Proceso de Propagación Histórica bajo Demanda
Software RMES

Centro de Desarrollo de Gestión Empresarial.
1 Oriente 1097 - Viña del Mar, Chile.
Fono:(56) (32)688987 - Fax:(56) (32)2684079
empresa@mes.cl

Junio, 2010

Resumen

El presente informe explicará el proceso de propagación de detenciones históricas para el software RMES. Se detallarán los puntos que hasta el momento el autor comprende a cabalidad. Secciones que no sean exhaustivamente explicadas serán completadas en el futuro cuando se tenga compresión completa de todo el proceso. Este documento esta dirigido particularmente a los actuales y futuros desarrolladores del software RMES, por lo que se utilizará en gran parte de éste la jerga técnica informática asociado al software.

Índice

1. Introducción

Uno de los procesos más importantes del software RMES es realizar la propagación de detenciones desde las raíces del árbol de jerarquía de sistemas, que representan a los equipos, hasta lo más alto de éste, que en visión sistémica representa a toda la planta en estudio. La importancia de este proceso radica en que los indicadores que se puedan obtener luego de la propagación son estimadores de gran precisión y muy representativos de la realidad, a diferencia de los estimadores probabilisticos obtenidos mediante estimación de parámetros que son muy inexactos para explicar el comportamiento de algún componente de la planta y que además no pueden ser ajustados a un rango de tiempo arbitrario.

Este proceso puede ser en línea (on line) o bajo demanda (on demand). El primero concepto se asocia a realizar el proceso cada vez que exista algún cambio que afecte al cálculo en algún punto de la planta; el segundo concepto se entiende como realizar el proceso completo sólo cuando se estime conveniente.

Este documento está enfocado a una propagación bajo demanda, es decir, realizar el proceso sólo cuando se requiera, específicamente, antes de querer obtener algún indicador de mantenibilidad histórica.

La idea en sí no es compleja, pero se necesita comprender mínimamente como se trajaban las detenciones en una planta. En general, todos los equipos en una planta presentan defectos en su funcionamiento, los cuales dentro del software son registrados como tuplas (falla o detención). Por ahora, sólo es necesario saber que una tupla como mínimo tiene una fecha de inicio y una duración. La representación gráfica de un conjunto de fallas se realiza mediante un cronograma de tiempo (en alguna "métrica de tiempo" como "horas"). Cada falla se representa con un bloque ubicado sobre el intervalo de tiempo asociado a la tupla, en donde la altura del bloque sólo es útil para hacer notar la existencia del intervalo dentro de la unidimensionalidad de este tipo de gráficos.

La visión sistémica de un planta permite ver cada componente como un compuesto, valga la redundancia, de algo "más grande", siendo este último objeto "más grande" también es un compuesto de algo superior. El ejemplo más simple es un planta con maquinarias en serie, un conjunto de máquinas alineadas, en donde la máquina siguiente en el proceso se nutre de lo entregado por la anterior. En este sencillo ejemplo se puede ver claramente que existe un gran padre que es "la planta" y un subconjunto de "hijos" que son las maquinarias. Sabemos que la planta completa se detendrá cuando cualquiera de los hijos falle. La figura 1 representa un escenario simulado de la planta en donde, las máquinas fallas en distintos momentos. Estas fallas se ven reflejadas directamente en la planta, debido a su configuración en serie.

Por ejemplo, si la planta de alguna manera tuviese sus máquinas organizadas en paralelo, el resultado sería distinto, dado que por definición la planta completa fallará sólo cuando todas las máquinas estén detenidas al mismo tiempo. La figura 2 ilustra esta situación.

Como se puede observar, la llegada de detenciones a la planta depende de la configuración de la planta, por ende, depende de criterios especiales para cada caso.

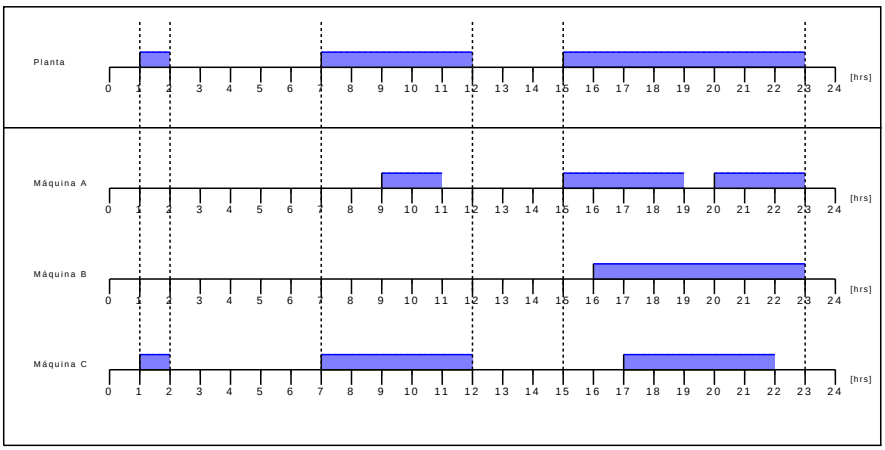

Figura 1: Detenciones de las maquinarias que suben hacia la planta en serie.

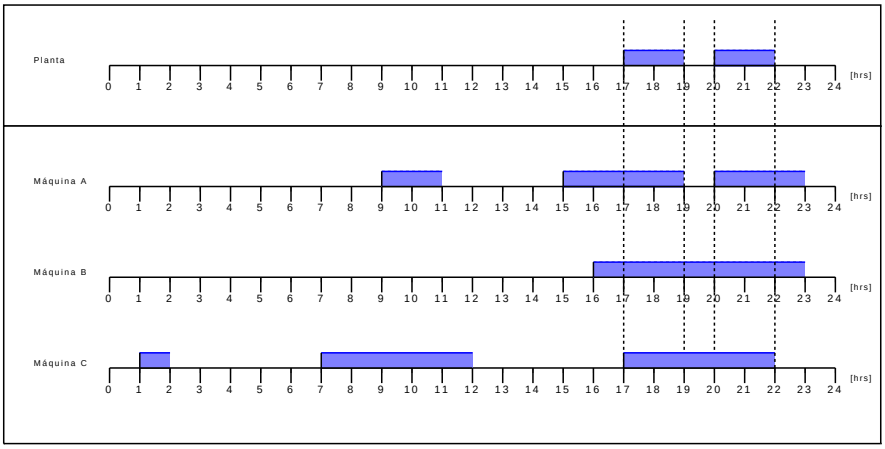

Figura 2: Detenciones de las maquinarias que suben hacia la planta en paralelo.

2. Implementación

2.1. Diagrama de Clases

Las clases que la figura 3 muestra se encuentran ubicadas en el paquete *cl.cgeconsultores.mes.main.historicCalculations* y representan la implentación realizada para la propagación bajo demanda en el software RMES.

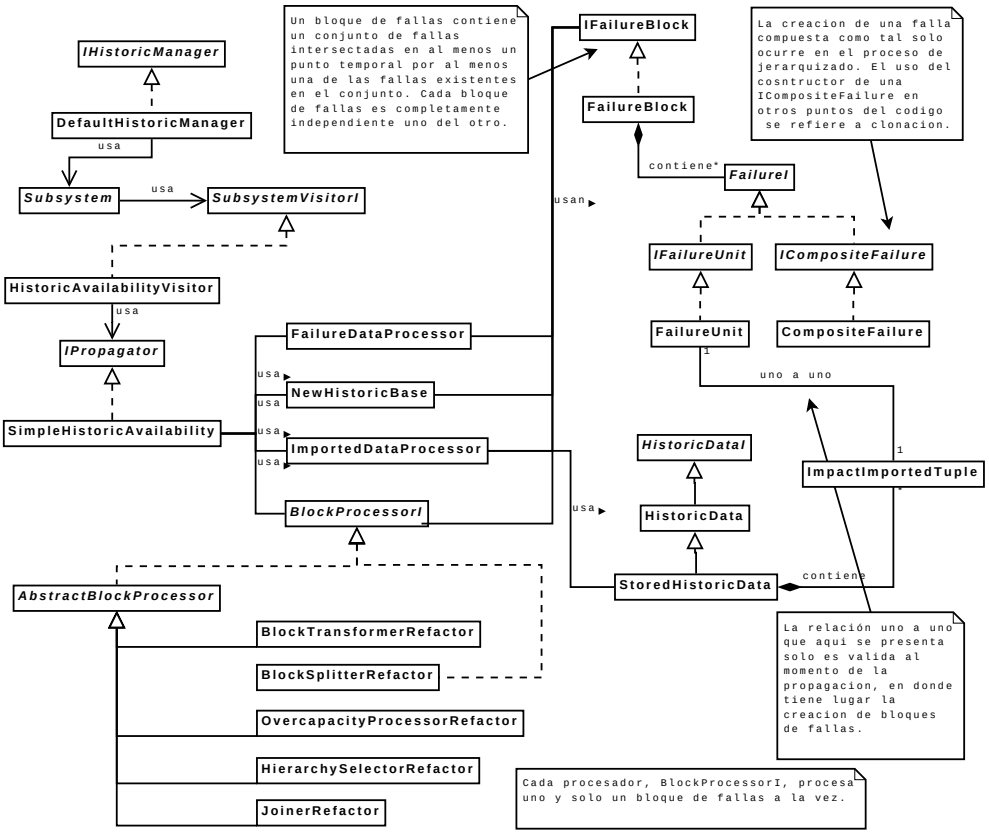

Figura 3: Diagrama de clases.

2.2. Llamada a la propagación

El proceso comienza cuando, en algún momento, se necesita realizar algún cálculo de mantenibilidad sobre uno o varios subsistemas, generalmente, sobre toda la planta. La interfaz *IHistoricManager* encapsula la llamada a la propagación con el método

```
1 public void propagateHistoricCalculations(Subsystem, Date, Date)
```

Dado que se trata de una interfaz, el método es sobreescrito en cada clase que la implemente. Este es el caso de la clase *DefaultHistoricManager*, la cual particularmente se encarga de hacer la propagación bajo demanda. En ella se puede encontrar un código muy simple:

```
1 public void propagateHistoricCalculations(
2   Subsystem rootVisit, Date from, Date to){
3   rootVisit.accept(new HistoricAvailabilityVisitor(from, to));
4 }
```

En la esquina superior izquierda de la figura 3 se representa esta relación. El objeto *rootVisit* es un *Subsystem* que generalmente es la planta completa (el subsistema contenido por la clase *Chain*). Como se puede observar, la propagación utiliza el paradigma de visitas (Patrón Visitor) y es posible utilizar cualquier nodo del árbol de subsistemas como raíz de la visita. Esta visita en particular se realiza desde las hojas del árbol hacia arriba, idea que se debe tener siempre presente a lo largo de este documento.

2.3. Falla y Bloque de Falla

Una Falla, representada por la interfaz *IFailure*, inicialmente es en sí una redución de una tupla, representada por la clase *ImpactImportedTuple*, que almacena información de una detención y su fin es el de agilizar el proceso de propagación, dado que es un objeto más liviano. Esto es ocurre al principio de la propagación, la transformación de tuplas a fallas, es decir, por cada detención ingresada a un subsistema (que sea abordado por la propagación), tendrá su contraparte como falla, más específicamente, como Falla Unitaria, representada por la interfaz *IFailureUnit*. Mientras avance la propagación, se crearán nuevas fallas que en resumidas cuentas, son subfallas (cortes de las fallas originales) o uniones de varias fallas (conocidas como Fallas Compuestas, representadas por la interfaz *ICompositeFailure*).

Un Bloque de Falla, representado por la interfaz *IFailureBlock*, es un objeto contenedor que contiene Fallas que se encuentren traslapadas. Esto quiere decir que dentro de un Bloque de Fallas cada falla al menos intersecta con al menos una existente dentro del bloque, arbitrariamente, lo denominaremos un conjunto cerrado. Cada Bloque de Falla es independiente uno de otro, en el sentido de que no comparten "espacio de tiempo", por definición, no existen traslapes entre bloques.

3. Los procesos

3.1. Vista General

Dado el uso del patrón Visitor para la implementación de la propagación bajo demanda, se realiza una visita en profundidad recorriendo desde las hojas hacia arriba hasta abarcar todos los nodos del árbol de subsistemas de la planta. En todo punto de la visita, en cada nodo o subsistema, se pueden enumerar los siguientes pasos que se realizan:

1. Procesamiento de Datos Importados: Crear bloques de fallas de los datos importados del subsistema: se realiza a todo nivel, siempre y cuando el nodo visitado contenga tuplas.
2. Procesamiento de Bloques Inferiores: Cruzar los bloques del subsistema actual con los bloques de los hijos: sólo se puede realizar para nodos que no sean hojas, o sea, no se aplica para *Equipment*'s.
3. Procesamiento Global de Bloques: consta de 4 subprocesos:
   a) Cortado (*BlockSplitterRefactor*): proceso de guillotinado de fallas.
   b) Chequeo de Sobrecapacidad (*OvercapacityProcesorRefactor*): chequeo de impactos para subsistemas en fraccionamiento o redundancia.
   c) Jerarquizado (*HierarchySelectorRefactor*): implementación de reglas de jerarquía que rige el comportamiento de las fallas que "subirán" al subsistema padre.
   d) Juntado (*JoinerRefactor*): Unión de fallas contiguas con propiedades similares.
4. Procesamiento Final: Crear la base histórica y asignarla al subsistema.

3.2. La clase *SimpleHistoricAvailability*

Como se puede observar en el diagrama de la figura 3, la clase *HistoricAvailabilityVisitor* implementa la interfaz *SubsystemVisitorI*, lo cual nos dice que esta clase es un visitador de subsistemas. El código de esta clase es simple:

```
1  public class HistoricAvailabilityVisitor implements SubsystemVisitorI {
2    private IPropagator propagator;
3
4    public HistoricAvailabilityVisitor(Date from, Date to) {
5      SimpleInterval interval = new SimpleInterval(from, to);
6      BlockProcessorI bt = new BlockTransformerRefactor();
7      this.propagator = new SimpleHistoricAvailability(interval, bt);
8    }
9
10   @Override
11   public void visit(Subsystem ss) {
12     this.propagator.calculateHistoricBase(ss);
13   }
14 }
```

La interfaz *IPropagator* simplemente fue creada con el fin de que en algún futuro se pueda implementar hebras al procesamiento de bloques. Por ahora, su única instancia conocida es la clase *SimpleHistoricAvailability*, a la que abordaremos nuestra atención.

Esta clase se encarga de coordinar todos los procesos principales descritos en la sección anterior. El código para el método *calculateHistoricBase(Subsystem)* es el siguiente:

```
1   @Override
2   public void calculateHistoricBase(Subsystem ss) {
3     this.blocks.clear();
4     this.ss = ss;
5
6     this.importedProcessor.processImportedData(ss, blocks, rangeOfAnalysis);
7     this.failureProcessor.processFailureData(ss, blocks, rangeOfAnalysis);
8     processBlocks();
9     createBase();
10  }
```

Se pueden ver claramente los 4 procesos principales: uno para los datos importados, otro para para juntar las fallas del subsistema actual con la de los hijos, otro referente al proceso completo de todos los bloques del subsistema y finalmente uno para la creación y asignación de la base histórica.

3.3. Procesamiento de Datos Importados

Es realizado por la clase *ImportedDataProcessor*. Primero, justa los datos importados del subsistema al rango de fechas en estudio, filtrando aquellas tuplas que estan fuera de rango y cortando aquellas que se ubican en los bordes. Sólo se consideran las tuplas que detienen al sistema. De las tuplas que seleccionadas, se transforman a fallas y se crean los primeros bloques. Cada bloque es independiente y contiene fallas traslapadas. Como corolario, si no existen traslapes en las fallas, se crearán tantos bloques como fallas existan. Un bloque de falla contiene al menos una falla.

3.4. Procesamiento de Bloques Inferiores

Es realizado por la clase *FailureDataProcessor*. Une la lista de bloques creados en el paso anterior con la lista de bloques de cada hijo. Básicamente, cada falla de cada hijo produce la unión de varios bloques de falla si éste es contenido por todos ellos.

3.5. Procesamiento Global de Bloques

Es realizado por la clase *BlockTransformerRefactor*. Consta de 4 subprocesos. Dentro de la clase *SimpleHistoricAvailability* en el método *processBlocks()* se puede encontrar el código que gatilla el procesamiento:

```
1   private void processBlocks() {
2     for (FailureBlock fb : blocks) {
3       fb.setSubsystem(ss);
4       blockTransformer.process(fb);
5     }
6   }
```

Dentro de la clase *BlockTransformerRefactor* en el método *process(FailureBlock)* se encuentran las llamadas a los subprocesos:

```
1   @Override
2   public void process(FailureBlock block) {
3     Subsystem ss = block.getSubsystem();
4     if (ss instanceof Equipment) {
5       Equipment equipment = (Equipment) ss;
```

```
6         if (equipment.getTreatAsSubsystem()) {
7           return;
8         }
9       }
10
11      block.sortFailures();
12      splitter.process(block);
13      if (ss instanceof FractionMachine &&
14          ((FractionMachine) ss).getSubsystemImpactSum() > 1.0) {
15        overcapacity.process(block);
16      } else if (ss instanceof RedundantMachine) {
17        overcapacity.process(block);
18      }
19      selector.process(block);
20      if (!(ss instanceof FractionMachine)) {
21        joiner.process(block);
22      }
23    }
```

3.5.1. Cortado

Es realizado por la clase *BlockSplitterRefactor*.

3.5.2. Chequeo de Sobrecapacidad

Es realizado por la clase *OvercapacityProcesorRefactor*.

3.5.3. Jerarquizado

Es realizado por la clase *HierarchySelectorRefactor*.

3.5.4. Juntado

Es realizado por la clase *JoinerRefactor*.

3.6. Procesamiento Final

Crear la base histórica y asignarla al subsistema.

```
1   private void createBase() {
2     NewHistoricBase hb = new NewHistoricBase();
3     for (FailureBlock fb : blocks) {
4       hb.addFailureBlock(fb);
5     }
6     ss.setNewHistoricBase(hb);
7   }
```