Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Tesis de Licenciatura

FXML + SEDA

Integrando dos enfoques para el manejo de concurrencia.

Autor:
Diego Gabriel Hara

Directores:
Sergio Yovine
Gervasio Perez

13 de Noviembre de 2013

Índice

1. Introducción **2**
- 1.1. Motivación . . . 2
- 1.2. Infraestructura . . . 2
  - 1.2.1. FXML . . . 2
  - 1.2.2. SEDA . . . 4
- 1.3. Objetivo . . . 5

2. Mapeo Lineal **6**
- 2.1. Dependencias weak. . . . 8
  - 2.1.1. Productor-Consumidor. . . . 9
  - 2.1.2. Dependencias weak con freshness . . . 11
- 2.2. Dependencias strong y biyectivas . . . 13
  - 2.2.1. Dependencias strong y biyectivas con freshness . . . 13
- 2.3. Pipelines no lineales . . . 14
- 2.4. Extensiones . . . 15
  - 2.4.1. SEIDE - Applicable . . . 15
  - 2.4.2. SEIDE - DispatcherRegistrationAware . . . 16
- 2.5. Templates . . . 18
  - 2.5.1. Caso 1 . . . 18
  - 2.5.2. Caso 2 . . . 18
  - 2.5.3. Caso 3 . . . 19

3. Mapeo InProc **21**

4. Mapeo Distribuido **25**
- 4.1. Distribuidor y Concentrador . . . 27
- 4.2. Ejemplo . . . 29

5. Caso de Estudio **30**

6. Conclusiones **35**

7. Trabajo Futuro **35**

Sinopsis

FXML es un lenguaje algebraico cuyo objetivo es poder expresar de manera formal la coordinación de código que se ejecuta de manera concurrente junto con dependencias de datos y control. SEDA es una arquitectura orientada a servicios de alta concurrencia que simplifica la construcción de servicios bien condicionados combinando el uso de *threads* y un modelo de programación basado en eventos.

El objetivo de este trabajo es encontrar la metodología para integrar el enfoque formal y el arquitectural que poseen FXML y SEDA respectivamente con el propósito de hacer que la ingeniería de software concurrente sea más simple y menos suceptible a errores. Se presentarán también estrategias que permitirán tratar distintos casos con distintas características para no sólo lograr una manera correcta, sino una manera conveniente de pasar del enfoque formal al arquitectural.

1. Introducción

1.1. Motivación

En la actualidad existe una gran variedad de problemas que pueden ser resueltos explotando las ventajas que proveen las grandes plataformas con múltiples unidades de procesamiento (multiprocesadores). Aprovechar estas ventajas ya no está limitado a grandes sistemas costosos, sino que ahora también existe *hardware multicore* accesible para pequeñas empresas o a nivel hogareño. Los problemas que pueden abordarse son heterogéneos, un típico ejemplo de esto puede ser un *Web Server* con una gran cantidad de peticiones en simultáneo, o bien cualquier otro tipo de servidor.

El desarrollo de *software* que aproveche la arquitectura *multicore* no es trivial, de hecho suele ser una fuente importante de errores [1]. Al desarrollo de software se le suma por ejemplo conceptos como la utilización de estructuras de datos *thread-safe*, planificación de I/O, etc. Entre los casos famosos se puede mencionar por ejemplo el del sistema Nasdaq OMX que provocó la demora de notificaciones de operaciones de la IPO de Facebook durante 30 minutos, generando inconsistencias en el precio de mercado que ocasionaron pérdidas millonarias [2].

Existen varios enfoques que permiten abordar y describir estos problemas, por un lado existen enfoques formales, muy cercanos al álgebra de procesos como por ejemplo *CCS*[3], *CSP*[4], *FXML*[5] [6], etc. También existen enfoques arquitecturales más cercanos a caracterizar problemas y resolverlos utilizando algún *framework* de modelado como por ejemplo *CNC*[7], *TBB*[8], *Hadoop*[9], *SEDA*[10] [11], etc. *En este trabajo se estudiará, la integración entre FXML y SEDA.*

1.2. Infraestructura

1.2.1. FXML

FXML es un lenguaje algebraico dotado de una semántica formal denotacional. El objetivo del lenguaje es poder expresar la coordinación de código que se ejecuta de manera concurrente junto con dependencias de datos y control. Un programa FXML es una composición asincrónica de procesos paralelos. Una variable no representa una dirección de memoria, sino la secuencia de todos los valores asignados.

El cuerpo de una especificación en FXML está compuesta por bloques llamados *pNodes*. Estos *pNodes* están compuestos por declaración de variables, asignaciones y por bloques *legacy* (C, Java, etc.). Las asignaciones son de la forma `x = f(y)`, dónde `f` es cualquier función que no tiene efecto colateral y semánticamente átomica. El bloque *legacy* puede utilizarse para encapsular código preexistente o lógica de negocio.

Estas unidades pueden ser compuestas de manera secuencial o paralela. Las operaciones que indican composición paralela son `par` y `for_all`. FXML permite además describir dependencias entre las instrucciones que componen a `par` o dentro del cuerpo del `for_all` utilizando *etiquetas*. Las dependencias de control o de datos pueden ser restringidas para lograr sincronización o restricciones en el grado de paralelismo.

Listing 1: FXML Ejemplo

```
1  dep W -> R
2  var int x
3  par
4    producer:
5       while(true) {
6          W : x = produce()    // write x
7       }
8    consumer:
9          while(true){
10            seq {
11               R : y = x      // read x
12               legacy{System.out.println(y)}
13            }
14         }
```

FXML además provee *labels* que permitirán nombrar de alguna manera a los bloques. Las dependencias y sus restricciones seran descriptas utilizando estos *labels*. Los *pNodes* dentro del `while` o `for` serán automáticamente indexados. La semántica del ciclo del productor en este caso será la secuencia $W_0W_1\ldots$, donde W_i es la i-ésima ocurrencia de la asignación etiquetada `W`.

La instrucción `dep W` $\rightarrow$ `R` indica la dependencia entre las ocurrencias de los *pNodes* etiquetados `W` y `R`. La flecha $\rightarrow$ indica que la variable `x` (mostrada en el código anterior) *debe* ser escrita al menos una vez antes de ser leida. Esta dependencia se llama *weak* dado que no se puede asegurar que todo valor producido es consumido, sólo que si se consumió algo es porque fue producido anteriormente. La figura 1 muestra una traza *weak* donde se puede observar por ejemplo que el valor generado W_1 nunca es leído por nadie. Las líneas verticales indican secuencialidad en el orden de las escrituras y lecturas y las horizontales indican el consumo de información.

Para indicar que no hay pérdida de información se introduce `dep W [strong]`$\rightarrow$ `R` (restringiendo la anterior) que indica que *todo* valor producido (escrito en `x`) es leído al menos una vez. La figura 2 muestra una traza *strong* donde se puede observar todo valor generado es leído por alguien. Tanto para las dependencias *weak* como para las *strong* no es importante la noción de orden entre las lecturas y escrituras (las dependencias de `W` $\rightarrow$ `R` se pueden cruzar).

También se puede especificar dependencias *biyecticas* `dep W[bij]` $\rightarrow$ `R`, tal como su nombre lo indica, define una biyección entre las escrituras del productor y las lecturas del consumidor. Un caso particular aún más fuerte, es el caso de las dependencias atadas a una determinada iteración, `dep W` $(i,i)\rightarrow$ `R`, en éste caso se leerá el valor producido en la i-ésima iteración del productor, en la i-ésima iteración del consumidor. La figura 3 muestra una traza donde no sólo cada valor generado es leído, sino que además las escrituras y lecturas coinciden en el número de la iteración.

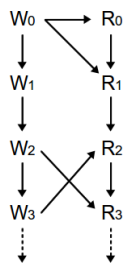

Figura 1: Traza Weak

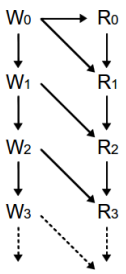

Figura 2: Traza Strong

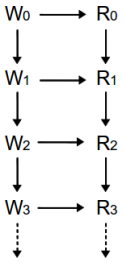

Figura 3: Traza (i,i)

La semántica de un programa FXML es el conjunto de todos los órdenes parciales (posiblemente infinitos) que son consistentes con las dependencias.

Un programa en *FXML* puede ser traducido a otro *equivalente* y *válido* si el comportamiento del nuevo es análogo y si no se agrega un grado de paralelismo que no tenía el anterior (permitiendo trazas espúreas). Básicamente el nuevo debe ser igual o más restrictivo que el original. (Ejemplo: Convertir todas las dependencias *weak* de una especificación en *strong*. La inversa de esto **no vale**, puesto que si se convierten dependencias *strong* en *weak* se estaría incurriendo exactamente en lo que no está permitido, agregar paralelismo - hay trazas nuevas permitidas que antes no lo estaban).

Para más detalle sobre FXML se recomienda leer la documentación formal[5] [6].

1.2.2. SEDA

SEDA (*Staged Event Driven Architecture*) es una arquitectura orientada a servicios de alta concurrencia que simplifica la construcción de servicios bien condicionados. SEDA combina el uso de *threads* y el modelo de programación basado en eventos para administrar la concurrencia, I/O, planificación y la gestión de recursos.

Se entiende por servicio en este trabajo a una aplicación (un programa o sistema) dedicado satisfacer las necesidades de un cliente. Un ejemplo simple de esto puede ser un *Web Server*, un *SMTP Server* o cualquier otro sistema que esté constantemente esperando recibir peticiones y las ejecuta (que puede terminar con una respuesta al usuario o no).

Las unidades básicas de construcción de una aplicación SEDA son los *Stages*, *EventHandlers* e *Incomming Queues*. Los *Stages* actuan básicamente como contenedores que agrupan el *Event Handler* (código de negocio) y la *Incomming Queue*. El *Stage* también tendrá asociada información de configuración como por ejemplo la cantidad de *threads* disponibles para procesar los eventos entrantes.

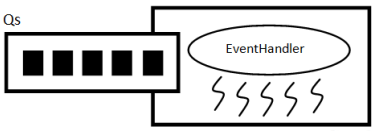

Figura 4: Stage S.

Un *Stage* recibe un evento en su *Incomming Queue*, este evento se desencola y se procesa. Se entiende por procesar a ejecutar el código del *EventHandler* asociado a ese *Stage*. El evento recibido puede tener parámetros asociados, por lo tanto el resultado de la ejecución podrá variar para el mismo evento en el mismo *Stage*. El código del *EventHandler* también podrá encolar un nuevo evento en otro *Stage*. Dependiendo de la cantidad de *threads* definida en el *Stage* se podrán ejecutar n eventos de manera concurrente.

Una aplicación SEDA esta compuesta básicamente por una red de *Stages* interconectados a través de colas. Esta arquitectura permite que el sistema no se degrade ante una ráfaga de pedidos que excedan la capacidad de procesamiento. Para lograr esto existen distintas políticas, pudiendo por ejemplo asignar una capacidad determinada al tamaño de las colas, y en caso de excederse rechazar el evento entrante o bien hacer que una petición tarde un poco más (porque los eventos permanecen más tiempo en las colas). Independientemente de la política elegida el sistema no se verá comprometido por ejemplo con la creación de excesivos *threads* o quizá con un consumo excesivo de memoria.

El aspecto fundamental de SEDA es que provee un modelo de programación que tiene en cuenta la carga del sistema y el rendimiento de la aplicación, es esclable y robusto dado que desacopla la in-

fraestructura (la red de *Stages* interconectados) de la lógica de negocio (el código del *EventHandler*).

Muchas aplicaciones comerciales, algunas con requerimientos de performance realmente altos utilizan esta arquitectura. Entre ellos se pueden nombrar Cassandra [13], SwiftMQ [14], LimeWire [15], etc.

Para más detalle sobre SEDA se recomienda leer la documentación formal[10] [11].

Se utilizará en el trabajo un framework llamado SEIDE [12] que modela SEDA para Java EE. SEIDE es un framework en el que autor de este trabajo ha colaborado en sus inicios y lo ha extendido, como se verá más adelante, especialmente para este trabajo.

1.3. Objetivo

El objetivo de este trabajo no es hacer una traducción automática entre estos dos enfoques sino encontrar la metodología (que responde a ciertas características) para integrarlos realizando derivaciones que permitan, partiendo del enfoque formal, llegar al arquitectural obteniendo una implementación adecuada.

FXML fue concebido para describir algoritmos paralelos sobre dominios finitos (Ej. Decodificar MPEG, Multiplicación de matrices, etc.), la derivación de programas desde *FXML* a un framework orientado a servicios (con dominios infinitos) presenta también un desafio.

Un beneficio importante de esta integración consiste en que existirá un formalismo que respalde el código que se va a ejecutar.

Al analizar un modelo SEDA las dos principales características que se observan son:

- Cantidad de eventos en el sistema.
- Tiempo de ejecución.

Una arquitectura SEDA permite que un programa o servicio se ejecute en múltiples *threads* cuidando que no se degrade el sistema cuando sube la carga del mismo. Para lograr esto se utilizan colas para obtener y acumular los eventos que llegan a cada *Stage*, ante una ráfaga o suba abrupta de la carga, los eventos se acumularán en las colas (hasta que haya un *thread* disponible) haciendo que los que están siendo ejecutados no se vean afectados por esta condición.

La utilización de esta arquitectura, de manera similar a un programa se ejecuta sobre una VM (JVM, .NET Framework, etc.), tiene un costo. Si alguien escribiera un programa creando threads directamente probablemente se ejecutaría más rápido que si utilizara SEIDE o cualquier otra implementación de SEDA, pero no obtendría los beneficios que la arquitectura provee. Básicamente claridad en el código, robustez y sencillez para modelar (templetización).

Sería sencillo tratar a cualquier caso (programa FXML con determinadas características que se presentará más adelante) de la misma manera, derivándolos de la misma forma, pero esto sería un error puesto que no todos los casos tienen las mismas características. No es lo mismo un proceso que utiliza pocos *threads* y cada uno ejecuta una tarea costosa, que un proceso que utiliza muchos *threads* y cada uno ejecuta una tarea económica.

Si se analizan ambos casos en el contexto de una arquitectura SEDA, el primero tendrá pocos eventos que viajen entre *stages* pero puede que el tiempo de ejecución de cada uno sea considerable. El segundo tendrá muchos eventos entre *stages* pero el tiempo de ejecución de cada uno será bajo. Lo ideal consiste en balancear estas dos características. Ni tener un sistema con pocos mensajes a costa de tener tareas pesadas, ni tener un exceso de mensajes acumulándose con tareas extremadamente livianas.

Partiendo de una solución genérica no dependiente del *framework* se podrán derivar entonces soluciones diferentes dependiendo principalmente de distintas estrategias, características intrínsicas del problema, hardware, etc. Se presentarán en las siguientes secciones 3 estrategias de derivación:

- Básica (o mapeo lineal).
  Que consiste en identificar un patrón de programa paralelo en *FXML* que se puede poner en correspondencia directa con un programa *SEDA*, convirtiendo cada *pNode* en un *Stage* e interconectandolos según las dependencias entre las asignaciones de variables compartidas.

- InProc
  Que consiste en manejar el paralelismo de un dominio finito dentro de un *Stage* para evitar la explosión de *Stages*.

- Distribuido
  Que consiste en distribuir la carga de procesamiento (paralelo) en distintos equipos, o dicho de otra manera, escalar horizontalmente.

Estas tres estrategias podrán componerse para encontrar una implementación que cumpla con el objetivo, la obtención de una implementación adecuada como resultado de las derivaciones sucesivas partiendo de *FXML*.

2. Mapeo Lineal

Cuando se desconoce la cantidad de eventos y el tiempo de ejecución o bien se conoce estas características y se sabe (o intuye) que están balanceadas la estrategia que se propone es la aplicación básica de SEDA que consiste en realizar mapeos lineales.

Tal como fue presentado en la seccion 1.3, el objetivo es partiendo del enfoque formal llegar al arquitectural o de manera más concreta de *FXML* a *SEDA*. Lo primero que se debe hacer es caracterizar el código *FXML*, no será posible (y probablemente carezca de sentido) tomar como punto de partida cualquier programa en *FXML*. Se comenzará entonces con la familia de programas en *FXML* que cumple determinadas características.

$$\begin{bmatrix} dep : d_1 ... d_k \\ var : v_1 ... v_s \\ par : p_1 ... p_n \end{bmatrix}$$

Def 1: Forma inicial FXML válida.

El programa en *FXML* debe tener dependencias, variables compartidas y una única instrucción `par` con n cantidad de bloques que pueden componerse de manera paralela. Cada uno de esos bloques o *pNodes* (p_i) deben también cumplir con ciertas características como se presenta a continuación.

Sea h_i una porción de código secuencial[1], un pNode p_i es de la forma:

$$p_i = \begin{cases} for\_all(k = 0; k < N_i; k++) \ \{ \\ \quad while(true) \ \{ \\ \quad\quad h_i \\ \quad \} \\ \} \end{cases}$$

Def 2: Forma inicial de p_i válida.

[1] Más adelante se presentará como tratar casos cuando h_i no es secuencial.

La razón por la que h_i está dentro de un `while` es porque, como se ha mencionado anteriormente, *SEDA* es una arquitectura orientada a servicio y en este trabajo esto implica un sistema que está constantemente esperando peticiones, es decir, no hay una condición de fin. Básicamente habrán N_i *threads* especializados en ejecutar todo el tiempo h_i.

N_i es la cota que indica el grado máximo de paralelismo del pNode, el caso particular $N_i = 1$ indica que el código (de cada *pNode*) se ejecutará de manera secuencial.

Básicamente un *pNode* p_i que cumple con las características pedidas, será mapeado a un *Stage* y las dependencias d_j indicarán comunicación entre esos *Stages*.

Los casos que se presentarán a continuación representan un caso particular donde existe la asignación de una única variable entre dos *pNodes*. Esta elección permite analizar de manera más sencilla cada caso sin perder generalidad, donde pueden existir múltiples asignaciones entre los *pNodes*. De esta manera vale que $k = s = n - 1$ en Def 1.

El *pNode* h_i puede entonces contener una etiqueta W_i y R_i (con x_i y x_j variables compartidas e y variable local) de la forma:

- $W_i : x_i = f_i(...)$
- $R_i : y = x_j$ con $i \neq j$.

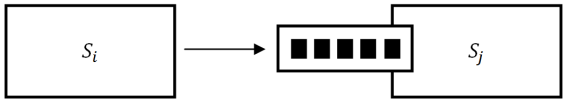

Figura 5: Stage i y Stage j

El siguiente cuadro muestra la relación entre los tipos de mapeos y la cantidad de threads disponible en cada *stage*.

Tipo de mapeo.	Cantidad de threads.
1 a 1	N_i=1 y N_j=1
1 a n	N_i=1 y $N_j \geq 1$
n a 1	$N_i \geq 1$ y N_j=1
n a m	$N_i \geq 1$ y $N_j \geq 1$

Cuadro 1: Relación entre tipo de mapeos y cantidad de threads.

La cantidad de *threads* disponibles para los productores o consumidores está definida por el número N_k asociado a p_k.

Las asignaciones de variables compartidas indexadas de un programa *FXML* entre distintos *pNodes* definen las dependencias entre distintos *stages*. Si todas las asignaciones (de variables compartidas) de un *pNode* son leidas en un único *pNode* y todas las lecturas (de variables compartidas) de un *pNode* son originadas en un único *pNode*, entonces los *stages* conformarán un pipeline lineal como se muestra en la Figura 6.

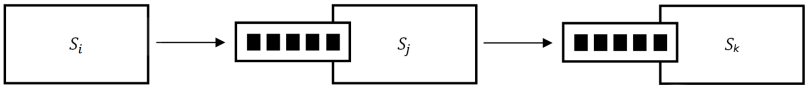

Figura 6: Stage i, Stage j y Stage k formando un pipeline lineal

Existen otros pipelines que no son lineales donde el flujo de datos puede dividirse o unirse con otro, a efectos prácticos en este trabajo cuando se hable de *pipeline* se estará hablando de *pipeline lineal*.

La problemática en el mapeo entre *FXML* y *SEDA* puede abordarse la mayoría de las veces utilizando este *pipeline* sin perder generalidad. Más adelante, en la sección 2.3 se presentarán los otros *pipelines* y algunas particularidades a tener en cuenta.

2.1. Dependencias weak.

Una dependencia `dep` $\{L_1 \rightarrow L_2\}$ es *weak* si toda ocurrencia del *pNode* $L_2 : q_2$ es precedida por una ocurrencia del *pNode* $L_1 : q_1$.

Dado un *stage* S:

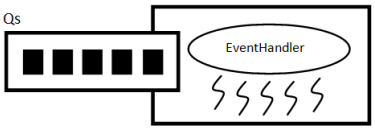

Figura 7: Stage S.

Q_s es la cola de entrada de eventos del *stage* S.

Una dependencia *weak* se implementa con una política de *eliminación de carga* sobre Q_s estableciendo un máximo en la capacidad de la misma. Si se desea encolar un evento y la capacidad está excedida, se ignorará el evento. Esto significa que se pueden perder eventos entrantes (escrituras), pero si se encola el evento en Q_s, será leído y procesado en el stage.

Esto hace que la implementación en *SEDA* pueda no tener todas las trazas admisibles por la semántica de *FXML*. Esto no es un problema puesto que *FXML* sólo pide que exista una inyección entre el modelo *FXML* y el conjunto de trazas de la implementación. (Ref. [5] - Sec 11.3)

2.1.1. Productor-Consumidor.

En esta sección se presenta el caso más simple donde hay un productor y un consumidor.

Listing 2: FXML Productor-Consumidor

```
1  dep W -> R
2  var int x
3  par
4    writer:
5         for_all(int i=0; i<N; i++) {
6            while(true) {
7               W : x = produce()
8            }
9         }
10   reader: for_all(int i=0; i<M; i++){
11              while(true){
12                 seq {
13                   R : y = x
14                        consume(y)
15                 }
16              }
17           }
```

Las variables `N` y `M` representan la cantidad de threads disponibles para producir y consumir valores respectivamente. Las funciones `produce()` y `consume(arg)` son como su nombre lo indica, funciones que producen y consumen un valor. No es importante en este caso su implementación y podrían estar ocultando la ejecución de una porción de código *legacy*.

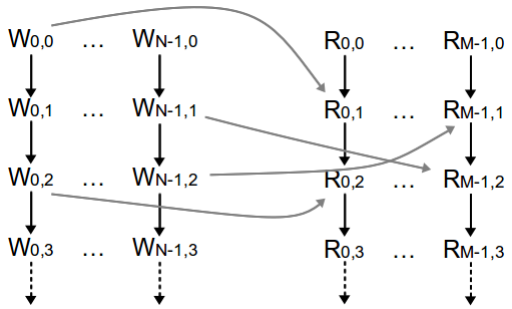

Figura 8: Traza de N Productores y M Consumudores - Dependencia Weak

La figura 8 muestra una posible traza de ejecución para el Productor-Consumidor del código 2.

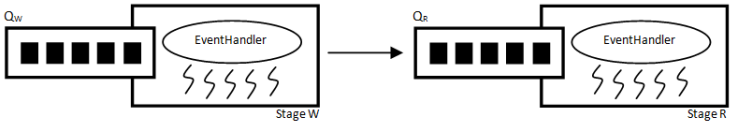

Figura 9: Stages que representan un Productor-Consumidor.

Un *event handler* está asociado a un *stage*, todo *event handler* debe implementar el método `execute` que es invocado cada vez que hay un elemento disponible en su cola de eventos (Q). Los h_i de *FXML* deberán convertirse en el código del método `execute`; la conversión será distinta si el *event handler* pertenece a un *stage* inicial o no. Si el *stage* es inicial es necesario poner explícitamente el `while(true)`, si no lo es no.

La escritura de una variable compartida se implementa mandando un mensaje de un *stage* a otro. En *FXML* esto sucede con la asignación propiamente dicha, en el código *SEIDE* el envío del mensaje está implementado con el método `W(x)`. De la misma manera, la lectura de una variable compartida se implementa obteniendo el mensaje recibido desde otro *stage*. Esta lectura se implementa en el método `R(message)`.

A continuación se muestra los *event handlers* de los *stages* Productor y Consumidor para la especificación *FXML* presentada en 2.

Listing 3: Productor-Consumidor EH_W

```
1  public class WriterEventHandler
2      implements DispatcherAware, EventHandler<InitMessage> {
3
4      private Dispatcher dispatcher;
5
6      @Override
7      public RoutingOutcome execute(InitMessage message) {
8
9          while (true) {
10             x = produce();
11             W(x);
12         }
13         return null;
14     }
15
16     private int W(int value) {
17
18        ProducerMessage message = new ProducerMessage();
19        message.setX(value);
20
21        Event event = new Event("READER", message);
22        dispatcher.execute(event);
23     }
24
25     private int produce() {
26         // legacy code
27     }
28
29     @Override
30     public void setDispatcher(Dispatcher dispatcher) {
31         this.dispatcher = dispatcher;
32     }
33 }
```

Listing 4: Productor-Consumidor EH_R

```
1  public class ReaderEventHandler
2      implements DispatcherAware, EventHandler<ProducerMessage> {
3
4      private Dispatcher dispatcher;
5
6      @Override
7      public RoutingOutcome execute(ProducerMessage message) {
8
9          int y = R(message);
10         consume(y);
11         return null;
12     }
13
14     private int R(ProducerMessage message) {
15         return message.getX();
16     }
17
18     private void consume(int value) {
19         // legacy code
20     }
21
22     @Override
23     public void setDispatcher(Dispatcher dispatcher) {
24         this.dispatcher = dispatcher;
25     }
26 }
```

La instrucción `for_all` incluida en ambos *pNodes* del código presentado en 2 no estará explicitamente escrita en el código resultante debido a que el *framework* se encarga de manejar el paralelismo, sólo es necesario definir el valor de N para cada *stage*.

2.1.2. Dependencias weak con freshness

$[min, max]$ *dep* $W \rightarrow R$ define, además de la dependencia, una restricción para la lectura de un valor v en R, escrito en W. No podrá ser leído si el tiempo transcurrido desde la escritura a la lectura no está en el intervalo $[min, max]$.

SEIDE no provee un mecanismo nativo para soportar estas restricciones, por lo tanto para proveer esta funcionalidad, se deberá agregar información de *freshness* en cada evento que va de un *stage* a otro. El *event handler* del productor (EH_W) deberá indicar el rango de validez del mismo dejando esa información en el mensaje. El consumidor deberá procesar el evento sólo si al momento de desencolarlo todavía está en rango (realizando un preproceso que consiste justamente en la verificación). Si no, se deberá descartar el evento obteniendo un resultado similar a si se hubiera perdido un evento por *eliminación de carga*.

A continuación se muestra la condición de *freshness* aplicado caso anterior.

Listing 5: FXML Productor-Consumidor - Freshness.

```
[min, max] dep W -> R
var int x
par
  writer:
     for_all(int i=0; i<N; i++) {
         while(true) {
            W : x = produce()
         }
     }
  reader: for_all(int i=0; i<M; i++){
        while(true){
             R : y = x
             consume(y)
        }
     }
```

Para detallar el código de EH_W y EH_R primero se debe introducir la clase *FreshnessRange*.

Listing 6: Clase FreshnessRange.

```
public class FreshnessRange {
    private long min;
    private long max;

    private long timeStamp;

    public TimeRange(long min, long max) {
        this.min = min;
        this.max = max;
        this.timeStamp = System.currentTimeMillis();
    }

    public boolean isInRange() {

        long currentTimeStamp = System.currentTimeMillis();
        long deltaTimeStamp = currentTimeStamp - this.timeStamp;

        return this.min <= deltaTimeStamp &&
               deltaTimeStamp <= this.max;
    }
}
```

Esta clase provee un método `inRange` que devuelve si el tiempo transcurrido entre la creación del mismo y la invocación del método, está en el rango comprendido por *min* y *max*.

A continuación se presenta una *primer versión del código* de EH_W y EH_R, más adelante, en la sección de extensiones a *SEIDE* se presentará una versión más refinada de los mismos.

Listing 7: Productor-Consumidor EH_W con Freshness

```
public class WriterEventHandler
    implements DispatcherAware, EventHandler<InitMessage> {

    private Dispatcher dispatcher;
    private int minRange = 0;
    private int maxRange = 100;

    @Override
    public RoutingOutcome execute(InitMessage message) {
        while (true) {
            int x = produce();
            W(x);
        }
        return null;
    }

    private int W(int value) {

        ProducerFreshnessMessage message =
             new ProducerFreshnessMessage();
        FreshnessRange freshnessRange =
             new FreshnessRange(this.minRange, this.maxRange);

        message.setFreshnessRange(freshnessRange);
        message.setX(value);

        Event event = new Event("READER", message);
        dispatcher.execute(event);
    }

    private int produce() {
        // legacy code
    }

    @Override
    public void setDispatcher(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }
}
```

Listing 8: Productor-Consumidor EH_R con Freshness

```
public class ReaderEventHandler
    implements DispatcherAware,
    EventHandler<ProducerFreshnessMessage> {

    private Dispatcher dispatcher;

    @Override
    public RoutingOutcome execute(ProducerFreshnessMessage message) {
        if (this.applies(message)) {
            int y = R(message);
            consume(y)
        }
        return null;
    }

    private int R(ProducerFreshnessMessage message) {
       return message.getX();
    }

    private void consume(int value) {
       // legacy code
    }

    private boolean applies(ProducerFreshnessMessage message) {
       return message.getFreshnessRange().isInRange();
    }

    @Override
    public void setDispatcher(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }
}
```

2.2. Dependencias strong y biyectivas

Una dependencia `dep` $\{L_1 \rightarrow L_2\}$ es *strong* si toda ocurrencia del *pNode* $L_1 : q_1$ es sucedida por una ocurrencia del *pNode* $L_2 : q_2$.

Dado un *stage* S similar al de la figura 7, una dependencia *strong* se implementa evitando la pérdida de eventos de $w \rightarrow r$. En otras palabras, para los ejemplos anteriores valdrá $Size(Q_R) = \infty$.

Para realizar esto basta con establecer 0 en el tamaño de la cola del *stage* R, esto implica que no se aplicará ninguna política de eliminación de carga.

Una dependencia biyectiva se implementa agregando a las condiciones de una dependencia *strong*, que Q_S sea de tipo FIFO.

2.2.1. Dependencias strong y biyectivas con freshness

En el caso de una dependencia *weak*, cuando no se cumple la condición de *freshness* al procesar un evento, lo que se hace es descartarlo de la misma manera que si se hubiera perdido por sobrepasar la capacidad de la cola asociada al *stage*. Básicamente, se puede descartar eventos por cualquier razón y sin embargo seguirá valiendo la condición de *weak*.

En los casos de *strong* o *biyectiva* no pueden perderse eventos. Si no hay condición de *freshness* con decir que $Size(Q_R) = \infty$ es suficiente, pero cuando se compone con esta condición surge un problema. ¿Qué es más importante? ¿El tipo de dependencia o la condición de *freshness*?

Cuando el tipo de dependencia (weak, strong, etc.) es más importante que la condición de *freshness*, si se vulnera esta última condición, se informará sobre ésta situación pero se procesará el evento normalmente. Cuando la condición de *freshness* es más importante que el tipo de dependencia, si se vulnera la primer condición (freshness) se informará e ignorará el evento. Por último, si ambas condiciones tienen la misma importancia, en caso de que se vulnere la condición de *freshness* se deberá abortar la ejecución.

Este comportamiento se implementará agregando a la verificación del rango, la verificación sobre la importancia entre el tipo de dependencia y la condición de *freshness* como se muestra a continuación.

Listing 9: Strong-Freshness-Applies

```
1  public boolean apply(FreshnessMessage message) {
2
3     boolean applies = message.getFreshnessRange().isInRange();
4
5     if (applies) {
6         // dejar en el log en caso de ser necesario - applies = true.
7     } else {
8         DependencyType dependencyType = getDependencyType();
9         Precedence precedence = getPrecedence();
10
11        if (dependencyType == DependencyType.WEAK) {
12            // dejar en el log que se descarta - applies = false
13        } else { // STRONG, BIYECTIVA, etc...
14                 if (precedence == Precedence.TYPE) {
15                     // dejar en el log que igual se procesará
16                     // modificar applies.
17                     applies = true;
18                 } else if (precedence == Precedence.FRESHNESS) {
19                     // dejar en el log que se descarta - applies = false
20                 } else {
21                     // EQUALS
22                     throw new IllegalStateException();
23                 }
24         }
25     }
26
27        return applies;
28 }
```

2.3. Pipelines no lineales

Al comienzo de esta sección se propuso trabajar con *pipelines lineales* donde todas las asignaciones (de variables compartidas) de un *pNode* son leidas en un único *pNode* y todas las lecturas (de variables compartidas) de un *pNode* son originadas en un único *pNode*.

Si las lecturas o escrituras pueden originarse en varios *pNodes* se conformará un pipeline con una bifurcación o unión como se muestra en las figuras 10 y 11 respectivamente.

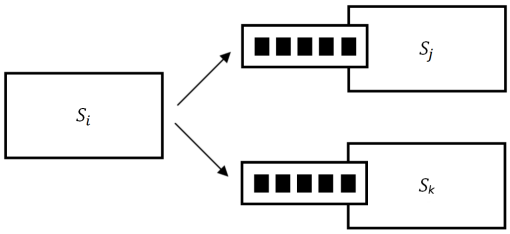

Figura 10: Bifurcación

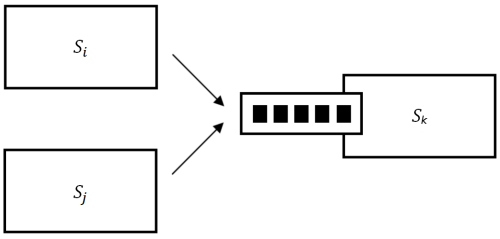

Figura 11: Unión

Si bien lo presentado anteriormente para el *pipeline lineal* puede aplicarse a los *no lineales*, se debe tener en cuenta que al trabajar con estos *pipelines* el *tipo de dependencia* es de suma importancia. Para el caso de una bifurcación, una dependencia al menos *strong* indicará un comportamiento de copia, donde S_j y S_k leerán siempre el valor generado en S_i. Para el caso de la unión, una dependencia al menos *biyectiva* generará un modelo vacío, que implica que no habrán trazas que puedan satisfacer la especificación.

2.4. Extensiones

En esta sección se presentarán modificaciones realizadas al framework que hará mas claro el código resultante.

2.4.1. SEIDE - Applicable

En un caso donde aplica la condición de *freshness*, el código presentado para el *event handler* del consumidor se muestra en Código 8. El método `execute` es el que se ejecuta al consumir el dato y como se puede ver, tiene la siguiente forma:

```
public RoutingOutcome execute(Message message) {
   if (applies) {
       doSomething()
   }
}
```

En éste código se puede ver que se mezcla la condición sobre si aplica la condición de *freshness* y el código propiamente a ejecutar. Para tener una traducción más clara, se extiende SEIDE agregando la interfaz `Applicable`.

```
public interface Applicable<T extends Message> {
    boolean apply(T message);
}
```

Si un *event handler* extiende de esta interfaz, deberá implementar el método *apply*. SEIDE ejecutará primero este método y en caso que devuelva *true*, ejecutará el código del `execute`. Esta extensión tiene como objetivo desacoplar la funcionalidad de la precondición (verificar si debe ser ejecutado) de la ejecución propiamente dicha.

El caso del Código 8, puede mejorarse ahora separando el código que valida el evento del de la ejecución propiamente dicha.

Listing 10: Productor-Consumidor EH_R con Freshness - Applicable

```
1  public class ReaderEventHandler
2      implements DispatcherAware,
3                 EventHandler<ProducerFreshnessMessage>,
4                 Applicable {
5
6      private Dispatcher dispatcher;
7
8      @Override
9      public RoutingOutcome execute(ProducerFreshnessMessage message) {
10         int y = R(message);
11         consume(y)
12     }
13
14     @Override
15     private boolean apply(ProducerFreshnessMessage message) {
16        return message.getFreshnessRange().isInRange();
17     }
18
19     private int R(ProducerFreshnessMessage message) {
20        return message.getX();
21     }
22
23     private void consume(int value) {
24        // legacy code
25     }
26
27     @Override
28     public void setDispatcher(Dispatcher dispatcher) {
29         this.dispatcher = dispatcher;
30     }
31 }
```

En este caso la verificación consiste solamente en la ejecución del método `isInRange`, pero en otro caso podría ser más complejo. Aquí se puede observar que queda desacoplada la verificación de si el evento está en el rango apropiado de la ejecución del mismo.

2.4.2. SEIDE - DispatcherRegistrationAware

Generalmente cuando se desea inicializar un objeto esto es realizado en el constructor. Un *event handler* es un objeto como cualquier otro y en su versión más simple está compuesto por su constructor y el método `execute`. Se presenta aquí la interfaz `DispatcherRegistrationAware`.

```
public interface DispatcherRegistrationAware {
    void register();
}
```

Si un *event handler* extiende de esta interfaz, deberá implementar el método `register`. SEIDE ejecutará este método luego de haberlo construido, pero antes de enviar el primer evento al mismo.

¿Qué diferencia hay entonces entre un constructor y el método `register` ? Si el código de inicialización - en el constructor o en el `register` - consiste en asignación de variables o código sencillo, la respuesta es ninguna. En cambio, si la inicialización consiste en algo más complejo como por ejemplo la creación de *threads* antes de comenzar a recibir eventos, éstos no deberían ser creados en el constructor puesto que no es correcto crear *threads* cuyo padre es un objeto que todavía no está creado (se entiende que el objeto está creado cuando termina de ejecutarse su constructor). En éste caso los *threads* deberán ser creados en el `register`, más adelante se presentarán casos donde sucede esto.

Completando el caso del código 7 del *event handler* del productor-consumidor EH_W con freshness se obtiene

Listing 11: Productor-Consumidor EH_W con Freshness - Register

```
1  public class WriterEventHandler
2      implements DispatcherAware, EventHandler<InitMessage>,
3      DispatcherRegistrationAware {
4
5      private Dispatcher dispatcher;
6      private int minRange;
7      private int maxRange;
8
9      @Override
10     public RoutingOutcome execute(InitMessage message) {
11         while (true) {
12             int x = produce();
13             W(x);
14         }
15         return null;
16     }
17
18     private int W(int value) {
19         ProducerFreshnessMessage message = new ProducerFreshnessMessage();
20         FreshnessRange freshnessRange = new FreshnessRange(this.minRange,
21                                                            this.maxRange);
22         message.setFreshnessRange(freshnessRange);
23         message.setX(value);
24
25         Event event = new Event("READER", message);
26         dispatcher.execute(event);
27     }
28
29     @Override
30     public void register() {
31         this.minRange=0;
32         this.maxRange=100;
33     }
34
35     private int produce() {
36         // legacy code
37     }
38
39     @Override
40     public void setDispatcher(Dispatcher dispatcher) {
41         this.dispatcher = dispatcher;
42     }
43 }
```

2.5. Templates

2.5.1. Caso 1

En esta sección se caracterizará el código de los *eventhandler* resultante de las transformaciones. Esta caracterización estará relacionada con el caso que se ha traducido desde *FXML* como veremos a continuación.

Analizando en detalle la parte del consumidor del código 2 se puede observar el código que se muestra a continuación. Este código corresponde a h_i del comienzo esta sección.

```
seq {
   R : y = x
   consume(y)
}
```

seq
read
process

Básicamente se puede ver la *lectura* de la variable compartida x y luego alguna clase se proceso con ese valor. En este caso en particular no se hace nada más con el valor leído luego de haber sido procesado.

Dado que *SEIDE* evita que se tenga que escribir el código que circunda h_i y que *JAVA* hace que el `seq` esté implícito, se puede caracterizar entonces el código anterior dividiéndolo en dos partes **[*read,process*]**. El código es el mostrado anteriormente en Código 4.

2.5.2. Caso 2

¿Qué pasaría si este consumidor fuese productor para otro *stage*? O en otras palabras, si éste fuera un *stage* intermedio en un *pipeline*. A continuación se analiza el caso.

```
seq {
   R : y = x1
   z = consume(y)
   W : x2 = z
}
```

seq
read
process
write

Aquí se puede observar la *lectura* de la variable compartida *x1*, un proceso con este valor y luego la *escritura* de la variable compartida *x2* con el resultado del proceso.

De la misma manera que el caso anterior podemos entonces caracterizar este caso dividiéndolo en tres partes **[*read,process,write*]** ignorando todo lo que no es h_i. La parte del código resultante para el método `execute()` es la que se muestra a continuación.

Listing 12: Traducción Caso 1 y 2

```
1     @Override
2     public RoutingOutcome execute(ProducerMessage message) {
3
4         int y = R(message);
5         int z = consume(y);
6         int x2 = z;
7         W(x2);
8         return null;
9
10    }
11
12    private int R(ProducerMessage message) {
13        return message.getX1();
14    }
15
16    private void W(int value) {
17        X2Message x2Message = new X2Message();
18        x2Message.setX2(value);
19        Event event = new Event("NEXTEVENT", x2Message);
20        dispatcher.execute(event);
21    }
```

En estos dos casos se puede observar que un *stage* recibe un evento, lo procesa y eventalmente se envía un evento a otro *stage*, son casos en los que está bien marcada la diferencia entre las etapas de lectura, proceso y escritura, pero ésto no siempre es así.

2.5.3. Caso 3

¿Qué sucede si por ejemplo se desea realizar un proceso (paralelizable) sobre los elementos de una *lista* o *array*? En este caso hay dos opciones:

- Pasar toda la lista de un *stage* a otro.
- Recorrer los elementos y pasar cada uno de un *stage* a otro.

En el primer caso, salvo que en siguiente *stage* se implemente una solución específica, no se estará aprovechando que el proceso es paralelizable. En cambio, en el segundo cada elemento de la lista podrá ser procesado en paralelo (dependiendo la cantidad de *threads* en el *stage*).

A continuación se caracteriza este último caso.

```
seq {
    R : y = x1
    for(z in y) {
       W : x2 = z
    }
}
```

seq
read
for(process)
write

En este caso las variables y y $x1$ son de algun tipo iterable y su asignación se corresponde con la etapa de lectura, el `for()` se corresponde con el proceso y por último, la asignación de $x2$ corresponde a la escritura. Aquí se puede ver que no está tan marcada la separación entre cada etapa que tiene la siguiente forma $\boldsymbol{read[,process,write]}^*$

La parte del código resultante para el método `execute()` es la que se muestra a continuación.

Listing 13: Traducción Caso 3

```
1     @Override
2     public RoutingOutcome execute(ProducerMessage message) {
3
4         List<Integer> y = R(message);
5         for(Integer z : y) {
6            int x2 = z;
7            W(x2);
8         }
9         return null;
10
11    }
12
13    private List<Integer> R(ProducerMessage message) {
14        return message.getX1();
15    }
16
17    private void W(int value) {
18        X2Message x2Message = new X2Message();
19        x2Message.setX2(value);
20        Event event = new Event("NEXTEVENT", x2Message);
21        dispatcher.execute(event);
22    }
```

Más adelante se presentará un ejemplo donde se puede ver este caso.

La inserción de código mencionada anteriormente en algunos casos puede ser fácilmente dividida entre **adaptadores entre negocio e infraestructura** (métodos `R(message)` y `W(value)` en el código, no confundir con los *labels*) y el **código propiamente de negocio**; esto puede verse en el código 12 para los casos 2.5.1 y 2.5.2 dónde cada parte está claramente separada. La separación para el caso 2.5.3 como puede verse en el código 13 no es tan sencilla puesto que el **código de negocio** y los **adaptadores** están unidos por alguna variable ligada.

3. Mapeo InProc

A continuación se presenta un nuevo problema, imprimir los números primos de la secuencia de Fibonacci. En el siguiente código *FXML* especifica este problema.

Listing 14: FXML Fibonacci-Prime-Print

```
1  dep F [bij]--> P
2  dep P [bij]--> PR
3
4  var int x1
5  var int x2
6  par
7    fib:
8      for_all(int i=0; i<N1; i++) {
9         while(true) {
10           seq {
11               x = R()                                  // read
12               f = fibonacci(x)                         // process
13               F: x1 = f                                // write
14        }
15     }
16   prime: for_all(int i=0; i<N2; i++){
17              while(true) {
18                  seq {
19                      P : y = x1                        // read
20                      boolean isPrime = true;
21                      for_all(int i=2;i<y;i++) {
22                         if (y % i == 0) {
23                            isPrime = false;
24                         }
25                      }
26                      if (isPrime) {
27                          x2 = y                        // write
28                      }
29                  }
30              }
31         }
32   print: for_all(int i=0; i<N3; i++){
33              while(true) {
34                  seq {
35                      PR : z = x2                      // read
36                      print(z)                         // process
37                  }
38              }
39          }
```

En este caso no importa cuál es el valor que toma inicialmente `x`, quién lo genera o desde dónde se recibe. El código especifica que se genera el $x-$ésimo número de la secuencia de Fibonacci, se verifica su primalidad y luego se lo imprime en caso de ser primo.

Seguramente el lector no hubiera escrito este código inicialmente si le hubieran pedido especificar el mismo problema, la idea es que cada p_i siga siendo como fue especificado al comienzo de la sección 2. En este caso los *pNodes* etiquetados con `fib` y `print` cumplen con lo pedido 2.5.2 y 2.5.1 respectivamente, pero `prime` no, puesto que se pueden observar dos `for_all` (líneas 16 y 21) anidados y este caso no está contemplado.

Una solución a este problema consiste en transformar el anidamiento en un mapeo lineal, esto se logra partiendo este último *pNode* en un productor y un consumidor dado que *SEDA* presenta un modelo plano donde todos los *stages* que componen el *pipeline* están al mismo nivel. El productor estará encargado de generar los $y-2$ números a dividir y el consumidor estará encargado de realizar las divisiones. A continuación se muestra como quedaría en *FXML* la transformación del *pNode* `prime`. Se excluyen `fib` y `print` que no sufren modificación alguna.

Listing 15: FXML Fibonacci-Prime-Print Transformado

```
1
2   prime_p: for_all(int i=0; i<N2_1; i++){
3               while(true) {
4                  seq {
5                     P : y = x1
6                     for(int i=2;i<y;i++) {
7                        x2 = new Tuple(y,i)
8                     }
9                  }
10              }
11          }
12  prime_c: for_all(int i=0; i<N2_2; i++){
13              while(true) {
14                 seq {
15                    Tuple t = x2
16                    boolean isDivisible = (t.first % t.second == 0)
17
18                    // put a new value if absent.
19                    // returns the value if present (null if was absent)
20                    Tuple lastValue = map.putIfAbsent(y,
21                                          new Tuple(isDivisible, 1))
22
23                    if (lastValue != null) {
24                        //update tuple <newValue, iteration++>
25                        lastValue.first = lastValue.first || isDivisible
26                        lastValue.second++
27
28                        //recieved n-2 messages and no divisor
29                        //send y to print
30                        if ((lastValue.second==t.first-2) &&
31                            (!lastValue.first)) {
32                            x3 = t.first // y
33                        }
34
35                        // clean map
36                        if (lastValue.secont==t.first-2) {
37                           map.remove(t.first) // y
38                        }
39                    }
40                    else
41                    {
42                        // nothing to do, added <y, (isDivisible,1)>
43                    }
44                }
45             }
46          }
```

Si bien la arquitectura SEDA no está orientada al cálculo de un valor para una entrada de datos específica, para introducir esta estrategia es útil este problema. En este caso, para saber si un número es primo se utiliza un algoritmo burdo, realizar $y-2$ divisiones, lo importante aquí es que hay un número *finito* de divisiones. Aquí se presenta un segundo problema, el dominio de iteración. *SEDA* es una arquitectura orientada a servicio con dominios infinitos y aquí se presenta un caso donde se debe realizar una operación (asignaciones o bien enviar eventos) una cantidad finita de veces $(y-2)$, más aún, se podría estar verificando de manera concurrente si el 5 y el 13 son primos, esto implicaría por ejemplo estar verificando de manera simultánea si $\{(2 \bmod 5) = 0\}$ y si $\{(2 \bmod 13) = 0\}$.

Para soportar este caso entonces no sólo se deberá enviar el número a dividir sino tambien el número que se desea probar si es primo. De este número se puede además inferir la cantidad de operaciones esperadas $(y-2)$. Como el *stage* consumidor, el que va a realizar las divisiones, es uno sólo y puede estar recibiendo divisiones para comprobar distintos valores deberá mantener alguna clase de estado o estructura que permita recolectar la informacion de eventos anteriores relacionados con el número que se desea verificar.

La estructura que mantiene el estado descripto es un mapa que va acumulando la cantidad de asignaciones recibidas y las divisiones para un cierto número (key=k). Cuando se reciben $y-2$ asignaciones si no hubo ningún divisor se envía el valor y a `print`. Independientemente de si hay divisores o no, al recibir la asignación $(y-2)$-ésima se limpia el mapa.

Como se puede observar, la traducción a realizar para soportar éste caso si bien no es dificil, no es trivial, además hay que definir estructuras para mantener estado dentro de cada *pNode* o *stage*, estructuras que tienen que ser *thread-safe* incurriendo posiblemente en operaciones bloqueantes.

En este caso, si y es suficientemente grande, aunque el sistema no se degradaría, tendríamos una sobrecarga de eventos encolándose para hacer sólo una división en el consumidor (además de los accesos al mapa). Aunque el código quedaría realmente simple, la utilización de esta estrategia parece excesiva puesto que *SEIDE* también agrega tiempo al procesamiento.

Este es un caso donde el *mapeo lineal* parece no ser una buena idea, se propone entonces un nuevo mapeo, el *mapeo InProc*.

Java 6 no cuenta con una versión de la instrucción `for()` paralela. Se presenta en este trabajo una versión de `for[i]()` y de `for[each]()` construída en base al `for()` convencional, `ExecutorService`, `Future` y `CountDownLatch` de Java. A continuación se muestra la implementación de cada una de ellas.

Listing 16: Parallel.ForEach

```
1  public static <T> void ForEach(Iterable<T> parameters,
2                                 final ForLoopBody<T> forLoopBody) {
3
4      ExecutorService executor = Executors.newFixedThreadPool(iCPU);
5      List<Future<?>> futures = new LinkedList<Future<?>>();
6
7      for (final T param : parameters) {
8          Future<?> future = executor.submit(new Runnable() {
9               public void run() {
10                  forLoopBody.run(param);
11              }
12         });
13
14         futures.add(future);
15     }
16
17     for (Future<?> f : futures) {
18        try {
19             f.get();
20        } catch (InterruptedException e) {
21             throw new RuntimeException(e);
22        } catch (ExecutionException e) {
23             throw new RuntimeException(e);
24        }
25     }
26
27     executor.shutdown();
28 }
```

Listing 17: Parallel.For

```
1  public static void For(int start, int stop,
2                         final ForLoopBody<Integer> forLoopBody) {
3
4       ExecutorService executor = Executors.newFixedThreadPool(iCPU);
5       final CountDownLatch cdl = new CountDownLatch(Math.abs(stop - start));
6
7       for (int i = start; i < stop; i++) {
8           final Integer k = i;
9           executor.submit(new Runnable() {
10               public void run() {
11                   forLoopBody.run(k);
12                   cdl.countDown();
13               }
14           });
15      }
16
17      try {
18         cdl.await();
19      } catch (InterruptedException e) {
20          throw new RuntimeException(e);
21      }
22
23      executor.shutdown();
24 }
```

Se puede paralelizar entonces la ejecución de un proceso sin utilizar SEDA cuando el caso lo amerite.

A continuación se presenta otra transformación (en vez partir `prime` en un productor y consumidor) utilizando el *mapeo InProc*.

Listing 18: Prime InProc

```
1  public class PrimeEventHandler
2      implements DispatcherAware,
3                 EventHandler<Message> {
4
5      private Dispatcher dispatcher;
6
7      @Override
8      public RoutingOutcome execute(Message message) {
9
10         int k = R(message);
11
12         final int finalK = k;
13         final Context ctx = new Context();
14
15         Parallel.For(2, k-2, new ForLoopBody<Integer>() {
16
17                 @Override
18                 public void run(Integer i) {
19                     ctx.addValue(finalK % i == 0);
20                 }
21             });
22
23         if (!ctx.getValue()) {
24                 W(k);
25         }
26     }
27
28     private int R(Message message) {
29        return message.getK();
30     }
31
32     private int W(int value) {
33
34         Message message = new Message();
35         message.setK(value);
36
37         Event event = new Event("PRINT", message);
38         dispatcher.execute(event);
39     }
40
41     @Override
42     public void setDispatcher(Dispatcher dispatcher) {
43         this.dispatcher = dispatcher;
44     }
45
46 }
```

Listing 19: Prime InProc Context

```
1  public class Context {
2
3      private AtomicBoolean hasDivisor = new AtomicBoolean(false);
4
5      public boolean getValue() {
6          return this.hasDivisor.get();
7      }
8
9      public void addValue(boolean value) {
10         this.hasDivisor.compareAndSet(value==true, true);
11     }
12 }
```

Básicamente el código que hace todo el proceso es el comprendido desde la línea 15 a 21. En éste caso se realizan las mismas $k-2$ divisiones pero no se mandan mensajes ni se utiliza la infraestructura de *SEIDE*. La ejecución del proceso será más económico y en comparación el código a generar es más sencillo.

4. Mapeo Distribuido

Cuando el tiempo de ejecución de un evento es alto, parecería razonable agregar *threads* al *stage* que lo va a procesar si hay procesadores disponibles en la computadora que ejecuta el programa. Muchas veces esto no es así, no se puede simplemente agregar procesadores a una computadora, pero lo que sí se puede es conseguir una computadora adicional para realizar el proceso.

Con SEDA distribuido la idea es poder tener réplicas de algunos *stages* en otras computadoras y proveer un mecanismo para que se distribuyan los eventos de manera transparente como si estuvieran en la misma computadora.

Estas réplicas pueden ser especificadas formalmente en *FXML* con hiper-dependencias(OR)[5][6].

Listing 20: Ejemplo Hiper-Dependencia OR

```
1  dep S1 [or]--> S2R1, S2R2
2  dep S2W1,S2W2 [or] --> S3
3  par
4    Stage1:
5             S1 : x1 = process()
6    Stage2a: ...
7             S2R1 : y = x1    \\ read
8                    ...       \\ process
9             S2W1 : x2 = y    \\ write
10
11   Stage2b: ...
12            S2R2 : y = x1    \\ read
13                   ...       \\ process
14            S2W2 : x2 = y    \\ write
15
16   Stage3: S3 : z = x2.
```

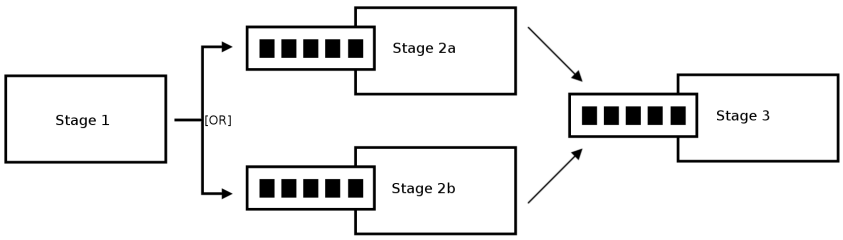

Figura 12: Hiper-Dependencia OR y SEDA

La hiper-dependencia OR hace que la dependencia entre las escrituras y lecturas puedan suceder entre *S1* y *S2R1* o *S1* y *S2R2*.

Para lograr ésto se utilizará una biblioteca llamada *Hazelcast* [16]. Esta es una herramienta que permite utilizar estructuras de datos similares a las de JAVA de manera concurrente y distribuidas en varias JVMs (en la misma computadora o en varias) de manera transparente. De esta manera (y sin entrar en detalle sobre la implementación de la herramienta) estaría resuelto el tema de la comunicación en el cluster utilizando colas compartidas.

En la figura 13 se muestra un esquema donde se define un *pipeline* distribuido. Esto puede representarse con dos instancias de la JVM ejecutádose en la misma computadora o bien en dos computadoras distintas.

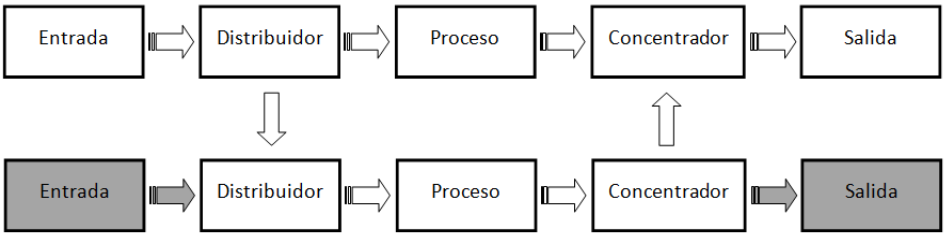

Figura 13: Pipeline Distribuido

Los *stages* que están grisados se encuentran desactivados en el segundo *pipeline*. Esto significa que hay una única entrada de datos y una única salida pero esto no tiene que ser siempre así, dependerá del problema a modelar pudiendo haber múltiples orígenes de información y múltiples salidas.

Ambos *pipelines* están conectados por dos *stages* especiales: **Distribuidor** y **Concentrador**. Estos dos *stages* son muy parecidos, ambos tienen dentro una cola que será compartida para todos los *stages* distribuidores y otra cola que será compartida entre todos los concentradores.

Estos *stages*, distribuidores o concentradores, tienen como como su nombre lo indica la *única* responsabilidad, reenviar o recolectar eventos.

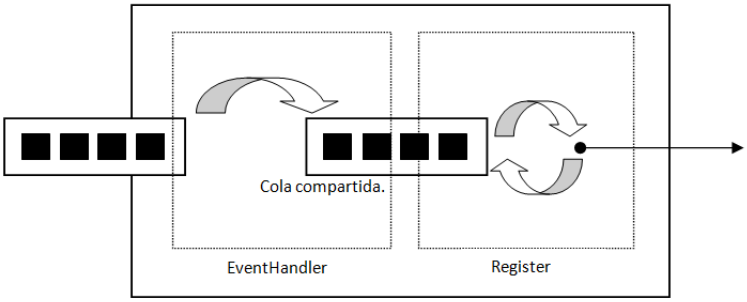

Figura 14: Stage de un Distribuidor

En la figura 14 se puede observar cómo está construido un *stage* distribuidor, el *event handler* tomará eventos de la cola del *stage* y los ofrecerá a la cola compartida. Luego de haber construido el *stage* (en la etapa de registro -DispatcherRegistrationAware-), se lanza una cierta cantidad de *threads* que consumirán los elementos de la cola compartida y los reenviarán al próximo *stage*.

La cola compartida es una *IQueue* de Hazelcast, esto hace que los elementos ofrecidos en el *pipeline* superior pueden ser tomados indistintamente en mismo *stage* o en el inferior para luego ser procesados. Si no hay elementos disponibles en la cola, al intentar tomar uno (`take()`) el *thread* quedará bloqueado esperando a que algún *event handler* ofrezca algun elemento a la misma.

La implementación del *stage* concentrador es muy parecida a la del distribuidor. La única diferencia es que solamente se consumirán elementos de la cola compartida (código lanzado en el evento de registro del *stage*) en los *pipelines* que deben continuar con la ejecución (en este caso el superior).

4.1. Distribuidor y Concentrador

Aquí se presenta el código del *Distribuidor* y del *Concentrador*, el objetivo es mostrar que son *stages* con *eventhandlers* que son construídos como cualquier otro.

Listing 21: Distributor Event Handler

```
1
2  public class DistributorEventHandler implements DispatcherAware,
3    EventHandler<Message>, Applicable<Message>, DispatcherRegistrationAware {
4
5    private Dispatcher dispatcher;
6
7    private String nextEventName;
8
9    private HazelcastInstance hazelcastInstance;
10
11   //hazelcast IQueue
12   private IQueue<Message> distributorQueue;
13
14   private static final String DISTRIBUTOR_QUEUE_NAME =
15      "distributor_" + DistributorEventHandler.class.getName();
16
17   @Override
18   public void register() {
19
20      this.distributorQueue = this.hazelcastInstance.getQueue(DISTRIBUTOR_QUEUE_NAME);
21
22      int threads = 1;
23
24      ThreadPoolExecutor tpe = new ThreadPoolExecutor(threads, threads, 1,
25          TimeUnit.DAYS, new LinkedBlockingQueue<Runnable>());
26
27      for (int i = 0; i < threads; i++) {
28
29          tpe.submit(new Runnable() {
30
31              @Override
32              public void run() {
33                 try {
34                   while (true) {
35                      Message m = DistributorEventHandler.this.distributorQueue.take();
36                      Event event = new Event(DistributorEventHandler.this.nextEventName, m);
37                      DistributorEventHandler.this.dispatcher.execute(event);
38                   }
39                 } catch (InterruptedException e) {
40                    throw new RuntimeException(e);
41                 }
42              }
43          });
44      }
45   }
46
47   @Override
48   public boolean apply(Message message) {
49      return true;
50   }
51
52   @Override
53   public RoutingOutcome execute(Message message) {
54      distributorQueue.offer(message);
55      return null;
56   }
57 }
```

El *eventhandler* del *joiner* es similar, la diferencia está en que hay que indicar cuál o cuáles son los *stages* que reenvian mensajes. Esto se puede ver en la Figura 13 en el *stage* **Salida** que está grisado.

Listing 22: Joiner Event Handler

```
1  public class JoinerEventHandler implements DispatcherAware,
2    EventHandler<Message>, Applicable<Message>, DispatcherRegistrationAware {
3
4    private Dispatcher dispatcher;
5
6    private String nextEventName;
7
8    private HazelcastInstance hazelcastInstance;
9
10   //hazelcast IQueue
11   private IQueue<Message> joinerQueue;
12
13   private boolean joinEventHandlerInstance;
14
15   private static final String JOINER_QUEUE_NAME =
16      "joiner_" + JoinerEventHandler.class.getName();
17
18   @Override
19   public void register() {
20
21      this.joinerQueue = this.hazelcastInstance.getQueue(JOINER_QUEUE_NAME);
22
23      if (joinEventHandlerInstance) {
24
25         int threads = 1;
26
27         ThreadPoolExecutor tpe = new ThreadPoolExecutor(threads, threads, 1,
28             TimeUnit.DAYS, new LinkedBlockingQueue<Runnable>());
29
30         for (int i = 0; i < threads; i++) {
31
32             tpe.submit(new Runnable() {
33
34                 @Override
35                 public void run() {
36                    try {
37                      while (true) {
38                         Message m = JoinerEventHandler.this.joinerQueue.take();
39                         Event event = new Event(JoinerEventHandler.this.nextEventName, m);
40                         JoinerEventHandler.this.dispatcher.execute(event);
41                      }
42                    } catch (InterruptedException e) {
43                       throw new RuntimeException(e);
44                    }
45                 }
46             });
47         }
48      }
49   }
50
51   @Override
52   public boolean apply(Message message) {
53      return true;
54   }
55
56   @Override
57   public RoutingOutcome execute(Message message) {
58      joinerQueue.offer(message);
59      return null;
60   }
61 }
```

Al igual que con el caso *InProc*, este mapeo puede utilizarse en cualquier caso, lo importante es poder elegir cuando conviene paralelizar utilizando *SEDA*, cuando conviene utilizar *InProc* y cuando *SEDA Distribuido* de manera de obtener resultados en el menor tiempo posible, sin generar un exceso de mensajes o un exceso de tareas costosas.

4.2. Ejemplo

En la sección 3 se presentó el problema de imprimir los números primos de la secuencia de Fibonacci, se utilizará el mismo ejemplo para mostrar *SEDA Distribuido*.

Si se planteara desde una especificación en *FXML* una traducción a *SEDA*, la figura 15 ilustra como quedaría el *pipeline* asociado a las dependencias del caso. Estas dependencias podrían ser *WEAK*, *STRONG*, etc.

Figura 15: Seda, Fibonacci - Prime - Print

En el caso de *SEDA Distribuido*, el *pipeline* quedaría de la siguiente forma:

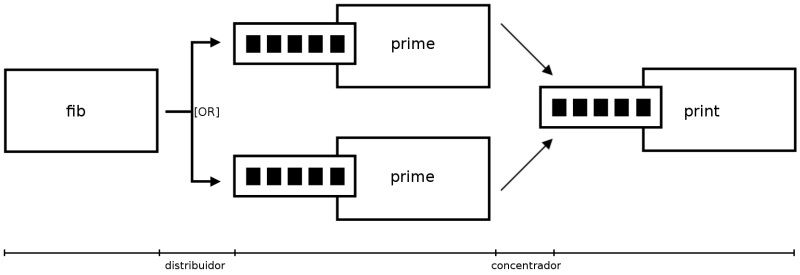

Figura 16: Seda Distribuido, Fibonacci - Prime - Print

Listing 23: Ejemplo Hiper-Dependencia OR

```
1  dep F [or, bij]--> PR1, PR2
2  dep PW1, PW2 [or, bij]--> PR
3  par
4    fib: ...                \\ tiene F
5
6    prime1: ...
7            PR1 : y = x1    \\ read
8                  ...       \\ process
9            PW1 : x2 = y    \\ write
10
11   prime2: ...
12           PR2 : y = x1    \\ read
13                 ...       \\ process
14           PW2 : x2 = y    \\ write
15
16   print: ...              \\ tiene a PR
```

En este caso los *pNodes* `prime1` y `prime2` son iguales, ambos leen de la variable compartida `x1` (cada valor es leído sólo por alguno de los dos). Existe en este caso una bifurcación. Luego, `prime*` enviará un evento a `print` en caso de ser primo. En este caso no existe sincronización alguna, `print` podrá recibir eventos de cualquier *stage* `prime*` en cualquier orden.

Un evento que se envía de `fib` a `prime` es recibido por **un solo stage**. De la misma manera que en los casos anteriores, el usuario deberá escribir sólo el código de negocio en los *stages* (fib, prime, print), la diferencia en este caso es como se encadenarán los *stages*. En este caso, en vez de hacer que las dependencias estén en `fib` → `prime` → `print`, deberán estar de la siguiente manera: `fib` → `distribuidor` → `prime` → `concentrador` → `print`.

5. Caso de Estudio

A continuación se presenta un caso donde se podrá ver múltiples conceptos enunciados anteriormente, **Reverse Links**. El problema consiste básicamente en revisar un conjunto de archivos `html`, parsearlos y ver todos sus links salientes, el resultado final debe ser un mapa que tenga por cada link, la lista de los archivos `html` que lo referencian.

Se presentará además aquí un ejemplo que completa lo presentado en la sección Templates 2.5

Se define entonces:

```
F : Lista de archivos
f : Archivo
L : Lista de links
l : link
map : mapa donde se almacenará la información
```

Dado que no es importante con respecto a la motivación del caso, se asume que el mapa (`map`) está inicializado y se hace un abuso en la semántica del método `get()`. Al realizar un `get` de una *key* inexistente, se devolverá una lista vacía y se la asociará a esa *key* (en vez de `null`).

Listing 24: FXML Reverse Links

```
1  var Map<String, Set<String>> map
2
3  F = readFiles();
4  for_all (i=0; i < F.size; i++) {
5     f = F[i];
6     L = getLinks(f);
7
8     for_all (j=0; j<L.size; j++) {
9        l = L[j];
10       map.get(l).add(f);
11    }
12 }
```

La función `getFiles` es una función *legacy* que retorna la lista de archivos a analizar, además la función *legacy* `getLinks` retorna la lista de links salientes de un archivo `html`. A continuación se presenta una transformación donde se paraleliza la obtención de los links con la inserción en el mapa.

Listing 25: FXML Reverse Links - Transformación

```
1  dep biyectiva p -> q
2  var String f
3  var List<String> L
4  var Map<String, Set<String>> map
5
6  F = readFiles();
7  par
8     for_all (i=0; i < F.size; i++) {
9        p: seq {
10          f = F[i];                          //write
11          L = getLinks(f);                   //process+write
12       }
13    }
14    for_all (i=0; i < F.size; i++) {
15       q: seq {
16          L1=L;                              //read
17          f1=f;                              //read
18       }
19       for_all (j=0; j<L1.size; j++) {        //process
20          l1 = L1[j];                        //process
21          map.get(l1).add(f1);               //process
22       }                                     //process
23    }
```

Nuevamente, se puede paralelizar aún más este caso, dividiendo la selección de los archivos del cálculo de los links de cada uno.

Listing 26: FXML Reverse Links - Segunda Transformación

```
1  dep biyectiva p -> q
2  dep biyectiva q -> r
3  var String f, f1
4  var List<String> L1
5  var Map<String, Set<String>> map
6
7  F = readFiles();
8  par
9     for_all (i=0; i < F.size; i++) {
10       p: seq {
11          f = F[i];                          //write
12       }
13    }
14    for_all (i=0; i < F.size; i++) {
15       p: seq {
16          f1 = f;                            //read
17          L = getLinks(f);                   //process
18          L1 = L;                            //write
19       }
20    }
21    for_all (i=0; i < F.size; i++) {
22       q: seq {
23          L2=L1;                             //read
24          f2=f1;                             //read
25       }
26       for_all (j=0; j<L2.size; j++) {        //process
27          l2 = L2[j];                        //process
28          map.get(l2).add(f2);               //process
29       }                                     //process
30    }
```

Por otra parte el código presentado no cumple con la estructura presentada al comienzo de la sección 2 ni con alguna de las formas presentadas en 2.5, puesto que el `for_all` de la línea 21 contiene otro en la línea 26. Este problema se puede solucionar con una tranformación más para la primer parte y con alguna de las siguientes maneras para la segunda:

- Partiendo el caso en Productor-Consumidor
- Utilizando mapeo *InProc*
- Secuencializando

En la siguiente transformación se muestra el resultado final, donde se ajusta el problema a la estructura deseada y en el que se ha decidido secuencializar el último `for_all`. Por otra parte, si simplemente se encapsula el código anterior para el primer *pNode* en un `for_all(...) { while (true) {`h_i`} }`, se estaría generando una cantidad incorrecta de iteraciones (`F.size()` $\neq \infty$).

Listing 27: FXML Reverse Links - Tercer Transformación

```
1  dep biyectiva p -> q
2  dep biyectiva q -> r
3  dep (i,i) wf -> rf
4  var String f, f1
5  var List<String> L1
6  var Map<String, Set<String>> map
7  par
8     wf: F=readFiles();                        //write
9     for_all (i=0; i<N1; i++) {
10       while(true) {
11          rf : auxF = F;                     //read
12          for (j=0; j < F.size; j++) {       //process
13             p: seq {                        //process
14                 f = auxF[i];                //write
15             }
16          }
17       }
18    }
19    for_all (i=0; i < N2; i++) {
20       while (true) {
21          p: seq {
22             f1 = f;                         //read
23             L = getLinks(f);                //process
24             L1 = L;                         //write
25          }
26       }
27    }
28    for_all (i=0; i < N3; i++) {
29       while (true) {
30          q: seq {
31             L2=L1;                          //read
32             f2=f1;                          //read
33          }
34          for (j=0; j<L2.size; j++) {        //process
35              l2 = L2[j];                    //process
36              map.get(l2).add(f2);           //process
37          }                                  //process
38       }
39    }
```

Para solucionar el problema del rango de iteración se agregará una dependencia más de tipo (i,i). Esta dependencia entre `wf` y `rf` hace que el `for_all(...) { while (true) {...} }` sea análogo a eliminar el `while()` y fijar `N1=1`. De esta manera se mantiene el rango de iteración original y el *pNode* posee la estructura deseada.

Esta transformación deja el código a generar suficientemente modularizado para caracterizarlo de la siguiente manera **[*read,process,write*]**, básicamente la forma de las 4 partes quedaría de la siguiente manera:

$$\textbf{[}\textit{write}\textbf{]} + \textbf{[}\textit{read}\textbf{,[}\textit{process,write}\textbf{]}^*\textbf{]} + \textbf{[}\textit{read, process,write}\textbf{]} + \textbf{[}\textit{read, process}\textbf{]}$$

En este último caso se observa que al cuarto *pNode* llegan el nombre del archivo y la lista de links, esta lista es recorrida y se agrega cada elemento al mapa. Las transformaciones que han sido realizadas en *FXML* han definido la forma o caracterización (template) del caso.

¿Qué pasaría si en vez de la transformación presentada en 26, se hubiera presentado ésta?

Listing 28: FXML Reverse Links - Segunda Transformación - Version 2

```
1  dep biyectiva p -> q
2  dep biyectiva q -> r
3  var String f, f1
4  var List<String> l1
5  var Map<String, Set<String>> map
6
7  F = readFiles();
8  par
9     for_all (i=0; i < F.size; i++) {
10       p: seq {
11          f = F[i];                          //write
12       }
13    }
14    for_all (i=0; i < F.size; i++) {
15       p: seq {
16          f1 = f;                           //read
17          L = getLinks(f);                  //process
18          for_all(j=0; j<L; j++) {          //process
19             l1=L[j];                       // write
20          }
21       }
22    }
23    for_all (i=0; i < F.size; i++) {
24       q: seq {
25          l2=l1;                            //read
26          f2=f1;                            //read
27          map.get(l2).add(f2);
28       }
29    }
```

Si bien se repiten los mismos problemas del caso anterior (no se cumple con la estructura presentada al comienzo de la sección 2 ni con alguna de las formas presentadas en 2.5), en este caso, en vez de mandar un mensaje con la lista completa de links, se envían muchos mensajes cada uno con un único link. Esta transformación (en comparación con la anterior) presenta un caso donde es mayor el grado de paralelismo que se puede lograr pero con un incremento en la cantidad de mensajes.

En la siguiente transformación se muestra el resultado final, donde se ajusta el problema a la estructura deseada, se ha decidido convertir el `for_all` en un *InProc* y se agrega la dependencia (i,i) para mantener el rango de iteración.

Listing 29: FXML Reverse Links - Tercer Transformación - Version 2

```
1  dep biyectiva p -> q
2  dep biyectiva q -> r
3  dep (i,i) wf->rf
4  var String f, f1
5  var List<String> l1
6  var Map<String, Set<String>> map
7  par
8     wf: F=readFiles();                              //write
9     for_all (i=0; i<N1; i++) {
10       while(true) {
11          rf: auxF = F;                             //read
12          for (j=0; j < F.size; j++) {              //process
13             p: seq {                               //process
14                 f = auxF[i];                       //write
15             }
16          }
17       }
18    }
19    for_all (i=0; i < N2; i++) {
20       while (true) {
21          p: seq {
22              f1 = f;                               //read
23              L = getLinks(f);                      //process
24              Paralell.For(j=0; j<L; j++) {         //process
25                 l1=L[j];                           //write
26              }
27          }
28       }
29    }
30    for_all (i=0; i < N3; i++) {
31       while (true) {
32           q: seq {
33              l2=l1;                                //read
34              f2=f1;                                //read
35              map.get(l2).add(f2);                  //process
36           }
37       }
38    }
```

Este caso quedaría caracterizado de la siguiente manera:

$$[write] + [read,[process,write]^*] + [read,[process,write]^*] + [read,\ process]$$

6. Conclusiones

La intención de este trabajo no ha sido sólo resolver un problema técnico, sino que además se ha deseado mostrar cómo se integran dos modelos conceptuales tratando de explotar lo mejor de cada uno para hacer la ingeniería de software concurrente más simple y menos suceptible a errores. La idea consistió en basarse en los *patterns* abstractos especificados en FXML para luego implementarlos en SEIDE, maximizando así los beneficios de la plataforma (claridad, robustez y facilidad para modelar) reduciendo la tarea de programación a prácticamente **insertar código** en algunos lugares determinados y de una manera determinada; usando FXML como formalismo "guía" para poner de manifiesto los aspectos comportamentales esenciales en lo que respecta el nivel de paralelismo existente en el problema (para después llevarlo a un paralelismo explotable en SEIDE según diferentes criterios de performance).

7. Trabajo Futuro

Este trabajo ha consistido en presentar y caracterizar problemas para luego, encontrar una solución adecuada. Quedan como línea de investigación analizar algunos aspectos que han sido dejados fuera del alcance del mismo, como por ejemplo, el problema de los dominios finitos, la utilización en profundidad de hiper-dependencias de *FXML* o la escritura de un traductor *FXML* $\rightarrow$ *SEIDE*.

Referencias

[1] Shan Lu, Soyeon Park, Eunsoo Seo and Yuanyuan Zhou Learning from Mistakes - A Comprehensive Study on Real Department of Computer Science, University of Illinois at Urbana Champaign, Urbana, IL 61801 - World Concurrency Bug Characteristics

[2] Joab Jackson Nasdaq's Facebook glitch came from 'race conditions' http://www.computerworld.com/s/article/9227350

[3] Robin Milner A Calculus of Communicating Systems - ISBN: 3540102353

[4] C.A.R Hoare Communicating Sequential Processes (Prentice Hall International Series in Computing Science) ISBN: 0131532715

[5] S. Yovine, I. Assayad, F.-X. Defaut, M. Zanconi, A. Basu. A formal approach to derivation of concurrent implementations in software product lines. In Process Algebra for Parallel and Distributed Processing, M. Alexander & W. Gardner, eds., Chapman and Hall/CRC Press, Taylor and Francis Group LLC, December 2008. ISBN: 978-1-4200-6486-5.

[6] I. Assayad, V. Bertin, F. Defaut, Ph. Gerner, O. Quévreux, S. Yovine. JAHUEL: A formal framework for software synthesis. *ICFEM'05*, Manchester

[7] Michael G. Burke Rice University Houston, Texas - Kathleen Knobe Intel Corporation Hudson, Massachusetts - Ryan Newton Intel Corporation Hudson, Massachusetts - Vivek Sarkar Rice University Houston, Texas - The Concurrent Collections Programming Model - Department of Computer Science - Rice University - December 2010

[8] Threading Building Blocks (Intel® TBB) - https://www.threadingbuildingblocks.org/

[9] Apache Hadoop® - http://hadoop.apache.org/

[10] Matthew David Welsh An Architecture for Highly Concurrent, Well-Conditioned Internet Services.

[11] Matt Welsh, David Culler, and Eric Brewer SEDA: An Architecture for Well-Conditioned, Scalable Internet Services

[12] German Kondolf SEIDE Framework - http://code.google.com/p/seide/ http://github.com/germanklf/seide

[13] Apache Cassandra http://cassandra.apache.org

[14] SwiftMQ A JMS Enterprise Messaging Server - http://www.swiftmq.com

[15] LimeWire http://www.limewire.com

[16] Hazelcast - In Memory Data Grid http://www.hazelcast.com