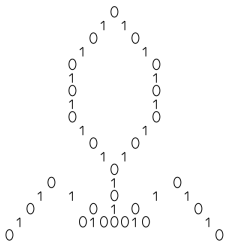
Universidad Nacional de Río Cuarto
Fac. de Cs. Exactas, Fco-Qcas y Naturales
Departamento de Computación



FuDePAN
Fundación para el Desarrollo de la Programación en Ácidos Nucleicos

Trabajo Final
Licenciatura en Ciencias de la Computación

CombEng

Implementation of a Distributed Combinatory Engine over FuD and its Application to Bioinformatic Problems

---

Autores
Bettiol, Favio
Diaz, Diego

Director
Biset, Guillermo

Co-Director
Gutson, Daniel

7 de diciembre de 2011

Agradecimientos

Este trabajo no se habría podido realizar sin la colaboración de muchas personas que nos han brindado su ayuda, sus conocimientos y su apoyo. Queremos agradecerles a todos ellos por cuanto han hecho por nosotros para que este trabajo saliera adelante de la mejor manera posible.

Queremos agradecerle especialmente a los dos directores de esta tesis: Daniel Gutson y Guillermo Biset. Ellos han sido los mentores de este proyecto y son quienes confiaron en nosotros para que lo llevemos adelante. Nos han guiado con mucha dedicación y paciencia durante el desarrollo del mismo, y, gracias a esto, hemos adquirido valiosos conocimientos, tanto en aspectos académicos como en muchos otros más.

A los autores del proyecto *RecAbs*, Emanuel Bringas y Mariano Bessone, por estar presentes siempre que necesitamos ayuda con su proyecto. Esperamos haberles sido de tanta utilidad como ustedes lo han sido para nosotros.

Al autor del proyecto *VAC-O*, Santigo Videla, y a los autores del proyecto *ASO*, Andrés Peralta Godoy y Ezequiel Velez, quienes fueron consultados varias veces por sus trabajos, de los cuales se hace un extenso uso de los mismos.

A Cecilia Cammisa y Daniel Rabinovich por enseñarnos y corregirnos innumerables cosas en cuanto a Biología. Cuando la situación lo requería, han ayudado con la mejor predisposición.

A Marcelo Arroyo y Nazareno Aguirre por su buena voluntad para ayudarnos con las formalidades requeridas por la universidad que este trabajo demandó.

Al Departamento de Computación y a su director, en su momento Nazareno, por facilitarnos las salas de máquinas para realizar pruebas a escalas un poco mayores.

A todos los miembros de la fundación **FuDePAN** y, en especial, a su presidente Daniel Gutson, por ser personas que, sin esperar nada a cambio, trabajan incansablemente sólo por su amor a la investigación.

Finalmente, y no por eso menos importante, sino todo lo contrario, queremos agradecerles muy especialmente a nuestras familias por habernos apo-

yado incondicionalmente a lo largo de toda la carrera. Esperamos que este trabajo, y el logro que el mismo conlleva, sea una mínima retribución a tantas cosas que ustedes nos han brindado durante los últimos 6 años de estudio.

Índice general

Índice de figuras

Índice de cuadros

Parte I

Preliminares

Capítulo 1

Introducción

Los virus tales como el HIV (Virus de Inmunodeficiencia Humana) no pueden reproducirse por sí mismos, sino que precisan de la maquinaria celular para lograrlo. Por esta razón, deben infectar a las células de un organismo vivo para duplicarse, es decir, hacer copias nuevas de ellos mismos. A menudo, el sistema inmunológico elimina las partículas virales que pueden ingresar en un organismo, no obstante, el HIV ataca el sistema inmunológico mismo, aquel que se encarga de eliminarlas.

Por otro lado, el SIDA (Síndrome de Inmunodeficiencia Adquirida) es una afección médica. A una persona infectada por el virus HIV se le diagnostica SIDA cuando su sistema inmunológico es demasiado débil para combatir las infecciones.

Desde su primera manifestación, allá por el año 1981, hasta la actualidad, el tratamiento de la infección por el HIV ha sido, y seguirá siendo, foco de numerosas investigaciones. Se han propuesto diferentes enfoques a lo largo de estos años como terapias para tratamiento del SIDA, pero la terapia más comúnmente utilizada consiste en una combinación de diferentes antirretrovirales.

Desde 1987, año en que se aprobó el uso de la *zidovudine*[1] en la prevención de la replicación del HIV, hasta 1995, la terapia consistía en la administración de un solo antirretroviral para disminuir la carga viral en sangre, tratamiento más conocido como monoterapia. En 1996, los avances significativos sobre el comportamiento del HIV y la expansión en la síntesis de diferentes clases de antirretrovirales, hicieron posible el paso de la monoterapia hacia terapias combinatorias de alta eficacia, cuyo fin radica en la administración simultánea de distintos antirretrovirales pertenecientes a diferentes grupos. Esta terapia con "cocktails" de antirretrovirales que recibe el nombre de Tratamiento

[1]Inhibidor nucleósido de transcriptasa reversa. Se vende bajo los nombres de **Retrovir** y **Retrovis**.

Antirretroviral de Gran Actividad o HAART (*Highly Active Antiretroviral Therapy*)[Cichocki, 2009] ha tenido un efecto dramático dado que, en esencia, la terapia de combinación sofoca las mutantes de HIV antes de que tengan chances de florecer.

Sin embargo, no todas las personas se adhieren del mismo modo al tratamiento antirretroviral. La adherencia es una cuestión de vital importancia ya que contribuye a evitar la resistencia a los fármacos, de otra forma, se puede dar lugar a la aparición de mutantes del HIV que ya no sean susceptibles a los efectos de la medicación que se toma.

Ahora bien, *¿El desarrollo de mutantes asociadas a la resistencia de los antirretrovirales, tiene alguna implicancia en la estructura secundaria del ARN Viral?*

Este trabajo intenta dar un abordaje inicial a lo antes mencionado, mediante el desarrollo de una aplicación de software que, tomando como entrada una secuencia inicial del virus y los antirretrovirales disponibles hasta el momento, permita analizar cómo las diferentes terapias pueden afectar, o no, a la **estructura secundaria** y si esta posible afección puede ser un factor en la evolución viral.

Para ello, se obtiene el valor ΔG del virus original y se lo compara con aquellos ΔG de las secuencias mutantes resultantes de aplicar una terapia y con secuencias que, a pesar de tener la misma secuencia aminoacídica que las resistentes, no se presentan en los pacientes.

Dado que todo este proceso es muy costoso computacionalmente, se opta por desarrollar la aplicación como una capa de un framework para aplicaciones distribuidas, desarrollado por integrantes de la fundación y que recibe por nombre **FuD** (**F**uDePAN **U**biquitous **D**istribution).

Como parte de este trabajo final, se realiza el acoplamiento de otra nueva capa al framework proveyendo, entre otras cosas, un motor combinatorio. Este último es quién facilitará la obtención de las posibles terapias.

La decisión de implementar esta última capa se debe a la gran cantidad de problemas que se presentan en la fundación, de caracter bioinformático, que requieren de la generación de combinaciones en diferentes sabores y el problema tratado aquí no escapa a ello.

Capítulo 2

Marco Teórico

En este capítulo se introducirá al lector en el framework **FuD** y en los conceptos básicos de biología para un mayor entendimiento de la aplicación implementada como parte de este trabajo final.

2.1. El Framework FuD

Computar grandes conjuntos de datos es una parte importante de las ciencias de la computación, mucha información importante es inherentemente complicada de comprimir y, además, los recursos para procesar estos conjuntos de datos son usualmente costosos. Las organizaciones sin fines de lucro, las instituciones educativas y similares deben encontrar otros caminos para administrar sus requerimientos de procesamientos mientras mantienen sus costos lo más bajo posible. Existen muchas herramientas para lograr una forma de computación distribuida en bajo costo.

FuD (del acrónimo **F**uDePAN **u**biquitous **D**istribution) consiste en el diseño e implementación de un framework abstracto para implementar algoritmos distribuidos independientes de las herramientas antes mencionadas, de los sistemas de comunicación subyacentes o algún otro problema específico.

Este framework es un sistema separado en las capas de *aplicación*, *administración* y *distribución* combinando los conceptos de cliente-servidor y Divide & Conquer. Tanto el cliente como el servidor se encuentran organizados en estos tres niveles (muy) separados, cada uno de ellos con una única responsabilidad bien definida.

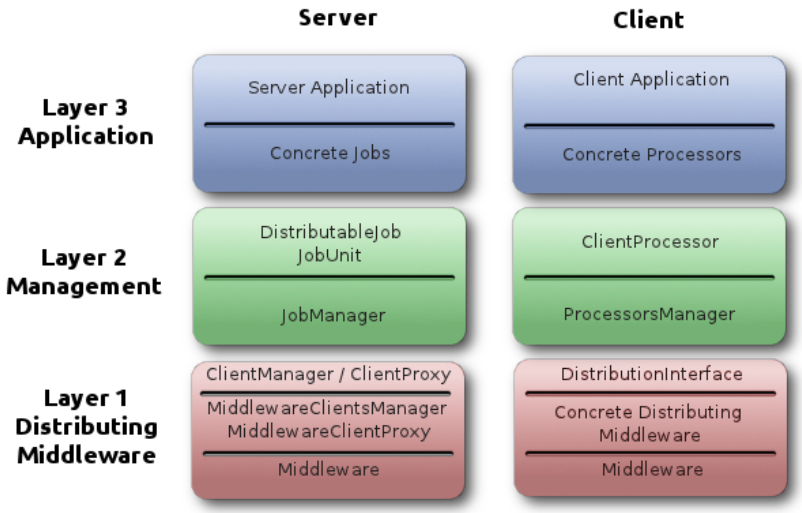


Figura 2.1: Vista abstracta de las capas en una instancia de uso de **FuD**

La comunicación entre los diferentes niveles se encuentra estrictamente limitada, es decir, por cada nivel existe un único punto de comunicación ya sea para comunicarse con la capa superior o con la inferior siguiendo el enfoque OSI para redes.

A continuación, muy brevemente, se explican cada una de las capas:

2.1.1. Application Layer (L3)

Este nivel provee los componentes que contienen todos los aspectos del dominio del problema. Estos aspectos incluyen todas las definiciones de los datos usados y su manipulación correspondiente, como así también todos los algoritmos relevantes para la solución al problema en general. Se debe tener en cuenta que bajo ninguna circunstancia esta capa pertenecerá a **FuD** pero su presencia es de ayuda para mostrar el uso del framework.

2.1.2. Job Management Layer (L2)

La responsabilidad de esta capa es manejar los trabajos que se desean distribuir como así también generar las unidades de trabajo que serán entregadas a los clientes para su procesamiento. Estas unidades de trabajo llegan a su cliente correspondiente gracias a la capa más baja, encargada de la distribución. Una vez finalizado el procesamiento, es el nivel 2 quien informa que todo ha terminado y otorga los resultados a la capa superior.

2.1.3. Distributing Middleware Layer (L1)

A partir de esta vista abstracta del diseño, se debe notar que L1 solo constituye un esquema de administración de clientes en particular. Tal y como se observa en la figura 2.1, en ambos lados las partes fijas son las intarfaces del middleware, mientras que las implementaciones concretas son las partes variables (por ejemplo, BOINC o MPI).

Notar que si se toma al proyecto **FuD** por separado, el mismo constituye una biblioteca que esta parcialmente implementada siempre y cuando el middleware de distribución estándar (boost::asio) sea removido. Esto lleva a que se pueden obtener diferentes sabores de la biblioteca **FuD** con sólo reemplazar la capa de distribución (L1) por alguna otra. Por ejemplo, se podría utilizar MPI [1], recomendado para clusters locales donde sus computadoras disponen de una interconexión veloz. Otra alternativa podría ser BOINC[2] que posee una riqueza extrema en procesamiento, pero ésta es llevada a cabo pagando el precio de la comunicación a través de Internet (la cual es bastante más baja en velocidad con respecto a otras configuraciones como Ethernet, infiniband, etc.). Estas diferentes implementaciones de la capa 1 deben ser intercambiables, es decir, uno puede variar entre ellas y el problema debe continuar siendo solucionable correctamente.

2.2. RNA Folding Free Energy

Como parte de esta tesis se implementó una aplicación de gran interés para la fundación **FuDePAN**[3]. La descripción e implementación de la misma se encuentra en el capítulo 8, en esta sección presentamos los conceptos básicos necesarios para comprender con mayor detalle la aplicación.

2.2.1. Organismos, Moléculas y Células

Organismos

Conjunto de biomoléculas orgánicas e inorgánicas organizadas de manera especifica para cumplir determinadas funciones e interactuar con el medio en que se encuentran.

[1] `http://www.mcs.anl.gov/research/projects/mpi/`
[2] `http://boinc.berkeley.edu/`
[3] http://www.fudepan.org.ar/

Moléculas Orgánicas

Hay dos tipos de moléculas orgánicas básicas:

- *Nucleótidos*: Pequeñas moléculas constituyentes de macromoléculas de mayor complejidad. Los 5 tipos de nucleótidos más importantes son:
  1. Adenina (A) compone el ADN y el ARN.
  2. Guanina (G) compone el ADN y el ARN.
  3. Timina (T) compone el ADN.
  4. Citosina (C) compone el ADN y el ARN.
  5. Uracilo (U) compone el ARN.
- *Aminoácidos*: Son moléculas que conforman las proteínas y son esenciales para la vida. A continuación se muestra una tabla conteniendo todos los aminoácidos y sus abreviaciones.

Nombre	Ab. 3 letras	Ab. 1 Letra
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartic acid	Asp	D
Cytesine	Cys	C
Glutamic acid	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

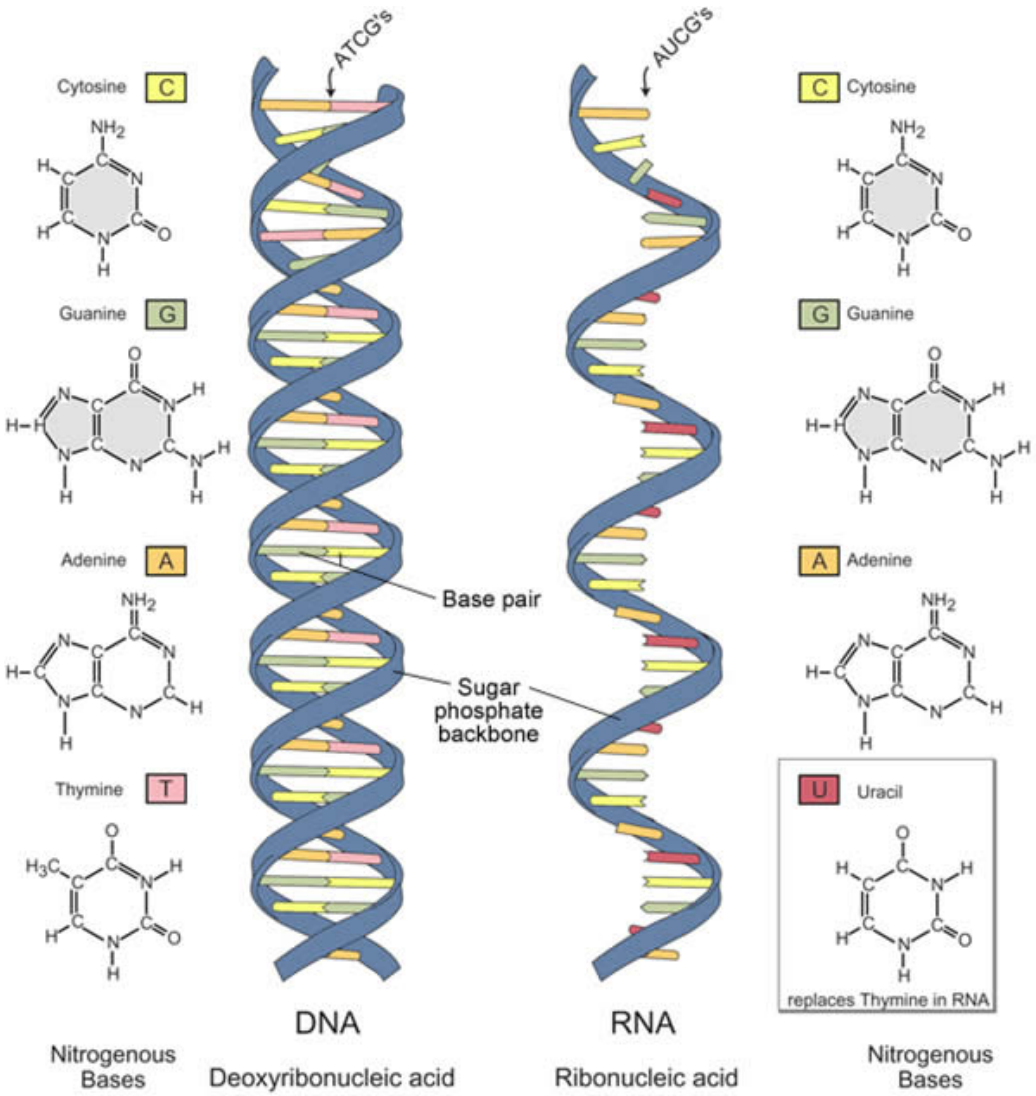
Macromoléculas

Son estructuras biológicas conformadas por un determinado número de moléculas orgánicas. Hay cuatro tipos de macromoléculas, Ácidos Nucleicos, Proteínas, Glúcidos y Lípidos. Sólo explicaremos las primeras dos ya que son las únicas relevantes para la aplicación:

- *Ácidos Nucleicos*: Están conformados por secuencias de nucleótidos específicas, que son utilizadas por todos los organismos para almacenar su información genética. La misma, está codificada mediante sucesivos codones, los cuales están conformados por tripletes de nucleótidos que codifican un aminoácido. Dos de los ácidos nucleicos más importantes son:

  1. **ADN (Ácido Desoxirribonucleico)**: Contiene la información genética para el desarrollo y el funcionamiento de los organismos vivos y de algunos virus, la cual se hereda de generación en generación. Está formado por una doble cadena de nucleótidos, en la que las dos hebras están unidas a través de las bases complementarias, según el modelo propuesto por Watson y Crick en 1953.
  2. **ARN (Ácido Ribonucleico)**: Consiste en una hebra de cadena simple, la cual usualmente es trascripta a partir de una porción de ADN y se utiliza posteriormente en la célula para la síntesis de proteínas. Algunos virus poseen ARN como único material genético cuya monohebra puede plegarse, dando lugar a lo que se conoce como estructura secundaria, la cual se analizará más adelante.

- *Proteínas:* Están formadas por secuencias de aminoácidos específicas, que adoptan una conformación tridimensional determinada debido a las interacciones electrostáticas existentes entre los residuos de sus aminoácidos constituyentes. Este arreglo tridimensional la habilita para realizar una función en particular.



Células

Son unidades básicas que conforman el organismo y dentro de éstas ocurren todas las funciones vitales. Además contienen la información genética en su ADN, la cual se transmite a las células hijas. Existen dos tipos:

1. *Procariotas*: Carecen de membrana nuclear.

2. *Eucariotas*: Poseen un núcleo bien definido mediante una membrana nuclear que contiene al ADN.

2.2.2. La Estructura Secundaria Del ARN

El plegamiento de una secuencia de ARN entre sus bases complementarias determina lo que se denomina *estructura secundaria de ARN*. Conocer la estructura secundaria es fundamental para comprender el funcionamiento de los distintos tipos de ARN y de la célula en general. Existen diferentes tipos de ARN, distinguiéndose entre ellos:

- *Messenger ARN* (mRNA).
- *Ribosomal ARN* (rRNA).
- *Transfer ARN* (tRNA).

La existencia del término *estructura secundaria* nos hace suponer también la existencia de una *estructura primaria*. Como se mencionó, dicha estructura queda representada mediante una secuencia de nucleótidos. También existe la estructura terciaria de ARN, pero se la deja a un lado por no ser relevante en este trabajo.

Definición 1. La estructura primaria del ARN, es una secuencia de nucleótidos de longitud n, $A = a_1a_2a_3 \dots a_n$ con $a_i \in \{A, U, G, C\}$

Definición 2. Dada una estructura primaria o secuencia de RNA de longitud n, la estructura secundaria es un conjunto S de pares (i, j) con $1 \leq i < j \leq n$ tal que para todo, $(i, j), (i', j') \in S$ se satisfacen las tres siguientes condiciones:

- $j - i > 3$
- $i = i' \Leftrightarrow j = j'$
- $i < i' \Rightarrow i < i' < j' < j \vee i < j < i' < j'$

2.2.3. Energía Libre y Estabilidad

El concepto de "Energía Libre" hace referencia a la energía total contenida en un sistema, como por ejemplo una molécula de ARN con una estructura secundaria, la cual le permite al mismo realizar trabajo. Una molécula de ARN puede estar dotada de una cierta cantidad de energía libre mediante interacciones eléctricas no neutralizadas en su estructura primaria, por lo tanto, esta energía se utilizará para plegar la misma hacia una estructura secundaria más estable. De este modo, un ARN con una estructura secundaria estable, implica una molécula en la cual sus interacciones eléctricas entre nucleótidos se hallan completamente neutralizadas. Se podría inferir que si una secuencia viral de ARN mutada tiene energía libre mínima, la misma no variará su estructura secundaria por sí sola.

2.2.4. Virus HIV

Los virus son entidades infecciosas microscópicas que pueden multiplicarse dentro de las células de un organismo dado. El virus HIV (*Human immunodeficiency virus*) infecta a células vitales del sistema inmunológico humano, tales como los *Linfocitos T* (específicamente los que contienen receptores $CD4^+$), los *macrófagos* y las *células dendríticas*. Las $CD4^+$ son un sub-grupo de los linfocitos, un tipo de glóbulos blancos, los cuales desempeñan un rol importante en el establecimiento y maximización del sistema de defensa de un organismo.

La infección con HIV lleva a un nivel bajo de células T $CD4^+$. En el instante en que éstas llegan a un nivel crítico, se pierde la inmunidad celular y el organismo, progresivamente, se vuelve cada vez más susceptible a otras infecciones oportunistas.

2.2.5. SIDA

El SIDA, del acrónimo **S**índrome de **I**nmunodeficiencia **A**dquirida, es una enfermedad que afecta a los humanos infectados por el HIV tipo 1. Se dice que una persona padece de SIDA cuando su organismo, debido a la inmunodeficiencia provocada por el HIV, no es capaz de ofrecer una respuesta inmune adecuada contra las infecciones.

Cabe destacar la diferencia entre estar infectado por el HIV y padecer de SIDA. Una persona infectada por el HIV es *seropositiva*[4] la cual luego desarrolla un cuadro de SIDA cuando su nivel de linfocitos T $CD4^+$ desciende por debajo de 200 células por mililitro de sangre.

2.2.6. Antirretrovirales

Los fármacos utilizados en tratamientos actuales para combatir las infecciones con HIV son denominados antirretrovirales. Existe una variedad de estos provenientes de distintos laboratorios, cada uno con sus respectivas características, los cuales se clasifican por su rango de acción en dos clases: los *Protease Inhibitors* (PI) y *Reverse Transcriptase Inhibitors*. Dentro de la segunda clase hay dos sub-tipos que son los *Nucleotide Reverse Transcriptase Inhibitors* (NRTI) y los *Non-Nucleotide Reverse Transcriptase Inhibitors* (NNRTI). Existen otras dos clases más recientes que son los *Fusion or Entry*

[4]el término seropositivo se aplica a una condición inmunitaria, caracterizada por la presencia de un anticuerpo específico en sangre, creado frente a un antígeno que puede provenir de un agente infeccioso como de uno no infeccioso.

Inhibitors y los *Integrase Inhibitors*. El objetivo de aplicar antirretrovirales a los pacientes es inhibir la replicación del virus por un tiempo indeterminado.

A continuación se muestra una tabla conteniendo los antirretrovirales aprobados por la FDA (Food and Drugs Administration `http://www.fda.gov/MedicalDevices/default.htm`).

Múltiples Clases Combinadas (Multi-class combinations):

Combinación	Comercial	Aprobación
EFV + TDF + FTC	Atripia	12-Jul-06
d4T + 3TC + NVP	-	Tentative only*
AZT + 3TC+ NVP	-	Tentative only*

Inhibidores de Transcriptasa Reversa Nucleósidos (NRTIs):

Abreviación	Genérico	Comercial	Aprobación
3TC	lamivudine	Epivir	17-Nov-95
ABC	abacavir	Ziagen	17-Dec-98
AZT o ZDV	zidovudine	Retrovir	19-Mar-87
d4T	stavudine	Zerit	24-Jun-94
ddI	didanosine	Videx EC	31-Oct-00
FTC	emtricitabine	Emtriva	02-Jul-03
TDF	tenofovir	Viread	26-Oct-01

Inhibidores de Transcriptasa Reversa No Nucleósidos (NNRTIs):

Abreviación	Genérico	Comercial	Aprobación
DLV	delavirdine	-	04-Apr-97
EFV	efavirenz	Sustiva	17-Sep-98
ETR	etravirine	Intelence	18-Jan-08
NVP	nevirapine	Viramune	21-Jun-96

NRTIs Combinados (Combined NRTIs):

Combinación	Comercial	Aprobación
ABC + 3TC	Epzicom (US)	02-Aug-04
ABC + AZT + 3TC	Trizivir	14-Nov-00
AZT + 3TC	Combivir	27-Sep-97
TDF + FTC	Truvada	02-Aug-04
d4T + 3TC	-	Tentative only*

Inhibidores de Proteasa (Protease Inhibitors) (PIs):

Abreviación	Genérico	Comercial	Aprobación
APV	amprenavir	Agenerase	15-Apr-99
FOS-APV	fosamprenavir	Lexiva (US)	20-Oct-03
ATV	atazanavir	Reyataz	20-Jun-03
DRV	darunavir	Prezista	23-Jun-06
IDV	indinavir	Crixivan	13-Mar-96
LPV/RTV	lopinavir + ritonavir	Kaletra	15-Sep-00
NFV	nelfinavir	Viracept	14-Mar-97
RTV	ritonavir	Norvir	01-Mar-96
SQV	saquinavir	Invirase	06-Dec-95
TPV	tipranavir	Aptivus	22-Jun-05

Inhibidores de Fusión o Entrada (Fusion or Entry Inhibitors):

Abreviación	Genérico	Comercial	Aprobación
T-20	enfuvirtide	Fuzeon	13-Mar-03
MVC	maraviroc	Celsentri	18-Sep-07

Inhibidores de Integrasa (Integrase Inhibitors):

Abreviaciaón	Genérico	Comercial	Aprobación
RAL	raltegravir	Isentress	12-Oct-07

2.2.7. Fracaso, o Fallo, Terapéutico Para El HIV

El fracaso terapéutico, o de tratamiento, ocurre cuando los medicamentos para HIV no pueden controlar la infección de un modo eficiente. Existen tres tipos de fracaso terapéutico: **fallo virológico**, **fallo inmunológico** y **fallo clínico**.

El fracaso virológico ocurre cuando los antirretrovirales no pueden disminuir la carga viral en sangre. De este modo, aunque el paciente esté tomando los medicamentos proscritos, la cantidad de virus en sangre no disminuye o bien se eleva repetidamente.

El fracaso inmunológico ocurre cuando el sistema inmunitario no responde a los medicamentos antirretrovirales. De este modo, aunque tome los medicamentos, el recuento de linfocitos CD4+ no disminuye ni se incrementa.

El fracaso clínico se presenta cuando, a pesar que el paciente toma los medicamentos antirretrovirales, persisten los síntomas de infección por VIH.

Los tres tipos de fracaso terapéutico pueden ocurrir solos o simultáneamente. Por lo general, primero ocurre el fracaso virológico, seguido por el fracaso inmunológico y luego se presenta el fracaso clínico. Pueden ocurrir en un lapso de meses o años entre sí mismos.

¿Cuáles son los factores de riesgo para el fracaso terapéutico?

Los factores que pueden incrementar el riesgo del fracaso terapéutico incluyen:

- Fracaso terapéutico previo
- **Resistencia farmacológica** (o resistencia al medicamento)
- Falta de adherencia al tratamiento
- El organismo no absorbe debidamente los medicamentos antirretrovirales
- Cualquier otra enfermedad o afección
- Mala salud antes de comenzar el tratamiento
- Efectos secundarios de los medicamentos o interacciones con otros medicamentos
- Abuso de sustancias tóxicas que ocasionan falta de adherencia al tratamiento.

2.2.8. Otros Términos

Adherencia al tratamiento: consiste en seguir (acatar) cuidadosamente un régimen de tratamiento. Eso incluye tomar la dosis correcta de un medicamento en el momento adecuado, exactamente como se lo recetaron.

Carga viral: cantidad de HIV en una muestra de sangre. La carga viral mide qué tan bien están controlando la infección los antirretrovirales.

Pruebas de resistencia farmacológica: pruebas de laboratorio para determinar si el VIH de una persona es resistente a cualquier medicamento antirretroviral.

Recuento de linfocitos CD4+: Un recuento CD4+ es la cantidad de linfocitos CD4+ en una muestra de sangre. El recuento CD4+ mide la salud del sistema inmunitario.

Resistencia farmacológica: El HIV puede mutar (EL ARN cambia su secuencia de nucleótidos para hacerse resistente a un antirretroviral).

El VIH alterado se puede multiplicar incluso en presencia de medicamentos antirretrovirales que normalmente matarían el virus. Uno o más medicamentos en un régimen de tratamiento pueden volverse ineficaces como resultado de la resistencia farmacológica.

Resistencia inmunológica: Las nuevas pruebas no muestran aumento del recuento de linfocitos CD4+ a pesar del tratamiento. Un descenso en el recuento CD4+ mientras el paciente toma antirretrovirales puede indicar también fracaso inmunológico.

Capítulo 3

Metodología de Trabajo

3.1. Prácticas De Software

- **Diseño:** Durante el desarrollo de **CombEng**, aproximadamente el 50 por ciento del tiempo fue dedicado al diseño. En este porcentaje esta incluido el diseño original y algunos cambios que debieron ser hechos durante la implementación. Para obtener como resultado un diseño que respeta, entre otras cosas, dos principios básicos: Simplicidad y ocultamiento de la información, se utilizaron Patrones de Diseño[Gamma et al., 2005] y UML.

- **Construcción de código:** Aproximadamente el 30 por ciento del tiempo fue dedicado a la construcción del código. Cada vez que se implemento un nuevo componente se chequeó la integración del mismo con todo el proyecto. Cuando superaba la prueba, el código era subido al **trunk** del repositorio.

- **Revisiones:** La mayoría de las operaciones *commits* realizadas (incluyendo código, diagramas, documentos de responsabilidades, etc.) fueron revisadas por al menos dos personas. No solo se resaltaron errores, sino también cuestiones en cuando a calidad y eficiencia. Cada vez que se encontró un error o sugerencia, una nueva revisión fue creada conteniendo la solución.

- **Seguimiento de issues:** Los Bugs y defectos del proyecto fueron reportados como Issues. Luego, para cada issue, se creo una nueva revisión conteniendo la solución.

3.2. Gestión de la Configuración

Para desarrollar **CombEng** y algunas aplicaciones de ejemplo (*Clothes-Changer*, *RNAFoldingFE*) fue necesario utilizar un manejador de versiones. Para ello se manipuló un repositorio SVN alojado en GoogleCode con el fin de poder seguir la pista a todos los archivos que componen el proyecto. Para más información acerca de qué es y cómo utilizar Subversion puede consultarse los libros de O'Reilly [Sussman et al., 2008].

3.3. GNU/Linux y Software Libre

Tanto el sistema operativo como todas las herramientas que se usaron para el desarrollo del proyecto son libres. Hasta el momento, la licencia libre mas usada es GPL (General Public Licence), una copia de la misma pude ser encontrada en:

`http://www.gnu.org/licenses/gpl-3.0.txt`

3.4. Herramientas

Durante la realización del proyecto fueron utilizadas distintas herramientas, todas ellas bajo licencia GPL. A continuación se muestra una lista de las más usadas.

3.4.1. GNU Toolchain

Este proyecto fue realizado bajo el sistema operativo GNU/Linux, éste cuenta con una serie de herramientas de gran utilidad e importancia. Las más utilizadas durante el desarrollo fueron:

- **GCC (GNU Compiler Collection)**: Es un conjunto de compiladores creados por el proyecto GNU. GCC es software libre y se distribuye bajo licencia GPL. Estos compiladores se consideran estándar para sistemas operativos derivados de GNU.[1]
- **GDB (GNU Debugger)**: Es un depurador portable que se puede utilizar en varias plataformas Unix y funciona para varios lenguajes de programación como C y C++ entre otros. GDB fue escrito por Richard Stallman en 1988, es software libre y se distribuye bajo licencia GPL. [2]

[1] `http://gcc.gnu.org/`

[2] `www.gnu.org/software/gdb/`

- **CMake**: Es una familia de herramientas diseñadas para construir, probar y empaquetar software. CMake se utiliza para controlar el proceso de compilación del software usando archivos de configuración sencillos e independientes de la plataforma.[3]

3.4.2. Latex

Todo este documento fue escrito en LaTeX. Leslie Lamport en 1984, con la intención de facilitar el uso de TeX (lenguaje de composición tipogrfica, creado por Donald Knuth), creó un sistema de composición de textos. El mismo está orientado especialmente a la creación de libros, documentos científicos y técnicos que contengan fórmulas matemáticas. A dicho sistema lo llamó LaTeX y está formado por un gran conjunto de macros de TeX.

3.4.3. Edición

- **Gedit**: Editor de texto plano.[4]
- **Kile**: Editor para LaTeX.[5]
- **vim+latexSuite**: Vim es un editor de texto altamente configurable construido para permitir la edición de texto eficientemente. Se trata de una versión mejorada del editor *Vi* distribuido en la mayoría de los sistemas UNIX. Además, latexSuite es un plugin para Vim, que integra una suite de LaTeX.

3.4.4. Gráficos

- **Gimp**: Editor de imágenes.[6]
- **Bouml**: Editor de diagramas UML.[7]
- **UMLet**: Editor de diagramas UML.[8]
- **Dia**: Editor de diagramas de propósito general.[9]

[3] `www.cmake.org`
[4] `http://projects.gnome.org/gedit/`
[5] `http://kile.sourceforge.net/`
[6] `www.gimp.org`
[7] `http://bouml.free.fr/`
[8] `www.umlet.com`
[9] `http://live.gnome.org/Dia`

3.4.5. Documentación

- **Doxygen**: Generador de documentación para múltiples lenguajes.[10]

3.4.6. Análisis Estático de Código

- gcc con flags
- cppcheck

3.4.7. Análisis Estadístico

- **R**: Lenguaje y entorno de programación para realizar análisis estadísticos y gráficos.[11].

[10] `www.doxygen.org`
[11] `www.r-project.org`

Parte II

El Motor Combinatorio

Capítulo 4

Combinatory Engine

4.1. Descripción del Problema

El desarrollo del motor combinatorio, a partir de este momento será denominado **CombEng**, se vio originado por la necesidad de **FuDePAN** (Fundación para el Desarrollo de la Programación en Ácidos Nucleicos) de contar con uno. En dicha fundación se presentan de manera recurrente, problemas de carácter bioinformático. Una familia de ellos se ve caracterizada por compartir una o más de las siguientes características:

- Requerir un motor combinatorio para la generación de árboles de combinaciones.
- Utilizar mecanismos de poda sobre dichos árboles.
- Requerir un sistema de puntuación por cada una de las combinaciones (*ranking* o *scoring*).

El principal objetivo de este proyecto fue el de acoplar a **FuD** una capa que permita, a usuarios sin altos conocimientos en programación, implementar soluciones a los problemas antes mencionados.

La elección de acoplar **CombEng** como una nueva capa del framework **FuD** en lugar de crear un proyecto aparte, se debió a que la mayoría de los problemas antes mencionados requieren un elevado nivel de cómputo, muchas veces "*imposible*" de realizar en una única computadora.

4.2. Nuevas Capas De FuD

A continuación, puede observarse un nuevo diagrama, al estilo OSI de redes, reemplazando al diagrama inicial del framework **FuD** mostrando las diferentes capas:

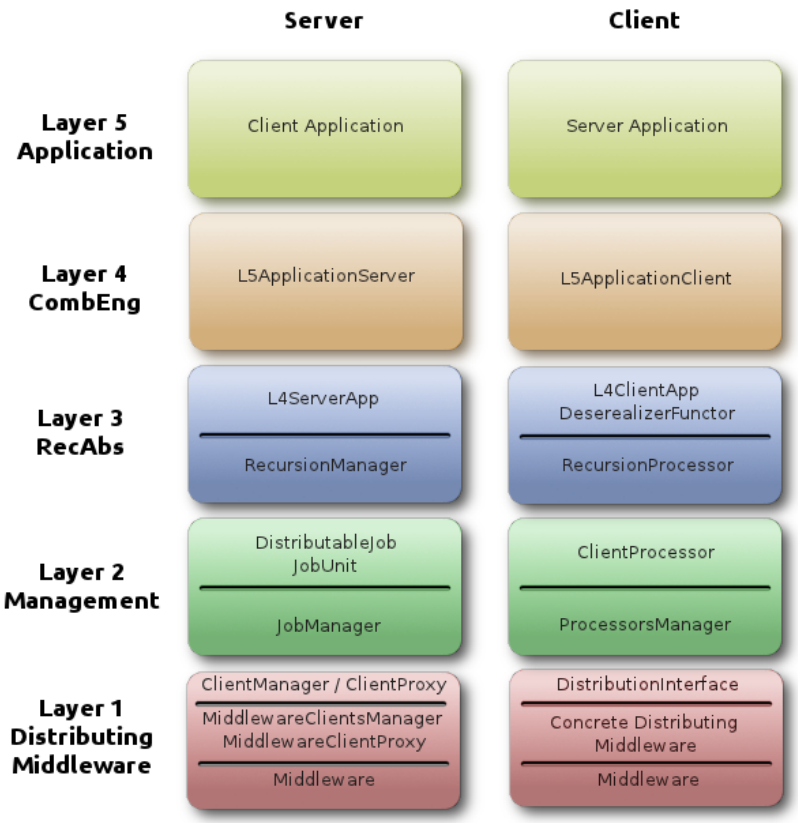


Figura 4.1: Vista abstracta de las nuevas capas en una instancia de uso de **FuD**

Claramente se puede observar la existencia de otra nueva capa, denominada **RecAbs**[1]. La misma constituye otro proyecto de la fundación y es con el que **CombEng** interactúa, por lo que a continuación se explica en qué consiste el mismo.

[1] `http://code.google.com/p/fud/source/browse/branches/FuD-duplex/layers/layer3-recabs/`

4.3. RecAbs

RecAbs, al igual que el presente trabajo, surge como una propuesta de **FuDePAN**. En tal fundación se necesitaba una biblioteca que facilitara la solución a un gran número de proyectos bioinformáticos, los cuales usan a la recursión como mecanismo fundamental y poseen muchos factores de implementación en común.

La meta de este proyecto es identificar una abstracción genérica con una estructura común a todas las soluciones recursivas a estos problemas de interés y, al mismo tiempo, proveer una solución algorítmicamente eficiente para dichos proyectos. Esta abstracción abarca la familia de problemas que cumplen las siguientes condiciones:

- la solución se adapte a un modelo recursivo. No obstante, se ajusta principalmente a aquellos problemas con definición inherentemente recursiva; y
- los nodos del árbol de recursión de la solución sean (horizontalmente) independientes.

Cabe destacar que, cualquiera sea el problema que se desee resolver por medio de **RecAbs**, debe adoptar la forma recursiva, donde el paso inductivo conduce a problemas cada vez mas pequeños que, necesariamente, terminarán en, por lo menos, un caso base. También es significativo el segundo punto, el cual expresa que sólo se pueden solucionar problemas con la recursividad más simple, es decir, una vez llegado a las hojas esa ejecución se da por terminada y se informa un resultado, es imposible volver a pasos anteriores buscando diferentes caminos, como si lo permiten algoritmos de backtracking, combinatorios, etc.

4.3.1. Arquitectura

Dado que **RecAbs** también forma parte del framework, presenta el modelo cliente-servidor. El Servidor tiene la responsabilidad de iniciar el proceso recursivo, administrar los pasos de recursión que realizan los clientes y llevar el control de los resultados, mientras que el cliente es el encargado de la resolución de un *functor recursivo* (ver 4.3.1). Durante la tarea de resolución, el cliente puede pedir ayuda y solicitar el envío tanto de functores intermedios como de resultados al servidor.

RecAbs es un proyecto bastante extenso, por lo que sólo se hará mención a aquellos componentes que interactúan de manera directa con **CombEng**, es decir, aquellas interfaces de L3 que poseen servicios o funcionalidades a ser implementadas por capas superiores.

Lado Servidor

RecursionManager: Es el "handler" del lado servidor. Cada paquete que sale de un cliente y llega a este *manager*, puede ser alguno de los siguientes tipos de paquetes:

- un **resultado**, parcial y relativo a la unidad de trabajo que se procesó, el cual es tratado de la manera que el usuario desee. Indica, en el cliente, que se ha llegado a una hoja en el árbol de recursión.
- un **mensaje intermedio** es cualquier dato transmitido de cliente a servidor sin llegar a una hoja o estado final en el árbol, y por lo tanto no es un resultado. También es enviado (en cliente) y tratado (en servidor) como el usuario disponga.
- una **unidad de trabajo** a distribuir, la cuál será enviada a un cliente ocioso. Un *trabajo* arribado es una partición del trabajo del cliente que lo envió.

Otro rol fundamental que juega el *manager* es la de iniciar el proyecto implementado sobre **RecAbs**. Para ello, el mismo cuenta con la ayuda de *L4ServerApp* que nos brinda el functor inicial, el cual es apilado para distribuirlo al primer cliente ocioso que se encuentre, donde así la iniciación del proceso recursivo.

L4ServerApp: La implementación de esta interfaz consta de los siguientes puntos:

- brindar el *functor* inicial,
- definir qué hará con los resultados a medida que lleguen, y
- definir el tratamiento de los mensajes intermedios (si es que los hubiera).

De estos requisitos, el primero es obligatorio mientras que los otros dos restantes son opcionales en la medida que la aplicación arroje resultados y mensajes intermedios.

Lado Cliente

RecursiveProcessor: Es el encargado de realizar la ejecución total (o parcial) del functor que fue asignado a un cliente.

DeserializerFunctor: Provee los servicios de transformación de un paquete serializado a un functor recursivo.

DistributionPolicy: Es una interfaz que ofrece varios "sabores" de distribución ya implementados, así como la posibilidad de extender este conjunto de políticas si el desarrollador quisiese. Cada política establece cuándo un cliente debe distribuir o cuándo debe parar y, en caso positivo, cuánto debe distribuir. Cuando se menciona *distribuir*, se hace referencia al envío de unidades de trabajo a clientes ociosos que el server dispone.

Común a Ambos Lados

RecursiveFunctor: es una abstracción a la *función recursiva* de una aplicación **RecAbs**. Una aplicación deberá implementar la interfaz definida por `RecursiveFunctor`, este nuevo *functor* (por ejemplo, `ConcreteFunctor`) representará la función que el usuario desee distribuir. Estos nuevos functores encapsulan la información específica de cada aplicación, la cual es necesaria y suficiente para ser procesada por los clientes.

SerializableRecursiveFunctor: Es un RecursiveFunctor para proyectos distribuidos, por lo que agrega el servicio de serialización.

4.4. ¿Qué es CombEng?

Como se mencionó anteriormente, **CombEng** constituye una capa de un framework para la implementación de aplicaciones distribuidas. Define una interfaz clara y sencilla para que un usuario pueda implementar sus aplicaciones independizándose de todas aquellas tareas involucradas en computación distribuida, como la división del trabajo, la administración de los clientes que realizan el procesamiento, la recolección de los resultados, etc.

CombEng no depende del problema a ser implementado, sino que define una estructura común para implementar aquellos problemas que compartan las características mencionadas en 4.1. La solución al problema constituirá una nueva capa, por encima de L4, la cual recibe el nombre de *Application Layer*

4.5. ¿Cómo Funciona Un Proyecto CombEng?

CombEng esta dividido en dos aplicaciones: *Cliente* y *Servidor*. Luego, durante la ejecución de cualquier proyecto **CombEng**, deberá existir exactamente un único servidor y cualquier número de clientes conectados al mismo. El servidor y sus clientes se relacionan de modo Master-Worker, en el sentido

de que el servidor (Master) es el encargado de llevar a cabo el progreso general del sistema y los clientes (Workers) son los que hacen el procesamiento de datos.

Para comprender cómo se lleva a cabo el procedimiento se introduce a continuación, la terminología del proyecto.

4.5.1. El Nodo

`L4Node` es un concepto abstracto de *estado* de una aplicación **CombEng**. Una aplicación tendrá que implementar la interfaz definida por `L4Node` (por ejemplo, `L5Node`). Estos nuevos nodos tienen la responsabilidad de encapsular la información de interés para la aplicación, disponer del conocimiento suficiente para procesar tal información, establecer el progreso general de la ejecución, entre otras.

4.5.2. La Política de Combinación

Una *Política de Combinación* define el modo en que una colección de elementos debe ser combinada. **CombEng** provee un conjunto de estas políticas y las hay en dos sabores, *simples* y *compuestas*. De todas formas, el desarrollador de una aplicación puede crearse tantas como sea necesario, sólo debe respetar las interfaces que L4 le impone. Las políticas compuestas establecen el comportamiento de dos o más políticas simples. Para más detalles, ver 5.2.4.

4.5.3. La Política de Poda

Una *Política de Poda* establece reglas que permiten acotar el espacio de búsqueda en un árbol de ejecución. Este tipo de políticas puede influenciar tanto en la performance como en el tiempo de ejecución de una aplicación. Cabe notar que en la mayoría de estas aplicaciones, los espacios de búsqueda son de gran tamaño.

4.5.4. La Aplicación Servidor

`L5ApplicationServer` es, como su nombre lo indica, la aplicación del lado servidor. Esta aplicación es la encargada de iniciar todo el proceso y, para éllo, necesita del nodo inicial (raíz del árbol de ejecución) junto con la colección de datos sobre la cual operará el motor combinatorio. Además, desempeña la tarea de recolectar y procesar los resultados que los clientes le envían.

En capítulos posteriores se introducen algunas *application layers*, reflejando con más claridad las nociones de nodo/estado, colección inicial de elementos para el motor combinatorio, y demás.

4.5.5. La Aplicación Cliente

`L5ApplicationClient` es la aplicación del lado cliente y es la encargada de procesar la información contenida en el nodo que el servidor le ha asignado.

Si bien el usuario desconoce cómo se lleva a cabo la distribución del trabajo en las capas inferiores del framework, aquí resulta de gran interés por lo que se realiza una breve descripción de ello[2].

Como se mencionó anteriormente, `L5ApplicationServer` es el que inicia todo el proceso. Luego, a grandes rasgos, el comportamiento de un cliente puede ser resumido en los siguientes pasos:

1. Recibe, del servidor, un nodo para procesar.
2. Se obtienen todos los estados posteriores (los nodos hijos de aquel que fue recibido).
3. Enviar información al servidor (resultados) en caso de ser necesario.
4. Continuar con el procesamiento de los nuevos nodos generados. Para ello, se consulta al servidor qué hacer:
   a) Si el servidor dispone de otros clientes conectados, el cliente corriente delegará tantos nodos como sea posible.
   b) Caso contrario, él mismo continuará con el proceso de ejecución.
5. Una vez finalizada la tarea encomendada por el servidor, notifica al mismo de tal situación y queda a la espera de un nuevo trabajo.

[2]Para más información, consulte *RecAbs* (`http://code.google.com/p/fud/source/browse/branches/FuD-duplex/layers/layer3-recabs/`)

4.6. Dependencias Externas

Además de las bibliotecas STL (Standard Template Library) de C++, **CombEng** depende de otra biblioteca muy importante, que se enuncia a continuación.

4.6.1. MiLi

MiLi es una colección de pequeñas bibliotecas C++, compuesta únicamente por *headers*. Sin necesidad de instalación, sin un *makefile*, sin complicaciones. Soluciones simples para problemas sencillos.

Esta biblioteca provee varias funcionalidades pequeñas en archivos cabecera (más conocidos en el ámbito de C/C++, extensión ".h"), y puede ser descargada junto con su documentación desde el siguiente enlace `http://mili.googlecode.com/`.

MiLi ha sido utilizada extensamente a lo largo del desarrollo de **CombEng** donde, particularmente, se destacan las funcionalidades provistas por *binary-streams* y *container-utils*.

- **binary-streams:** esta biblioteca permite empaquetar diferentes tipos de datos dentro de un único objeto usando operadores de stream (<< y >>). En la Tabla 4.1 se muestra un ejemplo de su uso.
- **container-utils:** esta biblioteca provee un conjunto de funciones, optimizadas para cada tipo de contenedor STL:
  - `find(container, element)`.
  - `find(container, element, nothrow)`.
  - `contains(container, element)`.
  - `insert_into(container, element)`.
  - `copy_container(from, to)`.

  Adicionalmente, esta biblioteca provee las siguientes clases:
  - `ContainerAdapter<T>`
  - `ContainerAdapterImpl<T, ContainerType>`

  Estos *container adapters* son una herramienta para lidiar con diferentes contenedores STL de manera homogénea, sin necesidad de conocer el tipo de contenedor que es (vector, list, map, set). Los usuarios pueden invocar al método `insert(const T&)` para insertar un elemento de

tipo T sin la necesidad de saber el tipo del contenedor. En el Cuadro 4.1 se muestra un ejemplo de su uso.

```
#include <iostream>
#include <string>
#include <vector>
#include "include/mili.h"

int main()
{
    std::vector<int> v(5,3); //all 3's
    v[1] = 1;
    v[4] = 7; //so it is [3,1,3,3,7]

    bostream bos;
    bos << 1 << 2 << 3 << std::string("Hello ") << v << 4 << std::string("↩
        World!");

    bistream bis(bos.str());

    int         nums[4];
    std::string str1;
    std::string str2;

    std::vector<int> v2;


    bis >> nums[0] >> nums[1] >> nums[2] >> str1 >> v2 >> nums[3] >> str2;

    for (int i=0; i < 4 ; ++i)
        std::cout << nums[i] << std::endl;

    std::cout << str1 << str2 << std::endl;

    std::cout << '[';
    for (size_t i=0; i < 5; ++i)
        std::cout<< v2[i] << ' ';
    std::cout << ']' << std::endl;
}
```

Cuadro 4.1: Código extraído de Mili::binary_streams.

```
#include <iostream>
#include <vector>
#include <string>
#include <queue>
#include "mili/mili.h"
using namespace std;

template <class T>
static void insert_elements(T& container);

int main()
{
    vector<int> v;
    v.push_back(1);
    map<string, string> m;
    m["hello"] = "good bye";
    m["Bonjour"] = "au revoir";
    m["hola"] = "adios";
    m["buenas"] = "adios";

    try
    {
        cout << contains(v, 2) << endl;          // will print 0 (false)
        cout << contains(m, "nothing") << endl; // will print 0

        cout << "map: " << endl;
        cout << remove_first_from(m, "au revoir") << endl; // will print 1
        cout << remove_all_from(m, "adios") << endl;       // will print 1

        cout << find(v, 1) << endl;          // will print 1 (true)
        cout << find(m, "hello") << endl;    // will print "good bye"
        cout << find(m, "world") << endl;    // will throw ElementNotFound
    }
    catch (ElementNotFound)
    {
        cerr << "Element not found!" << endl;
    }

    /* TEST - queue in container_utils::insert_into */
    queue<int> myqueue;
    insert_elements(myqueue);

    return 0;
}

template <class T>
static void insert_elements(T& container)
{
    insert_into(container, 100);
    insert_into(container, 100);
    insert_into(container, 100);
    insert_into(container, 200);
    insert_into(container, 300);
    insert_into(container, 400);
    insert_into(container, 400);
}
```

Cuadro 4.2: Código extraído de Mili::container_utils.

Capítulo 5

Consideraciones de Diseño

5.1. Diseño de Alto Nivel

Desde el punto de vista de la ingeniería, el diseño de software es una etapa crucial del desarrollo de software, en la cual sus implicaciones y deficiencias afectan el proyecto a lo largo de su ciclo de vida[Pressman, 1982], por lo que la toma de decisiones sobre el diseño es una tarea que debe ser llevada a cabo con especial atención y cuidado.

La técnica de diseño que se empleó fue la denominada Responsibility Driven Design[Wirfs-Brock and McKean, 2003a]. Esta técnica de diseño fue concebida en el año 1990 con la intención de cambiar la visión de los objetos como datos y algoritmos por roles y responsabilidades.

Además, se intentó que el diseño cumpliese con los principios fundamentales del diseño de software, comúnmente conocidos por el acrónimo "**SOLID**"[Martin, 2000] (**S**ingle Responsability, **O**pen-Closed, **L**iskov Substitution, **I** nterface Segregation y **D**ependency Inversion). De aplicar estos principios de manera conjunta, hacen más probable que un programador cree un sistema de fácil mantenimiento y extensible en el tiempo.

Single Responsibility Principle (SRP): No debe existir más de una razón para que una clase cambie. Esto significa que una clase con diferentes responsabilidades debe ser dividida en clases más simples.

Open-Closed Principle (OCP): Este principio establece que las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para su extensión pero cerradas para su modificación[Meyer, 1997].

Liskov Substitution Principle (LSP): Aquellas funciones que usan punteros o referencias a clases base deben ser capaces de utilizar objetos de

clases derivadas, sin saberlo. Barbara Liskov lo describió unos 8 años antes[1] como:

> *Lo que se intenta aquí es algo como la siguiente propiedad de sustitución: Si para cada objeto o_1 de tipo S existe un objeto o_2 de tipo T tal que para todos los programas P definidos en término de T, el comportamiento de P no se ve alterado cuando o_1 es sustituido por o_2, entonces S es un subtipo de T.*

Interface Segregation Principle (ISP): Los clientes no deben ser forzados a depender de interfaces que no usan, lo que puede ser interpretado como el hecho de que las interfaces deben tener usuarios que las usen de manera completa, no parcial. Si este último fuese el caso, entonces debe haber otra interfaz con el subconjunto de métodos que este usuario particular necesita.

Dependency Inversion Principle (DIP): Este principio establece dos puntos:

- Módulos de alto nivel no deben depender de módulos de nivel más bajo. Ambos deben depender de abstracciones.
- Abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

Observando el nuevo diagrama al estilo OSI del framework **FuD**(4.1) se puede observar claramente que el diseño se divide en dos partes, aplicaciones cliente y servidor. A su vez, cada una de estas partes se encuentra organizada en 5 capas separadas, donde cada una de ellas posee una responsabilidad bien definida.

Cabe destacar que el único enlace *real* entre las aplicaciones cliente y servidor estará en el nivel más bajo, es decir, en alguna implementación del middleware de distribución. Las restantes formas de comunicación son abstractas y deben atravesar la estructura de capas.

5.1.1. Application Layer (L5)

La capa de aplicación provee los componentes que contienen todos los aspectos del problema a resolver. Entre estos aspectos se incluyen todas las definiciones de los datos que se usan y la manipulación de los mismos, como así también, los algoritmos relevantes para la solución del problema.

[1] Barbara Liskov, "Data Abstraction and Hierarchy" (Mayo de 1988).

Es necesario aclarar que, bajo ningún punto de vista, esta capa forma parte del framework **FuD**. Como cualquier código de aplicación que hace uso de una biblioteca no forma parte de la misma. Sin embargo, la inclusión de esta capa tiene como principal objetivo servir de ayuda para la comprensión del funcionamiento de la biblioteca.

Las responsabilidades de esta capa son:

Lado Servidor: Proveer a la capa subyacente el nodo/estado inicial de la aplicación junto con la colección de elementos sobre la cual operará el motor combinatorio. Además, debe definir qué hacer una vez que arribe un paquete resultado.

Lado Cliente: Proveer la *deserialización* de un nodo. En 4.6.1 mencionamos el gran uso que se le da a MiLi en este proyecto, particularmente a los *binary streams*. Estos se utilizan para la *serialización* y *deserialización* de los paquetes que viajan desde el servidor hacia los clientes y viceversa. Serialización es el proceso de convertir una estructura de datos o un objeto en un formato que pueda ser almacenado (como por ejemplo un archivo, o un buffer de memoria) o transmitido a través de una conexión de red para más tarde, probablemente en otra computadora, realizar el proceso inverso, "resucitar" el objeto o estructura original[2].

Común a Ambos Lados: Se deben proveer, a la capa L4, las políticas de Combinación y de Poda (son las que determinan el comportamiento del motor combinatorio).

Además, se necesita definir el nodo de la aplicación, detallando, entre otras cosas, qué información contendrá y cómo se llevará a cabo su serialización. Como se mencionó en 4.5.1, el nodo representa un estado durante la ejecución y es lo que el servidor distribuye entre los clientes para su procesamiento.

[2] `http://en.wikipedia.org/wiki/Serialization`

5.1.2. Combinatory Engine (L4)

Esta capa es la que provee la estructura para definir soluciones a problemas que requieren un motor combinatorio.
Entre sus responsabilidades se encuentran:

Lado Servidor: Proveer a la capa subyacente el functor recursivo inicial[3] del problema y definir los pasos a seguir cada vez que se recibe un paquete de resultado.

Lado Cliente: Proveer la deserialización de un functor recursivo.

Común a Ambos Lados: Se debe proveer la serialización de un functor recursivo y la definición de la función *call*. Esta función es la que debe especificar cómo un functor recursivo debe generar sus hijos (estados siguientes), retornándolos en una lista de functores.

Dependiendo en que etapa se encuentre un functor, se ejecuta

- *un caso base:* devolver un resultado, o
- *un caso inductivo:* llenar la lista de functores hijos.

De este modo, la representación de cualquier función recursiva solo pasa por la implementación de la clase abstracta *SerializableRecursiveFunctor* de **RecAbs**, especificando la manera en que se deben generar los hijos.

Notar que entre **CombEng** y **RecAbs** hay ciertas responsabilidades que, si bien operan sobre diferentes tipos de instancias, se repiten. Tales son los casos de la serialización/deserialización de un nodo (o functor) o la de proveer el nodo (o functor) inicial. Lo que sucede es que **CombEng**, la capa 4, no conoce absolutamente nada del dominio del problema por lo que no sabe como realizar la serialización de un nodo, o que objetos iniciales proveer al motor combinatorio. Entonces, esta capa actúa como un puente derivando aquellas tareas que no conoce a la capa superior. De esta manera, ocurren relaciones de herencia en tres niveles. Todo esto se aclarará a continuación, en Diseñno de Bajo Nivel.

[3] el functor recursivo no es más que el nodo inicial que L5 provee a L4, con la diferencia de que el primero se encuentra serializado

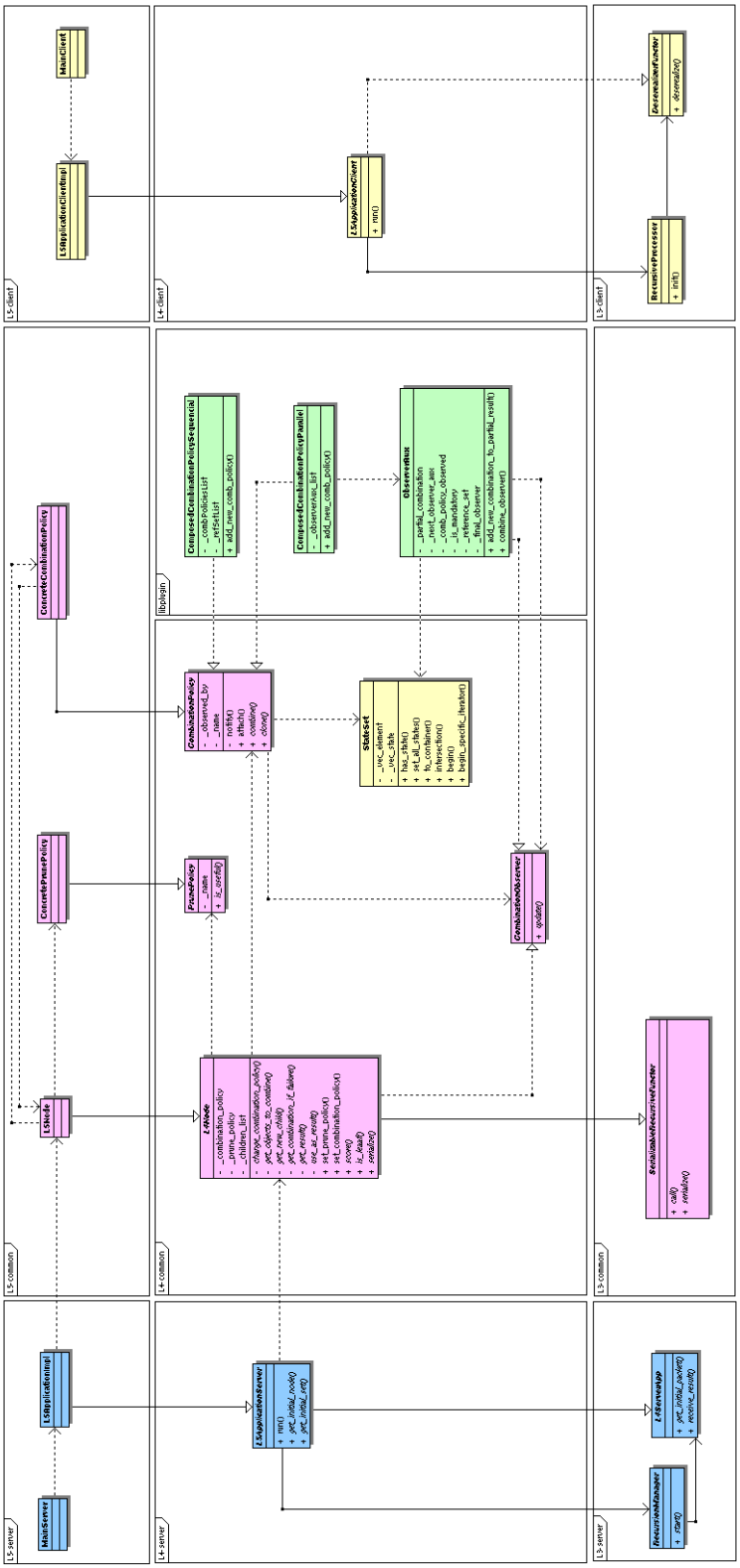


Figura 5.1: Diagrama de Clases de **CombEng** y sus capas inmediatas

5.2. Diseño de Bajo Nivel

Aquí se hace un refinamiento de las decisiones de diseño sobre aquellos componentes abstractos presentes en el diseño de alto nivel. Además, se analiza cómo estas clases están compuestas: los atributos de cada una, los métodos que declaran y sus interacciones con el resto del sistema.

Sin entrar en mucho detalle sobre las decisiones de diseño que se tuvieron en cuenta sobre cada módulo, se examina, a continuación, la capa **CombEng**. Para cada componente del módulo se muestra un diagrama de clases simplificado.

5.2.1. `L4node`

En la sección 4.5.1 solo se había hecho mención a un nodo de **CombEng** como un concepto abstracto de estado de una aplicación, pero como se puede observar en la figura 5.3, `L4Node` hereda de `CombinationObserver` y `SerializableRecursiveFunctor`, satisfaciendo otros dos roles que se explican a continuación:

- *L4Node es un observador:* el nodo observa a la política de combinación. Por cada nueva combinación que la política del nodo corriente genere, éste será notificado y evaluará, mediante su política de poda, si dicha combinación es útil. En caso afirmativo, creará un nuevo nodo del nivel 5 (de ahora en más lo llamaremos `L5Node`) y lo pondrá en su lista de hijos (`_children_list`).

- *L4Node es un functor recursivo:* para ello, el nodo del nivel 4 debe implementar los métodos que `SerializableRecursiveFunctor` define en el nivel 3, que son los métodos `call` y `serialize`.

  `void serialize(recabs::Packet& pkt)`

  Este método, por razones obvias de que el nodo en L4 no posee ningún tipo de información útil para la aplicación, es imposible brindar algún modo de serialización que resulte de utilidad. Debido a ésto, quien lo implementa es `L5Node`.

  `void call(recabs::ChildrenFunctors& children, recabs::Packet& result)`

  `call` está implementado usando los métodos virtuales definidos en esta clase, dado que, al igual que la serialización, en la capa 4 no se tiene información suficiente sobre el dominio del problema. En la figura 5.2 se muestra un diagrama de secuencia exhibiendo el comportamiento de este método.

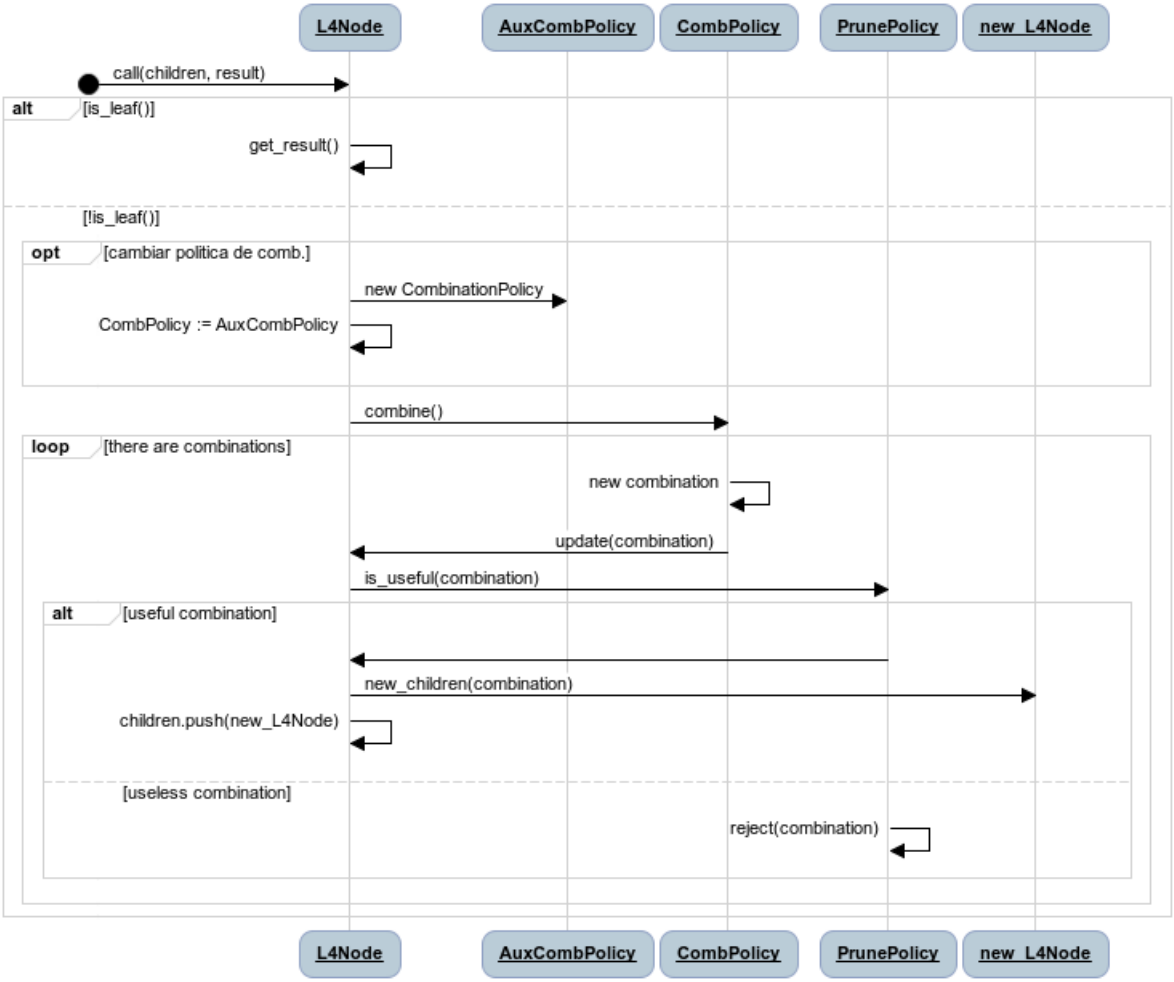


Figura 5.2: Diagrama de Secuencia del Método `call`.

- *L4Node es un estado de aplicación:* además de la serialización, el desarrollador de la aplicación debe implementar los siguientes métodos en su L5Node:
  - `void new_children(const std::list<T>& combination, std::list<L4Node*>&)`
  - `float score()`
  - `CombinationPolicy<T>* get_combination_if_failure(const std::string& failed_comb_policy)`
  - `CombinationPolicy<T>* change_combination_policy()`
  - `bool use_as_result()`
  - `recabs::Packet get_result()`
  - `void get_objects_to_combine(std::list<T>&)`
  - `bool is_leaf()`
  - `void serialize(recabs::Packet& pkt)`

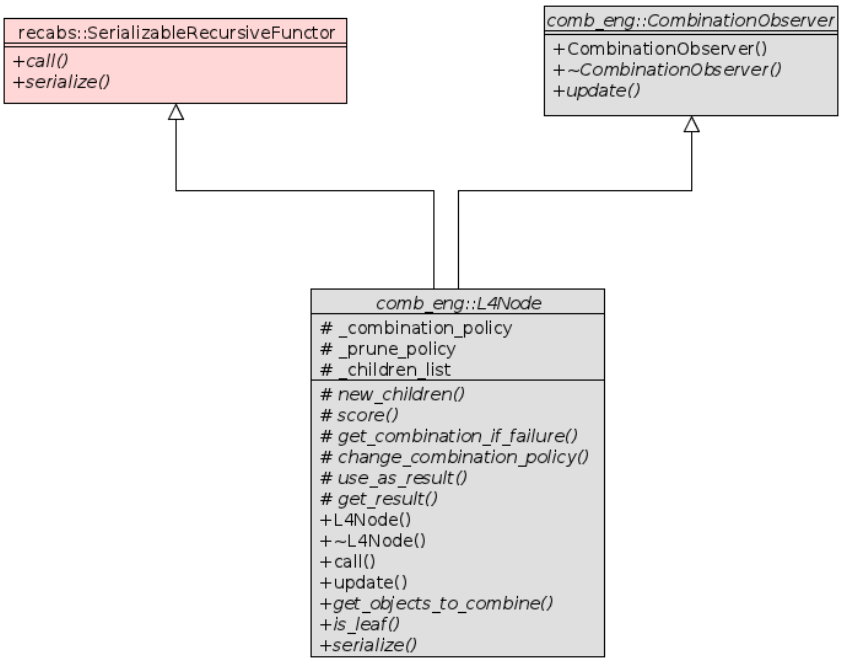


Figura 5.3: Clase `L4Node`.

5.2.2. `PrunePolicy`

La política de poda es la que define si una combinación es de utilidad o no. El desarrollador de la aplicación deberá implementar en su política de poda concreta el único método definido por la interfaz que se observa en la figura 5.4:

```
- bool is_useful(const std::list<T>combination)
```

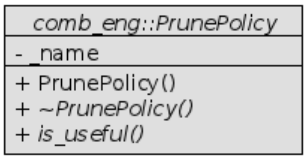


Figura 5.4: `PrunePolicy` class

5.2.3. `CombinationObserver`

Para las políticas de combinación se utilizó el patrón de diseño *Observer* (*veáse* A.3.1). Como se mencionó anteriormente, `L4Node` es un observador de una política, por lo que el mismo deberá implementar el método update definido en CombinationObserver:

```
void update(const std::list<T>& combination)
```

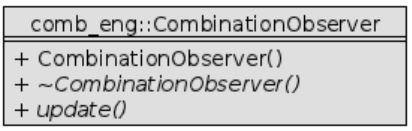


Figura 5.5: Clase `CombinationObserver`.

5.2.4. `CombinationPolicy`

La política de combinación es la que define la forma en que una colección de elementos serán combinados. El desarrollador deberá implementar en su política de combinación concreta, los métodos de la interfaz que se observan en la figura 5.6:

- `void combine(const std::list<T>& objects_to_combine, Status& combination_status)`.
  Como su nombre lo indica, es el que determina el comportamiento de la política, cómo se generan las combinaciones. Posee dos argumentos, el primero es la colección de elementos que debe combinar y el segundo es el estado con el que finaliza.

- `CombinationPolicy<T>* clone (CombinationObserver<T>* obs)`.
  Retorna una nueva instancia idéntica a la política de combinación sobre la cual se invoca.

Políticas De Combinación Provistas Por CombEng

Durante el desarrollo de las aplicaciones que se incluyen en este proyecto se han implementado diferentes políticas de combinación, algunas *simples* y otras *compuestas*. Las simples son algoritmos combinadores propiamente dichos, mientras que las compuestas combinan a dos o más políticas simples. Estas últimas no generan combinaciones por sí mismas, son las simples quienes se encargan de obtenerlas.

Dentro de las simples, están:

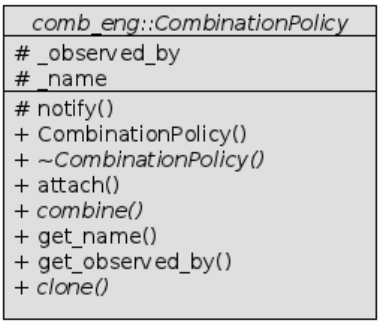


Figura 5.6: `CombinationPolicy` class

- `ListCombinationPolicy`: Esta política retornará uno a uno los elementos a combinar. El argumento `combination_status` informará un fallo si el conjunto de elementos se encuentra vacío.
- `NewtonianCombinationPolicy`: el método `combine` de esta política implementa el clásico algoritmo que retorna todos los subconjuntos de tamaño K, con $MIN \leq K \geq MAX$ [4]. El `combination_status` informa la ocurrencia de un fallo en caso de que la cantidad de elementos a combinar sea sea cero o menor a *MIN*.

Mientras que de las compuestas hay otras dos:

- `ComposedCombinationPolicySequencial`: Esta política establece una especie de nexo entre diferentes algoritmos combinatorios. Básicamente, toma un conjunto de combinadores y los hace combinar en secuencia, uno después de otro. Podría pensarse como una cola (FIFO) de combinadores, el primero en ingresar a la cola es el primero en combinar.

[4] MIN y MAX son establecidos por el usuario

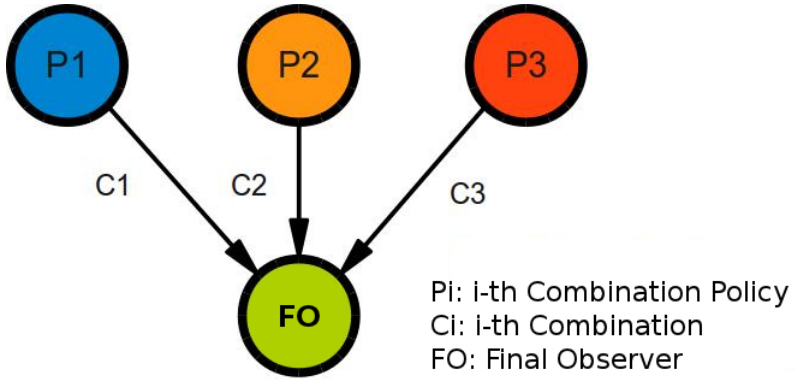


Figura 5.7: Política de Combinación Compuesta Secuencial.

- `ComposedCombinationPolicyParallel`: Esta política toma un grupo de algoritmos combinadores (políticas simples) y los pone a "correr" en paralelo. Es decir, cada combinación que genera la primer política se une con cada combinación generada por la segunda y así sucesivamente, hasta llegar a la última de éstas, la cual hará entrega de todo el paquete al observador final.

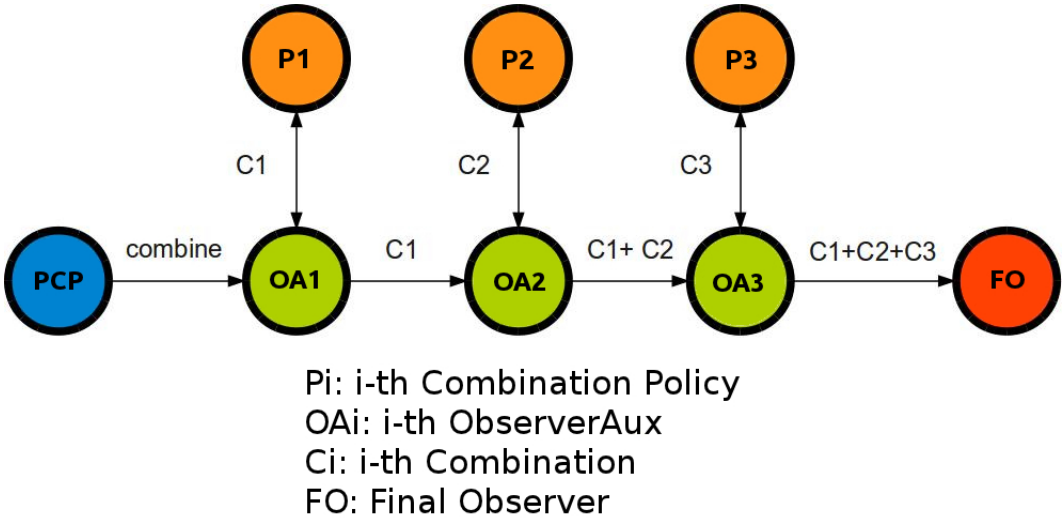


Figura 5.8: Política de Combinación Compuesta Paralela.

De la Figura 5.8 se puede observar la nueva entidad *ObserverAux* la cual realiza, repetitivamente hasta que no reciba más combinaciones, las siguientes tareas:

- Recibir una combinación generada por la política que observa.
- Unir dicha combinación con aquella que proviene del observador auxiliar anterior, siempre y cuando no se trate del primer algoritmo combinatorio (**P1**).
- Entregar la unión al observador auxiliar de la siguiente primitiva simple.

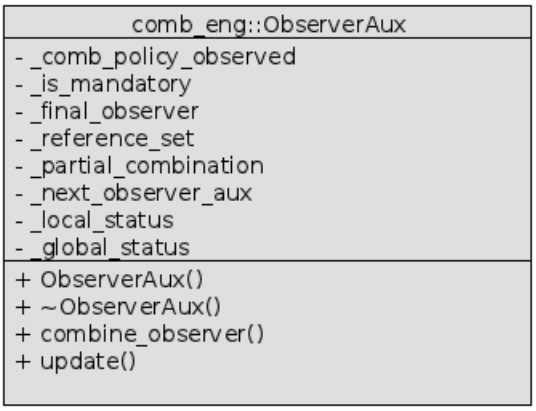


Figura 5.9: Clase `ObserverAux`.

A partir de la figura anterior, se explican cada uno de sus atributos:

`_comb_policy_observed`: tal y como su nombre lo indica, es la política a la que se esta observando.

`_is_mandatory`: a la hora de la creación de una nueva instancia de un ObserverAux, es necesario establecer si la misma será mandatoria o no. Cada una de las políticas simples opera sobre un conjunto de elementos (_reference_set) no necesariamente igual al resto, por lo que si una de las políticas falla (por ejemplo, no dispone de mas combinaciones) y es mandatoria, toda la política compuesta fallará. Por otro lado, si la misma no es mandatoria implica que este observador auxiliar actúe como un "puente", dejando pasar aquellas uniones que provienen de las primitivas anteriores a siguiente observador.

`_partial_combination`: es la combinación que recibe del observador auxiliar anterior a él. En el caso de que sea el primero, este atributo estará vacío.

`_next_observer_aux`: es aquel al que se le debe entregar la combinación parcial.

`_global_status` y `_local_status` son los que permiten llevar el estado de la política compuesta y de cada una de las primitivas simples, respectivamente.

5.2.5. `L5ApplicationServer`

L5ApplicationServer constituye el lado servidor de la aplicación. Quien desarrolle una aplicación sobre **CombEng**, deberá implementar, al menos, los siguientes métodos:

`get_initial_packet`: Retorna un paquete de datos representando al nodo inicial de la aplicación. Dicho nodo es enviado a un cliente, para que sea él quien inicie la ejecución de la aplicación.

`receive_result`: Durante el procesamiento de un nodo, los clientes pueden enviar resultados al servidor. Este método es quien decide qué hacer con los mismos.

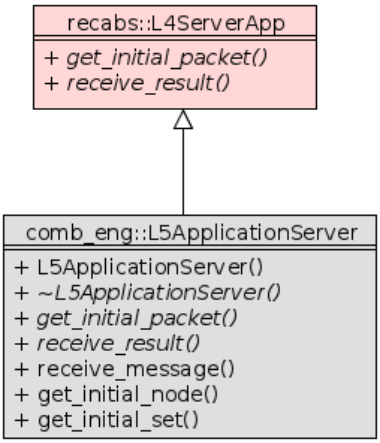


Figura 5.10: Clase `L5ApplicationServer`.

5.2.6. `L5ApplicationClient`

L5ApplicationClient constituye el lado cliente de la aplicación. El desarrollador deberá proveer la implementación de la única funcionalidad requerida por **CombEng**:

`deserialize`: Este método efectúa el proceso de conversión de un paquete que viaja por la red (*binary-stream*) a un nodo utilizable por la aplicación.

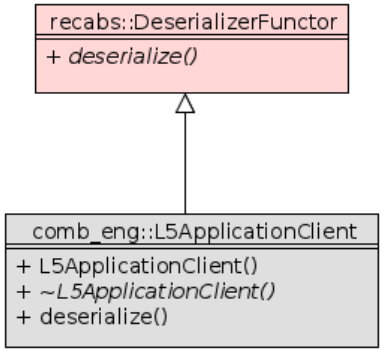


Figura 5.11: Clase `L5ApplicationClient`.

Capítulo 6

Implementación

6.1. Métricas de Código

Para analizar el código estáticamente se usaron herramientas como Cloc y CCCC. En esta sección se describen métricas de código generales obtenidas por ambas herramientas y se analizan los resultados obtenidos.

6.1.1. Métricas de CombEng

CombEng esta constituido por 13 archivos de cabecera con un total de 2072 líneas de texto a la fecha de publicación de este documento. La Tabla 6.1 resume los resultados obtenidos después de correr *cloc* sobre los archivos fuentes de **CombEng**. Los resultados revelan que **CombEng** no

Files		Line Types		
Type	**Count**	**Blank**	**Comment**	**Source**
`C++ header`	13	277	841	947
Total	13	277	841	947

Cuadro 6.1: Resultados de cloc para la capa **CombEng**

es un proyecto de gran tamaño. Sin embargo, estas medidas no reflejan su complejidad.

Dijkstra escribió un ensayo muy interesante[Dijkstra, 1988] donde reflejaba porqué las empresas no deberían considerar las Líneas de Código como una medida exacta de la productividad del software. Medir la "productividad de un programador" en términos del "número de líneas de código que produce por mes", fomenta la escritura de código insípido.

Por otro lado, a mayor número de líneas de código, mayor es la complejidad de un producto de software, pero solo en el sentido de que es más dificultoso de mantener y comprender, no tiene relación directa con la funcionalidad que él provee.

Continuando con los resultados de Cloc, otro dato interesante es la cantidad de líneas de comentarios en el proyecto y su porcentaje con respecto al total de líneas del proyecto:

$$\frac{\#comment\_lines}{\#comment\_lines + \#code\_lines}$$

Este valor ronda el 0.45, por lo que hay una cantidad de líneas de comentarios similar a las líneas de código. Para este porcentaje de líneas de comentarios hay una explicación, y es que incluso archivos realmente pequeños incluyen una cabecera ("heading") definiendo ciertos detalles del archivo, como sus autores, fecha de cración y la licencia por cual se rige.

Todo componente de software (clases, estructuras, funciones, atributos, etc.) conlleva una descripción detallada a ser interpretada por *doxygen* (el cual incluye perfiles de funciones) para la generación de documentación automática. La Tabla 6.2 muestra un ejemplo de la notación utilizada para doxygen exhibiendo además el porqué de las métricas de **CombEng** mencionadas anteriormente.

```
/**
*  It starts the node execution.
*  If the current node has some objects available,
*  it will combine them and the rest of its behaviour
*  is defined by both combination and prune policies.
*  The call method tries to fill the node's children list.
*
*  @param children : List of children obtained from the ↩
    excecution of this node.
*  @param result   : In case that a node wants to send a result ↩
    to the server,
*                    it has to fill this parameter with the ↩
    desired information.
*  @param when     : it establishes when the result packet has ↩
    to be sent to
*                    the server.
*/>
void call(recabs::ChildrenFunctors& children, recabs::Packet& ↩
    result, recabs::WhenToSend& when)
```

Cuadro 6.2: Comentario de una función

6.1.2. Métricas de la Aplicación RNAFoldingFreeEnergy

Los resultados que se obtuvieron para la aplicación, implementada como la capa más alta del framework, no distan demasiado de aquellos que se obtuvieron para **CombEng**. La aplicación **RnaFFE** se compone de 12 archivos en total, con una cantidad de 1759 líneas de texto. La tabla 6.3 resume los resultados obtenidos después de correr *cloc* sobre los archivos fuentes de **RnaFFE**.

Files		Line Types		
Type	**Count**	**Blank**	**Comment**	**Source**
C++ Header	7	158	442	532
C++ Source	5	84	245	298
Total	12	242	687	830

Cuadro 6.3: Resultados de cloc para la capa de aplicación **RnaFFE**

Cobertura de Código para RnaFFE

La tabla que se muestra a continuación exhibe la cobertura de código de los archivos más relevantes para la aplicación. Este test de cobertura fue realizado ejecutando a la aplicación **RnaFFE** con la secuencia de pseudonucleótidos *H2XB* y con los siguientes antivirales: *Didanosine, Abacabir, Emtricitabine, Lamivudine, Stavudine, Zidovudine, Tenofovir, Zidovudine, Nevirapine, Efavirenz, Atazanavir, Darunavir, Fosamprenavir, Indinavir, Lopinavir, Nelfinavir, Saquinavir, Tripanavir.*

	Lines of Code		Percentage
File	**Total**	**Executed**	**%**
main_client.cpp	17	17	100
main_server.cpp	40	24	60
rnaffe_application_client.cpp	23	23	100
rnaffe_application_server.cpp	53	53	100
rnaffe_node.cpp	81	57	70.37
Total	214	174	81.30

Cuadro 6.4: Resultados de cobertura para los archivos principales de **RnaFFE**

Capítulo 7

Aplicación de Prueba

Una vez finalizada la tarea de implementación del motor combinatorio, se continua con la elaboración de una aplicación sencilla cuya única finalidad es la comprobación de que el motor funcione de la manera esperada.

Esta aplicación, de nombre **Clothes Changer**, no requiere distribución de su trabajo debido a la simpleza en sus cálculos, pero aún así es de gran utilidad para poder probar algunas políticas de combinación como así también observar la nueva capa acoplada a **FuD**.

7.1. Descripción Del Problema

A partir de un conjunto de prendas de vestir, se desea conocer todas las posibles maneras en que una persona puede vestirlas, es decir, armar todas las combinaciones del conjunto de ropas. Además, se quiere hacer un "ranking" con las mejores 5 vestimentas.

Cada prenda pertenece a una de las siguientes categorías (notar que una vestimenta es factible siempre y cuando esté consituida por una prenda de cada una de estas categorías).

- Pantalones.
- Remeras.
- Medias.
- Calzados.

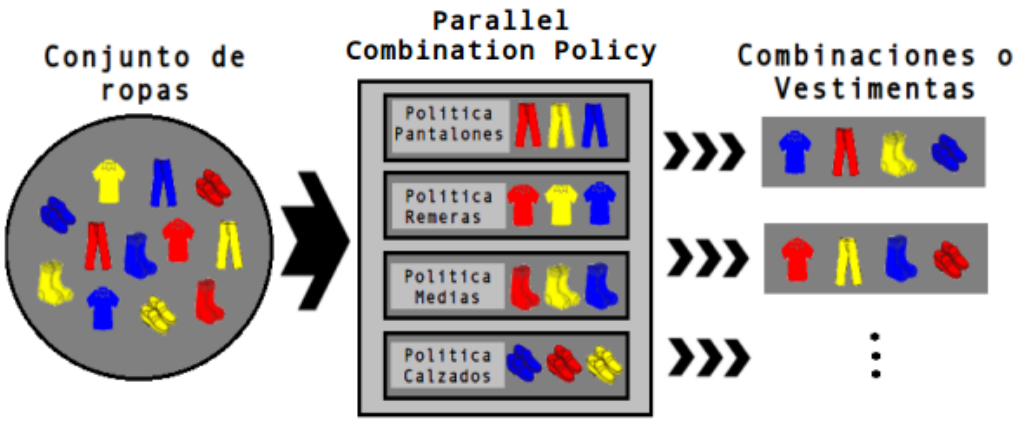
7.2. Solución

Debido a que la aplicación fue pensada como un caso de estudio simple para el motor combinatorio, la misma ha sido implementada respetando la interface que **CombEng** propone. La solución planteada puede ser descompuesta en dos partes:

- Generar todas las posibles vestimentas.
- Calcular un puntaje o *score* para cada vestimenta, representando qué tan buena es la combinación de prendas.

7.2.1. Generación De Vestimentas

Para generar todas las posibles vestimentas se utiliza una política de combinación paralela (ver 5.2.4), donde cada política simple que la compone es del tipo *ListCombinationPolicy*, cada una actuando sobre una única categoría de ropa.



7.2.2. Cálculo Del Score Para Una Vestimenta

Las tablas que se muestran a continuación determinan qué tan bien "combinan" dos prendas (un pantalón con una remera, o un pantalón con un par de medias, etc.). El valor o puntaje de una vestimenta se obtiene realizando la suma entre las diferentes tablas, a partir de las prendas de vestir que la componen (las tablas fueron inicializadas con valores aleatorios).

Tabla Remera-Pantalón

Ropa y Color	Rem. Blanca	Rem. Amarilla	Rem. Roja	Rem. Azul
Pant. Blanco	5	2	4	7
Pant. Amarillo	4	5	6	4
Pant. Rojo	4	4	2	7
Pant. Azul	3	5	4	9

Tabla Pantalón-Medias

Ropa y Color	Pant. Blanco	Pant. Amarillo	Pant. Rojo	Pant. Azul
Med. Blancas	3	6	9	7
Med. Amarillas	5	1	2	9
Med. Rojas	4	5	3	1
Med. Azules	2	4	8	4

Tabla Medias-Calzados

Ropa y Color	Med. Blancas	Med. Amarillas	Med. Rojas	Med. Azules
Cal. Blanco	6	2	7	1
Cal. Amarillo	3	4	3	2
Cal. Rojo	8	7	6	8
Cal. Azul	1	9	2	3

A modo de ejemplo, supongamos que una vestimenta generada por el motor combinatorio consta de las siguientes prendas:

- *Remera Azul*
- *Pantalón Blanco*
- *Medias Rojas*

- *Calzado Blanco*

Para calcular el score de esta vestimenta, indexamos las tablas de acuerdo a las ropas que la componen y sumamos los valores. En este caso:

- *Remera Azul* y *Pantalón Blanco* = **7**
- *Pantalón Blanco* y *Medias Rojas* = **4**
- *Medias Rojas* y *Calzado Rojo* = **6**

Dando un valor para la vestimenta de ***17***.

Parte III

La Aplicación Rna Folding Free Energy

Capítulo 8

Aplicación Rna Folding Free Energy

8.1. Motivación

En la actualidad, los tratamientos para personas infectadas con HIV consisten de una intensiva terapia antirretroviral. Esta terapia puede ser vista como una sucesión de aplicaciones de antirretrovirales en el tiempo, donde en cada paso de la sucesión se le suministra al paciente una combinación de uno o más antirretrovirales.

Cuando a una persona se le aplica un antirretroviral, el virus comienza a mutar hasta que logra hacerse resistente. Ésto implica que el antirretroviral deje de cumplir sus funciones principales y se deba buscar uno nuevo para continuar con el tratamieinto. El orden en que se aplican los antirretrovirales y cómo éstos son combinados, son factores muy importantes que determinan, entre otras cosas, el tiempo que transcurrirá hasta el momento en que el virus sea resistente a todos los antirretrovirales.

Hasta el momento, el impacto del escape a los antirretrovirales sobre la estructura secundaria no ha sido estudiada. Esta herramienta pretende recopilar información sobre como varía la Energía Libre en la estructura secundaria del RNA viral a medida que los antirretrovirales son aplicados sobre la persona infectada.

8.2. Solución

Para alcanzar el objetivo propuesto, la idea principal puede ser resumida en el siguiente pseudo-algoritmo:

1. Tomar una secuencia en representación del virus y todos los antirretrovirales conocidos hasta el momento.

2. De todos los antirretrovirales, seleccionar solo aquellos que realmente pueden ser aplicados a la secuencia (un antirretroviral es aplicable sobre una secuencia cuando esta última no presenta resistencia alguna al mismo). De no haber ninguno aplicable, saltar al paso 7.

3. Generar combinaciones de hasta tres antirretrovirales, utilizando aquellos seleccionados en el paso anterior.

4. Aplicar cada combinación de antirretrovirales a la secuencia y obtener todas las secuencias mutantes.

5. Calcular la FE de cada mutación.

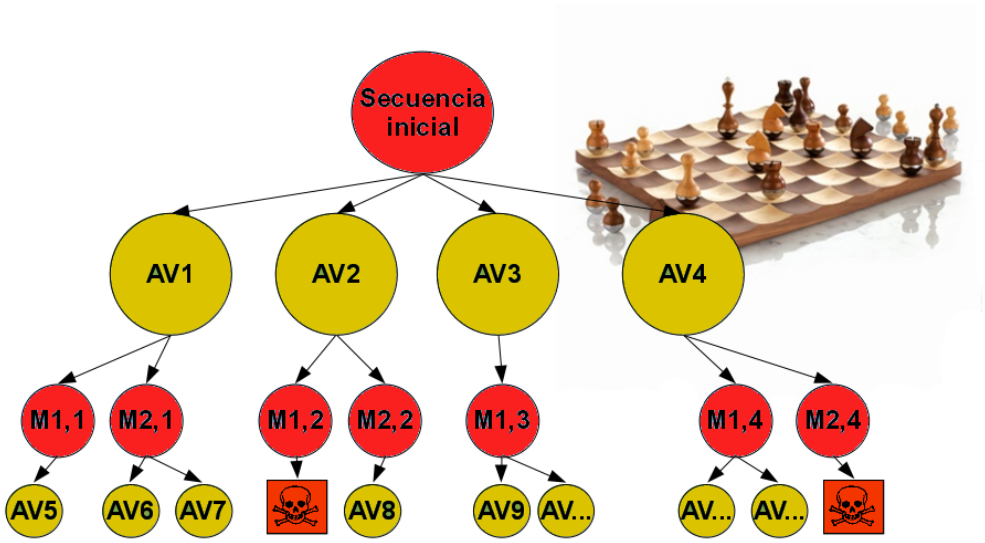
6. Ejecutar este proceso para cada una de las mutaciones obtenidas, recibiendo ahora como entrada cada una de esas secuencias mutadas. Luego, saltar al paso 8.

7. Informar resultado.

8. Terminar.

Al realizar estos pasos se genera un árbol donde cada nodo contiene, entre otras cosas, la secuencia del virus y la energía libre (**FE**) para dicha secuencia. Cabe destacar que una hoja se caracteriza por tener una secuencia a la que ningún antirretroviral puede ser aplicado.

Para lograr el objetivo de ver cómo el suministro de antirretrovirales afecta la estructura secundaria del RNA viral, basta con seguir los distintos caminos del árbol y observar como varía la FE a medida que avanza la terapia. En la siguiente sección se explicará más detalladamente como se implementó la solución.



En la imagen se muestra como se va generando el árbol, los círculos de color rojo representan los nodos y los de color amarillo, los antirretrovirales que se aplican para obtener la mutación del nodo hijo. Por otro lado, los recuadros que encierran una calavera establecen que la mutación obtenida es resistente a todos los antirretrovirales, por lo que esas ramas del árbol han llegado a su fin.

8.3. Implementación

En una visión general, como se mencionó en otras oportunidades, la solución consta de armar un árbol y analizar los distintos caminos del mismo. Debido a la gran cantidad de nodos que contiene este árbol, a lo costoso que es calcular la energía libre de una secuencia y a la necesidad de contar con un motor combinatorio, se decidió implementar la herramienta como la capa de aplicación del framework **FuD**. Ésto nos brinda dos facilidades imprescindibles:

1. La posibilidad de poder realizar los cómputos de manera distribuida.

2. Realizar, eficientemente, las combinaciones de antirretrovirales usando distintas políticas de combinación.

Según lo mencionado en 5.1.1, la herramienta constituye el nivel más alto del framework (la capa 5) y debe respetar la API definida por la capa subyacente, es decir, la capa **L4-CombinatoryEngine**.

8.3.1. Datos de Entrada

La herramienta recibe como entrada:

- *Secuencia Inicial del Virus*: Archivo que contiene una secuencia de nucleótidos representando el ADN del virus.

```
CCTCAGGTCACTCTTTGGCAACGACCCCTCGTCACAATAAAGGTA
GATAGGGGGGCAACTAAAGGAAGCTCTATTAGATACAGGAGACGG
CAGATGATACAGTATTAGAAGAAATGAGTGATTTGCCAGGAA...
```

- *Conjunto de Antirretrovirales*: Archivo .xml conteniendo la base de datos de antirretrovirales.

```
<antivirals>
  ...

  <antiviral id="Didanosine" num ="1" type ="↩
     tRT" class ="cNRTI">
    <resistance pos="164" aminos="R"/>
    <resistance pos="173" aminos="V"/>
  </antiviral>

  ...

</antivirals>
```

8.3.2. Representaciones de Datos

Secuencia de nucleótidos

Se representa a las secuencias de nucletidos como cadenas de caracteres. Denotamos a las mismas se denotan como expresiones regulares.

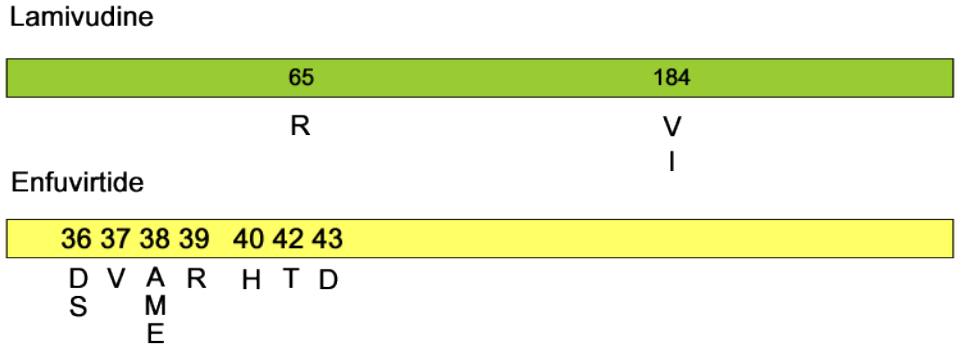
- $nuc\_arn = a|u|c|g|\_$
- $nuc\_adn = a|t|c|g|\_$

Al igual que los nucleótidos, se representan los genes de ADN y ARN.

- $gen\_arn = (nuc\_arn)^+$
- $gen\_adn = (nuc\_adn)^+$

Antirretrovirales

Un Antirretroviral consta de, principalmente, un listado de posiciones de resistencia. Estas posiciones establecen en qué moléculas del virus el antirretroviral se comporta de manera efectiva. En el diseño propuesto, estas posiciones no son sólo más que un par $< pos, aminos >$ donde $pos \geq 0$ y *aminos* es una lista de aminoácidos. Es decir, la posición de la resistencia describe un valor entero en el cual los aminoácidos actúan sobre el virus.



Además de este listado de posiciones, se incluye otro tipo de información para el antirretroviral, como un identificador, un nombre, una clase y un tipo.

8.3.3. Predicción de la Estructura Secundaria

Esta predicción consiste en obtener la *estructura secundaria* a partir de la estructura primaria (ver 2.2.2). Existen dos tipos de algoritmos para realizar esta tarea:

- **Predicción por *minimal free energy (mfe)***: Propuesto e implementado por Michael Zuker en el año 1981 [Zuker and Sankoff, 1984], utiliza programación dinámica para encontrar la estructura secundaria que minimiza la energía libre.
- **Predicción comparativa**: Utiliza diferentes métodos para comparar secuencias y estructuras con el fin de obtener una estructura por "consenso"[Gardner and Giegerich, 2004].

Si bien la *predicción comparativa* presenta un incremento en la fidelidad de los resultados obtenidos con respecto a la *predicción por mfe*, el primer tipo de

predicción requiere la existencia de un conjunto de secuencias relacionadas entre sí (homólogas) y ésto no siempre es posible. En particular, en este trabajo sólo interesa poder realizar predicciones de la estructura secundaria a partir de una sola secuencia, por lo que la predicción comparativa fue descartada.

Entre las implementaciones de la predicción por mfe, se destacan **RNAfold**[Hofacker et al., 1994] y **Mfold**. Ambas implementan el algoritmo propuesto por Michael Zuker con complejidad $\mathcal{O}(N^3)$ donde N es la longitud de la secuencia. A continuación se puede ver un ejemplo de una predicción realizada con **RNAfold**.

```
combeng@fudepan:~$ RNAfold

  Input string (upper or lower case);
  ....,....1....,....2....,....3....,....4...
  AAAGGCAACGGCCAU
  length = 15
  AAAGGCAACGGCCAU
  ...(((....)))..
  minimum free energy =  -4.40 kcal/mol
```

El resultado obtenido es, precisamente, la energía libre y la estructura secundaria representada con paréntesis y puntos. Donde los pares de paréntesis indican las bases "apareadas" o "unidas" y los puntos, las bases libres. Para este trabajo, solo es relevante la energía libre.

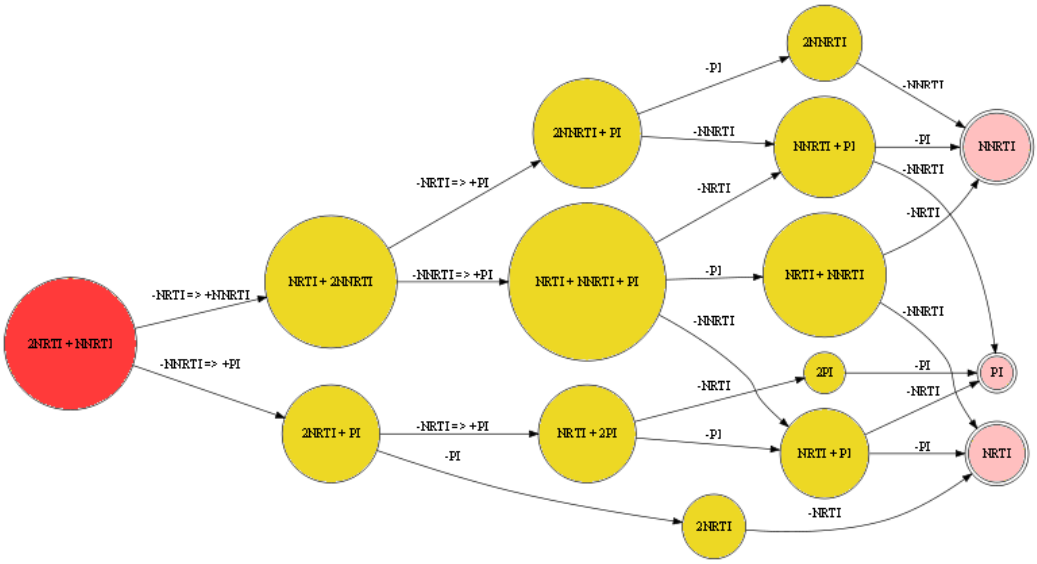
8.3.4. Selección de los Antirretrovirales a Aplicar

Como se muestra en la siguiente sección, cada nodo del árbol de ejecución dispone de una secuencia de nucleótidos. Para saber qué antirretrovirales combinar y así avanzar en la terapia, es necesario obtener de la base de datos de antirretrovirales sólo aquellos a los que la secuencia del nodo no es resistente. Además de la condición anterior, se seleccionan, preferentemente, aquellos que representen más obstáculos para el virus, logrando que al mismo le sea más costoso volverse resistente.[1]

[1]Para una explicación más detallada, examinar el capítulo 6 de ASO (`http://aso.googlecode.com`)

8.3.5. Preferencia en Combinación de Antirretrovirales

Los antirretrovirales individualmente no suprimen la infección por VIH a largo plazo, por lo cual, actualmente se usan combinaciones de éstos. Las combinaciones de antirretrovirales actúan incrementando el número de obstáculos para la mutación viral, manteniendo bajo el número de copias virales. En la figura 8.3.5 se muestra una Máquina de Estados Finita, de ahora en adelante FSM (por su acrónimo en inglés), representando cómo los antirretrovirales son combinados de acuerdo a la disponibilidad de los mismos. Como se explicó anteriormente, hay tres tipos de antirretrovirales (**NRTI**, **NNRTI**, **PI**). Dependiendo de la cantidad de cada uno que pueden ser aplicados a la secuencia del nodo corriente, se escoge una política de combinación según se detalla en la figura.



A modo de ejemplo supongamos que:

- El nodo raíz del árbol contiene la secuencia inicial del virus, llámese **X**.
- Hay más de dos antirretrovirales de cada tipo que se pueden aplicar a la secuencia **X**.
- La política de combinación del nodo raíz es la representada por el primer estado de la FSM, **2NRTI+1NNRTI**.

Para este caso, la herramienta generará todas las combinaciones de tres antirretrovirales, conteniendo dos del tipo **NRTI** y uno del tipo **NNRTI**. Luego,

por cada mutación obtenida al aplicar una combinación, se genera un hijo y se le ordena que se ejecute con la misma política de combinación. En el momento en que el hijo debe ser ejecutado pueden ocurrir dos cosas:

1. La cantidad de antirretrovirales de cada tipo que se le pueden aplicar a la mutación es suficiente para aplicar la misma política de combinación del nodo padre.

2. La cantidad de antirretrovirales de cada tipo que se le pueden aplicar a la mutación no es suficiente para aplicar la política de combinación del padre. En este caso, dependiendo del tipo del antirretroviral cuya cantidad es insuficiente, se cambia la política de combinación para ejecutar el nodo. Estos cambios se realizan acorde a la FSM mostrada.

8.3.6. Generación de Mutaciones a Partir de un Conjunto de Antirretrovirales

Un antirretroviral esta compuesto, entre otras cosas, por un conjunto de resistencias. Suponga que, a modo de ejemplo, se toma el antirretroviral Abacavir, el cual tiene como una de sus resistencias el aminoácido **R** en la posición **65**.

Luego, si el triplete número **65** de la secuencia codifica el aminoácido **R**, significa que dicha secuencia es resistente a tal antiviral.

En este caso, el triplete **ATT** codifica el aminoácido **R**, lo cual hace a la mutación resistente al antirretroviral Abacavir.
Para lo obtención de las mutaciones a partir de un conjunto de antirretrovirales, se desarrolló un algoritmo que puede ser resumido en los siguientes pasos:

1. Obtener el producto cartesiano de las resistencias de cada antirretroviral. Suponga el siguiente ejemplo,

   Antirretrovirales:

   $$A = \{(1,G),\ (2,T)\}$$
   $$B = \{(3,G),\ (2,T),\ (4,R)\}$$

   En un par *(X, Y)*: X representa la posición de la resistencia e Y el aminoácido al cual es resistente.

   Luego, el producto cartesiano entre A y B es:

   $$A \text{ x } B = \{\{(1,G),\ (3,G)\},\ \{(1,G),\ (2,T)\},\ \{(1,G),\ (4,R)\},\ \{(2,T),\ (3,G)\},\ \{(2,T),\ (2,T)\},\ \{(2,T),\ (4,R)\}\}$$

2. Aplicar las siguientes reglas a cada subconjunto S del producto cartesiano:

   a) Si $\exists(x,y),(z,w) \in S : x = z \wedge y \neq w \mapsto$ Eliminar S del producto cartesiano [2].

   b) Si $\exists(x,y),(z,w) \in S : x = z \wedge y = w \mapsto$ Eliminar una de las repeticiones de S.

   Una vez aplicadas las reglas, el producto cartesiano del paso anterior queda reducido a:

   $$A \text{ x } B' = \{\{(1,G),\ (3,G)\},\ \{(1,G),\ (2,T)\},\ \{(1,G),\ (4,R)\},\ \{(2,T),\ (3,G)\},\ \{(2,T)\},\ \{(2,T),\ (4,R)\}\}$$ [3]

3. A partir del conjunto obtenido en el paso anterior (A x B'), se crea un nuevo conjunto (T) conteniendo aquellos subconjuntos de A x B' que tengan la menor cantidad de elementos.

   Es decir,

   Sea $S_1 ... S_n \in$ A x B' : $S_i \in$ T **Sii** $\forall S_j \in$S : $count(\mathrm{S}_i) \leq count(\mathrm{S}_j)$.

   (*count*: función que toma un conjunto y retorna su cantidad de elementos)

   Continuando con el ejemplo, A x B' se reduce a:

   $$A \text{ x } B' = \{\{(2,\ T)\}\}$$

[2] Esto se debe a que no existen secuencias en las que para una posición haya dos aminoácidos.

[3] Sólo se aplicó la regla del inciso b).

4. Por cada subconjunto S_i del conjunto obtenido en el paso anterior, se obtiene una mutación. Esto se logra reemplazando en la secuencia de entrada (no mutada), cada una de las resistencias de S_i.

   En el ejemplo anterior, debido a que hay un único conjunto con mínima cantidad de elementos, se obtendrá solamente una mutación. La misma será el resultado de reemplazar el aminoácido que haya en la posición *2* de la secuencia de entrada, por *T*.

Es importante notar que todas las mutantes obtenidas al aplicar un conjunto de antirretrovirales a una secuencia, son resistentes todos los antirretrovirales del conjunto.

8.3.7. El nodo RNAFFE

Los nodos del árbol están formados principalmente por:

- La secuencia corriente del virus: representa el ADN corriente del virus.
- Política de combinación: Especifica la manera en que van a ser combinados los antirretrovirales.
- La terapia que se ha aplicado hasta el momento. Es decir, el nodo contiene una lista de conjuntos de antirretrovirales representando cuales han sido aplicados y en que orden se aplicaron los antirretrovirales que llevaron a obtener la secuencia mutante corriente.
- Lista de FE's: Su objetivo es almacenar la información de como varió la energía libre a medida que se avanzó en la terapia.

8.3.8. Obtención de Los Hijos Para Un Nodo

Debido a que para obtención de las distintas combinaciones de antirretrovirales se utilizó el patrón de diseño *Observer*[4], cada vez que una combinación es generada, el nodo es notificado con dicha combinación. Ante tal evento, el nodo reacciona de la siguiente manera:

1. Utilizando su secuencia y la combinación de antirretrovirales, obtiene una lista de secuencia mutantes, las cuales son resistentes a cada uno de los antirretrovirales de la combinación.
2. Con cada secuencia mutante se crea un hijo, al cual se le adjunta la nueva combinación de antirretrovirales como un paso más en la terapia. En este punto es calculada la **FE** de la mutación.

[4]Para mayor información, puede consultar A.3.1

8.3.9. Fallo Virológico

En 2.2.7 se observó la utilidad del término "fallo virológico" en el campo de la medicina. Un fallo virológico tiene lugar como resultado de una mixtura poco certera entre una mala adherencia al tratamiento y el desarrollo de resistencias.

En la aplicación, el término es utilizado para especificar esto último, la resistencia del virus. Suponga el caso en que se estén combinando dos antirretrovirales de tipo NRTI con uno de tipo NNRTI. Ahora bien, al aplicar una posible combinación (2**NRTI**+1**NNRTI**) se obtiene una secuencia de mutantes. Continuando con la supuesta ejecución, por cada una de las mutantes se debe crear un nodo hijo, pero antes se consulta con qué política de combinación debe ser creado. Esto se debe a que puede ser posible que que sobre una de las mutantes ahora sólo un antirretroviral del tipo NRTI sea aplicable, lo que implica que la política de combinación que se utilizaba antes ya no es viable. Luego, el nuevo nodo hijo dispondrá de otra política de combinación para la generación de sus mutantes acorde a la FSM (véase 8.3.5).

8.3.10. Datos De Salida

La salida obtenida tras una ejecución de la herramienta, es un archivo conteniendo cada camino del árbol. Dicho de otra forma, el archivo contiene todas las terapias posibles, además de mostrar cómo fue variando la energía libre en cada paso de dichas terapias. Este archivo, es utilizado por R para generar los resúmenes estadísticos necesarios (gráficos, tendencias, etc).

8.4. Dependencias Externas

En esta sección se mencionan las dependencias de la aplicación. La mayoría de ellas pertenecen a **FuDePAN**. Además, dos nuevas bibliotecas fueron desarrolladas, ambas como parte de este trabajo final.

- **Biopp**: Biblioteca C++ para Biología Molecular. Esta biblioteca es la que provee las estructuras de datos y métodos para manipular secuencias de nucleótidos. Además, se realizaron ciertas extensiones para este trabajo, particularmente la serialización/deserialización de ciertas estructuras utilizadas. Para más información, visite el sitio web de la misma, `biopp.googlecode.com`.
- **Fx-parser**: FXP es un parser XML de alto nivel para C++. De hecho, FXP es un adaptador de datos que se sitúa por encima de un parser

SAX (Simple API for XML Parsing). Utilizado para parsear el archivo .xml que contiene la base de datos de antivirales[5].

- **GetOpt_pp**: GetOpt forma parte de la *glibc* (The GNU C Library). La función GetOpt provee un mecanismo estructurado y eficiente para procesar opciones por línea de comandos para un programa de aplicación. El proyecto GetOpt_pp es una otra versión de lo antes mencionado, escrita en C++ y pertenece a **FuDePAN**. (`getoptpp.googlecode.com`).

8.4.1. Nuevas Bibliotecas

Para lograr una mejor modularidad en la aplicación *RNAFoldingFE* fueron creadas dos bibliotecas: Antivirals y RnaFolding. La mayoría de código de ambas bibliotecas es código perteneciente a otros proyectos de la fundación. El trabajo aquí realizado fue el de agrupar el contenido relevante para este trabajo, como así también extenderlo, proveyendo funcionalidades de gran necesidad (serialización/deserialización de ciertas clases, optimizaciones, correcciones, etc.)

- *Antivirals*: Define todas las estructuras de datos y métodos para la manipulación de secuencias de nucleótidos y antirretrovirales, como aplicar un conjunto de éstos sobre una secuencia y asi obtener sus mutantes, la carga de la base de datos de antirretrovirales, entre otras. El proyecto **ASO** es quién proveyó la mayor parte del código existente en la biblioteca. Para más información acerca de ASO, puede consultar `aso.googlecode.com`.

- *RnaFolding*: Provee, entre otras, las funcionalidades necesarias para obtener la energía libre de una secuencia de nucleótidos. Casi la totalidad de su código proviene del proyecto **VAC-O**. Al igual que el proyecto anterior, puede consultar su código, su documentación, sus resultados en la URL `vac-o.googlecode.com`.

Estas bibliotecas, además de ser necesarias para nuestra tesis, serán utilizadas por futuros proyectos de la fundación.

[5] `fx-parser.googlecode.com`

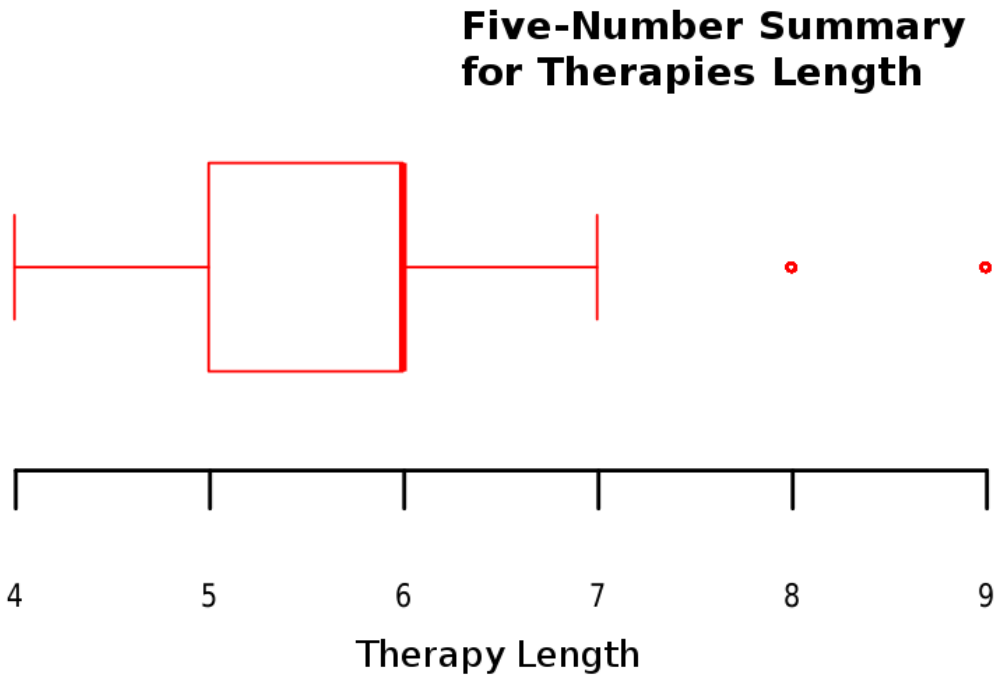
8.5. Resultados

La herramienta fue ejecutada utilizando los antirretrovirales de la base de datos de la *International AIDS Society*[6][Johnson and Richman, 2008] del año 2010 y, como secuencia inicial, la secuencia HXB2 en representación del virus. Cabe destacar que sólo el 50 por cierto de los caminos del árbol fueron computados dada a la escasez de recursos computacionales con los que se contaba.

Los resultados aquí presentes fueron obtenidos utilizando el lenguaje y entorno para computación estadística y gráficos conocido como **R**[7]. Todos ellos tuvieron como entrada de datos el archivo de resultados originado durante la ejecución de la aplicación.

8.5.1. Promedio de Longitud de Terapia

Esta estadística representa la cantidad de fallos virológicos que ocurrieron hasta que el virus se hizo resistente a todos los antirretrovirales. Dependiendo de cómo vaya mutando el virus, varía la longitud de la terapia.



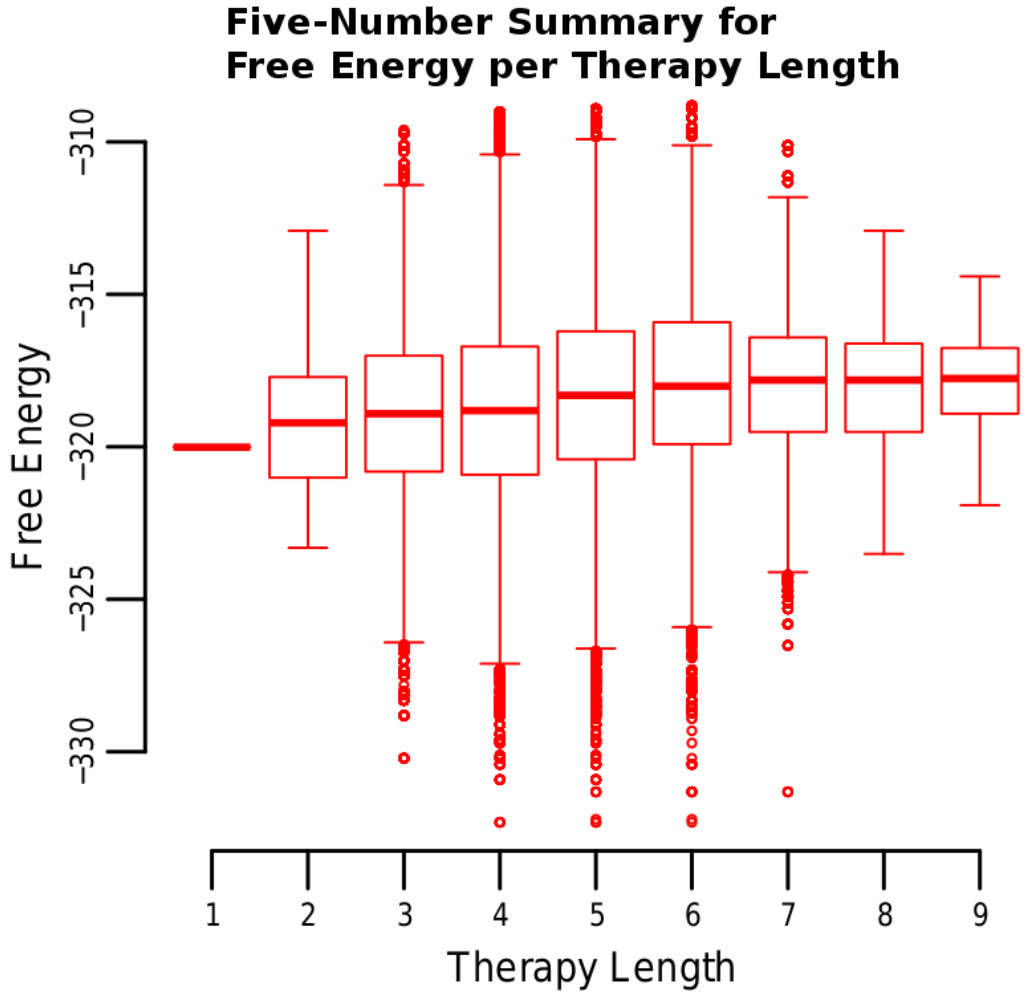
El gráfico revela que la mayoría de las terapias tienen longitud 6 y otra gran cantidad tienen como longitud, 5. Con una menor frecuencia se encuentran

[6] www.iasusa.org
[7] www.r-project.org

las terapias con longitudes de 6 y 7, mientras que en muy pocos casos, las mismas alcanzaron valores de 8 o 9. La importancia de estas estadísticas residen en el hecho de que mientras más larga es la terapia, más tiempo el paciente podrá ser mantenido bajo tratamiento, inhibiendo así el desarrollo de la enfermedad.

8.5.2. Sobre la Energía Libre de las Mutaciones en Relación a la Longitud de Terapia

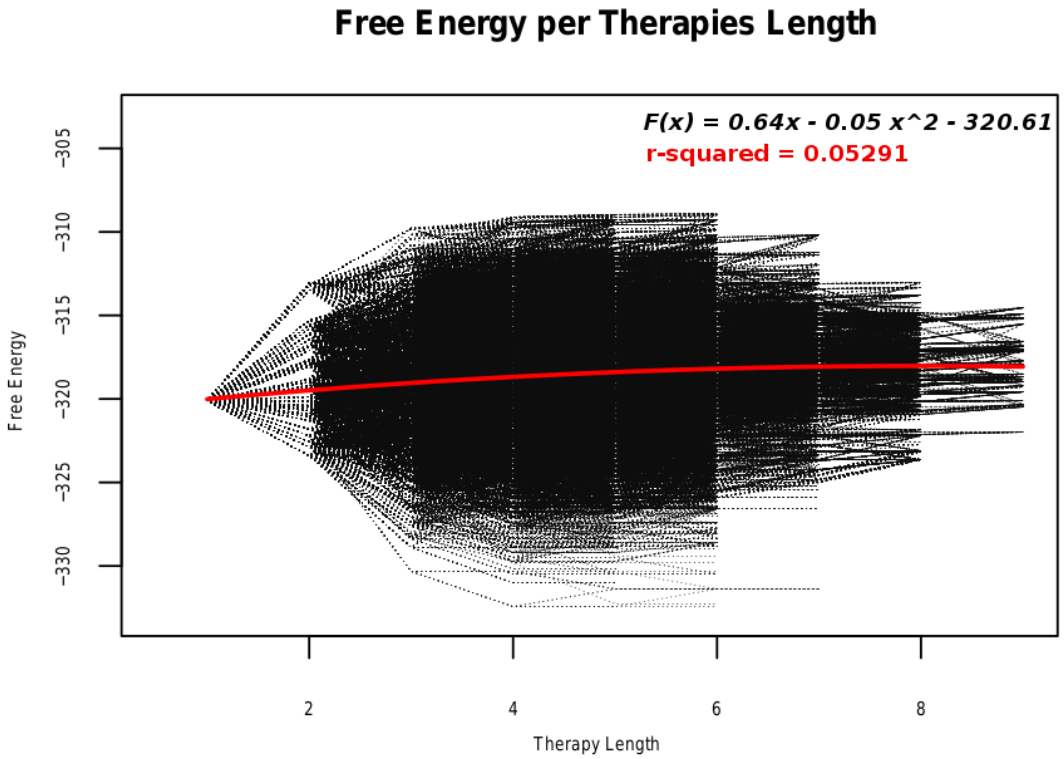
Cuando un antirretroviral es aplicado, la secuencia de nucleótidos que representa el ADN del virus sufre una modificación, es decir, se produce una secuencia mutante y su Energía libre varía. La idea de este resumen estadístico es mostrar cómo la energía libre de las mutaciones se incrementa o decrementa acorde se avanza sobre la terapia.



Tal y como se puede observar en el gráfico, a medida que se avanza en las terapias, los valores promedio para la energía libre se incrementan de manera moderada. Si bien estas variaciones son pequeñas, las mismas informan que, en la mayoría de los casos, a lo largo de un tratamiento el virus sufre cierta desestabilización.

8.5.3. Estimación de Tendencia de la Energía Libre

Los datos representados en el siguiente gráfico no varían respecto del punto anterior, la única diferencia es que se exhiben de otra manera. Esto se realizó para obtener información acerca de la tendencia[8] de la energía libre a medida que se van aplicando los antirretrovirales de la terapia.



Como muestra el gráfico, los datos se encuentran de manera dispersa. La línea roja que cruza el gráfico de nubes representa la tendencia. La fórmula

[8] www.en.wikipedia.org/wiki/Trend_estimation

de dicha línea esta dada por la ecuación:

$Y = 0.64X + -0.05X^2 - 320.61$

y su R^2 es de 0.05291. Si bien el R^2 es pequeño, lo cual significa que la ecuación no aproxima con gran exactitud, se esperan obtener resultados más precisos cuando la herramienta compute el árbol de ejecución de manera completa.

Capítulo 9

Conclusiones

Como resultado de este trabajo se ha obtenido una nueva capa para el framework **FuD**, de nombre **CombEng**. La misma ha cumplido con todas las expectativas y ha cubierto cada uno de los requerimientos planteados desde un comienzo. Si bien sólo se ha desarrollado una aplicación relevante, y algunas otras como prueba, se puede concluir que el motor combinatorio se encuentra apto para resolver cualquier problema que requiera de la combinación de cualquier tipo de elementos. Aquellas aplicaciones que hacen un uso correcto del framework **FuD** y, por ende, de la nueva capa, no deberán preocuparse por asuntos relacionados a la programación paralela. Esto permite realizar el procesamiento de las combinaciones, muchas veces de gran costo computacional, de manera distribuida y sin la necesidad de tener grandes conocimientos en el área.

En cuanto a la aplicación **RnaFFE**, se han obtenido numerosos y valiosos datos que hasta el momento no eran conocidos. Tanto la aplicación como los datos recolectados recibieron buenas críticas tras ser presentados en el 2do Congreso Argentino de Bioinformática y Biología Computacional. La información obtenida sirvió para dar un abordaje inicial a como varía la energía libre a medida que se avanza en una terapia antirretroviral. Debido a la falta de recursos y tiempo, la aplicación no pudo ser ejecutada en su totalidad, por lo que los datos que fueron recopilados pertenecen al 60 % del árbol total. No obstante, la información adquirida fue suficiente para obtener ciertas estadísticas y tendencias interesantes.

9.1. Trabajo a Futuro

A continuación se muestran las tareas que quedan pendientes en este trabajo. Las mismas se encuentran agrupadas según pertenezcan a la aplicación *RNAFoldingFE* o a la nueva capa acoplada a **FuD**(**CombEng**).

9.1.1. Trabajo Futuro Para la Aplicación *RNAFolding-FE*

- Optimizar la implementación de la FSM que define la manera en que los antirretrovirales son combinados.
- Implementar políticas de poda para achicar el espacio de búsqueda.
- Implementar la funcionalidades necesarias para hacer que la aplicación sea independiente de la arquitectura.
- Implementar una interfaz gráfica que permita visualizar los resultados intuitivamente.

9.1.2. Trabajo futuro para CombEng

- Implementar nuevas políticas de combinación.

Appendices

Apéndice A

Patrones De Diseño

A.1. Qué Son Los Patrones De Diseño

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Un patrón describe un problema que ocurre varias veces en un sistema, y la base de la solución a ese problema. Los patrones de diseño son el resultado del consenso de los profesionales en el área y brindan herramientas a los diseñadores de sistemas para no escoger malos caminos, valiéndose de documentación disponible en lugar de simplemente la intuición.

A.2. Patrones

Un patrón de diseño es una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que se debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reusable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias. Cada patrón de diseño se focaliza sobre un problema o issue particular de diseño (DOO).

En general, un patrón tiene cuatro elementos esenciales:

1. **El nombre del patrón** es un manejador que se usa para describir un problema de diseño, su solución, y consecuencias.

2. **El problema** describe cuando aplicar el patrón, explica el problema y su contexto. También describe problemas de diseño específicos tales como C̈omo representar un algoritmo como un objeto.̈ Además, describe

la estructura de clases y objetos que son sintomáticas de un diseño inflexible. A veces, el problema puede incluir una lista de condiciones que deben ser reunidas antes de que tenga sentido aplicar el patrón.

3. **La solución** describe los elementos que integran el diseño, sus relaciones, responsabilidades y colaboración. La solución no describe un diseño particular concreto o implementación, porque un patrón puede ser aplicado en muchas situaciones diferentes. De hecho, el patrón provee una descripción abstracta de un problema de diseño y como una disposición general de los elementos lo resuelve.

4. **Las consecuencias** son los resultados y compromisos de aplicar el patrón. Estas son fundamentales para evaluar alternativas de diseño y para la comprensión de los costos y beneficios de aplicar el patrón. Las consecuencias de un patrón incluye su impacto sobre la flexibilidad del sistema, expansión o portabilidad.

A.3. Patrones Utilizados En CombEng

A lo largo del proceso de diseño se encontraron problemas que coincidían con patrones. Estos patrones se explican brevemente a continuación:

A.3.1. Observer

Define una dependencia uno a muchos entre objetos, de forma que si un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. En la Fig. A.1 se ve el modelo estándar del patrón.

Aplicabilidad

Se usa el patrón *Observer* cuando:

- Una abstracción tiene dos aspectos, uno dependiente del otro. Encapsular estos aspectos en objetos por separado le permite variarlo y reutilizarlos de manera independiente.
- Un cambio de un objeto requiere cambiar a los demás, y no se sabe cuantos objetos hay que cambiar.
- Un objeto debe ser capaz de notificar a otros objetos sin hacer suposiciones acerca de qué son estos objetos. En otras palabras, no se quiere que los objetos estén muy acoplados.

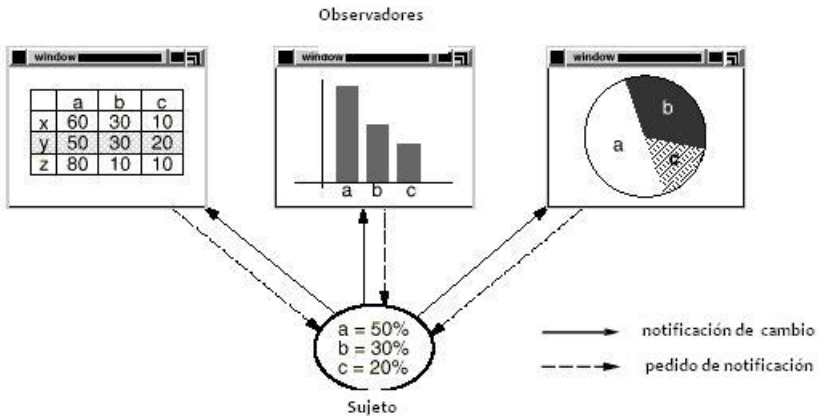


Figura A.1: Patrón Observer.

Bibliografía

[Booch et al., 2005] Booch, G., Rumbaugh, J., and Jacobson, I. (2005). *Unified Modeling Language User Guide*. Second edition.

[Cichocki, 2009] Cichocki, M. (2009). Highly active antiretroviral therapy.

[Cichocki, 2010] Cichocki, M. (2010). The history of hiv.

[Dijkstra, 1988] Dijkstra, E. W. (1988). On the cruelty of really teaching computing science.

[Foster, 1995] Foster, I. (1995). *Designing and Building Parallel Programs*. Addison-Wesley.

[Gamma et al., 2005] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2005). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Gardner and Giegerich, 2004] Gardner, P. P. and Giegerich, R. (2004). A comprehensive comparison of comparative rna structure prediction approaches. *BMC Bioinformatics*.

[Hofacker et al., 1994] Hofacker, I. L., Fontana, W., Stadler, P. F., Bonhoeffer, L. S., Tacker, M., and Schuster, P. (1994). Fast folding and comparison of rna secondary structures. *Monatshefte für Chemie*, 125(2).

[Johnson and Richman, 2008] Johnson, C. and Richman (2008). Update of the drug resistance mutations in hiv-1.

[Lamport, 1994] Lamport, L. (1994). *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts.

[Leonard and Schaffer, 2006] Leonard, J. and Schaffer, D. (2006). Antiviral rnai therapy: emerging approaches for hitting a moving target.

[Martin, 2000] Martin, R. C. (2000). *Design Principles and Design Patterns*.

[Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction, Second Edition*. Prentice Hall Professional Technical Reference, Santa Barbara.

[Niko Beerenwinkel and Selbig, 2006] Niko Beerenwinkel, Thomas Lengauer, M. D. R. K. H. W. K. K. D. H. and Selbig, J. (2006). Methods for optimizing antiviral combination therapies. 19.

[Oetiker et al., 2008] Oetiker, T., Partl, H., Hyna, I., and Schlegl, E. (2008). *The Not So Short Introduction to LaTeX 2ε*.

[Perks, 2003] Perks, M. (2003). Best practices for software development projects.

[Pressman, 1982] Pressman, R. S. (1982). *Software Engineering - A Practitioner's Approach - Fourth Edition*.

[Stroustrup, 1991] Stroustrup, B. (1991). *The C++ programming language 2nd ed*. Addison-Wesley.

[Stroustrup, 1997] Stroustrup, B. (1997). *The C++ Programming Language, Third Edition.* Addison-Wesley, Murray Hill, New Jersey.

[Sussman et al., 2008] Sussman, B. C., Fitzpatrick, B. W., and Pilato, C. M. (2008). *Version Control with Subversion*. O'Reilly.

[Wirfs-Brock and McKean, 2003a] Wirfs-Brock, R. and McKean, A. (2003a). *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley.

[Wirfs-Brock and McKean, 2003b] Wirfs-Brock, R. and McKean, A. (2003b). *Object Design: Roles, Responsibilities and Collaborations*. Addison-Wesley.

[Zuker and Sankoff, 1984] Zuker, M. and Sankoff, D. (1984). Rna secondary structures and their prediction. *Bulletin of Mathematical Biology*, 46(4).