Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tesis

# Validación de protocolos de comunicación en programas JAVA

Pablo Santillán
Director: Hernán Melgratti

Diciembre 2016

Dirección: Pabellon I, Ciudad Universitaria – C1428EGA
Tel: (54.11) 4576-3359 — Conmutador: (54.11) 4576-3390 al 96 (int 701/702) —
E-mail: pablogs2006@gmail.com

# Agradecimientos

A Hernán, por su compromiso y dedicación, un gran director y una excelente persona, demostrando su profesionalismo y compromiso en cada momento.

A Gabo, Gaspar y Guille, mis compañeros de grupo de las primeras materias, las personas con las cuales di los primeros pasos en la facultad.

A Bruno y Ricky, compañeros de la segunda mitad de la carrera, con quienes pasamos noches estudiando para rendir parciales y finales, y me brindaron su amistad. Sin la ayuda, apoyo y compromiso de ellos no estaría hoy aca.

A Chris, amigo incondicional, que me ayudó y orientó en todo momento. Cada palabra de apoyo y fuerza me ayudaron a no bajar los brazos y a siempre ir por mas. Gracias por planificar cientos de veces el cronograma adecuado para mi.

A mis abuelos, que donde estén, se que me acompañan, y que están orgullosos de este momento.

A mis hermanos, que siempre me apoyaron y me desearon lo mejor a lo largo de esta etapa.

A mi mujer y mi hijo, por bancarme días y noches encerrado estudiando, de no compartir momentos con ellos para poder dedicarle tiempo a la facultad, pero saber que estaban ahí, apoyándome en cada momento. Los amo.

A mis papás, por brindarme la posibilidad de estudiar, por luchar siempre por nosotros, por nunca bajar los brazos, y por hacer lo imposible para que yo esté hoy en este lugar. Todo este esfuerzo y dedicación es por ustedes. Gracias por tanto.

# Resumen

Muchos de los sistemas de hoy en día tienen comportamiento distribuido. Los sistemas distribuidos realizan sus tareas interactuando con otros componentes, con lo cual la coordinación, especificación y análisis de las interacciones es un problema crucial. Es por eso que se buscan especificaciones precisas que garanticen la correctitud en las interacciones (protocolos) entre los componentes distribuidos.

Existen resultados teóricos que tratan estos problemas de correctitud. Recientemente, se ha propuesto formalizar cada componente como máquinas finitas comunicantes (Communicating machines) y verificar que su composición es correcta sintetizando a partir de las mismas un protocolo global o coreografía. El problema, se reduce entonces a modelar cada componente del sistema como una máquina de estados finito comunicante y luego verificar que la composición de las mismas es correcta.

En este trabajo abordaremos el problema de generar estas abstracciones a partir de código Java. Utilizaremos programas que se comunican a través de Sockets y aplicamos la ténica para comprobar la correcta implementación de casos de estudio.

# Índice general

# Índice de figuras

# Capítulo 1

# Introducción

La mayoría de los sistemas de hoy en día involucran concurrencia o comportamiento distribuido. Ellos corren concurrentemente o en hardware de multiprocesos, interactuando con los otros sistemas y accediendo a los datos o recursos computacionales a través de la red. El uso de sistemas distribuidos tiene sus ventajas, como por ejemplo la escalabilidad, concurrencia, distribución de funcionalidad, y tolerancia a fallas, que son detalladas y explicadas en [13], [17], [15], [12], [11], pero también tiene sus desventajas.

Con el fin de realizar una tarea, componentes de un sistema distribuido tienen que coordinar sus ejecuciones interactuando con otros componentes, con lo cual la especificación y el análisis de las interacciones a través de sus componentes es un problema crucial.

La comunicación basada en mensajes es un mecanismo de interacción común usado en sistemas distribuidos o concurrentes, donde cada sistema se comunica con los otros a través del envío y recepción de mensajes. Una técnica estandar para especificar la estructura de comunicación de sistemas que se comunican a través del intercambio de mensajes son las coreografías. Una especificación coreográfica identifica el conjunto de mensajes permitidos y el orden en que deben ser intercambiados entre los componentes de un sistema distribuido. Una coreografía es realizable si existe una manera de implementar un conjunto de componentes que realicen dicha coreografía, es decir que interactúan en la manera descripta por la coreografía, como se menciona en [3],[4].

Una manera abstracta de representar la implementación de un sistema distribuido son los **Communicating finite states machine (CFSM**), que representan procesos los cuales se comunican a partir del intercambio de mensajes asincrónicos via canales FIFO. Su principal ventaja es que permiten caracterizar simplemente las propiedades escenciales de la comunicación, tales como errores y deadlocks.

El uso de especificaciones coreográficas es un proceso muy valorado a la hora de garantizar correctitud en las interacciones entre los componentes de los sistemas distribuidos, ya que en conjunto con otros conceptos (como CFSM), ayuda a validar la correctitud de una comunicación.

Hoy en día, existen algoritmos teóricos que hacen uso de estas especificaciones en conjunto con los CFSM con el fin de garantizar que el protocolo utilizado por procesos que se comunican intercambiando mensajes, es correcto. Un ejemplo es [10], que viene acompañado por una herramienta que realiza este tipo de validaciones. El desafío es utilizar la técnica para verificar aplicaciones.

## 1.1. Objetivos

El objetivo de este trabajo es tomar programas reales que se comuniquen a través del envío y recepción de mensajes, y garantizar que la comunicación entre los mismos es correcta, tomando como base el resultado teórico presentado en [10].

Nos vamos a basar en programas escritos en JAVA, cuya comunicación se realiza a través de sockets. Para probar que la comunicación es correcta, vamos a dar un algoritmo que toma como entrada los programas que intervienen en la comunicación, analiza estáticamente su comportamiento y como resultado construye las **CFSM** que son requeridas como entrada para la herramienta descripta en [10], que decide si la composición de las máquinas es correcta o no.

## 1.2. Estructura de la tesis

La tesis esta organizada de la siguiente manera. En la sección 2 se reportan las nociones fundamentales sobre **CFSM** y **GMC**. Definimos el algoritmo teórico del cual hacemos uso para validar los protocolos de comunicación. Luego definimos los programas sobre el cual vamos a aplicar la técnica y las condiciones que estos programas deben cumplir. Finalmente, mencionamos las herramientas que tomamos como base para analizar los programas reales.

En el capítulo 3 definimos el algoritmo que vamos a implementar para realizar la validación. Se hace una descripción de alto nivel del funcionamiento. Luego se describe en profundidad las técnicas utilizadas por el algoritmo, y finalmente se compone nuestro algoritmo con el algoritmo teórico que realiza la validación.

En el capítulo 4 aplicamos el algoritmo sobre programas JAVA de pequeña escala que fueron seleccionados como casos de estudio y analizamos los resultados.

Por último, en el capítulo 5 presentamos las conclusiones y hablaremos sobre los trabajos futuros.

# Capítulo 2

# Background

La importancia que los sistemas distribuidos tienen hoy en día, introducen siempre la necesidad de especificaciones precisas, que sirven para garantizar la correctitud de las interacciones (protocolos) entre los componentes distribuidos.

Para lograr esto, *Multiparty sesion types* es un tipo de disciplina que puede garantizar una comunicación segura para procesos distribuidos, a través de una especificación corogreáfica (llamada especificación global) de las interacciones entre un conjunto de componenentes [7].

Una especificación global puede ser proyectada a especificaciones end-point (llamadas especificaciones locales), para obtener el comportamiento local de componentes. La metodología de ingeniería de software asociada con coreografías es usualmente un enfoque uni-direccional (top-down) en el ciclo de vida del desarrollo de software. Esta metodología tiene influencia en la industria ya que permite a los desarrolladores chequear sus componentes separadamente contra las correspondientes proyecciones de la coreografía, garantizando la correcta implementación del protocolo.

Sin embargo, los enfoques basados en coreografías no soportan completamente el ciclo de vida del desarrollo de software, ya que por ejemplo, se carece de algoritmos para obtener modelos globales cuando se modifican proyecciones locales de componentes.

A continuación se describe un enfoque reciente que permite completar el ciclo y sintetizar a partir de las proyecciones locales, la coreografía global.

## 2.1. De CFSM a Global Graph

En [10] se propone un algoritmo para construir coreografías a partir de especificaciones de comportamiento de componentes que interactúan a través del envío y recepción de mensajes asincrónicos. Estas interacciones son representadas por *CFSMs*.

Dicho algoritmo toma como entrada un conjunto de CFSMs y produce una coreografía expresada como un global graph [7], un modelo gráfico relacionado con

coreografías BPMN 2.0 [1].

CFSMs representan procesos que se comunican por el intercambio de mensajes asincrónicos via canales FIFO [7]. Se sabe que la caracterización de algunas propiedades de la comunicación, tal como decidir si es libre de deadlocks, es un problema NP−hard. En este algoritmo, se da una condición de decidibilidad, llamada **"generalised multiparty compatibility (GMC)"**, que caracteriza un conjunto de sistemas para los cuales preguntas como la de arriba puede ser decidible.

El algoritmo entonces produce un global graph a partir de un conjunto de CFSMs que cumplen la condición de decidibilidad. El global graph es construido a partir de la transformación de CFSMS en redes de Petri seguras, utilizando el algoritmo en [3]. Se destaca que un gran número de sistemas cumplen con GMC.

*Definición:* una CFSMs es un sistema de transición finita dado por una 4−upla M = (Q, $q_0$, A, $\delta$) donde

- Q es un conjunto finito de estados,
- $q_0 \in Q$ es el estado inicial,
- A es el alfabeto,
- $\delta \subseteq Q$ x Act x Q es el conjunto de transiciones.

Las transiciones de una CFSM son etiquetadas por acciones; la etiqueta ***sr!a*** representa el envío de un mensaje "a" desde la máquina s a r y, ***sr?a*** representa la recepción del mensaje "a" enviado por la máquina r.

Un sistema *S* se define como el conjunto de *CFSM* de los componentes que participan en la comunicación.

A partir del sistema de comunicación S, se construye el sistema de transiciones sincrónicas *TS(S)*, que representa todas las posibles ejecuciones sincrónicas del sistema S, y se chequea si TS(S) cumple con la condición de decidibilidad GMC.

GMC se basa en 2 condiciones:

- **Representabilidad**: para cada máquina, cada traza y cada opción están representadas en *TS(S)*.
- **Propiedad de bifurcación**: en cualquier lugar que exista una opción en *TS(S)*, una única máquina toma la decisión, y cada uno de los otros participantes, es conciente de que rama fue elegida, o no participa de en la elección.

*Representabilidad* garantiza que *TS(S)* contiene suficiente información para decidir propiedades de seguridad de cualquier ejecución de (S); y la *propiedad de Bifurcación* asegura que, si una bifurcación en *TS(S)* representa una opción, esa opción es "bien formada".

Como corolario, si un sistema *S* es *GMC*, entonces el sistema es seguro, es decir, no se cumplen las siguientes propiedades:

- **Orphan message**: representa un mensaje que es enviado por una máquina, pero que no es recepcionado por ningún otra.
- **Deadlock**
- **Unspecified reception**: representa un estado donde una máquina espera la recepción de un mensaje que no es enviado por ningún otra.

El algortimo para construir un Global Graph $G$ a partir de un sistema de transiciones sincrónicas *TS(S)* consiste de los siguientes pasos:

1. se aplica el algoritmo de Cortadella [5] para derivar un TS(S) a una red de Petri $\mathbb{N}$;
2. se transforma $\mathbb{N}$ para tener solo 1 lugar inicial;
3. se juntan las transiciones donde sea posible, para hacer explícitas las uniones y bifurcaciones del flujo;
4. se transforma la red anterior en un pre Global Graph;
5. se limpia el pre Global Graph de vértices innecesarios para obtener el Global Graph final.

Una representación gráfica de los pasos puede verse en la figura 2.1.

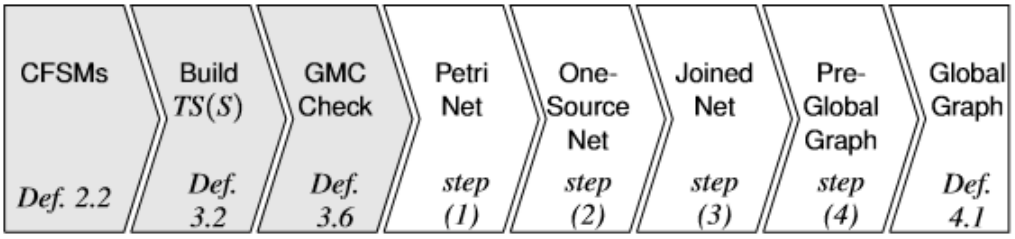


Figura 2.1: Contrucción de Global Graph

En este trabajo denotaremos a este algoritmo como Global Graph Builder (**GGB**).
*GGB* toma como entrada una representación textual del sistema (S) de comunicación a analizar (ver figura 2.2).

## 2.2. Análisis de programas

En este trabajo se van a tomar programas reales, y mediante un procedimiento, generar las CFSM que sirvan como input para el algoritmo mencionado en 2.1. Nos vamos a basar en programas escritos en JAVA, que se comunican entre sí a través del envío y recepción de mensajes por sockets.

```
.outputs
.state graph
alice0 1 ! bwin alice1
alice0 2 ! cwin alice1
alice1 1 ? sig alice2
alice1 2 ! score alice3
alice2 2 ! score alice4
alice3 1 ? sig alice4
alice4 3 ! free alice0
.marking alice0
.end

.outputs
.state graph
bob0 0 ? bwin bob1
bob1 2 ! close bob2
bob0 2 ? blose bob2
bob2 0 ! sig bob0
.marking bob0
.end
```

Figura 2.2: Input herramienta

Para generar las CFSMs a partir de los programas, se analizaron 2 posibilidades. La primer opción es analizar el código fuente y aproximar el comportamiento del programa. En este caso, en necesario realizar un parsing del código fuente y analizarlo con el fin de obtener las CFSM buscadas. La otra opción es analizar el código compilado (.class) en lugar del código fuente. Una de las ventajas de trabajar con clases compiladas es que, obviamente, no necesitamos el código fuente. Otra ventaja de trabajar con código compilado es que es posible analizar, modificar o generar transformaciones de clases en tiempo de ejecución.

El Análisis, la generación y transformación de programas son técnicas muy usadas que pueden aplicarse en diversas situaciones, por ejemplo:

1. Análisis de programas para simples parseo sintáctico hasta complejos analizadores semánticos.

2. La generación de programas utilizadas en compiladores.

3. La transformación de programas puede ser utilizado para optimizar programas, para monitorear código.

En este trabajo se opta por analizar código compilado, debido a las ventajas mencionadas anteriormente y que, además, hay muchas herramientas que ayudan a entender, analizar y manipular el código compilado.

Si bien se puede realizar el análisis sin tener el código fuente, en este trabajo es necesario incorporar alguna forma de poder reconocer los componentes que participan en la comunicación. Esto se logra mediante el uso de anotaciones propias, que deben agregarse en clases que poseen componentes que interactúan en la comunicación (mas detalles en sección 5.1).

## 2.3. Análisis estático de programas

Una clase compilada mantiene la información estructural y muchos de los símbolos del código fuente. En la figura 2.3 se ilustra de manera esquemática la estructura exacta descrita en la especificación de la Java virtual machine.

En particular:

1. Una sección que describe los modificadores, como público o privado, nombre, interfaces, anotaciones de una clase.

2. Una sección por cada field declarado en la clase. Cada sección describe el modificador, nombre, y tipos de los fields.

3. Una sección por cada método y constructor de una clase. Cada sección describe el modificador, el nombre y el tipo de salida de cada método, como así también el código compilado del método en forma de una secuencia de instrucciones de byteCode.

| | |
|---|---|
| Modifiers, name, super class, interfaces | |
| Constant pool: numeric, string and type constants | |
| Source file name (optional) | |
| Enclosing class reference | |
| Annotation* | |
| Attribute* | |
| Inner class* | Name |
| Field* | Modifiers, name, type |
| | Annotation* |
| | Attribute* |
| Method* | Modifiers, name, return and parameter types |
| | Annotation* |
| | Attribute* |
| | Compiled code |

Figura 2.3: Estructura .Class

En las clases compiladas, las clases o interfaces son representadas con nombres internos. Las demás situaciones, tales como los fields, son representados con "descriptores de tipo" (ver figura 2.4).

| Java type | Type descriptor |
|---|---|
| `boolean` | `Z` |
| `char` | `C` |
| `byte` | `B` |
| `short` | `S` |
| `int` | `I` |
| `float` | `F` |
| `long` | `J` |
| `double` | `D` |
| `Object` | `Ljava/lang/Object;` |
| `int[]` | `[I` |
| `Object[][]` | `[[Ljava/lang/Object;` |

Figura 2.4: Descriptores de tipo

Los descriptores para los tipos primitivos son simples caracteres, el descriptor de una clase es el nombre interno de la clase precedido de la letra "L" y terminado con ";". Por ejemplo, el descriptor de la clase Socket de Java es Ljava/net/Socket;

Para analizar los bytecodes, existen distintas herramientas que lo analizan y brindan la posibilidad de manipularlo. Ejemplos de estos son BCEL [6], ASM [2], JULIA [16], entre otros. La ventaja de estos frameworks es que permiten analizar las funciones dentro de una clase y armar rápidamente diagramas de flujos por cada método.

### 2.3.1. ASM, uso y funcionamiento

En este trabajo se opta por utilizar ASM [4], como framework de análisis de bytecodes.

ASM es un framework de análisis y manipulación de java byteCodes. También puede ser utilizado para modificar clases existentes o agregar clases, directamente en forma dinámica. El propósito principal es analizar y manipular el byteCode mediante conceptos de nivel superior a los bytes, tales como constantes, String, tipos y clases, conceptos más amigables para el usuario.

Algunas de las características salientes de ASM son:

1. Fue diseñado para ser rápido, flexible y simple.
2. Tiene una API simple, modular y bien diseñada, que hace que su uso sea simple.
3. Tiene buena documentación y cuenta con un plugin para eclipse.
4. Es open source y puede usarse mediante distintas APIs.

ASM provee dos APIs para analizar el código: "core API", la cual provee una representación de clases basadas en eventos, y "tree API", que provee una representación de clases basadas en objetos. En esta tesis se utiliza la representación basada en eventos, ya que es fácil de manipular y no requiere crear y guardar en memoria un árbol de objetos, como en la otra representación.

Con el modelo basado en eventos, una clase se representa como una serie de eventos ordenados, donde cada evento representa un elemento de una clase, tal como el header, un field, un método, una instrucción, etc.

ASM utiliza el patrón Visitor en su modelo orientado a eventos. En la programación orientada a objetos y en la ingeniería de software, el patrón de diseño "Visitor" es un patrón que intenta separar un algoritmo de una estructura de objetos sobre el cual opera [8]. Un resultado práctico de esta separación es la capacidad de agregar nuevas operaciones a estructuras de objetos existentes, sin tener que modificar dichas estructuras. En este patrón, se crea una clase Visitor que toma una referencia a una instancia como parámetro, e implementa el objetivo de la operación a partir del doble dispatch.

ASM Provee tres componentes principales para generar, analizar y transformar código:

- **ClassReader**, es la clase que parsea una clase compilada dada como un array de bytes, y llama a los correspondientes visitors de la instancia **ClassVisitor** pasada como parámetro del método *accept* de la misma. Es comunmente llamado un "Aceptor".
- **ClassWriter**, esta clase es una subclase de la clase abstracta **ClassVisitor**, que construye clases compiladas directamente en forma binaria.

- **ClassVisitor**, clase que delega todas las llamadas que esta recibe a métodos de otra instancia de ClassVisitor.

En el siguiente diagrama se muestra como interactúan las clases antes mencionadas con un diagrama de secuencias (ver figura 2.5). La principal interacción esta dada por las clases **ClassReader** y **ClassVisitor**, a partir de la invocación del método *accept* de la clase ClassReader, en donde se pasa como parámetro una instancia de la clase ClassVisitor. A partir de esta invocación, la instancia de la clase ClassReader analiza estáticamente una clase, e invoca a los diferentes métodos de la clase ClassVisitor, con la información analizada. El primer lugar, informa nombre de la clase, interfaces que implementa y accesos. Luego, informa en el siguiente orden los fields, métodos, clases internas y atributos que posee la clase;

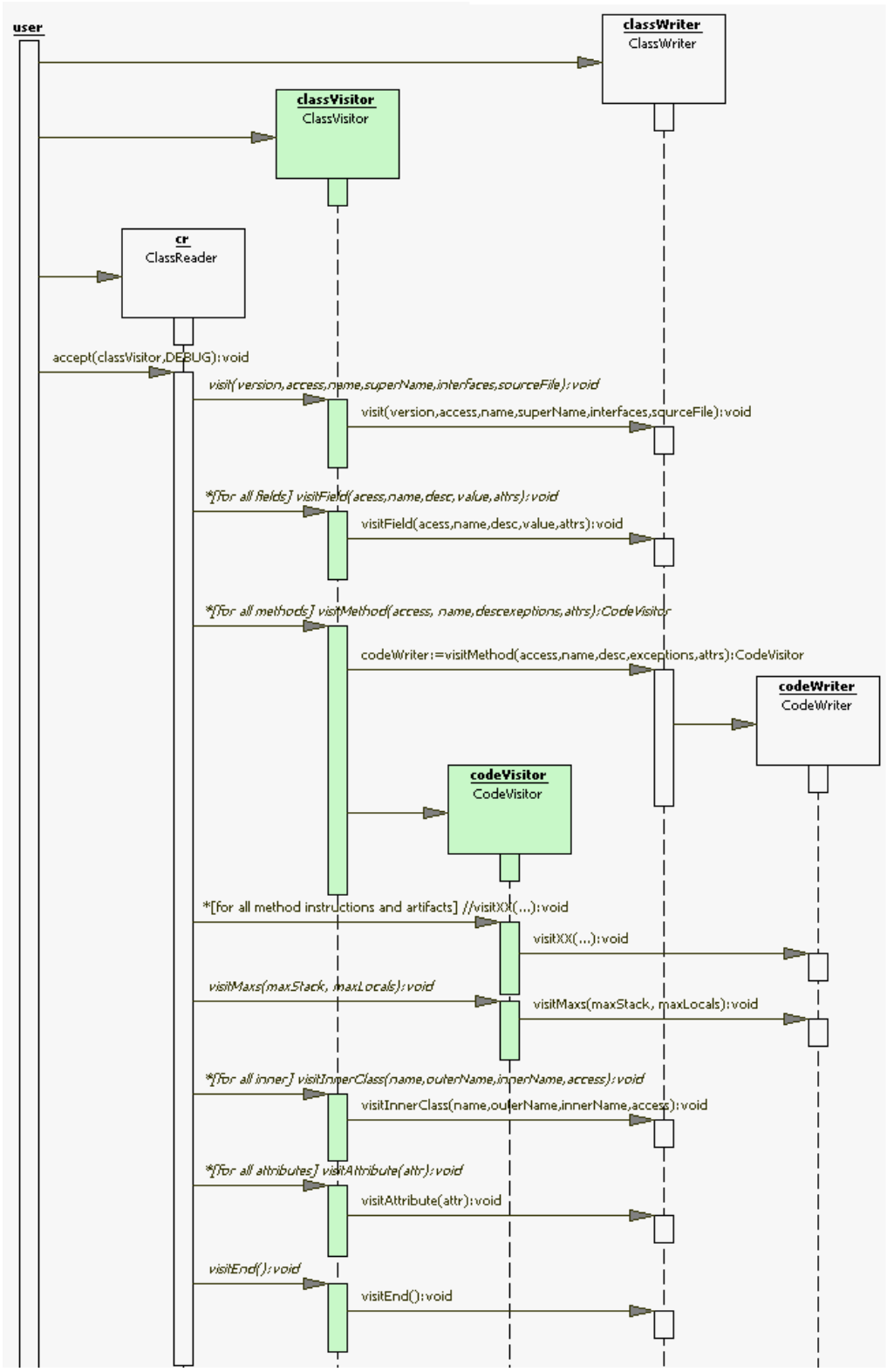


Figura 2.5: Diagrama de secuencias para una transformación típica de bytecode

# Capítulo 3

# Algoritmo de síntesis de CFSMs

Como mencionamos anteriormente, en este trabajo nos vamos a centrar en programas escritos en JAVA, que se comunican entre sí a través del envío y recepción de mensajes por sockets. Para desarrollo de nuestro algoritmo hacemos las siguientes hipótesis simplificadoras sobre las características de los programas:

1. Cada programa debe poseer una función "Main", que se tomará como punto inicial del algoritmo para generar las CFSMs.

2. Los mensajes que se intercambian deben ser Strings o Enums.

3. La comparación de los mensajes se debe realizar a través de la función "equals" de JAVA.

4. Los programas deben conocer a los otros programas que participan en la comunicación.

5. El envío de mensajes se realiza a través de la clase "PrintWriter" de JAVA.

6. La recepción de mensajes se realiza a través de la clase "BufferedReader" de JAVA.

En un sistema, es muy probable que la comunicación entre los componentes esté distribuida entre distintas clases y funciones dentro de cada componente del sistema. Lo que vamos a hacer entonces, es analizar estas clases y funciones y aplicar una serie de algoritmos con el fin de obtener la CFSM que reporesenta el comportamiento del sistema.

Luego, aplicando el mismo procedimiento a cada componente que participa de la comunicación, se van a obtener todas las CFSMs que son requeridas para la síntetis de la la coreografía global y validar si la comunicación entre los componentes es segura.

## 3.1. Descripción general

En este punto vamos a dar una descripción general de los pasos que aplica nuestro algoritmo cuando recibe las clases que correponden a un componente que interviene en la comunicación.

1. Input: Nuestro algoritmo va a recibir como entrada una clase compilada (archivo con extensión .class), o un conjunto de clases compiladas (archivo con extensión .jar)

2. Utilizando ASM, vamos a analizar las clases que recibimos, y por cada clase, los métodos dentro de ellas, en busca de 2 cosas:

   *a*) Por un lado, vamos a buscar componentes relevantes en la comunicación con otros procesos. Estos componentes pueden ser Sockets, ServerSockets, mensajes, componentes de lecturas de mensajes (BufferedReader), componentes de escritura de mensajes (PrintWriter), entre otros. Estos componentes son importantes porque nos dicen con que otros procesos se está comunicando el proceso que estamos analizando, y cuáles son los mensajes que se están intercambiando. La detección de estos elementos relevantes se hace mediante el uso de autómatas de reconocimiento, que serán detallados en secciones siguientes.

   *b*) Por otro lado, analizamos el flujo de ejecución del programa dentro del método que esta siendo analizado. A partir de este flujo, podemos deducir el orden en que son intercambiados los mensajes. Parte del flujo analizado puede tener invocación a otros métodos que también tienen que ser analizados. En dicho caso, se marca esta invocación como una referencia a un flujo de ejecución. Se describe detalladamente en sección 3.2.

   Como postcondición de este paso, se tiene por cada método de cada clase un grafo que modela el flujo de ejecución, en el cual solo se mantienen las instrucciones que son relevantes para la comunicación por sockets.

3. Como siguiente paso del algoritmo se procede a unificar los componentes relevantes encontrados. Estas unificaciones pueden ser, entre otras

   - sockets con sus elementos de escritura
   - sockets con sus elementos de lectura
   - mensajes con el socket que lo envió
   - mensajes con el socket que lo tiene que recibir

   Al finalizar este paso, podemos deducir los mensajes que son intercambiados con cada componente de la comunicación, ya que ahora conocemos el origen y destino de cada mensaje.

4. Como siguiente paso del algoritmo, se procede a generar un grafo de transiciones, cuyas transiciones modelan la relación entre los mensajes intercambiados entre el proceso que esta siendo analizado, y los demás procesos. Se analizan el flujo de ejecución y componentes encontrados por cada método, y se aplican las siguientes acciones:

   - Si el método no tiene componentes que representen mensajes, ni referencia a otros métodos, entonces se descarta.
   - Si el método tiene una referencia a otro método, entonces se aplica el mismo paso recursivamente en el método referenciado. Cada método analizado es marcado, para no volver a ser analizado por otra referencia.
   - Por cada componente descubierto que represente mensajes, se procede a armar un grafo de transiciones, donde las transiciones representan los mensajes y los estados representan los sockets que envían y reciben ese mensaje. Para modelar la relación entre las transiciones, se toma en cuenta el flujo de ejecución asociado al método.

5. Se busca por cada clase, el método "Main", que tomamos como punto inicial para generar la CFSM. A partir del método "Main", se analizan las referencias a otros métodos, y se unifica el grafo de transiciones de los métodos referenciados, al grafo de transiciones del método "Main", aplicando primero recursión sobre los métodos referenciados.

   Al final de este paso, queda un grafo de transiciones que parte del método "Main" y contiene todas las transiciones que existen a partir de el. Dichas transiciones representan los mensajes que fueron intercambiados con los otros procesos.

6. Como parte final del algoritmo, se recorre el grafo generado y se procede a transformar dicho grafo en el modelo de CFSM que recibe *GGB*.

Finalmente, se aplica este algoritmo sobre cada componente que participa en la comunicación, y luego se toman todos los CFSM generados y se arma un único archivo que es el parámetro de entrada de *GGB* (ver figura 3.1).

## 3.2. Detección de componentes relevantes al problema

Para poder construir el CFSM buscado, lo que vamos a hacer es analizar cada .class en busca de métodos, y por cada método vamos a construir un grafo que contenga los elementos importantes que son utilizados a la hora de comunicarse con los otros componentes. Luego, tomando como base el grafo asociado al método main

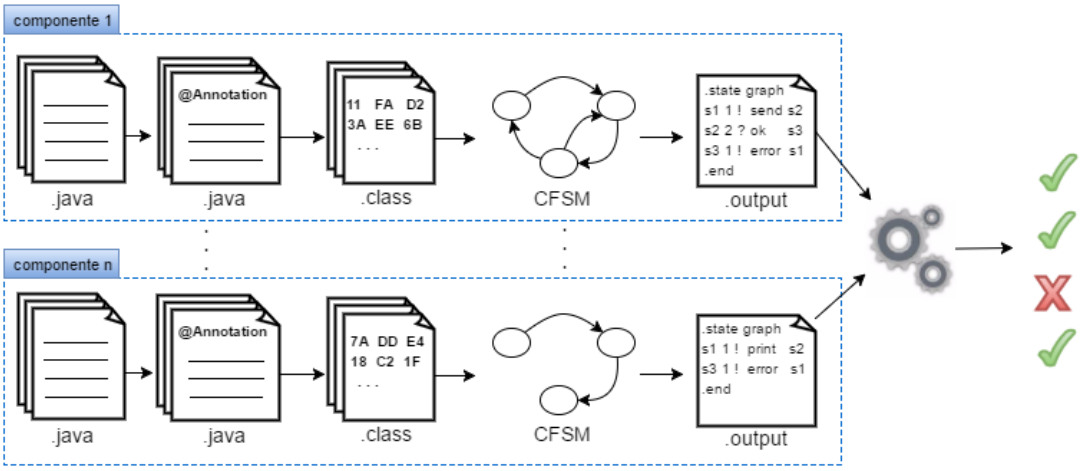


Figura 3.1: Idea general

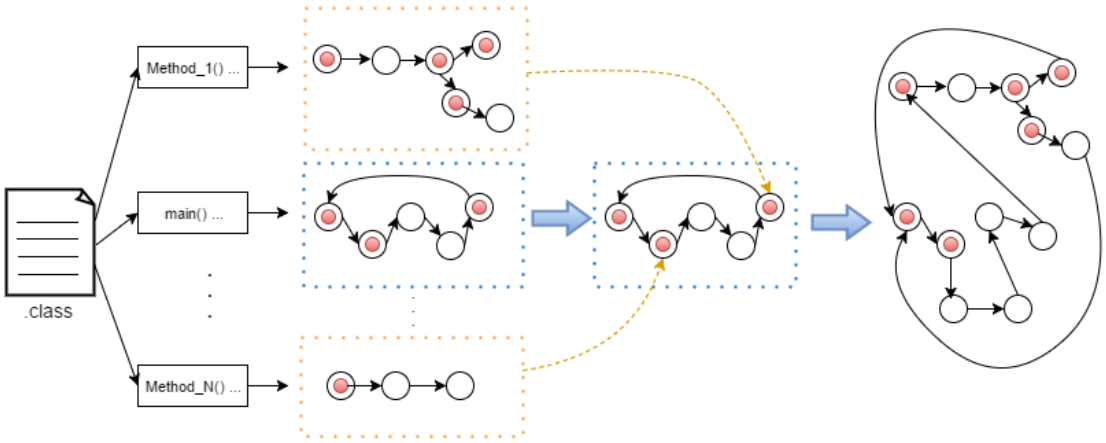


Figura 3.2: Construcción de flujos

de cada clase, se aplicarán una serie de algoritmos con el fin de construir el CFSM final asociado a un componente.

Tomando como base la clase ClassReader de ASM, analizaremos las clases y sus métodos, generando por cada uno de ellos un flujo de control que contiene los elementos relevantes utilizados en la comunicación entre los componentes. Luego, se procederá a unificar los flujos tomando como base el método main, como se muestra en el diagrama 3.2. Finalmente, se realizará un análisis sobre el flujo final, para obtener el FSM buscado y a partir del mismo, generar el input esperado por la herramienta.

En las siguientes secciones se describen los detalles de la implementación del análisis de código. El diagrama de secuencias de la figura 3.3) ilustra el comportamiento global del algoritmo. Las colaboraciones en la figura se explican en las siguientes secciones.

Como se observa en la figura 3.3), cuando se crea una instancia de la clase ClassReader perteneciente al framework ASM, se indica cual es la clase que se va a analizar (la clase es pasada en el parámetro "classPath" del método "buildFSM").

## 3.3. ClassAnalyzer

Cuando se ejecuta la función **accept(analyzer)** de la clase ClassReader expuesta por el framework ASM, se procede a visitar la clase que fue pasada como parámetro al momento de crear la instancia. La función *accept* analiza el bytecode de la clase y notifica mediante eventos los elementos que va encontrando, como métodos, anotaciones, etc. El receptor de estos eventos será la clase **ClassAnalyzer** definida en este trabajo, que hereda la clase ClassVisitor expuesta por el framework ASM, y que conocerá, además de otras cosas, como nombre de la clase y clases internas de la clase, cuales son los métodos que posee la clase a visitar.

Los eventos que ClassAnalyzer sabe manejar son:

1. visit: este evento es disparado cuando la clase ClassReader de ASM visita el header de una clase. De este evento, ClassAnalyzer obtiene el nombre de la clase visitada, y lo guarda de manera local para tener referencia que la clase fue visitada.

2. visitInnerClass: este evento es disparado cuando la clase ClassReader de ASM detecta que la clase visitada contiene clases internas. Sirve para conocer que la clase contiene clases internas que también tienen que ser analizadas. Luego que la clase principal es analizada, se corre el algoritmo para construir el CFMS de cada una de las clases internas encontradas por ClassReader, que fueron guardadas por ClassAnalyzer. En muchos casos, estas clases internas pueden representar Threads, que se utilizan para procesar interacciones con distintos sockets.

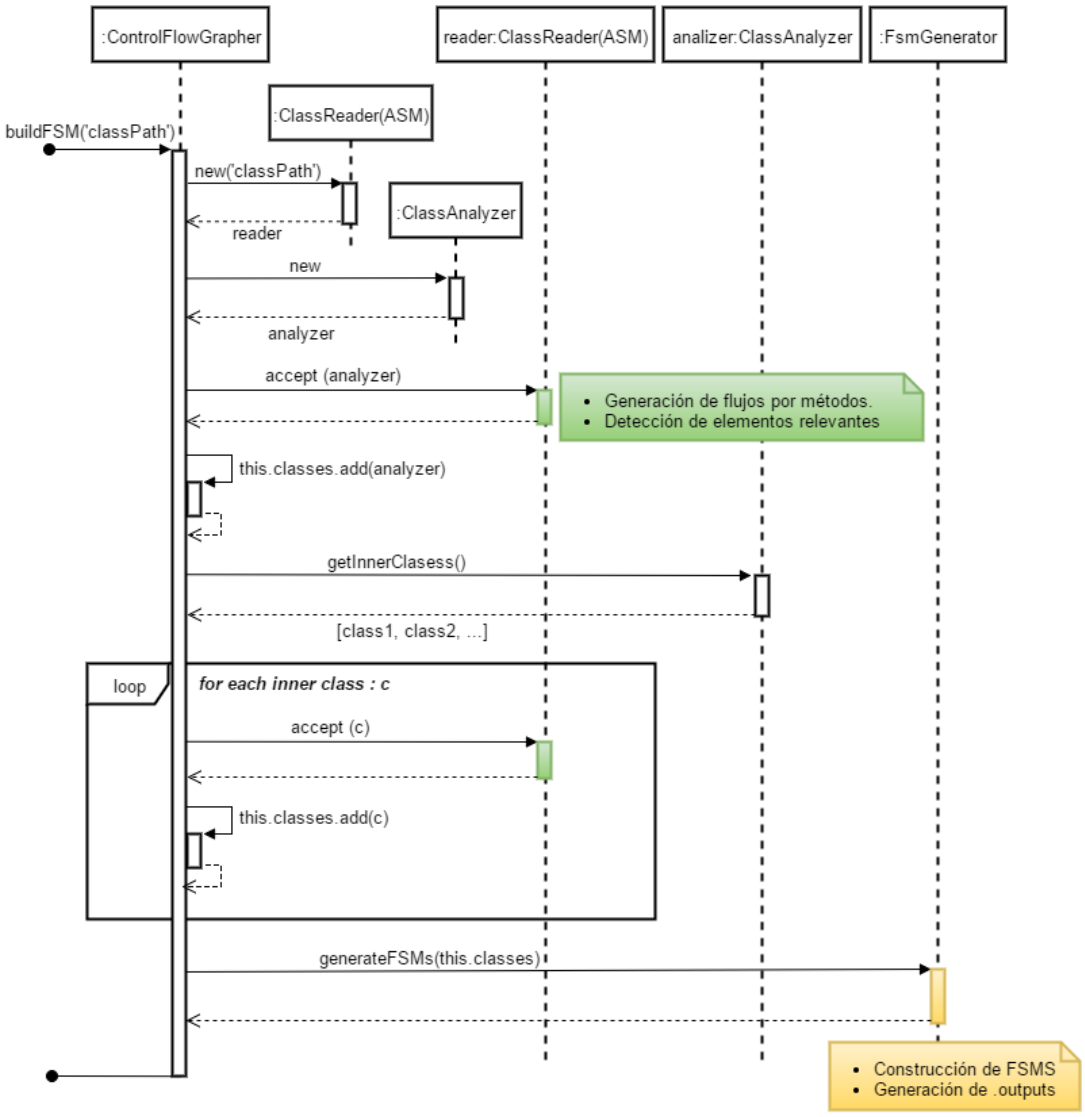


Figura 3.3: Construcción de flujos−Diagrama de secuencias

3. visitMethod: este evento es disparado cada vez que el Reader de ASM detecta un método dentro de la clase. Nuestra clase guarda de manera local todos los métodos que fueron encontrados por el Reader.

El diagrama de secuencias en figura 3.4 describe lo mencionado.

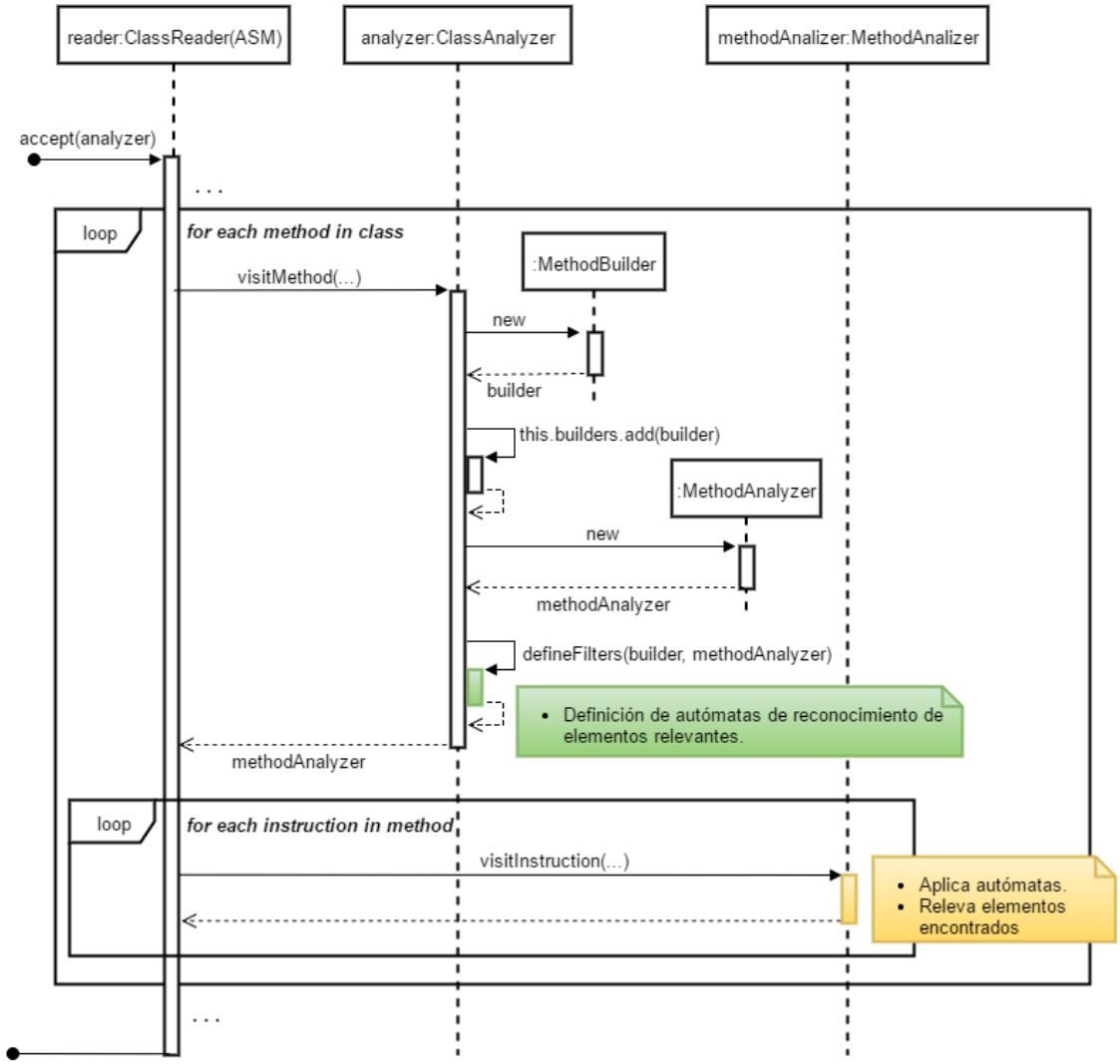


Figura 3.4: Class analyzer−Diagrama de secuencias

Cada vez que ASM visita un método de la clase analizada, se dispara un evento que notifica que un método fue encontrado. Ante este evento, ASM espera como respuesta una clase que se encargue de manejar los eventos asociados a un método. Dicha clase debe heredar de la clase MethodVisitor expuesta por ASM.

La clase MethodAnalyzer definida en este trabajo hereda de MethodVisitor, y una instancia de ella se devuelve como respuesta ante el evento visitMethod.

## 3.4. MethodAnalyzer

Esta clase es la encargada de manejar los eventos lanzados por el framework cuando visita un método. Los eventos relevantes para esta clase son los eventos asociados a las instrucciones que posee un método. Ante un evento enviado por el ASM relacionado a la visita de una instrucción, esta clase decodifica la instrucción y notifica a las clases encargadas de descubrir elementos importantes en la comunicación entre los componentes, que una nueva instrucción fue visitada.

Para ello, se implementa el patrón de diseño Observer, donde los observadores son, en este caso, filtros que se encargan de descubrir los elementos relevantes. Estos filtros son registrados por la clase ClassAnalizer cada vez que se instancia la clase MethodAnalyzer.

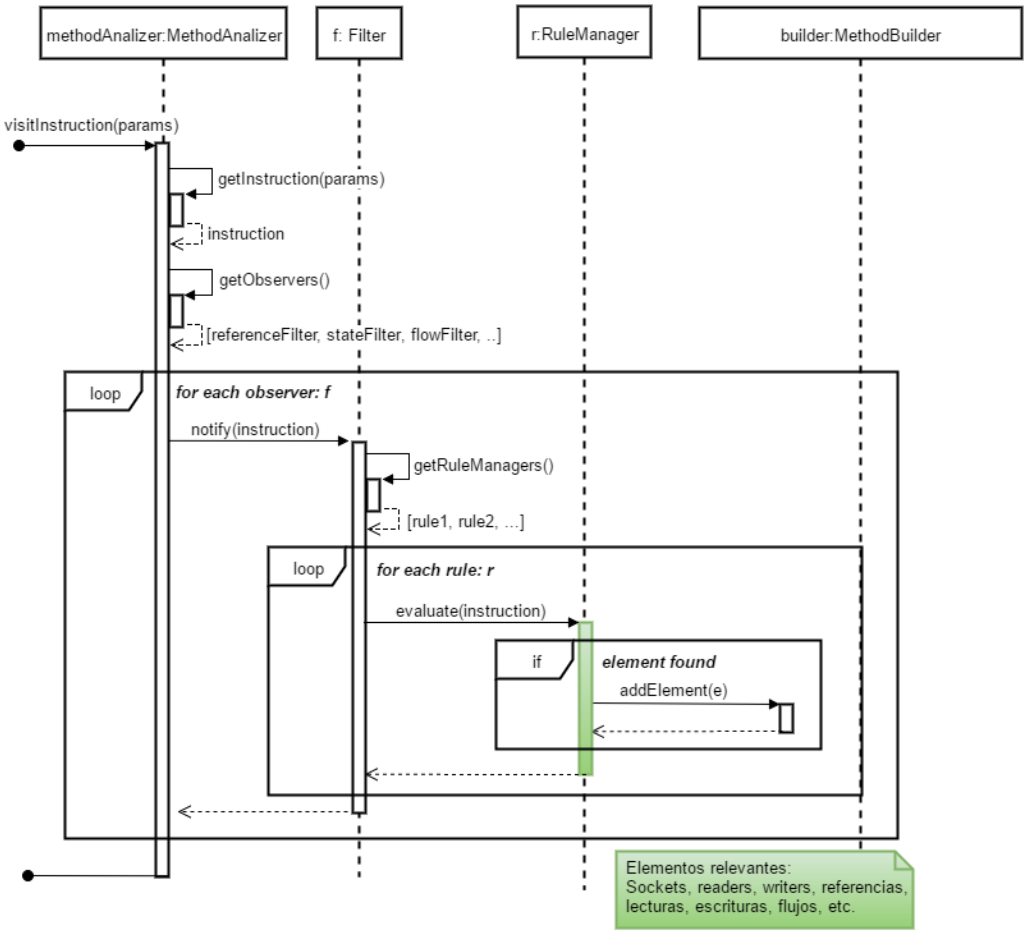


Figura 3.5: Method analyzer−Diagrama de secuencias

## 3.5. Filtros

La función de los filtros es encontrar los elementos que intervienen en la comunicación entre los componentes, y ver cuáles son los mensajes que se intercambian entre los mismos. Además, también deben identificar el orden con el cual los mensajes son intercambiados durante la comunicación.

Los elementos relevantes que analizan estos filtros son:

1. instancias de las clases java/net/Socket y java/net/ServerSocket, para analizar los socket que intervienen.

2. instancias de las clases java/io/BufferedReader y java/io/PrintWriter, para analizar la forma en la cual los sockets se comunican.

3. mensajes que son representados como Strings y que se comparan a través de la función "equals".

Para detectar estos elementos y ver la forma en que interactúan, se analizaran las instrucciones de los métodos y la relación que existe entre las mismas. Dichas instrucciones son instrucciones del código ya compilado, es decir, instrucciones de byteCode. Cada filtro tiene la responsabilidad de descubrir un elemento en particular.


Figura 3.6: Server Socket−Filtro

Un filtro se define como un autómata de reconocimiento cuya función es analizar las instrucciones de un método y definir si estas instrucciones representan un elemento relevante de la comunicación. Por cada componente a descubrir, se define un autómata cuyas transiciones representan la secuencia de instrucciones que deben suceder para que se encuentre una instancia del componente que está analizando. El estado final del autómata es un estado de éxito, lo que indica que se encontró una instancia del elemento que el autómata modela.

Cada filtro analiza la instrucción y valida si esta es válida en su autómata. Si la instrucción forma parte de su autómata, entonces avanza por la transición que modela la instrucción y luego chequea en que estado quedó. Si el autómata llegó a un estado de éxito, entonces se notifica que un nuevo componente fue encontrado por causa de esa instrucción. Si el estado al cual se movió el autómata es un estado intermedio, entonces espera por las próximas instrucciones del método. Si en cambio,

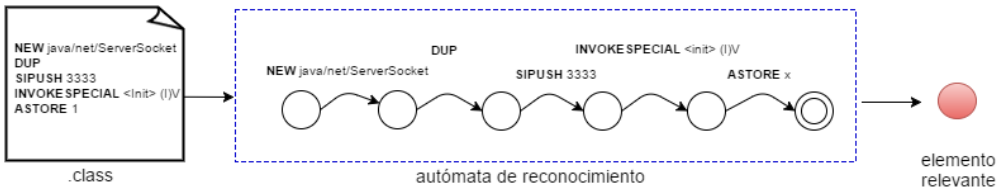


Figura 3.7: Server Socket−Reconocimiento

la instrucción no forma parte del autómata, entonces vuelve al estado inicial, ya que la secuencia de instrucciones no se corresponde con la de su autómata.

De esta manera, los elementos relevantes a la comunicación son descubiertos a medida que las instrucciones son visitadas. Los filtros mencionados no solo se encargan de analizar los componentes relevantes, sino que también analizan los mensajes que se intercambian, los flujos de las instrucciones y otros aspectos que son relevantes a la hora de armar el flujo de cada método.

Cada vez que un elemento relevante es encontrado por un filtro, el mismo es notificado por el filtro a una instancia de la clase MethodBuilder, que es la clase encargada de mantener todos los elementos encontrados, y es la clase principal utilizada a la hora de aplicar los algoritmos con el fin de obtener los CFSM buscados.

### 3.5.1. Tipos de filtros

Existen distintos tipos de filtros encargados de encontrar distintos elementos relevantes. Cada filtro se encarga de encontrar uno en particular. A continuación se describen los filtros usados en este trabajo y la función que cumple cada uno de ellos:

#### Input State Filter

Este filtro es el encargado de encontrar instancias de la clase java/net/ServerSocket utilizados en las funciones. Cada vez que un ServerSocket es encontrado, se crea un instancia de la clase “InputState”, que modela dicha instancia encontrada.

Se modelan dos autómatas para realizar el descubrimiento, donde cada uno controla una forma distinta de asignación de las instancias creadas.

*Creación de ServerSockets y asociación a field*:

Se crea un automata que se encarga de descubrir la creación de sockets, cuya instancia es asociada a un field de la clase. En este caso, el filtro guarda el nombre de la variable de la clase a la cual se asigna la instancia creada.

*Creación de ServerSockets y asociación a variable*:

Se crea un automata que se encarga de descubrir la creación de sockets, cuya instancia es asociada a una variable locale del método. En este caso, el filtro guarda el índice de la variable a la cual se asigna la instancia creada.

### Output State Filter

Este filtro es el encargado de encontrar instancias de la clase java/net/Socket utilizados en las funciones. Cada vez que un Socket es encontrado, se crea un instancia de la clase "OutputState", que modela dicha instancia encontrada.

Se modelan tres autómatas para realizar el descubrimiento, donde cada uno controla una forma distinta de encontrar dichas instancias.

*Creación de Socket y asociación a field*:

Se crea un autómata que se encarga de descubrir la creación de sockets, cuya instancia es asociada a un field de la clase. En este caso, el filtro guarda el nombre de la variable de la clase a la cual se asigna la instancia creada.

*Socket pasado como parámetro*:

Se crea un automata que se encarga de descubrir el pasaje de sockets por parámetro.

*Creación de Socket y asociación a variable*:

Se crea un autómata que se encarga de descubrir la creación de sockets, cuya instancia es asociada a una variable locale del método. En este caso, el filtro guarda el índice de la variable a la cual se asigna la instancia creada.

### Reference Filter

Este filtro es el encargado de encontrar las referencias a métodos que se hacen dentro del código. Cada vez que una referencia es encontrada, se crea una instancia de la clase "ReferenceState", que modela dicha referencia. Se guardan las siguientes propiedades:

- name: representa el nombre del método que es referenciado.
- owner: representa la clase donde se encuentra el método referenciado.

### Local Variable Filter

Este filtro es el encargado de encontrar los nombres de las variables locales utilizadas en un método. Cuando un objeto es asociado a una variable local, dentro de las instrucciones de byteCode, estas se representan con dos pasos: primero, se realiza la asignación del objeto a un índice, en donde el índice representa el índice de la variable utilizada, dentro de las variables del método. Por otro lado, se recorrer las variables existentes en el método, y por cada variable, se imprime el índice y el nombre de la misma.

Las variables que son relevantes a nuestro problema son las variables que representan:

- variables de clases java/net/Socket
- variables de clases java/net/ServerSocket

- variables de clases java/io/PrintWriter
- variables de clases java/io/BufferedReader

### Transition Send Filter

Cuando en un método se ejecuta la función "println" de la clase java/io/PrintWriter, se agrega una transición, cuya información es definida de la siguiente manera:

- message: contiene el String enviado en la función println.
- origen: se crea un OwnerState que representa al socket que envió el mensaje.
- destino: se crea un OutputState que representa al socket al cual se le envía el mensaje. Sobre este state se realizan una serie de operaciones: por un lado, se marca el state como virtual, que quiere decir que este state no es real, sino que representa a algún state que tuvo que haber sido descubierto por algún filtro. Por otro lado, se crea y asocia a dicho socket un Writer, que va a servir como base para realizar la búsqueda del state original. Este Writer contiene el nombre o índice de la variable asociada al writer al cual se aplicó el "println".

### Block Filter

Una de las funcionalidades de ASM es analizar los bloques de instrucciones que existen en un método. Este filtro se encarga de descubrir los bloques existentes, y formar una relación entre los mismos. Cada vez que un nuevo bloque es descubierto, se marca dicho bloque como el bloque actual del método. Si existía un bloque actual anterior, entonces se define el nuevo bloque como sucesor del mismo, con una relación de precedencia.

Un bloque es representado por un label.

### Flow Filter

Un flow filter es un autómata que se encarga de reconocer las instrucciones que provocan una bifurcación en el flujo del programa. Las instrucciones que analizan en este trabajo son:

- instrucción *return*: representa a la instrucción return. En este caso, la acción que se realiza es eliminar el bloque actual, si es que existe.
- instrucción *if*: representa la instrucción if. En este caso, ASM define a qué bloque se realiza el salto. La acción que se realiza entonces es agregar el bloque al que se realiza el salto como sucesor del bloque actual, si es que existe.
- instrucción *goto*: representa la instrucción goto. En este caso, la acción que se realiza es eliminar el bloque actual, si es que existe.

### Socket Name Annotation Filter

Este filtro es el encargado de encontrar las anotaciones que modelan los nombres de los componentes que participan en la comunicación.

Luego de visitar las instrucciones de un método, la clase MethodBuilder asociado al método contiene todos elementos relevantes que fueron encontrados. Además, se genera un grafo con el flujo del mismo, en donde los nodos mantienen referencia a los elementos encontrados. El siguiente diagrama describe como es el grafo resultante luego de analizar un método.

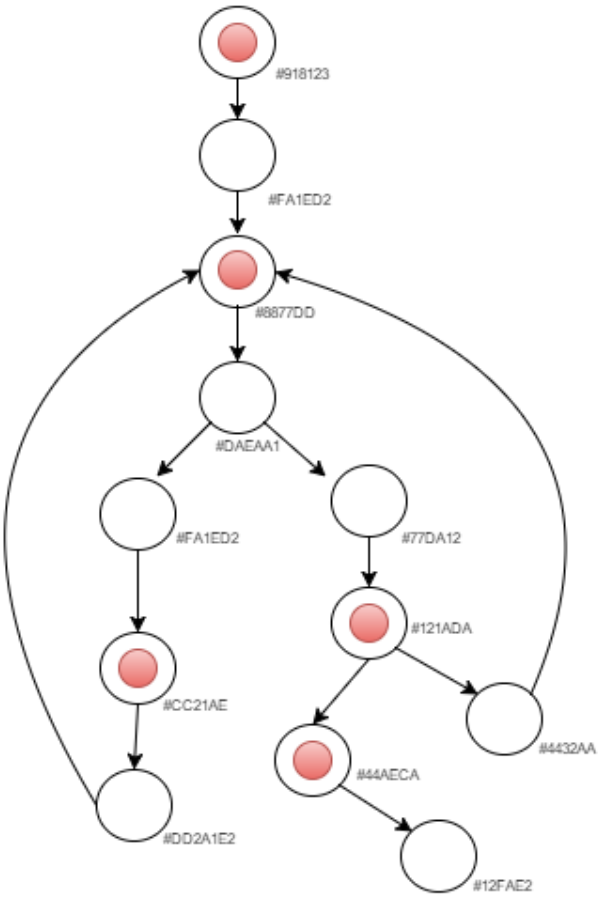


Figura 3.8: Componentes detectados

## 3.6. Method Builder

Esta clase se encarga de mantener todos los componentes relevantes que fueron encontrados por cada uno de los filtros. Estos son:

1. InputStates: mantiene una lista de las instancias de ServerSocket que fueron encontradas.

2. HandlerInputs: mantiene una lista de las clases encargadas de procesar los mensajes que se intercambian con un ServerSocket, que fueron encontradas.

3. OutputStates: mantiene una lista de las instancias de Socket que fueron encontradas.

4. Readers: mantiene una lista de todos los BufferedReader que fueron encontrados.

5. Writers: mantiene una lista de todos los PrintWriter que fueron encontrados.

6. Transiciones: mantiene una lista de todas las transiciones que fueron encontradas.

7. ReferenceStates: mantiene una lista de todas las referencias a métodos que fueron encontrados.

8. LocalVariables: mantiene una lista de todas las variables locales que fueron encontradas.

9. Blocks: mantiene una lista de todos los bloques que fueron encontrados.

10. Flow: mantiene una lista de todas las instrucciones de flujo que fueron encontradas.

11. SocketNames: mantiene una lista de todas las anotaciones que contienen un nombre de socket que fueron encontradas.

12. IntanceReaders: mantiene una lista de todas las instrucciones que leen un String de un Reader a través de una instrucción readLine.

Una de las principales funciones que tiene la clase MethodBuilder es mantener una relación entre los bloques de código que se fueron descubriendo. Esta relación es importante porque en base a ella, se va a definir el flujo de instrucciones del método.

Existen dos maneras de agregar un bloque de código:

- una forma es por el descubrimiento de un nuevo bloque por alguno de los filtros. Cada vez que un nuevo bloque es descubierto, se pone el nuevo bloque como bloque actual del método, y se actualiza el viejo bloque actual, marcando el nuevo bloque descubierto como sucesor del mismo, en el caso que existiese un bloque actual anterior.

- otra forma es por el descubrimiento de un nuevo flujo por alguno de los filtros. Un nuevo flujo puede darse, por ejemplo, por el descubrimiento de una instrucción Jump. En este tipo de instrucciones (las instrucciones de flujo), la instrucción notifica hacia que bloque se realiza el flujo. En este caso, se agrega una relación entre el bloque actual y el bloque de la instrucción de flujo, marcando el bloque de la instrucción como sucesor del bloque actual, si es que el bloque actual existe. En el caso que el bloque de la instrucción de flujo no exista, se crea un nuevo bloque que represente el mismo.

Los bloques están representados por un Label, que es una forma de distinguirlos.

## 3.7. Estados y transiciones

En este trabajo, un estado modela el uso de un socket, y las transiciones entre estos estados modelan los mensajes que estos intercambian. Debido a que parte de este trabajo es descubrir cuales son los sockets que participan y cómo se relacionan, se modelan distintos tipos de estados que son utilizados para distintos fines y en distintos momentos del proceso de construcción de los CFSM. Al final del algoritmo, los estados relevantes son aquellos que representan a sockets o estados finales.

Los distintos estados son modelados con clases diferentes, donde todas las clases heredan de una clase abstracta denominada State. La clase State mantiene la siguiente información:

1. outgoingTransitions: representa las transiciones que salen desde este estado. Dicho en otras palabras, representa las transiciones cuyo estado origen es el estado en cuestión.

2. incomingTransitions: representa las transiciones que entran al estado. Dicho en otras palabras, representa las transiciones cuyo estado destino es el estado en cuestión.

3. socketName: representa al nombre del socket contenido en el estado.

Los distintos estados utilizados son:

- InputState: representa a las instancias de la clase java/net/ServerSocket que son utilizadas en las clases analizadas. Esta clase mantiene la siguiente información:

    1. varName: representa el nombre de la variable en donde se almacena la instancia de la clase java/net/ServerSocket que fue creada. Este atributo es asignado cuando la instancia del socket es almacenada en una variable de la clase.

2. varIndex: representa el índice de la variable en donde se almacena la instancia de la clase java/net/ServerSocket que fue creada. Este atributo es asignado cuando la instancia del socket es almacenada en una variable local al método

- OutputState: representa a las instancias de la clase java/net/Sockets que son utilizadas en las clases analizadas . Esta clase mantiene la siguiente información:

  1. varName: representa el nombre de la variable en donde se almacena la instancia de la clase java/net/ServerSocket que fue creada, o el nombre de un parámetro. Este atributo es asignado cuando la instancia del socket es almacenada en una variable de la clase, o cual el socket haya sido pasado como parametro de una función.
  2. varIndex: representa el índice de la variable en donde se almacena la instancia de la clase java/net/Socket que fue creada. Este atributo es asignado cuando la instancia del socket es almacenada en una variable local al m]étodo.

- OwnerState: representa a un estado desde el cual se realiza una transición de escritura hacia un socket. Cuando se detecta que se envía un mensaje hacia algún socket a través del uso de la función "println", se crea una transición a la cual se le indica que el estado origen es una instancia de esta clase. Estos owner state son reemplazados en el proceso de armado de las CFSM, por el correspondiente socket que es el actor principal del método.

- FinalState: representa un estado final. Estos estados son construidos durante el proceso de armado de flujo de los métodos. En el momento que se analizan los bloques para armar el grafo, si en bloque no tiene sucesores, entonces se crea un nuevo estado final y se indica que el bloque representa este estado final.

- ReferenceState: indica una referencia a un método. Estos estados son construidos por un filtro, cuando se analizan los componentes relevantes a nuestro problema. En el proceso de armado de las CFSM, se reemplazan estos estados por los correspondientes grafos de los métodos a los cuales se hace referencia.

- TemporalState: representa estados temporales. Los estados temporales son creados cuando se crean las transiciones. Por defecto, al crear una transición, el estado origen y destino son estados temporales. Estos estados son eliminados en el proceso de armado de las CFSM, ya que son reemplazados por los sockets que participan en la transición.

## 3.8. Transiciones

Una transicion consta de:

- un estado origen.
- un estado destino.
- un mensaje.
- un índice de variable.

En este trabajo se modelan dos tipos de transiciones: por un lado están las transiciones que son descubiertas por los filtros. Estas transiciones representan envíos de mensajes hacia sockets, o recepción de mensajes de los mismos. Se generan cuando:

1. se ejecuta el método "println" de la clase java/io/PrintWriter. Esto determina que se está enviando un mensaje a un socket, que es el que está asociado a dicho PrintWriter.

2. Se realiza la comparación de una variable contra un String a través del método "equals", y la variable fue asignada con el resultado de la instrucción "readLine" de la clase java/io/BufferedReader. Esto determina que el método está esperando un mensaje desde un socket.

Por otro lado, existen otro tipo de transiciones denominadas Transiciones lambda, que son generadas en diferentes casos, a saber:

- se genera una transición lambda cuando se ejecuta una instrucción de lectura sobre una clase BufferedReader, invocando al método "readLine". Esta transición se utiliza para indicar que en este momento, se está recibiendo un mensaje desde un socket, pero que no se sabe que mensaje es, ya que para saber el mensaje se tiene que analizar que valores son interpretados por el String al cual fue asociado el valor del resultado de la instrucción "readLine".

- se generan transiciones lambda para unir bloques que no tienen transiciones, en el proceso de construcción del grafo de transiciones

## 3.9. Unificación de componentes

Sobre los flujos generados y los elementos relevantes encontrados en cada método de las clases analizadas, se aplican algoritmos con el fin de generar el CFSM buscado.

## 3.10. Asociación entre elementos encontrados

Este algoritmo consiste en asociar los sockets encontrados con sus correspondientes elementos de lectura y escritura. Cada método analizado tiene asociado elementos relevantes que fueron descubiertos, además de un grafo que modela el flujo de información del mismo.

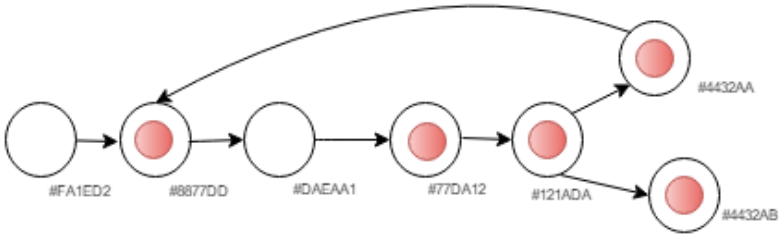


Figura 3.9: Flujo de ejecución y componentes

Parte de los elementos encontrados se corresponden a sockets, serverSockets, y elementos de lectura y escritura asociados a los mismos.

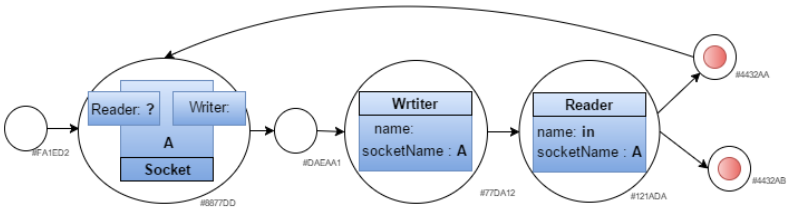


Figura 3.10: Componentes en el flujo

Cada socket encontrado, contiene o el nombre de la variable a la cual hace referencia (si el socket es creado como un field o es pasado como parámetro a una función), o el índice de la variable, si el socket fue creado como una variable local.

Por cada socket que no tiene nombre de variable asociado, se busca y asocia el nombre de la variable que le corresponde. Para ello, se filtran las variables (del método al que pertenece el socket) de tipo Socket a partir del indice de la variable que contiene el socket. Una vez que se encuentra la variable, se obtiene el nombre y se asocia el mismo al socket. A partir de este momento, todos los sockets contienen el nombre de variable asociado.

Por cada clase analizada y por cada método encontrado dentro de la clases, se obtienen todos los elementos de escritura (Writer).

Cada writer contiene o, el nombre de la variable del socket asociado, o el indice de la variable. Luego, por cada writer, se obtiene el socket al cual corresponde, filtrando los mismos por índice o nombre de variable, y se realiza la asociación entre ellos.

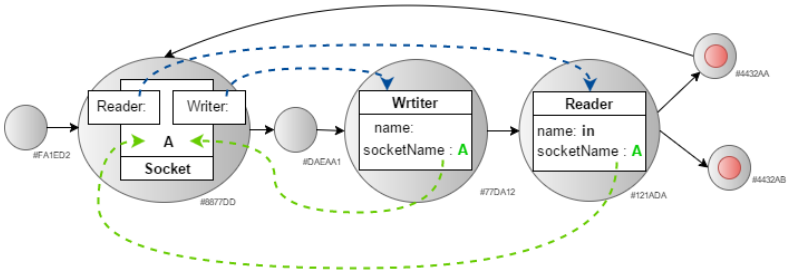


Figura 3.11: Unificación de componentes

El siguiente paso consiste en asociar los ServerSockets encontrados con los Handler (Threads) encargados de manejar los mensajes que este recibe.

Cada ServerSocket encontrado, contiene o el nombre de la variable a la cual hace referencia (si el ServerSocket es creado como un field o es pasado como parámetro a una función), o el índice de la variable, si el socket fue creado como una variable local.

Por cada ServerSocket que no tiene nombre de variable asociado, se busca y asocia el nombre de la variable que le corresponde. Para ello, se filtran las variables (del método al que pertenece el ServerSocket) de tipo ServerSocket a partir del índice de la variable que contiene el ServerSocket. Una vez que se encuentra la variable, se obtiene el nombre y se asocia el mismo al ServerSocket.

A partir de este momento, todos los ServerSockets contienen el nombre de variable asociado. Luego, por cada clase analizada y por cada método encontrado, se obtienen todos los ServerSockets. Por cada uno de ellos, se busca si posee un handler asociado, filtrando los Handlers por nombre o índice de variable. Si se encuentra, se procede a realizar la asociación.

## 3.11. Grafo de transiciones

En esta parte del algoritmo, nos vamos a centrar en las transiciones encontradas y las referencias a métodos dentro de los métodos. Tomando como base el flujo asociado a un método, y filtrando solo los elementos relevantes mencionados, vamos a generar un grafo que modele la relación entre las transiciones encontradas.

La idea es recorrer los bloques e ir armando transiciones entre los mismos, tomando como referencia las transiciones que existen en los mismos. Para ello, se realiza un algoritmo recursivo, cuya recursión se realiza entre los bloques del método, tomando como caso base el primer bloque del método.

Algoritmo: recibe 2 parámetros: el bloque actual y la última transición encontrada. La primera vez que se invoca al algoritmo, el bloque actual es el primer bloque del método y la última transición encontrada es nula.

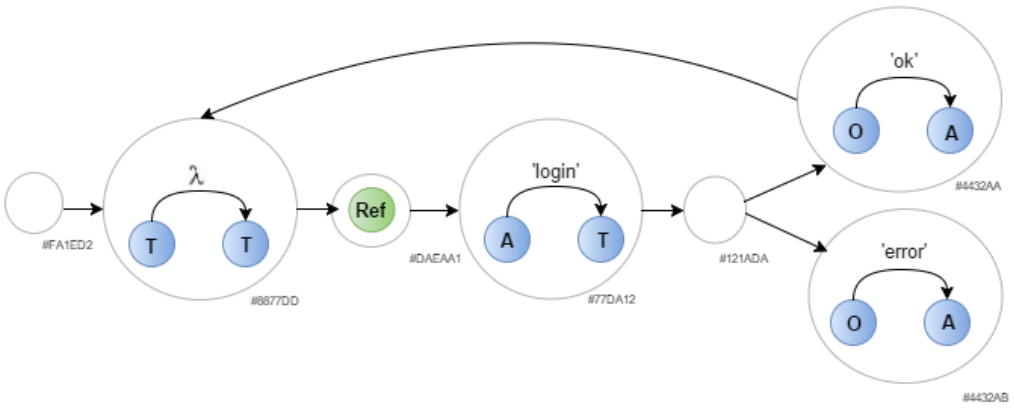


Figura 3.12: Relación entre transiciones

Si el bloque ya fue visitado, entonces obtengo la primer transición del bloque y se genera una transición lambda entre el estado destino de la última transición encontrada y el estado origen de la primer transición del bloque. Si la última transición encontrada es nula, entonces no se realiza ninguna acción.

Si el bloque no fue visitado, entonces se realiza las siguientes acciones:

1. obtengo la transición incluida en el bloque (si es que existe).
2. marco el bloque como visitado, para evitar ciclos.
3. Si el bloque no tiene sucesores, entonces se agrega una transición lambda desde el estado final de la transición del bloque hacia un estado final.
4. Si el bloque tiene sucesores, entonces por cada sucesor, se realiza las siguientes acciones:

   *a*) si el bloque tiene estados de referencia a métodos, se agrega una transición lambda entre la última transición encontrada y el estado de referencia.

   *b*) si el bloque tiene una transición, entonces aplico recursión con el sucesor del bloque y transición perteneciente al bloque. La recursión va a generar un grafo de las transiciones que existe desde el sucesor del bloque. Una vez que la recursión finaliza, se agrega una transición lambda entre la última transición encontrada antes del bloque, y la transición del bloque.

   *c*) si el bloque no tiene una transición, entonces aplico recursión con el sucesor del bloque y la última transición encontrada.

Finalmente, descartamos los bloques y solo nos quedamos con las transiciones y estados existentes en los mismos. Como resultado de este algoritmo, se obtiene un grafo cuyas transiciones son:

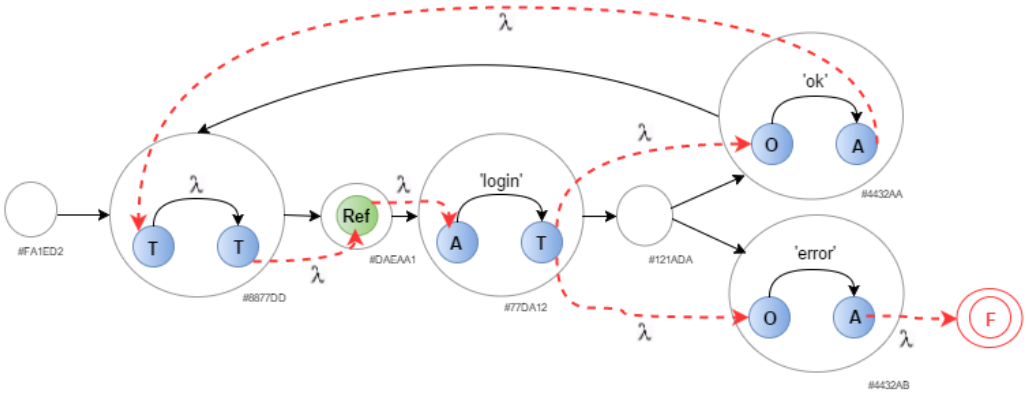


Figura 3.13: Unificación de transiciones

1. transiciones pertenecientes a los bloques
2. transiciones lambda entra las transiciones encontradas en el bloque.
3. transiciones lambda a estados de referencias. Estos estados de referencia representan métodos que están dentro de la clase del método analizado, o en otras clases.

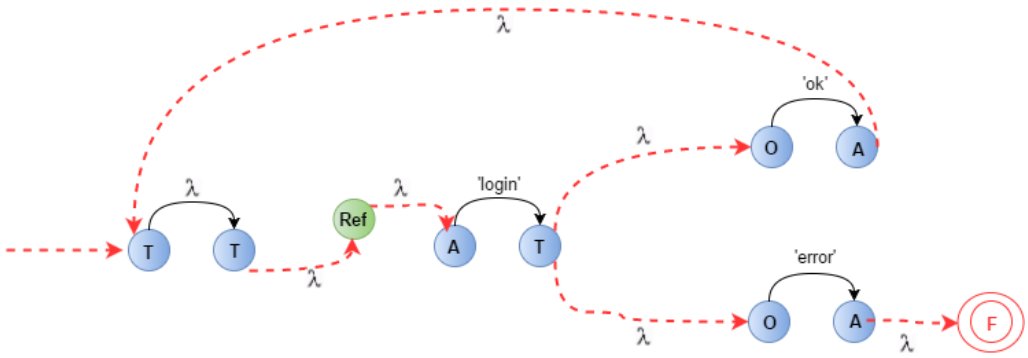


Figura 3.14: Resultado de la unificación

## 3.12. Construcción de Fsm

Una vez generado el grafo con las transiciones, se va a procesar el mismo con el fin de generar el grafo final que va a ser pasado a la herramienta con el fin de validar el protocolo. Para lograr esto, se aplica un algoritmo, que se detalla a continuación:

El primer paso del algoritmo consiste en encontrar dentro de cada clase analizada, el método main, ya que se toma este método como entrada para generar la máquina de estados. Una vez que se obtiene el método Main, se toma el grafo correspondiente

al método y se aplican los siguientes pasos del algoritmo, para dejar el grafo en el estado definitivo que estamos buscando.

### 3.12.1. Completar estados de referencia

En esta parte del algoritmo se recorren los estados del grafo en busca de estados de referencia. Por cada estado de referencia encontrado, se busca el grafo correspondiente a dicho estado y se procede a reemplazar el estado por el grafo. Para realizar la búsqueda del grafo asociado al estado de referencia, se toma el "owner" del estado (recordemos que cada estado de referencia posee una propiedad "owner" que representa el método al cual hace referencia), y en base a este, se busca entre todas las clases analizadas un método cuyo nombre sea igual al "owner" del estado, y se obtiene el grafo asociado.

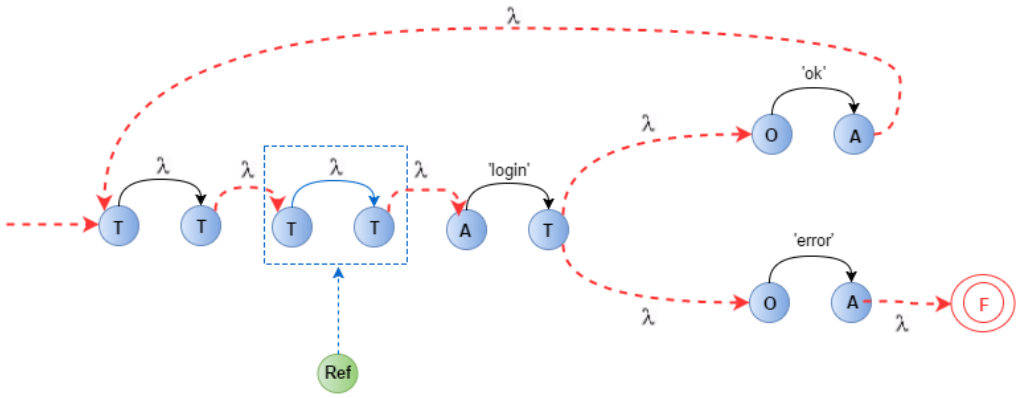


Figura 3.15: Completar referencias

Si no se encuentra el grafo asociado, entonces se elimina el estado de referencia. Para realizar el remplazo del estado por el grafo, se realizan las siguientes acciones:

En primer lugar, se procede a apuntar las transiciones entrantes al estado de referencia al estado inicial del grafo. Procedimiento:

1. obtener el estado inicial del grafo referenciado.

2. obtener las transiciones del grafo en cuestión cuyo estado destino es el estado de referencia.

3. por cada transición encontrada, poner estado destino el estado inicial del grafo referenciado

En segundo lugar, se procede a poner como origen de las transiciones salientes del estado de referencia los estados finales del grafo referenciado. Procedimiento:

1. obtener los estados finales del grafo referenciado.

2. obtener las transiciones del grafo en cuestión cuyo estado origen es el estado de referencia.

3. por cada estado final y por cada transición saliente, se crea una copia de la transición, cuyo estado origen es el estado final.

En tercer lugar, se procede a eliminar el estado de referencia, ya que fue reemplazado por el grafo. Referencias circulares: En caso que existan referencias circulares, el algoritmo lanza una excepción y no se puede obtener el grafo.

### 3.12.2. Remover estados temporales

Este paso del algoritmo se encarga de remover los estados temporales. Para ello, primero se deben localizar todos los estados temporales existentes en el grafo.

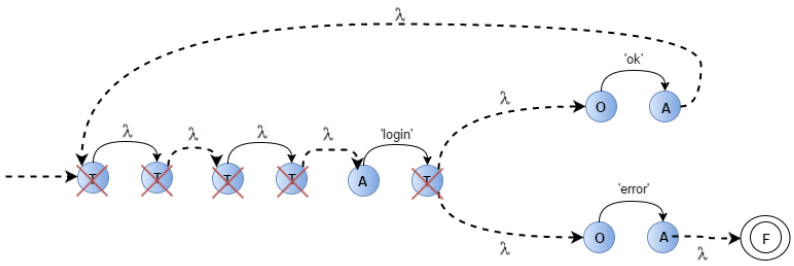


Figura 3.16: Eliminación de estados temporales

Todos los estados temporales tienen transiciones lambda salientes y/o entrantes. Se aplica un algoritmo recursivo sobre el grafo. Por cada estado, si el estado es un estado temporal, se procede a eliminarlo. Cuando se elimina, se realizan 2 cosas:

1. se remueven las transiciones que apuntan a mismo estado.

2. todas las transiciones que llegan a este estado y se apuntan a los destinos de este nodo.

Luego, se hace recursión sobre los sucesores del estado. Finalmente, el grafo queda libre de estados temporales, quedando solo estados que representan a sockets intervinientes en la comunicación.

### 3.12.3. Remover transiciones lambda

Este proceso se encarga de remover las transiciones lambda existentes en el grafo. Para ello, primero debemos localizar todas las transiciones lambda existentes en el grafo.

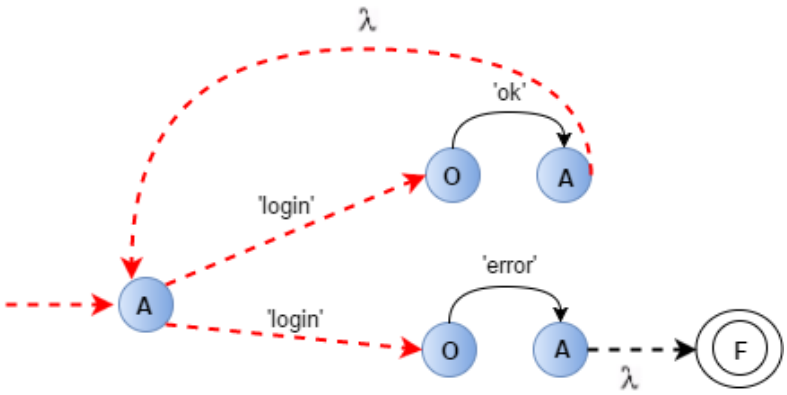


Figura 3.17: Grafo sin estados temporales

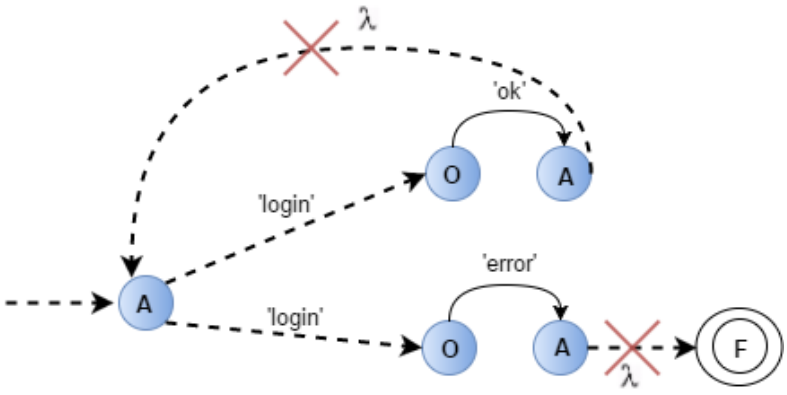


Figura 3.18: Eliminación de transiciones lambda

Es un algoritmo recursivo sobre los estados del grafo, empezando por el estado inicial del grafo. Por cada estado, se obtienen todas las transiciones lambda salientes del mismo. Por cada una de ella, se procede a eliminarla, realizando las siguientes acciones:

1. se remueve la transición
2. por cada transición entrante al estado, se apunta dicha transición al siguiente estado de la transición lambda que se acaba de eliminar.

Luego, se aplica revisión sobre los sucesores del estado actual. Finalmente, el grafo queda libre de transiciones lambdas.

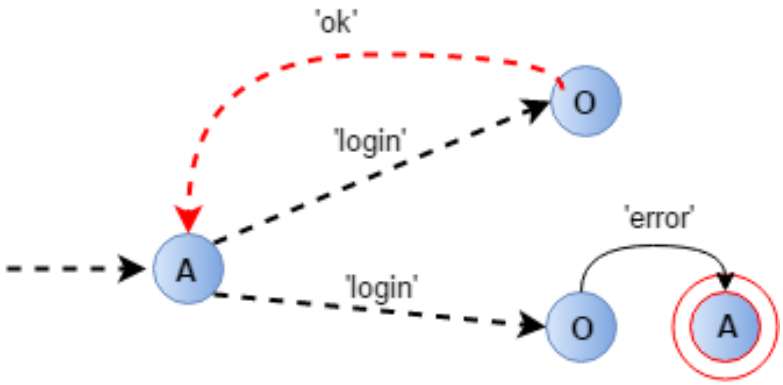


Figura 3.19: Grafo sin transiciones lambda

### 3.12.4. Generar el CFSM

Una vez que se remueven los estados temporales y las transiciones lambda, en el grafo quedan los estados y transiciones definitivos. En este paso, se recorre el grafo, enumerando los estados y las transiciones, y se genera por último un CFSM, que es una clase que contiene una lista de estados y una lista de transiciones.

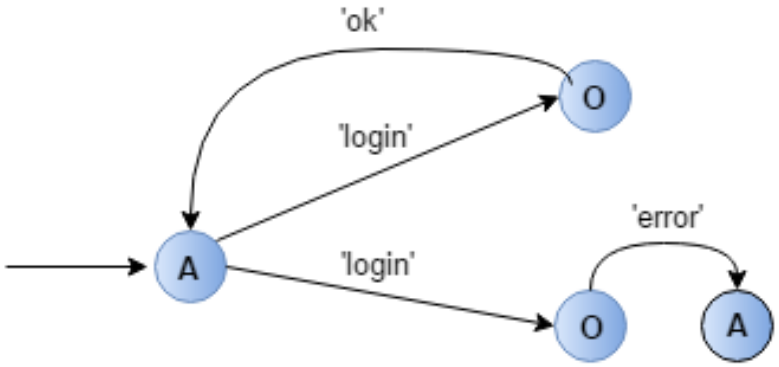


Figura 3.20: Grafo final

# Capítulo 4

# Uso de la herramienta

En esta sección, ilustraremos algunos ejemplos que hemos estudiado, y analizaremos el resultado obtenido. Con el fin de aumentar la comprensión de los ejemplos, obviaremos algunos detalles de implementación, como el manejo de excepciones y manejo de conexiones.

Sobre cada ejemplo a estudiar, realizaremos los siguientes pasos:

1. Daremos una descripción de los servicios involucrados en la comunicación, detallando el código de los programas, del cual se generará el .class que tomamos como base para realizar en análisis.

2. Corremos el algoritmo detallado en este trabajo, para generar las CFSM, que utilizaremos como input para la herramienta.

3. Ejecutamos la herramienta tomando como entrada las CFSM generadas, ilustramos los gráficos generados y analizaremos el resultado obtenido.

4. Por último, realizaremos cambios sobre las implementaciones de los servicios, introduciendo errores, para que, luego de volver a correr la herramienta, analizar como se comporta con estos cambios realizados.

Los siguientes ejemplos irán aumentando en orden de complejidad, ilustrando en cada uno de ellos distintas problemáticas y soluciones sobre los mismos.

## 4.1. Caso de estudio 1

Empezaremos con un sistema simple, en donde se modela un sistema compuesto por una aplicación Cliente y una aplicación Servidor, que se comunican a través de sockets.

El flujo de ejecución comienza cuando la aplicación Cliente realiza una petición de *login* al Servidor, el cual devuelve la respuesta de login exitoso. Luego de que el

cliente realice las operaciones que necesita, realiza una petición de *logout* al Servidor, el cual finaliza la conexión.

### Paso 1

En este ejemplo, la aplicación Cliente es representada por la máquina 0, y la aplicación Servidor es representada por la máquina 1.

A continuación, se muestra el código de las aplicaciones:

```
1  public class Cliente {
2
3     @SocketName(socketName="1", variableName="serverEndpoint")
4     public static void main(String[] args) {
5        Socket serverEndpoint = new Socket("localhost", 3331);
6
7        PrintWriter out = getWriter(serverEndpoint);
8        BufferedReader in = getReader(serverEndpoint);
9
10       // send 'login' message to server
11       out.println("login");
12
13       String serverResponse = in.readLine();
14
15       if (serverResponse.equals("loginSuccess")){
16          // process logic ...
17          out.println("logout");
18       }
19
20       serverEndpoint.close();
21    }
22
23    public static PrintWriter getWriter(Socket socket){
24       return new PrintWriter(socket.getOutputStream(), true);
25    }
26
27    public static BufferedReader getReader(Socket socket){
28       return new BufferedReader(new
             InputStreamReader(socket.getInputStream()));
29    }
30 }
```

Figura 4.1: Aplicación Cliente.

```
1  public class Server {
2
3      @SocketNames(names = {
4          @SocketName(socketName = "0", variableName = "clientEndpoint"),
5          @SocketName(socketName = "1", variableName = "serverEndpoint") })
6      public static void main(String[] args) {
7          System.out.println("init server...");
8          ServerSocket serverEndpoint = new ServerSocket(3331);
9          while (true) {
10             Socket clientEndpoint = serverEndpoint.accept();
11
12             BufferedReader in = getReader(clientEndpoint);
13             PrintWriter out = getWriter(clientEndpoint);
14
15             String inputLine = in.readLine();
16
17             if (inputLine.equals("login")) {
18                 out.println("loginSuccess");
19
20                 String request = in.readLine();
21                 if (request.equals("logout")) {
22                     // process logout ...
23                 }
24             }
25
26             clientEndpoint.close();
27         }
28
29         serverEndpoint.close();
30     }
31
32     public static PrintWriter getWriter(Socket socket){
33         return new PrintWriter(socket.getOutputStream(), true);
34     }
35
36     public static BufferedReader getReader(Socket socket){
37         return new BufferedReader(new
               InputStreamReader(socket.getInputStream()));
38     }
39 }
```

Figura 4.2: Aplicación Servidor.

### Paso 2

Tomando como base los ".class" correspondientes a estas clases, ejecutando nuestro algoritmo para obtener la representación textual del sistema, que luego será utilizada como input por GGB, obtenemos

```
.outputs
.state graph
q1 1 ! login q2
q2 1 ? loginSuccess q3
q3 1 ! logout q4
.marking q1
.end

.outputs
.state graph
q1 0 ? login q2
q2 0 ! loginSuccess q3
q3 0 ? logout q4
.marking q1
.end
```

Figura 4.3: Output generado.

### Paso 3

Luego, al correr GGB, genera el global graph, indicando que la comunicación entre estos componentes es segura. En la figura 4.4 podemos ver las FSMs que GGB generó a partir de las especificaciones textuales brindadas. En la figura 4.5 podemos ver el "global graph" generado.

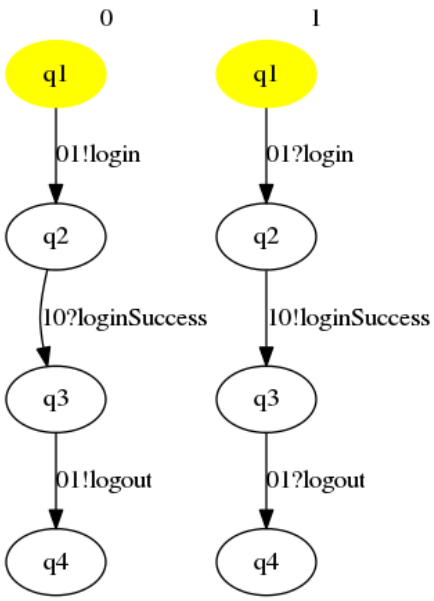


Figura 4.4: FSMs generadas

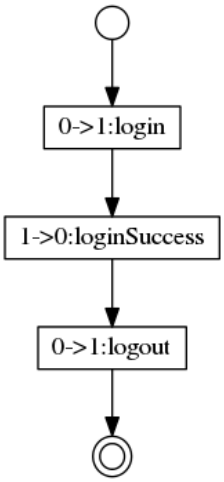


Figura 4.5: Global Graph

**Paso 4**

Supongamos ahora que el equipo encargado de la aplicación Servidor realiza modificaciones sobre la implementación, pudiendo ahora ante el pedido de un "Login", devolver distintas respuestas, como login exitoso y login fallido.

El código con la modificación realizada es el siguiente:

```
1  public class Server {
2
3      ...
4      public static void main(String[] args) {
5          ...
6          while (true) {
7              ...
8              if (inputLine.equals("login")){
9                  boolean loginSuccess = processLogin();
10                 if (loginSuccess){
11                     out.println("loginSuccess");
12
13                     String request = in.readLine();
14                     if (request.equals("logout")){
15                         // process logout ...
16                     }
17                 } else {
18                     out.println("loginError");
19                 }
20             }
21
22             clientEndpoint.close();
23         }
24         serverEndpoint.close();
25     }
26
27     private static boolean processLogin() {
28         // some logic ..
29         return false;
30     }
31 }
```

Figura 4.6: Aplicación Servidor con nuevo mensaje

Con el código modificado, volvemos a correr el algoritmo y GGB. En este caso, GGB indica que hay un error, y que la comunicación no es segura. El error se debe a que el nuevo mensaje "loginError" agregado en la aplicación Servidor no es recepcionado por la aplicación Cliente, generando así un *Orphan message* (se envía un mensaje que no es recepcionado por ningún componente de la comunicación).

En la figura 4.7 podemos ver las FSMs que GGB generó a partir del código con la modificación. En la figura 4.8 podemos ver una ejemplo del error lanzado por GGB.

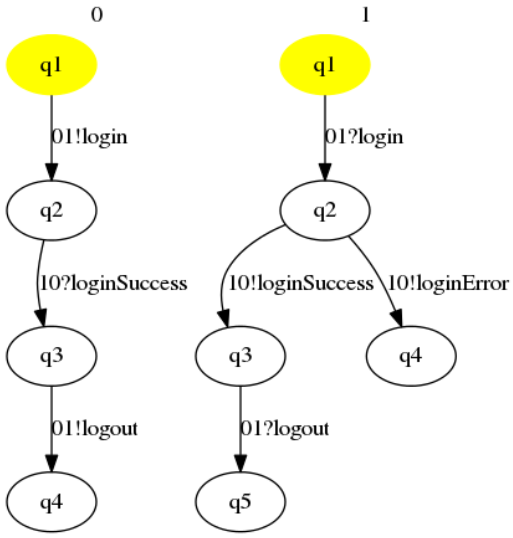


Figura 4.7: FSMs generadas

```
gmc: check start
gmc:    Parsing CFSMs file...
gmc:    factor 0
gmc:    dir "outputs"
gmc:    TS:     (nodes 4, transitions 3)
gmc:    0-bounded TS:   (nodes 4, transitions 3)
gmc:    Branching representability:     [Bp 1 "q2",Bp 1 "q4"]
gmc:    Branching Property:             []

chosyn: ...minimising projections
chosyn: {---Testing machine 0 for language equivalence---}
chosyn: Is machine 0 representable? true
chosyn: {---Testing machine 1 for language equivalence---}
chosyn: Is machine 1 representable? false
chosyn: {---Is the system Language-equivalence Representable (part (i))? False---}
chosyn: {---Calling petrify...---}
chosyn: {---Generating Global Graph...---}
gg: global graph synthesis
chosyn: All done:
```

Figura 4.8: Error por consola. Indica que la máquina 1 no es representable. La transición del estado “q2” al estado “q4” no es recepcionada.

Para solucionar el problema, el equipo Server da aviso al equipo Cliente de la no recepción del mensaje, el cual modifica su código agregando la recepción del mismo.

```
1  public class Cliente {
2      ...
3      // send 'login' message to server
4      out.println("login");
5
6      String serverResponse = in.readLine();
7
8      if (serverResponse.equals("loginSuccess")){
9          // process logic ...
10         out.println("logout");
11     }
12
13     if (serverResponse.equals("loginError")){
14     // do something ...
15     }
16 }
```

Figura 4.9: Aplicación Cliente: recepción de nuevo mensaje.

Finalmente, con el código de la aplicación Cliente modificado, volvemos a correr el algoritmo y GGB, el cual vuelve a indicar que la comunicación es segura.

En la figura 4.10 podemos ver las FSMs que GGB generó a partir del código con la modificación. En la figura 4.11 podemos ver el global graph generado.

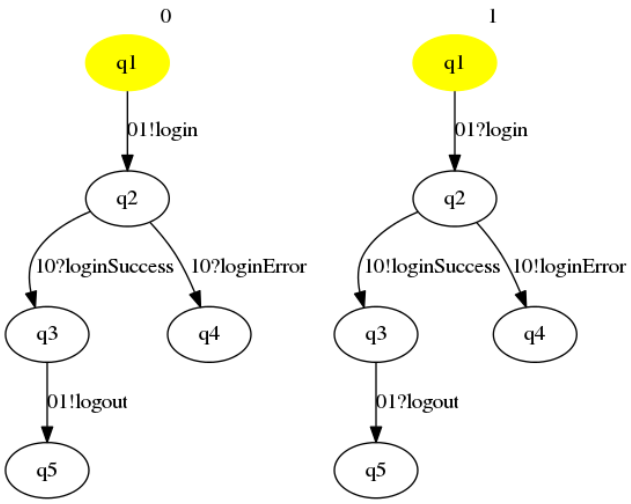


Figura 4.10: FSMs generadas con aplicación cliente modificada.

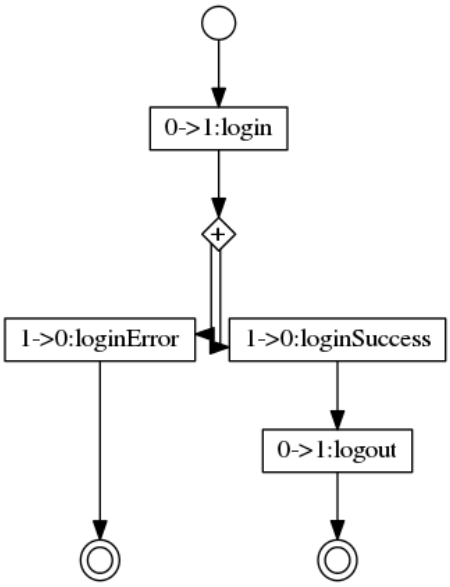


Figura 4.11: Global graph final.

En los siguientes ejemplos se asume que cada clase ilustrada contiene las funciones "getWriter(Socket socket)" y "getReader(Socket socket)" denotadas en este ejemplo, las cuales devuelven respectivamente el PrintWriter y BufferedReader asociados al socket pasado como parámetro en cada función, utilizados para enviar y recibir mensajes a partir del socket.

Por otro lado, también se asume que cada clase contiene las anotaciones "@SocketNames" necesarias para identificar a cada uno de los componentes con los que la clase se comunica.

## 4.2. Caso de estudio 2

Como segundo ejemplo, se modela un sistema de otorgamiento de becas a los alumnos de una universidad. En este sistema, los alumnos completan exámenes, y en base al promedio de los mismos, la universidad analiza si le otorga una beca al alumno o no.

Los servicios involucrados en el sistema son los siguientes:

- **Estudiante**: completa un examen. Se representa con la máquina 0.
- **Analizador**: evalúa el examen y le asigna una nota. Se representa con la máquina 1.
- **Estadísticas**: analiza el promedio de notas del alumno y se fija si le corresponde una beca o no. Se representa con la máquina 2.
- **Histórico**: mantiene todos los exámenes que un alumno rindió. Se representa con la máquina 3.
- **Becas**: maneja las becas asignadas a los alumnos. Se representa con la máquina 4.

El flujo de ejecución comienza cuando el servicio *Estudiante* envía un mensaje al servicio *Analizador* indicando que se completó un examen. El servicio *Analizador* evalúa el examen y le pone una nota. Una vez que tiene el resultado, envía el examen junto con su resultado al servicio *Estadísticas*, el cual se encarga de validar si corresponde o no una beca al estudiante. Para realizar dicho análisis, obtiene el histórico de notas del servicio *Histórico*. Finalmente, luego de realizar el análisis, si decide que le corresponde una beca, el servicio *Estadísticas* llama al servicio *Becas* indicando que se tiene que generar una beca para el alumno.

**Paso 1**

A continuación, se muestra el código relacionado a las aplicaciones:

```java
public class Estudiante {

    @SocketNames(names = {...})
    public static void main(String[] args) {
        Socket analyzerSocket = new Socket("localhost", 3331);

        PrintWriter out = getWriter(analyzerSocket);
        BufferedReader in = getReader(analyzerSocket);

        // some logic...

        out.println("complete");

        String serverResponse = in.readLine();

        if (serverResponse.equals("done")){
            System.out.println("process ok");
        }

        if (serverResponse.equals("scholarshipCreated")){
            System.out.println("new scholarship created...");
        }

        analyzerSocket.close();
    }
}
```

Figura 4.12: Aplicación Estudiante.

```java
public class Analizador {

    private static Socket statisticSocket;
    private static PrintWriter statisticOut;
    private static BufferedReader statisticIn;

    @SocketNames(names = {...})
    public static void main(String[] args) {
        System.out.println("init analyzer server...");

        statisticSocket = new Socket("localhost", 3332);
        statisticOut = getWriter(statisticSocket);
        statisticIn = getReader(statisticSocket);

        ServerSocket serverSocket = new ServerSocket(3331);
        while (true){
            Socket studentSocket = serverSocket.accept();
            BufferedReader studentIn = getReader(studentSocket);
            PrintWriter studentOut = getWriter(studentSocket);

            String inputLine = studentIn.readLine();

            if (inputLine.equals("complete")){
                evaluateTest();
                statisticOut.println("addTest");
                String statisticResponse = statisticIn.readLine();

                if (statisticResponse.equals("withoutScholarship")){
                    studentOut.println("done");
                }

                if (statisticResponse.equals("withScholarship")){
                    studentOut.println("scholarshipCreated");
                }
            }

            statisticSocket.close();
            studentSocket.close();
        }

        serverSocket.close();
    }

    private static void evaluateTest() {
        // some logic ...
    }
}
```

Figura 4.13: Aplicación Analizador.

```
1 public class Historico {
2
3    @SocketNames(names = {...})
4    public static void main(String[] args) {
5       System.out.println("init historical server...");
6
7       ServerSocket serverSocket = new ServerSocket(3333);
8       while (true){
9          Socket statisticSocket = serverSocket.accept();
10         BufferedReader statisticIn = getReader(statisticSocket);
11         PrintWriter statisticOut = getWriter(statisticSocket);
12
13         String inputLine = statisticIn.readLine();
14
15         if (inputLine.equals("getHistorical")){
16            // some logic...
17            statisticOut.println("historicalOk");
18         }
19
20         statisticSocket.close();
21      }
22
23      serverSocket.close();
24   }
25 }
```

Figura 4.14: Aplicación Histórico.

```
1  public class Estadistica {
2
3     @SocketNames(names = {...})
4     public static void main(String[] args) {
5        System.out.println("init analyzer server...");
6        Socket historicalSocket = new Socket("localhost", 3333);
7        PrintWriter historicalOut = getWriter(historicalSocket);
8        BufferedReader historicalIn = getReader(historicalSocket);
9
10       Socket scholarshipSocket = new Socket("localhost", 3334);
11       PrintWriter scholarshipOut = getWriter(scholarshipSocket);
12       BufferedReader scholarshipIn = getReader(scholarshipSocket);
13
14       ServerSocket serverSocket = new ServerSocket(3332);
15       while (true){
16          Socket analyzerSocket = serverSocket.accept();
17          BufferedReader analyzerIn = getReader(analyzerSocket);
18          PrintWriter analyzerOut = getWriter(analyzerSocket);
19          String inputLine = analyzerIn.readLine();
20          if (inputLine.equals("addTest")){
21             historicalOut.println("getHistorical");
22             String historicalResponse = historicalIn.readLine();
23             if (historicalResponse.equals("historicalOk")){
24                boolean createScholarship = analyzeStatistics();
25                if (createScholarship){
26                   scholarshipOut.println("createScholarship");
27                   String schoResponse = scholarshipIn.readLine();
28                   if (schoResponse.equals("scholarshipCreated")){
29                      analyzerOut.println("withScholarship");
30                   }
31                } else {
32                   analyzerOut.println("withoutScholarship");
33                }
34             }
35          }
36          analizerSocket.close();
37       }
38       serverSocket.close();
39    }
40
41    private static boolean analyzeStatistics() {
42       // some logic ...
43       return false;
44    }
45 }
```

Figura 4.15: Aplicación Estadística.

```java
public class Becas {

    @SocketNames(names = {...})
    public static void main(String[] args) {
        System.out.println("init scholarship server...");

        ServerSocket serverSocket = new ServerSocket(3334);

        while (true){
            Socket statisticSocket = serverSocket.accept();
            BufferedReader statisticIn = getReader(statisticSocket);
            PrintWriter statisticOut = getWriter(statisticSocket);

            String inputLine = statisticIn.readLine();

            if (inputLine.equals("createScholarship")){
                // some logic...
                statisticOut.println("scholarshipCreated");
            }

            statisticSocket.close();
        }

        serverSocket.close();
    }
}
```

Figura 4.16: Aplicación Beca.

### Paso 2

Tomamos como base los ".class" correspondientes a las clases, y ejecutamos nuestro algoritmo para obtener la representación textual del sistema que luego será utilizada como input por GGB.

### Paso 3

Luego, al correr GGB, genera el global graph, indicando que la comunicación entre estos componentes es segura. En la figura 4.17 podemos ver las FSMs que GGB generó a partir de las especificaciones textuales brindadas. En la figura 4.18 podemos ver el global graph generado.

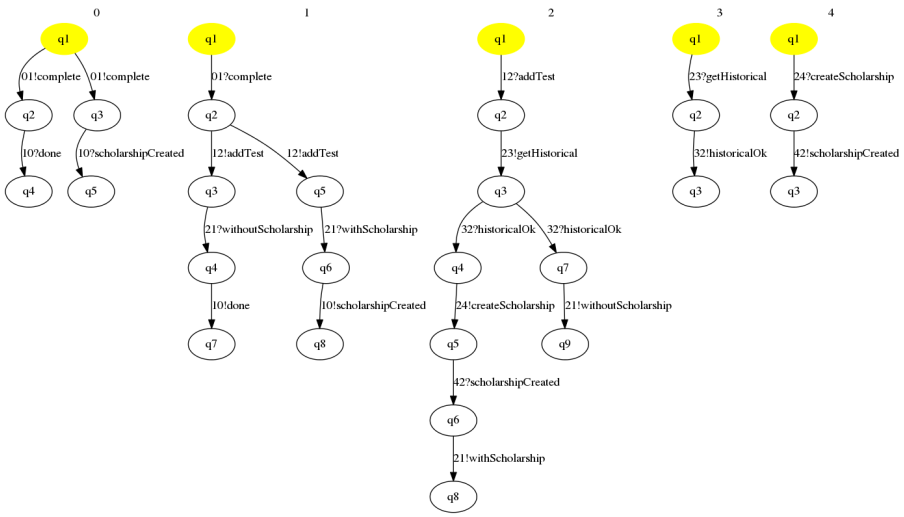


Figura 4.17: FSMs generadas

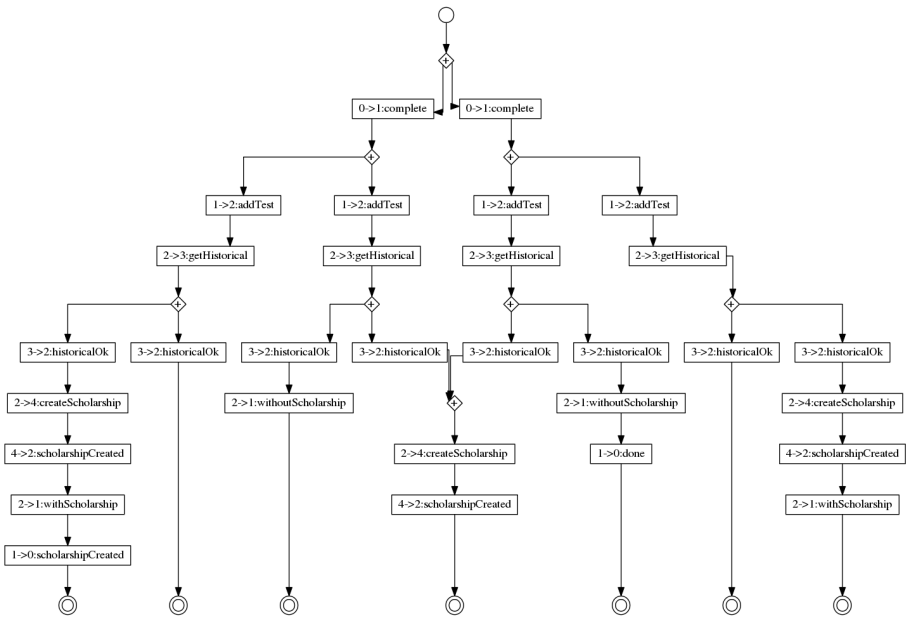


Figura 4.18: Global Graph

### Paso 4

Supongamos que ahora, las becas se tienen que activar para que pueda ser aplicada a un alumno, y esto sucede si se rindieron mas de una cantidad n de exámenes. Entonces, se pide a los equipos encargados de manejar las aplicaciones de *Estadísticas* y *Becas*, que agreguen dicha funcionalidad en el código. Luego de realizar el pedido, el equipo encargado del servicio *Scholarship* agrega la funcionalidad, pero el equipo de *Estadísticas* se olvida de agregar la misma.

El siguiente código es el modificado por el equipo de *Becas*:

```
1  public class Becas {
2  ...
3  public static void main(String[] args) {
4      ...
5      while (true){
6          String inputLine = statisticIn.readLine();
7
8          if (inputLine.equals("createScholarship")){
9              // some logic ...
10             statisticOut.println("scholarshipCreated");
11         }
12
13         if (inputLine.equals(“activateScholarship”)) {
14         // some logic ...
15         }
16     }
17     ...
18 }
19 }
```

Figura 4.19: Modificación código Becas.

Al ejecutar nuevamente GGB con el código modificado, la misma lanzó un error, identificando que existe un estado en donde se espera por un mensaje que no es enviado por ningúna componente de la comunicación, generando así un *unspecified reception configuration*.

En la figura 4.20 podemos ver las FSMs que GGB generó a partir del código con la modificación. En la figura 4.21 podemos ver un ejemplo del error lanzado por GGB.

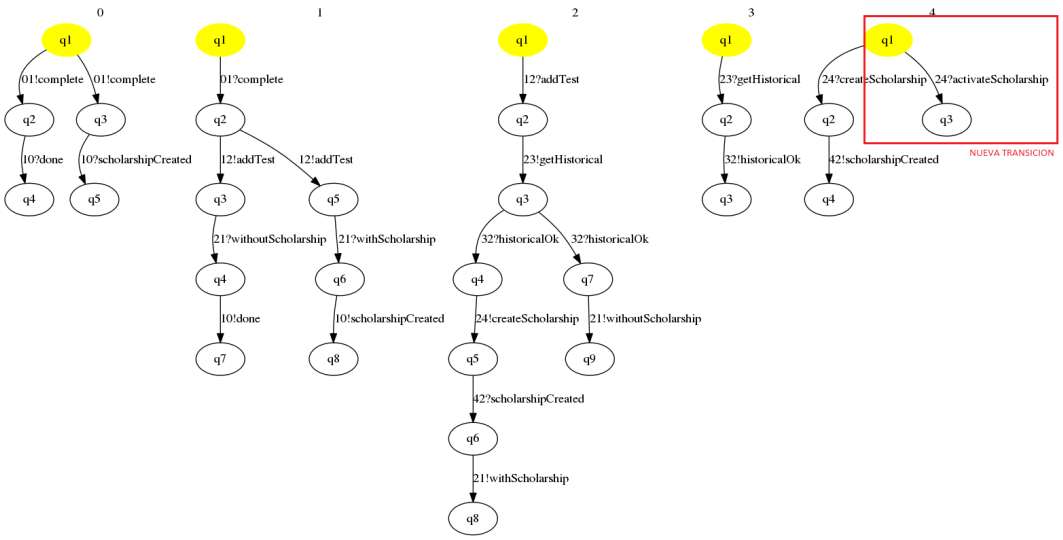


Figura 4.20: FSMs generadas

```
gmc: check start
gmc:    Parsing CFSMs file...
gmc:    factor 0
gmc:    dir "outputs"
gmc:    TS:     (nodes 33, transitions 32)
gmc:    0-bounded TS:   (nodes 33, transitions 32)
gmc:    Branching representability:     [Bp 4 "q1",Bp 4 "q3"]
gmc:    Branching Property:             []

chosyn: ...minimising projections
chosyn: {---Testing machine 0 for language equivalence---}
chosyn: Is machine 0 representable? true
chosyn: {---Testing machine 1 for language equivalence---}
chosyn: Is machine 1 representable? true
chosyn: {---Testing machine 2 for language equivalence---}
chosyn: Is machine 2 representable? true
chosyn: {---Testing machine 3 for language equivalence---}
chosyn: Is machine 3 representable? true
chosyn: {---Testing machine 4 for language equivalence---}
chosyn: Is machine 4 representable? false
chosyn: {---Is the system Language-equivalence Representable (part (i))? False--
-}
chosyn: {---Calling petrify...---}
chosyn: {---Generating Global Graph...---}
gg: global graph synthesis
chosyn: All done:
```

Figura 4.21: Error por consola. Indica que la máquina 1 no es representable. La transición del estado “q1” al estado “q3” nunca es invocada.

Luego, el equipo de *Estadística* agrega el envío del mensaje faltante.

```
1  public class Estadistica {
2      ...
3      public static void main(String[] args) {
4          ...
5          while (true){
6              ...
7
8                  if (historicalResponse.equals("historicalOk")){
9                      boolean createScholarship = analyzeStatistics();
10
11                     if (createScholarship){
12                         scholarshipOut.println("createScholarship");
13                         String scholarshipResponse =
                               scholarshipIn.readLine();
14
15                         if
                               (scholarshipResponse.equals("scholarshipCreated")){
16                             analyzerOut.println("withScholarship");
17                         }
18                     } else {
19                         boolean activateScholarship = mustActivateScholarship();
20                         if (activateScholarship) {
21                         scholarshipOut.println("activateScholarship");
22                         analyzerOut.println("withScholarship");
23                         } else {
24                             analyzerOut.println("withoutScholarship");
25                         }
26                     }
27                 }
28             }
29         }
30         ...
31     }
32     ...
33
34     private static boolean mustActivateScholarship() {
35     // some logic ...
36     return true;
37     }
38 }
```

Figura 4.22: Código Estadísitica: agrega envío de mensaje faltante.

Finalmente, con el código de la aplicación *Estadísticas* modificado, volvemos a correr el algoritmo y GGB, el cual vuelve a indicar que la comunicación es segura.

En la figura 4.23 podemos ver las FSMs que GGB generó a partir del código con la modificación. En la figura 4.24 podemos ver el nuevo global graph generado.

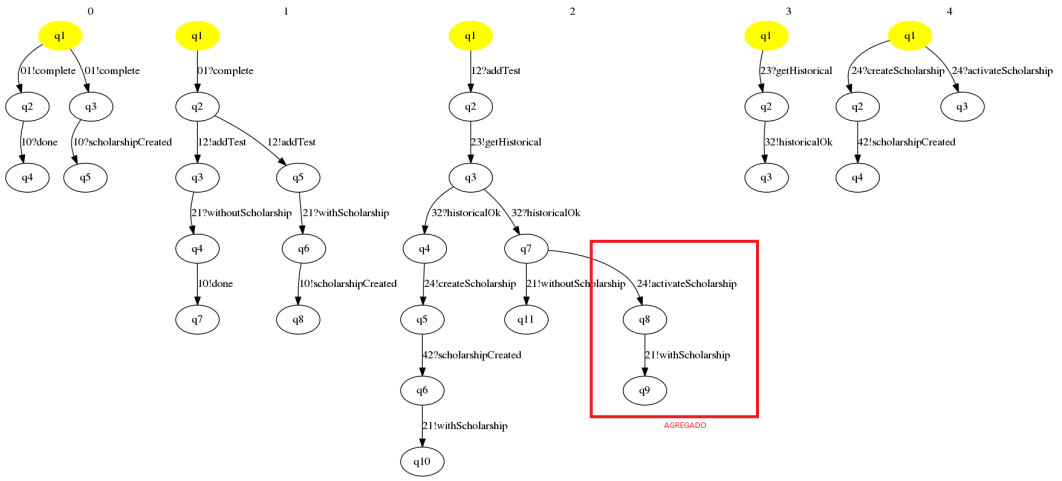


Figura 4.23: FSMs generadas con aplicación Statistic modificada.

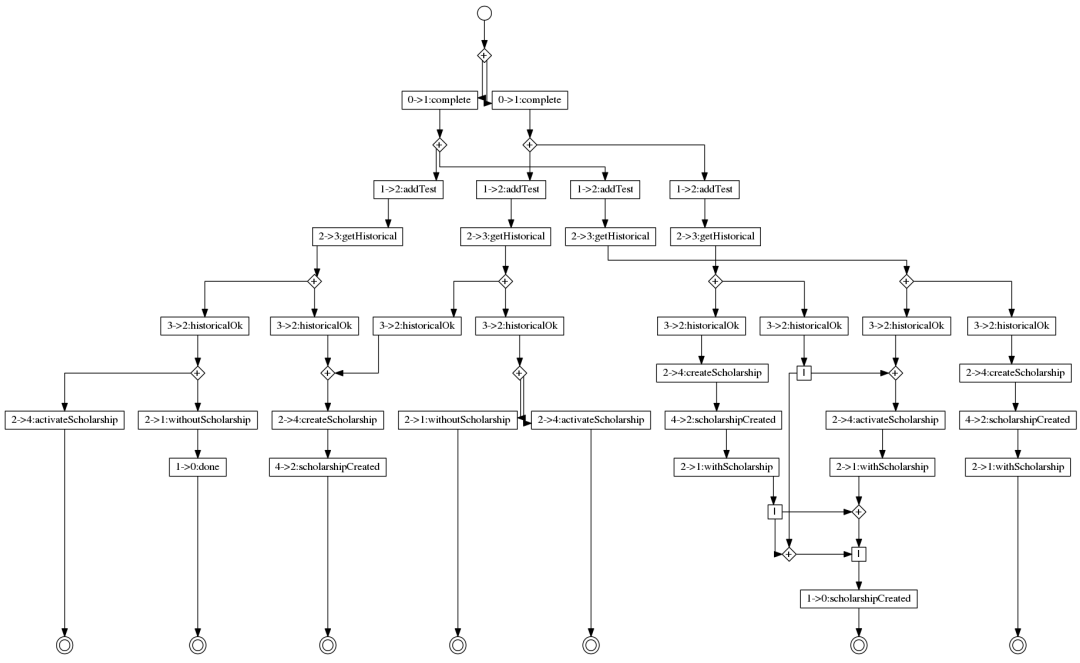


Figura 4.24: Global graph final.

## 4.3. Caso de estudio 3

Como último ejemplo, presentamos un sistema que modela la compra de un vuelo en una aplicación web. Los servicios involucrados en la compra son los siguientes:

- **Cliente**: se encarga de manejar el pedido de un cliente. Se representa con la máquina 0.
- **Vuelos**: se encarga de manejar el proceso de reserva, que consiste en hacer la reserva en la aerolinea, invocar al pago de la tarjeta e invocar a la aplicación que genera el voucher de la compra. Se representa con la máquina 1.
- **Aerolinea**: se encarga de hacer la reserva del vuelo en la aerolínea. Se representa con la máquina 2.
- **Pagos**: se encarga de realizar el cobro. Invoca la validación de la tarjeta. Se representa con la máquina 3.
- **TarjetaDeCredito**: se encarga de realizar validaciones sobre la tarjeta de crédito utilizada, y ejecutar el pedido de débito en la misma. Se representa con la máquina 4.
- **Voucher**: se encarga de generar el voucher e invocar el envío por email. Se representa con la máquina 5.
- **Mailing**: se encarga de realizar el envío por email del voucher generado. Se representa con la máquina 6.
- **Cuenta**: maneja las reservas realizadas por el cliente. Se representa con la máquina 7.

El flujo de ejecución comienza cuando el servicio *Cliente* envía el mensaje "book" al servicio *Vuelos*, con la intención de realizar la compra. Cuando *Vuelos* recibe el mensaje, ejecuta las siguientes operaciones: en primer lugar, realiza la reserva del vuelo en la aerolínea correspondiente. Para eso, invoca al servicio de *Aerolinea*, el cual realiza la reserva en la aerolínea. En segundo lugar, realiza el cobro del vuelo, invocando al servicio de *Pagos*. *Pagos* llama al servicio de *TarjetaDeCredito* para que valide la tarjeta ingresada. Si la tarjeta es válida y tiene saldo, entonces devuelve ok. Sino devuelve el error correspondiente. Una vez que la tarjeta es validada, *Pagos* realiza el cobro de la misma. En tercer lugar y como último paso, invoca al servicio que genera el voucher. *Voucher* se encarga de generar la factura e invoca al servicio *Mailing* para que envíe el mail al cliente con la factura generada.

**Paso 1**

A continuación, se muestra el código relacionado a las aplicaciones:

```
1  public class Cliente {
2
3      @SocketName(names={...})
4      public static void main(String[] args) {
5          Socket flightSocket = new Socket("localhost", 3336);
6          PrintWriter out = getWriter(flightSocket);
7          BufferedReader in = getReader(flightSocket);
8
9          out.println("book");
10
11         String flightResponse = in.readLine();
12
13         if (flightResponse.equals("bookOk")){
14             System.out.println("booking ok!!");
15         }
16
17         if (flightResponse.equals("bookError")){
18             System.out.println("booking error");
19         }
20
21         if (flightResponse.equals("bookOkWithoutVoucher")){
22             System.out.println("booking ok without voucher");
23         }
24
25         flightSocket.close();
26     }
27 }
```

Figura 4.25: Aplicación Cliente.

```
1  public class Vuelos {
2     private static Socket airlineSocket, paymentSocket, voucherSocket;
3     private static PrintWriter airlineOut, paymentOut,voucherOut;
4     private static BufferedReader airlineIn, paymentIn, voucherIn;
5
6     @SocketName(names={...})
7     public static void main(String[] args) {
8        System.out.println("init flight server...");
9        createAirlineSocket();
10       createPaymentSocket();
11       createVoucherSocket();
12       ServerSocket serverSocket = new ServerSocket(3336);
13       while (true){
14          Socket clientSocket = serverSocket.accept();
15          BufferedReader clientIn = getReader(clientSocket);
16          PrintWriter clientOut = getWriter(clientSocket);
17
18          String inputLine = clientIn.readLine();
19          if (inputLine.equals("book")){
20             airlineOut.println("airlineBook");
21             String airlineResponse = airlineIn.readLine();
22             if (airlineResponse.equals("airlineBookError")){
23                clientOut.println("bookError");
24             }
25             if (airlineResponse.equals("airlineBookOk")){
26                // process payment
27                paymentOut.println("processPayment");
28                String paymentResponse = paymentIn.readLine();
29                if (paymentResponse.equals("paymentError")){
30                   clientOut.println("bookError");
31                   return;
32                }
33                if (paymentResponse.equals("paymentOk")){
34                   // generate vouchers
35                   voucherOut.println("generateVoucher");
36                   String voucherResponse = voucherIn.readLine();
37                   if (voucherResponse.equals("voucherError")){
38                      clientOut.println("bookOkWithoutVoucher");
39                   } else if (voucherResponse.equals("voucherOk")){
40                      clientOut.println("bookOk");
41                   }
42                }
43             }
44          }
45       }
46       serverSocket.close();
47    }
48
49    private static void createPaymentSocket() {
50       paymentSocket = new Socket("localhost", 3334);
51      paymentOut = getWriter(paymentSocket);
52      paymentIn = getReader(paymentSocket);
53    }
54
55    private static void createAirlineSocket() {
56       airlineSocket = new Socket("localhost", 3338);
57      airlineOut = getWriter(airlineSocket);
58      airlineIn = getReader(airlineSocket);
59    }
60
```

```java
public class Aerolinea {

    @SocketName(names={...})
    public static void main(String[] args) {
        System.out.println("init airline server...");
        ServerSocket serverSocket = new ServerSocket(3338);

        while (true){
            Socket flightSocket = serverSocket.accept();
            BufferedReader in = getReader(flightSocket);
            PrintWriter out = getWriter(flightSocket);

            String inputLine = in.readLine();

            if (inputLine.equals("airlineBook")){
                boolean hasError = book();

                if (hasError){
                    out.println("airlineBookError");
                } else {
                    out.println("airlineBookOk");
                }
            }
        }

        serverSocket.close();
    }

    private static boolean book() {
        // do logic ...
        return false;
    }
}
```

Figura 4.27: Aplicación Aerolínea.

```
1  public class Pagos {
2
3     private static Socket creditCardSocket;
4     private static PrintWriter creditCardOut;
5     private static BufferedReader creditCardIn;
6
7     @SocketName(names={...})
8     public static void main(String[] args) {
9        System.out.println("init payment server...");
10
11       createCreditCardSocket();
12
13       ServerSocket serverSocket = new ServerSocket(3334);
14
15       while (true){
16          Socket flightSocket = serverSocket.accept();
17          BufferedReader flightIn = getReader(flightSocket);
18          PrintWriter flightOut = getWriter(flightSocket);
19
20          String inputLine = flightIn.readLine();
21
22          if (inputLine.equals("processPayment")){
23             creditCardOut.println("validateCard");
24
25             String creditCardResponse = creditCardIn.readLine();
26
27             if (creditCardResponse.equals("cardNotValid")){
28                flightOut.println("paymentError");
29             } else if (creditCardResponse.equals("insuficientCredit")){
30                flightOut.println("paymentError");
31             } else if (creditCardResponse.equals("cardOk")){
32                flightOut.println("paymentOk");
33             }
34          }
35       }
36
37       serverSocket.close();
38    }
39
40    private static void createCreditCardSocket() {
41       creditCardSocket = new Socket("localhost", 3337);
42       creditCardOut = getWriter(creditCardSocket);
43       creditCardIn = getReader(creditCardSocket);
44    }
45 }
```

Figura 4.28: Aplicación Pagos.

```
1  public class TarjetaDeCredito {
2
3      @SocketName(names={...})
4      public static void main(String[] args) {
5          System.out.println("init credit card server...");
6          ServerSocket serverSocket = new ServerSocket(3337);
7
8          while (true){
9              Socket paymentSocket = serverSocket.accept();
10             BufferedReader in = getReader(paymentSocket);
11             PrintWriter out = getWriter(paymentSocket);
12
13             String inputLine = in.readLine();
14
15             if (inputLine.equals("validateCard")){
16                 boolean cardIsValid = cardIsValid();
17
18                 if (!cardIsValid){
19                     out.println("cardNotValid");
20                 } else {
21                     boolean hasCredit = hasCrdit();
22
23                     if (!hasCredit){
24                         out.println("insuficientCredit");
25                     } else {
26                         out.println("cardOk");
27                     }
28                 }
29             }
30         }
31
32         serverSocket.close();
33     }
34
35     private static boolean hasCrdit() {
36         // do logic ...
37         return false;
38     }
39
40     private static boolean cardIsValid() {
41         // do logic ...
42         return false;
43     }
44 }
```

Figura 4.29: Aplicación Tarjeta de crédito.

```
1  public class Voucher {
2     private static Socket accountSocket, mailingSocket;
3     private static PrintWriter accountOut,mailingOut;
4     private static BufferedReader accountIn, mailingIn;
5
6     @SocketNames(names={...})
7     public static void main(String[] args) {
8        System.out.println("init voucher server...");
9        createMailingSocket();
10       createAccountSocket();
11       ServerSocket serverSocket = new ServerSocket(3333);
12       while (true){
13          Socket flightSocket = serverSocket.accept();
14          BufferedReader flightIn = getReader(flightSocket);
15          PrintWriter flightOut = getWriter(flightSocket);
16          String inputLine = flightIn.readLine();
17          if (inputLine.equals("generateVoucher")){
18             accountOut.println("addReservation");
19             String accountResponse = accountIn.readLine();
20             if (accountResponse.equals("reservationAdded")){
21                mailingOut.println("sendEmail");
22                String mailingResponse = mailingIn.readLine();
23                if (mailingResponse.equals("emailNotSent")){
24                   flightOut.println("voucherError");
25                }
26                if (mailingResponse.equals("emailSentOk")){
27                   flightOut.println("voucherOk");
28                }
29             }
30          }
31       }
32       serverSocket.close();
33    }
34
35    private static void createAccountSocket() {
36       accountSocket = new Socket("localhost", 3339);
37       accountOut = getWriter(accountSocket);
38       accountIn = getReader(accountSocket);
39    }
40
41    private static void createMailingSocket() {
42       mailingSocket = new Socket("localhost", 3335);
43       mailingOut = getWriter(mailingSocket);
44       mailingIn = getReader(mailingSocket);
45    }
46 }
```

Figura 4.30: Aplicación Voucher.

```
1  public class Mailing {
2
3      @SocketNames(names={...})
4      public static void main(String[] args) {
5          System.out.println("init mailing server...");
6          ServerSocket serverSocket = new ServerSocket(3335);
7
8          while (true){
9              Socket voucherSocket = serverSocket.accept();
10             BufferedReader voucherIn = getReader(voucherSocket);
11             PrintWriter voucherOut = getWriter(voucherSocket);
12
13             String inputLine = voucherIn.readLine();
14
15             if (inputLine.equals("sendEmail")){
16                 boolean hasError = send();
17
18                 if (hasError){
19                     voucherOut.println("emailNotSent");
20                 } else {
21                     voucherOut.println("emailSentOk");
22                 }
23             }
24         }
25
26         serverSocket.close();
27     }
28
29     private static boolean send() {
30         // do logic ...
31         return false;
32     }
33 }
```

Figura 4.31: Aplicación Mailing.

```
1  public class Cuenta {
2
3      @SocketNames(names={...})
4      public static void main(String[] args) {
5          System.out.println("init account server...");
6          ServerSocket serverSocket = new ServerSocket(3339);
7
8          while (true){
9              Socket voucherSocket = serverSocket.accept();
10             BufferedReader in = getReader(voucherSocket);
11             PrintWriter out = getWriter(voucherSocket);
12
13             String inputLine = in.readLine();
14
15             if (inputLine.equals("addReservation")){
16                 addReservation();
17                 out.println("reservationAdded");
18             }
19         }
20
21         serverSocket.close();
22     }
23
24     private static void addReservation() {
25         // logic to add reservation..
26     }
27 }
```

Figura 4.32: Aplicación Cuenta.

### Paso 2

Tomamos como base los .class correspondientes a las clases, y ejecutamos nuestro algoritmo para obtener la representación textual del sistema que luego será utilizada como input por el la herramienta.

### Paso 3

Luego, al correr la herramienta, genera el global graph, indicando que la comucación entre estos componentes es segura. En la figura 4.33 podemos ver las FSMs que la herramienta generó a partir de las especiciones textuales brindadas. En la figura 4.34 podemos ver el global graph generado.

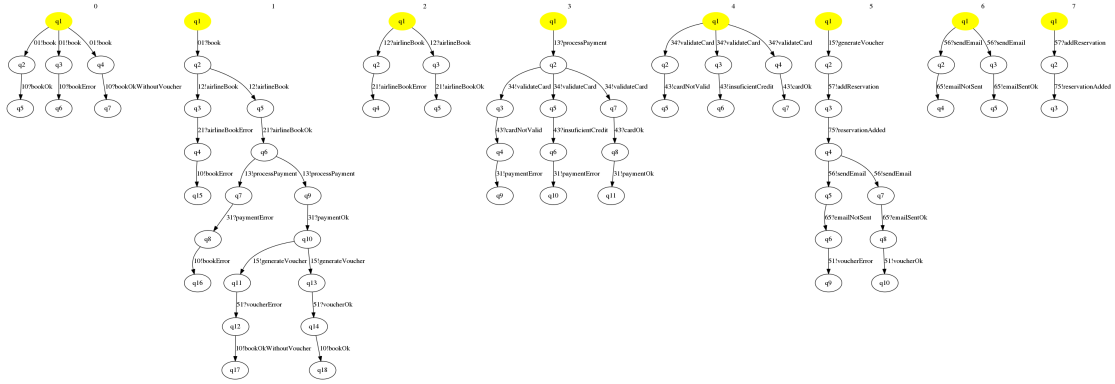

Figura 4.33: FSMs generadas

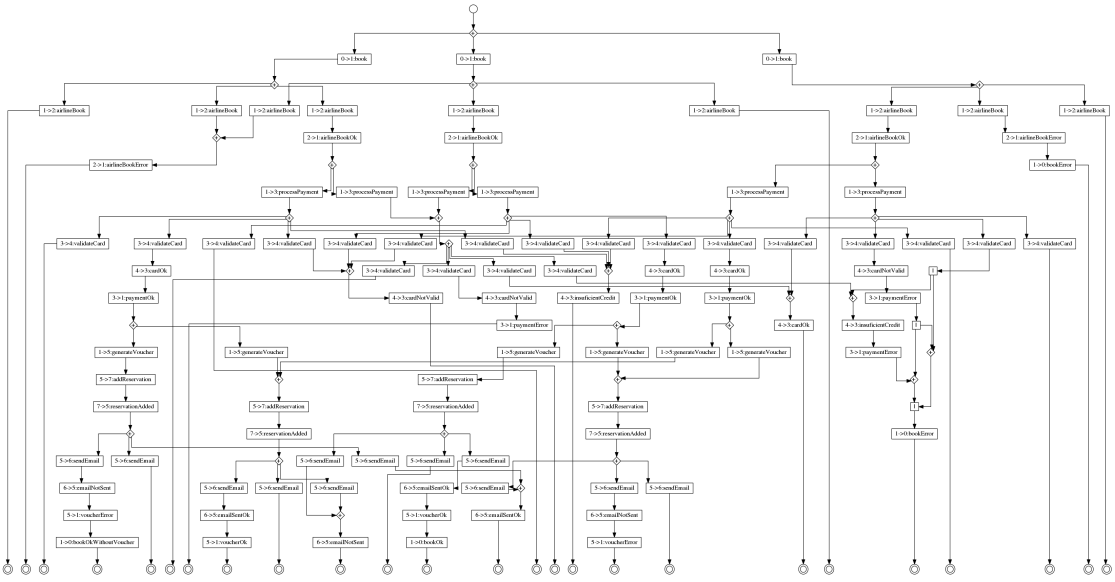

Figura 4.34: Global Graph

### Paso 4

Supongamos ahora que el equipo encargado del servicio *TarjetaDeCredito* decide remover las respuestas, cuando se procesa el mensaje "validateCard", ya que asumen que estas respuestas no tienen sentido. Luego, modifica en código para remover las respuestas:

```
1  public class TarjetaDeCredito {
2      ...
3      public static void main(String[] args) {
4          while (true){
5              ...
6              String inputLine = in.readLine();
7
8              if (inputLine.equals("validateCard")){
9                  processCreditCard();
10             }
11         }
12
13         serverSocket.close();
14     }
15
16     private static void processCreditCard() {
17         // do logic ...
18     }
19 }
```

Figura 4.35: Aplicación Tarjeta de crédito con código modificado.

Al ejecutar nuevamente GGB con el código modificado, la misma lanzó un error, identificando que existe mensajes que están en la espera de una respuesta, y las mismas no existen en la comunicación.

En la figura 4.36 podemos ver un ejemplo del error lanzado por GGB.

Finalmente, el equipo de CreditCard se advierte del error gracias al algoritmo, y vuelve a enviar las respuestas, haciendo que la comunicación entre los sistemas vuelva a ser segura.

```
gmc: check start
gmc:    Parsing CFSMs file...
gmc:    factor 0
gmc:    dir "outputs"
gmc:    TS:     (nodes 47, transitions 46)
gmc:    0-bounded TS:   (nodes 47, transitions 46)
gmc:    Branching Property:             []
gmc:    Branching representability:     [Bp 0 "q2",Bp 0 "q4",Bp 0 "q5",Bp 0 "q7",Bp 1
16",Bp 1 "q17",Bp 1 "q18",Bp 1 "q7",Bp 1 "q8",Bp 1 "q9",Bp 3 "q10",Bp 3 "q11",Bp 3
"q9",Bp 5 "q1",Bp 5 "q10",Bp 5 "q2",Bp 5 "q3",Bp 5 "q4",Bp 5 "q5",Bp 5 "q6",Bp 5
4",Bp 6 "q5",Bp 7 "q1",Bp 7 "q2",Bp 7 "q3"]

chosyn: ...minimising projections
chosyn: {---Testing machine 0 for language equivalence---}
chosyn: Is machine 0 representable? false
chosyn: {---Testing machine 1 for language equivalence---}
chosyn: Is machine 1 representable? false
chosyn: {---Testing machine 2 for language equivalence---}
chosyn: Is machine 2 representable? true
chosyn: {---Testing machine 3 for language equivalence---}
chosyn: Is machine 3 representable? false
chosyn: {---Testing machine 4 for language equivalence---}
chosyn: Is machine 4 representable? true
chosyn: {---Testing machine 5 for language equivalence---}
chosyn: Is machine 5 representable? false
chosyn: {---Testing machine 6 for language equivalence---}
chosyn: Is machine 6 representable? false
chosyn: {---Testing machine 7 for language equivalence---}
chosyn: Is machine 7 representable? false
chosyn: {---Is the system Language-equivalence Representable (part (i))? False---}
chosyn: {---Calling petrify...---}
```

Figura 4.36: Error por consola.

## 4.4. Limitaciones

El algoritmo desarrollado en este trabajo presenta algunas limitaciones a la hora de analizar los programas JAVA, entre las cuales podemos mencionar:

- Referencia circular: al momento de analizar el programa, en el caso que el algoritmo detecte una referencia circular entre métodos, lanza una excepción indicando que no puede analizar el programa. Esta limitación esta dada por la forma en la cual el algoritmo reemplaza los grafos asociados a referencias de métodos, con el fin de obtener el grafo global de transiciones: supongamos que el método **A** tiene una referencia al método **B**, y el método **B** tiene una referencia al método **A**. Entonces, cuando el algoritmo analiza el método **A** para obtener su grafo, realiza las siguientes operaciones:

  1. Busca las referencias a métodos incluídas en el método **A**. En este caso, encuentra el método **B** como referencia.
  2. Obtiene el grafo asociado al método **B**, realizando el mismo procedimiento que para el método **A**.
  3. Al analizar las referencias en el método **B**, encuentra referencia al método **A**, con lo cual intenta obtener el grafo asociado a dicho método.

4. Vuelve a realizar el prodimiento con el método **A**, generando así una recursión infinita entre los métodos.

Para detectar la referencia circular, el algoritmo mantiene en una lista de "métodos incompletos" cada método sobre el que intentó obtener el grafo y el mismo incluía una referencia. Luego, al aplicar recursión en los métodos referenciados en busca de sus grafos, si alguno de esos métodos está incluído en dicha lista, entonces detecta la referencia circular y lanza la excepción.

- Recursión: la recursión se puede ver como un caso particular de una referencia circular. Al igual que antes, en el caso que el algoritmo detecte recursión, lanza una excepción indicando que no es posible analizar dicho programa.

- Pasaje de componentes por parámetro: no se analizó en este trabajo los componentes detectados en un método (Socket, ServerSocket, BufferedReader, PrintWriter, etc) que son pasados como parámetros entre funciones. Para poder cumplir con esta funcionalidad, el algoritmo debería mantener, además de las referencias a métodos, las referencias de las variables que se pasan como parámetro a dicho método, y luego realizar una "asociación" entre las variables pasadas como parámetro y las variables utilizadas en el método. El único caso en el cual el algoritmo si mantiene esta relación es cuando un objeto Socket resultante de invocar la función *accept* de un ServerSocket, es pasado como párametro en el constructor de una clase que extienda de la clase *Thread* de JAVA. Este caso fue analizado ya que es una técnica habitual que estos sockets sean manejados por threads distintos.

- Copia de variables: los casos en los que se utilice *aliasing* para referirse a componentes detectados no son tenidos en cuenta por el algoritmo. Solo la variable a la cual se asignó el componente es la que se va a analizar.

# Capítulo 5

# Conclusiones y Trabajo futuro

En este trabajo, se dio un algoritmo que analiza programas escritos en Java, cuya comunicación se hace a través del envío y recepción de mensajes usando sockets, cuyo fin es validar que el protocolo utilizado es correcto. Se da una técnica en la cual, ante cualquier cambio en alguno de los componentes de la comunicación, se puede verificar si la especificación global del sistema sigue siendo correcta o no.

Se aplicó la técnica en algunos programas reales, analizando el comportamiento de los mismos, y validando la correcta comunicación. Se realizaron modificaciones en los programas para analizar como se comportaba el algoritmo ante dichos cambios. En todos los programas modificados, se ilustraron los problemas que se originaron al realizar modificaciones sobre el código.

La posibilidad de realizar validaciones en el protocolo de comunicación de sistemas reales puede ser de gran ayuda a la hora de realizar modificaciones sobre los mismos, ya que permite garantizar que ante estos cambios, el sistema general sigue siendo seguro.

Por otro lado, las simplificaciones sobre los programas tomadas en este trabajo reducen la población de programas que pueden ser de análisis del algoritmo. La escritura de algoritmos no es una tarea estandar, muchas veces está ligado a la cultura del desarrollador, y en consecuencia, puede realizarse de muchas maneras diferentes.

De todos modos, el algoritmo puede ser extendido de manera no compleja con el fin de soportar código que hoy en día no es reconocido por el mismo. En conclusión, cada programador podría extenderlo para que soporte la implementación de los sistemas desarrollados por ellos mismos, y poder así validas su comunicación.

## 5.1. Trabajos futuros

Como trabajos futuros, se pueden clasificar los mismos como trabajos de mejora del algoritmo y trabajos de ampliación.

Por un lado, se puede mejorar el algoritmo brindado en este trabajo realizan-

do modificaciones sobre decisiones tomadas o extendiendo algunas funcionalidades implementadas. Se pueden mencionar:

1. agregado de componentes de lectura. Hoy en día, el trabajo solo interpreta objetos de la clase InputBuffers de Java como componente de lectura. Se podrían agregar otras clases de Java que actúen como readers. Para ello, se tendría que definir nuevos filtros que sepan interpretar estas nuevas clases.

2. agregado de componentes de escritura. Al igual que los componentes de lectura, se podrían agregar nuevos filtros que interpreten distintos componentes de lectura, y asi extender la funcionalidad del algoritmo.

3. agregar validaciones lógicas en los flujos. Mediante este procedimiento, se podrían ignorar transiciones encontradas que lógicamente nunca se ejecutarían en el programa en tiempo de ejecución.

4. reconocimientos de protocolos de comunicación mas complejos. En este trabajo, solo se interpretan mensajes que son instancias de Strings o Enum. Se podría extender el algoritmo para que también pueda interpretar mensajes mas avanzados, como objetos.

5. Por último, se podría implementar una herramienta web en donde se puedan ingresar los componentes que participan en la comunicación, y que la herramienta procese el algoritmo. En este caso, se podrían sacar las anotaciones que reconocen los componentes, y podría ser parte de la herramienta configurar los nombres de los componentes, sin necesidad de agregar las anotaciones para hacer esta tarea.

Por otro lado, se puede ampliar el algoritmo desarrollado para poder aplicarlo sobre sistemas cuyo protocolo de comunicación no esté basado en sockets. Hoy en día, muchos sistemas complejos están construidos bajo el estilo arquitectónico de Microservices, que pueden ser considerados una especialización o extensión de arquitecturas orientas a servicios (SOA), usadas para construir sistemas distribuidos. Es un enfoque para el desarrollo de una sola aplicación como un conjunto de servicios pequeños, cada uno ejecutando en su propio proceso, muchas veces comunicados a través de HTTP mediante APIs REST. En consecuencia, se podría ampliar la técnica desarrollada y utilizarla en protocolos HTTP.

# Bibliografía


[1] Business process model and notation. http://www.bpmn.org.

[2] Objectweb. asm. web pages at http://asm.objectweb.org/.

[3] Samik Basu and Tevfik Bultan. Choreography conformance via synchronizability. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 795–804. ACM, 2011.

[4] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 191–202. ACM, 2012.

[5] Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexandre Yakovlev. Deriving petri nets for finite transition systems. *IEEE Trans. Computers*, 47(8):859–882, 1998.

[6] M. Dahm. Byte code engineering. in java-information-tage 1999. http://jakarta.apache.org/bcel/, Sept. 1999.

[7] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012.

[8] Erich Gamma. Design patterns - past, present & future. In Sebastian Nanz, editor, *The Future of Software Engineering.*, page 72. Springer, 2010.

[9] Matthias Güdemann, Gwen Salaün, and Meriem Ouederni. Counterexample guided synthesis of monitors for realizability enforcement. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification*

*and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, volume 7561 of *Lecture Notes in Computer Science*, pages 238–253. Springer, 2012.

[10] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 221–232. ACM, 2015.

[11] Sudheer R Mantena. Transparency in distributed systems by.

[12] Krishna Nadiminti, Marcos Dias De AssunÃ§Ã£o, and Rajkumar Buyya. Distributed systems and recent innovations: Challenges and benefits.

[13] B. Clifford Neuman. Scale in distributed systems. In *Readings in Distributed Computing Systems*, pages 463–489. IEEE Computer Society Press, 1994.

[14] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. *IJBPIM*, 1(2):116–128, 2006.

[15] Benjamin H. Sigelman, Luiz AndrÃ© Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and An Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure.

[16] F. Spoto. The julia generic static analyser. available at www.sci.univr.it/ spoto/julia, 2005.

[17] Lukasz Swierczewski. The distributed computing model based on the capabilities of the internet. *CoRR*, abs/1210.1593, 2012.