UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
DOCTORADO EN CIENCIAS APLICADAS

MÉRIDA – VENEZUELA

“Middleware Reflexivo para la Gestión Automática de Aplicaciones Orientadas a Servicios usando la teoría de Firmas (signatures) de Fallas”

Autor: Juan Ernesto Vizcarrondo Rojas

Tutores: Dr. Jose L. Aguilar C.

Cotutor: Dr. Francklin Rivas E.

Proyecto de Grado presentado ante la ilustre Universidad de los Andes como requisito parcial para optar al Título de Doctor en Ciencias Aplicadas

MÉRIDA, MAYO 2016

AGRADECIMIENTOS

El logro de este trabajo es la suma de esfuerzo, fe y dedicación, pero por sobre todo es el resultado de las palabras de aliento y confianza de mis padre Guillermo Vizcarrondo, mi hermano Leonardo y mi sobrino Leonardo Jesús a quienes nunca podré pagarles el amor y la comprensión que me brindaron para alcanzar esta meta.

Al Prof. José Aguilar por todo el apoyo y los conocimientos que me brindó para la realización de este proyecto.

A los Prof. Francklin Rivas, Ernesto Exposito y Audine Subias por su apoyo, su orientación pedagógica, a pesar de la distancia.

A la Universidad de Los Andes por haberme permitido formarme como Doctor en Ciencias Aplicadas.

A la Fundación Centro Nacional de Desarrollo e Investigación en Tecnologías Libres (CENDITEL) por su gran apoyo en el desarrollo de este trabajo.

Al Laboratoire d'analyse et d'architecture des systèmes LAAS-CNRS y todos sus miembros por permitirme formar parte de su equipo.

Al Proyecto PCP "tareas de supervisión y mantenimiento en entornos distribuidos organizacionales" PCP 2010000307.

A mis compañeros en el doctorado MSc. Ruben Leal y PHD Code Diop con quienes compartí todos estos años.

A todos muchas gracias….

RESUMEN

En la composición de servicios, la falla de un servicio puede generar la propagación de errores en otros servicios, y por lo tanto, como consecuencia de ello, puede generar la falla de todo el sistema. Estas fallas, frecuentemente no pueden ser detectadas y corregidas a nivel local (servicio individual), por lo que es necesario el desarrollo de arquitecturas que permitan el diagnóstico y corrección de fallas, tanto a nivel individual (servicio) como a nivel global (composición). En este trabajo se propone una arquitectura de middleware reflexivo basado en computación autonómica, para el diagnóstico distribuido de composiciones de servicios, llamado ARMISCOM.

El middleware no tiene un diagnosticador central, el diagnóstico de las fallas se realiza a través de la interacción de los diagnosticadores presentes en cada servicio de la composición, utilizando para ello una extensión del paradigma de crónicas propuesta en este trabajo, para permitir el reconocimiento de patrones totalmente distribuido, y un conjunto de patrones temporales generales para la identificación de las clásicas fallas en los sistemas SOA. Adicionalmente, las estrategias de reparación se definen a través del consenso de los reparadores igualmente distribuidos en los servicios de la composición, utilizando el concepto “regiones equivalentes” definido en este trabajo, para la corrección de las fallas en una aplicación SOA equivalente.

ARMISCOM se implemento usando el bus de servicios OpenESB, Protégé, el lenguaje CQL, entre otras herramientas computacionales. Sobre esa implementación, se realizaron un conjunto de experimentos en 2 aplicaciones SOA, para probar las diferentes propuestas realizadas en este trabajo: la extensión del paradigma de crónicas distribuido, el mecanismos de reconocimiento de patrones basado en eventos distribuidos, los distintos patrones

de fallas basadas en crónicas distribuidas propuestos para el diagnóstico de fallas en aplicaciones SOA, y finalmente, las capacidades autonómicas y reflexivas del middleware para diagnosticar y corregir las fallas en escenarios distintos.

Palabras Claves: Middleware Reflexivos, Tolerancia a fallas en aplicaciones SOA, Computación Autonómica, Reconocimiento de patrones temporales, Crónicas, Diagnóstico de Fallas distribuido.

ABSTRACT

In the composition of services, a service failure can generate error propagation in other services, and therefore, consequently, can generate the failure of the entire system. Such failures, often cannot be detected and corrected locally (single service), so it is necessary to develop architectures to enable diagnosis and correction of faults, both at individual (service) as global (composition) levels. In this work we propose a reflective middleware architecture based on autonomic computing, for the distributed diagnosis of service compositions, called ARMISCOM.

The middleware has not a central diagnoser, so that the diagnosis of faults is performed through the interaction of the diagnosers present in each service of the composition, using an extension of the proposed on this work, to allow a pattern recognition fully distributed, and a set of general temporal patterns to identify the classical faults in SOA systems. Additionally, the repair strategies are defined through consensus of the repairers distributed equally in the services of the composition, using the notion of “equivalent regions” defined in this work, to fix the faults in a SOA application.

ARMISCOM was implemented using the OpenESB bus, Protégé, the CQL language, among other computational tools. On this implementation, We have performed a set of experiments on two SOA applications, to test the different proposals carried out in this work: the extended paradigm of chronicles distributed, the mechanism of patterns recognition based on distributed events, the different fault patterns based on distributed chronicles proposed for the faults diagnostic in SOA applications, and finally, the autonomic and reflective middleware capabilities to diagnose and fix faults in different scenarios.

Keywords: Reflective middleware, fault tolerance of SOA applications, Autonomic Computing, Temporal patterns recognition, Chronicles, Distributed Fault Diagnostic.

Índice de contenido

Índice de Figuras

Índice de Tablas

Índice de Algoritmos

Listado de Acrónimos

ARMISCOM: Autonomic Reflective MIddleware for management Service COMposition.

AS: Application Server .

CarDeCRS: Chronicles Applied to error Recognition in Distributed Environments, through CRS .

CRS: Chronicle Recognition System.

CQL: Continuos Query Language .

DES: Discrete Event Systems .

EAI: Enterprise Application Integration .

ESB: Enterprise Service Bus.

IDE: Integrated Development Environment .

IVSS: Instituto Venezolano de los Seguros Sociales.

MAPE: Monitoring, Analisis, Plan and Execute.

MAPE-K: Monitoring, Analisis, Plan and Execute – Knowledge .

MOM: Message Oriented Middleware .

OWL: Web Ontology Language.

QoS: Quality of Service.

SLA: Service Level Agreement.

SOA: Service Oriented Architecture.

SOAP: Simple Object Access Protocol.

SOAR: Service Oriented Architecture with Reflection.

SQL: Structured Query Language.

UDDI: Universal Description, Discovery and Integration.

WS-CDL: Web Services Choreography Description Language.

WSDL: Web Services Description Language.

XML: eXtensible Markup Language (lenguaje de marcas Extensible en español)

XSD: XML Schema Definition .

XSLT: Extensible Stylesheet Language Transformations.

Capítulo 1
Introducción

1.1. Generalidades

El desarrollo de aplicaciones utilizando el paradigma Arquitectura Orientada a Servicios (SOA), consiste en: un patrón de desarrollo de software en el que una aplicación es descompuesta en pequeñas unidades, lógicas o funcionalidades, denominadas servicios. SOA permite un despliegue de aplicaciones distribuidas muy flexible, con bajo acoplamiento entre sus componentes de software, los cuales interactúan en ambientes distribuidos heterogéneos. En general, SOA provee un estándar que hace posible la interoperatividad, la reusabilidad de distintas piezas de software, entre otros aspectos. En particular, SOA ha sido utilizado para organizar y utilizar recursos distribuidos [1].

Tradicionalmente, la base de SOA son los Servicios Web, que son una familia de componentes de software distribuido que pueden ser gestionados a través de Internet [2]. Los Servicios Web [3] son entidades computacionales que se pueden agregar a otros Servicios Web para el desarrollo de aplicaciones más complejas, los cuales pueden ser publicados, localizados e invocados a través de Internet.

La integración de servicios facilita la cooperación [4]. Este estilo de programación basado en la composición y en la reutilización de servicios (nuevas aplicaciones basadas en servicios existentes), es posible por el apropiado ensamblaje de otros servicios ya existentes [4]. En general, los enfoques de ensamblaje de servicios capturan las interacciones entre los participantes y las relaciones entre ellos (dependencias en el flujo de control y de datos,

restricciones de tiempo, dependencias de transacciones, entre otras). El ensamblaje de servicios puede ser realizado por orquestación y/o coreografía [5]. Así, la coreografía captura las interacciones desde una perspectiva global, dado que todos los participantes son tratados de la misma manera (abarca todas las interacciones entre los servicios que son pertinentes con respecto al objetivo de la coreografía). Por otro lado, la orquestación se puede ver como un solo proceso [5], en el cual se requiere de un conductor para realizar el ensamblaje de las operaciones. Así, la orquestación se hace desde la perspectiva del conductor.

Ahora bien, los servicios son propensos a fallas, pues son inherentemente dinámicos, y no pueden ser asumidos como siempre estables [6], ya que durante la evolución natural de un servicio (por ejemplo, modificaciones de sus interfaces) se pueden alterar las características del trabajo que previamente realizaba (un servicio se le puede eliminar una operación que ejecutaba, modificar parte de su lógica interna, entre otras cosas), llegando en algunas ocasiones a ser un servicio totalmente distinto al que se tenia inicialmente. También están las fallas inherentes a su utilización, que corresponde a los casos donde la plataforma distribuida presenta anomalías funcionales (por ejemplo, los procesadores o medios de comunicación fallan).

En general, la gestión de las fallas en aplicaciones basadas en servicios, es un problema complejo, por las características distribuidas, dinámicas, heterogéneas, entre otras. En ese sentido, la detección y corrección de las fallas en aplicaciones basadas en servicios es difícil, por lo que es necesario el desarrollo de arquitecturas y mecanismos que permitan la detección, el diagnóstico y la corrección de fallas, tanto a nivel individual (servicio) como global (composición). A todo lo anterior, clásicamente se le ha denominado “tolerancia a las fallas”. Así, en el transcurso del tiempo han surgido un conjunto de middlewares con distintos

enfoques [19, 20, 21, 23], para garantizar la tolerancia a fallas de las aplicaciones basadas en servicios, pero presentan en general algunos problemas: de escalabilidad, son orientados sólo a problemas en la composición o en los servicios, no establecen mecanismos de reparación, entre otros.

En este trabajo se diseña e implementa un middleware reflexivo para la gestión distribuida de fallas, tal que el diagnóstico de fallas se realiza a través de la interacción de los diagnosticadores presente en cada servicio, y las estrategias de reparación se desarrollan a través del consenso de los reparadores, también localizados en los diferentes servicios, disminuyendo la cantidad de mensajes intercambiados por reconocedor y distribuir el procesamiento de eventos entre los distintos diagnosticadores. Dicha arquitectura está basada en el enfoque de la computación autonómica, con el fin de facilitar su construcción basada en métodos, algoritmos y herramientas, que permitan el desarrollo de un sistema de auto-sanación, el cual es proactivo, con la finalidad de detectar, diagnósticar y corregir las fallas, en tiempo real.

1.2. Definición del Problema de Estudio

El uso de las aplicaciones SOA actualmente es muy extenso. Ahora bien, dichas aplicaciones son propensas a fallas debido a los servicios que las componen, las cuales muchas veces son difíciles de determinar, por la composición de servicios en la que se basan. El fallo de un servicio produce la propagación de errores en los demás servicios implicados en la composición, y por lo tanto, genera la falla de toda la aplicación. También, existen las fallas propias al proceso de composición, asociadas a los problemas de comunicación y de indisponibilidad de los sitios donde se ejecutan los servicios. Por las anteriores razones, es necesario el desarrollo de arquitecturas y mecanismos que permitan la detección, el diagnóstico y la corrección de fallas en las aplicaciones SOA, tanto de sus

servicios como en la composición de los mismos.

Por otro lado, las firmas de fallas han resultado ser una técnica que facilita la identificación de las causas de fallas en los sistemas, la cual ha sido usada exitosamente en diferentes ámbitos, por ejemplo para el diagnóstico de la propagación de fallas entre los distintos componentes que conforman un sistema [8]. En particular, el desarrollo de técnicas de reconocimiento de firmas desarrolladas para sistemas de control [8], permiten establecer mecanismos para desarrollar sistemas capaces de identificar las distintas causas de las fallas presentes en los sistemas. Su uso, para el desarrollo de un catálogo de fallas presentes en los servicios web, provee un mecanismo para solventar los distintos problemas de fallas afrontados por servicios web [9].

Una manera de implementar firmas de fallas consiste en representarlas en formas de Crónicas. Las Crónicas permiten modelar las relaciones temporales entre eventos observables en un sistema, de tal manera de describir patrones de comportamientos del mismo. Dousson presentó los trabajos iniciales sobre las Crónicas [10, 11], definiéndolas como un conjunto de patrones, cada uno caracterizado por eventos observables y restricciones temporales entre ellos y con respecto al contexto, representando una interpretación de lo que está ocurriendo en la dinámica del sistema en estudio en un momento dado [12]. Así, cada crónica representa una situación o escenario de desempeño normal o anormal del sistema. La mayoría de los mecanismos de reconocimiento de crónicas se basan en una arquitectura central (un solo diagnosticador), la cual verifica la llegada de los eventos y sus restricciones temporales, permitiéndole así reconocer las situaciones de fallas. La implementación del mecanismo centralizado no resulta muy eficiente a nivel computacional con respecto a las arquitecturas distribuidas, y es poco escalable, cuando se busca su

implementación en sistemas distribuidos con muchos componentes.

Algunos trabajos han utilizado el formalismo de crónicas para el diagnóstico de fallas [11, 13, 14, 15, 16], pero poco se ha realizado en el contexto distribuido. El enfoque más cercano a la propuesta de este trabajo provee un mecanismo de reconocimiento descentralizado de supervisión [16]. Dicho trabajo se comentará en la sección de antecedentes, y básicamente, propone una arquitectura compuesta por diagnosticadores locales, que supervisan un conjunto de eventos en su ámbito de acción, y en caso de algún suceso invocan a un diagnosticador global, que es el encargado de realizar el reconocimiento de la crónica (infiere la falla).

Por otra parte, la mayoría de los actuales middleware [19, 20, 21] se conciben como arquitecturas capaces de adaptarse ellas mismas, basadas en una inferencia de como es su conducta actualmente, y las condiciones que presenta el ambiente en el que se desenvuelven. Uno de estos tipos de middlewares son los reflexivos, que al tener una auto-representación, les es posible realizar reconfiguraciones dinámicamente. De esta manera, logran mejorar el desempeño de las aplicaciones que se ejecutan sobre el middleware, basándose en la optimización de las propiedades no funcionales, tales como tiempo de respuesta, disponibilidad, tolerancia a fallas, escalabilidad, seguridad, entre otros.

En este mismo orden de ideas, la Computación Autonómica representa un modelo de computación de auto-gestión, permitiéndole a una aplicación adaptarse a los cambios en el ambiente sin la intervención humana. La Computación Autonómica ha sido usada para auto-gestionar capacidades como: Auto-configuración, Auto-reparación, Auto-Optimización y Auto-Protección. En el caso de las Arquitecturas SOA, se han empleado middlewares denominados Buses de servicios (ESB, por sus siglas en Inglés), que permite el despliegue e

integración de servicios, a los cuales actualmente se le están incorporando componentes basados en Computación Autonómica (Autonomic Enterprise Service Bus) [26].

El principal problema con estas arquitecturas es que han sido implementadas como arquitecturas centralizadas y semi-centralizadas, que en el mejor de los casos están compuestas por componentes locales, distribuidos a través de los servicios que conforman la composición, los cuales son coordinados por un componente central, que obtiene una visión completa del problema para inferir una estrategia de reparación. Esto trae como consecuencia el problema de escalabilidad de la arquitectura, cuando las aplicaciones contienen un gran número de componentes. También, otro problema de estas arquitecturas es que las firmas de fallas que se proponen son estáticas, sin posibilidad de adaptarlas a las características propias de las aplicaciones que en un momento dado se estén ejecutando sobre el middleware.

Así, en este trabajo se plantea el desarrollo de un middleware reflexivo autonómico para la gestión de fallas en aplicaciones basadas en servicios, cuya arquitectura es distribuida, tal que el diagnóstico de fallas se realiza a través de la interacción de los diagnosticadores presentes en cada servicio, utilizando las crónicas para el diagnóstico de las fallas. También, las estrategias de reparación se desarrollan a través del consenso entre los reparadores distribuidos entre los servicios. Por otro lado, se usa el formalismo de crónicas que permite el modelado de los patrones de fallas, pero de manera distribuida, por lo que se extiende dicho formalismo para introducir la noción de crónicas distribuidas [18]. Finalmente, se plantean patrones de fallas, lo que posibilita que el middleware pueda adaptarlos a las características propias de las aplicaciones que en un momento dado se estén ejecutando en él.

1.3. Antecedentes

Las fallas en las aplicaciones orientadas a servicios se pueden clasificar en base a si ocurren en el mismo servicio y/o a nivel de la secuencia de llamadas en la composición. En [6] se propone una taxonomía para el análisis de los posibles fallas partiendo de los efectos percibidos, tanto a nivel local (servicio) como a nivel de la composición, lo que representa un excelente punto de partida para este trabajo. Las fallas son analizadas separadamente, y se muestran los efectos percibidos tanto a nivel del sub-flujo (parte del flujo que se encuentra en falla), como en el resto de servicios que forman parte de la composición. La taxonomía las clasifica en fallas de infraestructura, de desarrollo de la aplicación, y de iteraciones. Además, se hace un primer intento por correlacionar las fallas y los posibles mecanismos a implementar para resolverlos.

Por otro lado, se han propuesto varias arquitecturas para la gestión de fallas y recuperación en la composición de servicios web. En [19, 20] se presentan un enfoque de gestión de fallas basada en una clasificación de los tipos de fallas por niveles: Infraestructura, servicios y la aplicación SOA, y la correlacionan con los mecanismos de reparación y la manera de implementarlos (de tal forma de determinar si el proceso de reparación debe ser reactivo o proactivo). Adicionalmente, proponen un middleware centralizado compuesto por tres módulos. En [21] se propone un middleware reflexivo llamado SOAR, para la gestión centralizada de las fallas en la composición de servicios, el cual controla y adapta el sistema completo. El middleware se compone de dos niveles de una torre reflectante: el primero (nivel base) es responsable de la descripción de las características básicas del sistema SOA (Aplicación SOA), y el segundo nivel (nivel meta) es responsable de la supervisión y la adaptación del sistema SOA. Para realizar la reflexión, el nivel meta realiza su conexión causal con las entidades de

la aplicación SOA: consumidores, aplicación que sirve de intermediario (Bus), y proveedores. El nivel meta analiza los mensajes intercambiados por los servicios, basado en criterios de calidad de servicio (QoS) como cantidad de requerimientos exitosos y tiempo de respuestas (proceso de introspección). En el momento que sucede una degradación de QoS el middleware modifica las conexiones de la aplicación. La intersección en el middleware se realiza al conectar o desconectar dinámicamente los servicios que forman parte de una tarea conjunta.

En el mismo orden de ideas, en [22] se propone una arquitectura centralizada para las reparación de los servicios web, llamada arquitectura self-healing (auto-sanación). Para determinar la reparación del sistema SOA utiliza indicadores de calidad sobre los servicios web, los cuales son: disponibilidad, tiempo de respuesta, escalabilidad, entre otros. La arquitectura consta de tres módulos: el de seguimiento y medición (es responsable de la observación y el mantenimiento de los registros de los parámetros de QoS que son relevantes), el motor de diagnóstico y de estrategias de decisión (detecta la degradación del sistema e identifica los posibles planes de reparación), y finalmente, el módulo de reconfiguración (que implementa el plan de reparación). En [23] se propone una arquitectura descentralizada compuesta por 2 niveles. El primer nivel utiliza un diagnosticador local para cada servicio que es parte de la composición, que se comunica con un diagnosticador global (central) para el diagnóstico de la composición global. El diagnosticador global es responsable de la coordinación de los diagnosticadores locales, haciendo uso del intercambio de mensajes para encontrar el servicio y la actividad responsable de la falla. Además, esta entidad global también es capaz de implementar mecanismos para la recuperación de la composición. El diagnóstico es realizado usando un mecanismo de reconocimiento de crónicas descentralizado, modelado con redes de Petri, las cuales se enriquecen con información acerca de las pre y post condiciones. En general,

cada diagnosticador local diagnóstica un conjunto de patrones de fallas definidos previamente (off-line), que se propagan al diagnósticador central. Las crónicas instanciadas por los diagnosticadores locales determinan la ocurrencia de un error. Por otra parte, en [24] se propone una solución semi-centralizada compuesta de diagnosticadores locales distribuidos por cada servicio que forma parte de la composición, que son coordinados por un diagnosticador global que diagnóstica el estado de la composición, y en caso de una falla, realiza tareas de reparación. Cada diagnosticador local razona para proveer una hipótesis local sobre la conducta observada (a nivel del servicio), y no la comparte con el resto de los diagnosticadores locales. La conducta del sistema es descrita como un conjunto de restricciones entre variables a nivel de cada servicio, las cuales representan la presencia o ausencia de fallas localmente. El diagnosticador global es el encargado de recopilar la información de los diagnosticadores locales (ver las variables locales que describen la ausencia/presencia de fallas), verificar la compatibilidad de los diagnósticos locales, para después realizar un diagnóstico global.

En cuanto a la computación autonómica, ha sido usada previamente para realizar la auto-reparación en aplicaciones SOA. En [25] se propone una arquitectura centralizada basada en Computación Autonómica llamada JOpera project, encargada de realizar la auto-reparación a una composición SOA. Para esto, define un lazo de control MAPE encargado de balancear la carga, realizando una configuración autonómica de la composición. El sistema recolecta la información de toda la aplicación para verificar que se encuentra balanceada la carga. La política de optimización es determinada por un grupo de criterios (por ejemplo, carga de trabajo), con el fin de realizar una auto-configuración del sistema. Otra arquitectura centralizada ha sido implementada en un bus de servicios para monitorear QoS [26], la cual puede ser convertida en

descentralizada al colocar instancias del ESB para sub-flujos de la composición, logrando de esta forma obtener una federación de ESBs que monitorean sub-flujos. Ahora bien, con el enfoque centralizado no pudieron lograr el diagnóstico como un todo de la aplicación, por los problemas de escalabilidad propios de las arquitecturas centralizadas. En el otro caso, se divide la composición en sub-flujos, y se asignan instancias del ESB para monitorearlos, logrando una federación de ESB que funciona de manera descentralizadas, contrarrestando los problemas de escalabilidad. Otras implementaciones distribuidas han sido desarrolladas en [27, 28], colocando lazos MAPE para auto-reparar cada servicio web, activando o desactivando servicios.

Algunos trabajos que utilizan Crónicas para determinar las fallas presentes en los sistemas son [11, 13, 14, 15, 16], comúnmente usando un diagnosticador centralizado o global. Ahora bien, algunos estudios han propuesto un esquema distribuido de diagnóstico de fallos usando crónicas, en que los diagnosticadores locales se asignan a diferentes componentes del sistema, y son sincronizados para obtener un diagnóstico global [29]. En particular, proponen una verificación distribuida de restricciones utilizando eventos locales y globales, añadiendo la introducción de retrasos (*delay*) entre las comunicaciones presentes en los diferentes diagnosticadores. Los diagnosticadores son colocados en cada componente del sistema, y cada uno realiza por separado el diagnóstico de la crónica global, propagando los eventos comunes que consideran de importancia a los demás diagnosticadores vecinos, y al recibirlos y juntarlos con los eventos propios del componente les es posible tener una visión global del sistema. En [30] proponen otro mecanismo para el diagnóstico de fallos distribuido usando crónicas, para lo cual descomponen las crónicas en tantas sub-crónicas como componentes tenga el sistema, y las observaciones se comunican entre los diagnosticadores, para obtener así la información necesaria que no está

disponible localmente. El sistema de reconocimiento permite reconocer cada sub-crónica, y el resultado se comunica a un diagnosticador global para construir el diagnóstico total, basado en una historia que contiene la unión de cada sub-crónica que compone el sistema en estudio. Otro enfoque utilizando una arquitectura descentralizada para el diagnóstico de fallas en los servicios de composición utilizando crónicas se presenta en [16]. Esta arquitectura se compone de dos niveles, en el primer nivel los diagnosticadores locales se colocan para cada servicio que es parte de la composición, que se comunica con un diagnosticador global (central) para el diagnóstico de la composición entera. El diagnosticador global es responsable de coordinar los diagnosticadores locales, mediante el intercambio de mensajes, para encontrar el servicio y la actividad responsable del fracaso. En [31] proponen una arquitectura totalmente distribuida entre los diagnosticadores locales, donde cada diagnosticador envía los eventos recibidos a diagnosticadores vecinos, lo que permite el reconocimiento de la crónica global.

Una herramienta para el reconocimiento de crónicas, llamado CRS, ha sido desarrollado por Dousson [10, 32]. Es el encargado de analizar el flujo de eventos y reconocer en tiempo real, cualquier patrón de eventos que coincida con una situación descrita por las crónicas. Cuando un nuevo evento se registra en el sistema, nuevas instancias de crónicas son generadas en el conjunto de hipótesis. CarDeCRS [12] y Matrac [71] son extensiones de CRS, que permiten el diagnóstico descentralizado de crónicas [12], colocando diagnosticadores locales que realizan un diagnostico parcial de las fallas en la composición de servicios, y son coordinados por un diagnosticador global para obtener un diagnóstico completo. Las tres herramientas, CRS, CarDeCRS y Matrac, utilizan el lenguaje CRS, que no permite operadores matemáticos en las restricciones de las variables atemporales [33], y su uso como un componente de una plataforma SOA tampoco ha sido

desarrollado, reduciendo su portabilidad.

Como se ve en el estado de arte, existen trabajos sobre middleware reflexivos y otras arquitecturas, para tolerancia a fallas en aplicaciones SOA, pero al estar enmarcadas en arquitecturas centralizadas o semi-centralizadas[1] , no gestionan apropiadamente el problema de escalabilidad cuando la aplicación aumenta de tamaño. También, existen trabajos en computación autonómica para soportar aplicaciones SOA, pero al igual que antes, la mayoría de los trabajos presentan un enfoque centralizado o descentralizado para obtener el diagnóstico global del sistema, teniendo el mismo problema de escalabilidad. En los casos de los trabajos que proponen una arquitectura distribuida, se enfocan en el diagnóstico de fallas a nivel de cada servicio, sin considerar el diagnóstico global de fallas en la composición. Finalmente, a nivel de crónicas se han desarrollado enfoques autodenominados distribuidos, pero con limitaciones similares a las planteadas anteriormente, por considerar un coordinador central que recolecta el diagnóstico de cada crónica local. Adicionalmente, los reconocedores de crónicas empleados hasta los momentos no permiten operadores matemáticos en las restricciones de las variables atemporales, lo que no posibilita la detección de ciertos tipos de patrones con restricciones complejas que se deben modelar matemáticamente. Esto comúnmente es implementado usando componentes intermedios, lo que limita el reconocimiento de patrones en tiempo real, lo cual restringe su uso en diagnosticadores que se enmarcan en una arquitectura SOA. Nuestra propuesta trata de solventar todos estos problemas, al distribuir el middleware en cada servicio, coordinando acciones entre sí para obtener un diagnóstico global de la composición y su reparación. Además, se usa un entorno/lenguaje de desarrollo para la implantación del reconocedor de crónicas, adecuado para ser usado en arquitecturas SOA.

1 En las arquitecturas semi-centralizadas, los componentes son distribuidos, pero son coordinados por un ente central.

1.4. Objetivos de la Tesis

El objetivo general de este trabajo es: "Diseñar y Desarrollar un Middleware Reflexivo para la Gestión Autonómica de Fallas en Aplicaciones Orientadas a Servicios Web, basado en un reconocimiento distribuido de la crónicas". Los objetivos específicos son:

- Diseñar la arquitectura de un middleware reflexivo para realizar el diagnóstico y reparación distribuida de fallas en aplicaciones SOA.
- Especificar un conjunto de patrones que permitan el diagnóstico de fallas en Aplicaciones Orientadas a Servicios Web.
- Diseñar e implementar mecanismos para definir y reconocer crónicas en un ambiente distribuido, que representan fallas en las Aplicaciones Orientadas a Servicios Web.
- Diseñar y desarrollar herramientas de programación bajo software libre, que implementen un middleware reflexivo para la gestión de fallas en Aplicaciones Orientadas a Servicios Web, basado en los mecanismos de gestión de crónicas previamente desarrollados.
- Definir e implementar casos de prueba utilizando el middleware reflexivo.

1.5. Alcance y Organización de la Tesis

Este trabajo propone el diseño e implementación de un middleware que pueda realizar el diagnóstico y la reparación distribuida de las fallas en aplicaciones orientadas a servicios. Para esto, en un primer momento se procede a realizar una revisión de los trabajos dedicados a la problemática de auto-reparación y

arquitecturas para la gestión de fallas en aplicaciones orientadas a servicios, en particular, las taxonomías, modelos centralizados o semi-centralizados, etc.

Posteriormente, se presenta el diseño del Middleware, denominado ARMISCOM, el cual es un Middleware Reflexivo Autonómico de dos niveles: en el nivel base se encuentra la aplicación y el conjunto de normas y definiciones que rigen esas interacciones (Sistema SOA), y en el nivel meta se encuentra el sistema de gestión de fallas de las aplicaciones SOA basado en computación autonómica. En específico, para contrarrestar el problema de escalabilidad y rendimiento, el nivel meta se diseña para que sea totalmente distribuido.

Para realizar el diagnóstico de las fallas, se usa un mecanismo de reconocimiento de patrones temporales llamado crónicas. En particular, en este trabajo se propone una extensión a las crónicas, para lograr un reconocimiento de las mismas completamente distribuido. En ese mismo orden de ideas, a continuación se presenta el diseño de un conjunto de patrones temporales distribuidos y genéricos, para el diagnóstico de fallas en aplicaciones SOA, basado en crónicas distribuidas. Además, como no existe un sistema para el reconocimiento de crónicas distribuidas, seguidamente se presenta el diseño e implementación de una herramienta basada en el lenguaje CQL [67, 69], que permite el reconocimiento de crónicas distribuidas. Dicho sistema es basado en software libre, y puede ejecutarse en un ambiente de servicios web.

En cuanto a la reparación de las fallas, se propone el diseño de una ontología para correlacionar las fallas con los mecanismos de reparación, y se implementa como un servicio web utilizando el motor de inferencia FACT++[2]. Adicionalmente,

2 FaCT++ es un razonador basado en tablas, de expresiones en Lógica descriptiva (DL), cubre los lenguajes OWL y OWL2.

se diseña e implementa una metadata[3], para almacenar el conjunto de mecanismos de reparación disponible en un momento dado en un sitio.

Por último, se procede a realizar un conjunto de experimentos, para comprobar el correcto funcionamiento de todos los mecanismos diseñados para ARMISCOM, utilizando 2 casos de prueba: una aplicación vinculada a fábrica de muebles, y otra aplicación de comercio electrónico.

De esta manera, la tesis se organiza de la siguiente forma: En el Capítulo 2 se exponen los fundamentos teóricos necesarios para comprender este trabajo. Así, en ese apartado se describen los servicios web y sus problemas de fallas, los middlewares reflexivos, la computación autonómica, y el formalismo usado para el diagnóstico de fallas, es decir, las crónicas.

El Capítulo 3 presenta la arquitectura del middleware reflexivo autonómico para el diagnóstico de fallas en Aplicaciones Orientadas a Servicios Web, el cual es totalmente distribuido, llamado ARMISCOM. Así, ARMISCOM no está coordinado por ningún componente central, el diagnóstico de las fallas se realiza a través de la interacción de los diagnosticadores presente en cada servicio, y la reparación se desarrolla a través del consenso de los reparadores distribuidos también a través de los servicios.

El desarrollo del mecanismo para el diagnóstico de fallas completamente distribuido es mostrado en el Capítulo 4. Además, en ese Capítulo se desarrollan el conjunto de patrones temporales genéricos distribuidos para el diagnóstico de las fallas en las Aplicaciones Orientadas a Servicios Web. Todos los patrones desarrollados en ese Capítulo son basados en crónicas.

3 En nuestro caso, una metadata es una colección de datos que describen regiones de eventos,mecanismos, etc., en un sitio.

En el Capítulo 5 se presenta la implementación de ARMISCOM en OpenESB, tal que cada componente de ARMISCOM es implementado como un servicio web. Adicionalmente, se presenta la implementación del mecanismo distribuido de reconocimiento de crónicas propuesto en este trabajo, usando para ello el lenguaje CQL, la ontología de Fault-Recovery y el motor de inferencia FACT++.

En el Capítulo 6 se presenta un conjunto de experimentos para comprobar el funcionamiento del modelo de crónicas distribuida propuesto en este trabajo, y se analizan los resultados obtenidos. Seguidamente, el Capítulo 7 presenta un conjunto de experimentos sobre ARMISCOM, para analizar su funcionamiento. Los casos de prueba mostrados en este apartado permiten comprobar el correcto funcionamiento de cada uno de los componentes de ARMISCOM .

Por último, el Capítulo 8 contiene las conclusiones generales de la tesis y los trabajos futuros que se podrían realizar como continuación de este trabajo.

Capítulo 2
Marco Teórico

En este Capítulo se introducen los fundamentos teóricos de los diferentes aspectos considerados en este trabajo, en específico sobre las aplicaciones orientadas a servicios, los middlewares reflexivos, la computación autonómica, y las crónicas.

2.1. Arquitectura Orientada a Servicios (SOA)

El paradigma SOA representa un modelo en el cual la lógica de la aplicación es descompuesta en pequeñas y distintas unidades lógicas (denominadas servicios), que al agruparse, comprenden la lógica de la aplicación como un todo [3, 34]. Estas unidades son independientes, y pueden ser distribuidas y usadas por diferentes aplicaciones. Así, SOA es un enfoque para crear sistemas por la composición de servicios autónomos, independientemente de las tecnologías sobre las que descansan.

En específico, un servicio define una funcionalidad concreta del sistema [35], que puede estar interactuando con otros (servicios) a través de una estructura muy bien definida de intercambio de mensajes [36]. Un servicio realiza una tarea concreta y sin memoria, es decir, atiende las operaciones que le fueron encomendadas. Así, al no conservar estados de transición impide que el servicio emita respuestas distintas para un mismo requerimiento. Un ejemplo es un servicio de verificación de la cédula de identidad de una persona, al ser invocado por usuarios, sistemas, e incluso otros servicios, a través de una interfaz estándar predefinida [36], debe dar la misma respuesta. Un servicio al ensamblarlo o

acoplarlo con otros servicios (composición), realiza tareas más complejas, como por ejemplo, el servicio anterior podría ser ensamblado con otros servicios para determinar las cotizaciones de un trabajador en el Instituto Venezolano de los Seguros Sociales (IVSS)[4] [36].

Para que un servicio siga el paradigma SOA, debe poseer las siguientes propiedades [34, 36]:

- Debe ser débilmente acoplado, tal que minimice sus dependencias con el resto del ambiente donde existe.
- Debe poseer un contrato de servicio que muestre la descripción de la tarea que realiza y la manera de comunicarse con él.
- Debe ser autónomo para realizar su función.
- Debe ser reusable por terceros en tareas más grandes.
- Debe ocultar los detalles de la lógica de la tarea que realiza al mundo exterior.
- Debe permitir su acoplamiento y coordinación, para formar servicios compuestos.
- Debe minimizar la retención de la información especifica de una actividad.
- Debe ser diseñado pensando en que será utilizado por personas o aplicaciones externas, por lo que es necesario realizar una descripción que permita, a partir de mecanismos de descubrimiento de servicios, inferir el

4 El IVSS es una institución pública de Venezuela dedicada a la protección de la Seguridad Social de todos sus beneficiarios trabajadores.

lugar donde se encuentra ubicado y como puede ser invocado.

2.1.1. Servicios Web

Los Servicios Web son pequeñas unidades lógicas que encapsulan la función de una tarea especifica, y pueden ser invocadas usando un conjunto de protocolos y estándares, proveyendo una eficiente forma para compartir la lógica distribuida de aplicaciones a través de múltiples máquinas, que están corriendo bajo diferentes sistemas operativos en Internet [37]. En general, alrededor de los Servicios Web se han definido un conjunto de mecanismos estandarizados, para exponer y consumir datos y lógica de aplicaciones sobre protocolos de Internet. Dicha tecnología se basa en estándares abiertos, para definir y reglamentar los mecanismos que permiten la comunicación entre distintos sistemas.

Así, la interoperabilidad de los sistemas se logra a partir de una serie de protocolos y estándares que permiten la comunicación y ejecución de ellos [38] (ver Figura 2.1):

- **SOAP (Simple Object Access Protocol):** es un protocolo basado en XML para el intercambio de mensajes, el cual es independiente de cualquier protocolo de transporte. Como SOAP está basado en XML, es soportado por un gran número de ambientes, tal que los mensajes pueden ser enviados sobre un gran número de protocolos.

- **XSD[5] (XML Schema Definition):** es un lenguaje utilizado para describir las estructuras y restricciones de los contenidos en un documento XML, y se basa en espacios de nombres (namespaces)[6]. En el caso de los servicios

5 Es un vocabulario para expresar las reglas de los datos que usa un WSDL.

6 Es un contenedor abstracto en el que un grupo de uno o más identificadores únicos pueden existir, y sirve para diferenciarlos de otros grupo.

web, sirve para describir las estructuras de datos utilizadas y sus restricciones, en los mensajes intercambiados al momento de invocar y obtener respuesta de un servicio.

- **WSDL (Web Services Description Language):** es un lenguaje de definición de interfaces basado en XML, el cual separa la definición de la función de la implementación de un servicio. Provee a los desarrolladores y sistemas, la manera de entender la operación de los servicios y los mecanismos de invocación.

- **UDDI (Universal Description, Discovery and Integration):** define una especie de directorio de servicios disponibles, y sus proveedores. Su característica interna consiste en una especificación basada en el lenguaje XML, que permite describir los servicios disponibles con sus proveedores (incluye el lugar donde se encuentra el WSDL), para que puedan ser localizados fácilmente por otras personas y/o aplicaciones.

El protocolo SOAP y sus estándares posibilitan que un servicio pueda ser invocado externamente, permitiendo que todo el proceso pueda ser realizado sin la intervención humana. Para esto, al momento de desarrollar un servicio web se coloca disponible en un catálogo UDDI para que pueda ser encontrado. Cuando una persona o aplicación busca y encuentra el servicio que más se adapte a sus necesidades, el catálogo le provee la dirección donde se encuentra la descripción completa del servicio usando los estándares WSDL (operaciones disponibles) y XSD (tipos de datos y sus restricciones usados en las operaciones). Una vez que se obtiene la ubicación y forma en que debe ejecutarse un servicio, se realiza la invocación del servicio haciendo uso del protocolo SOAP, logrando de esta manera una comunicación entre los dos puntos (invocador y proveedor) por medio del intercambio de datos en XML.

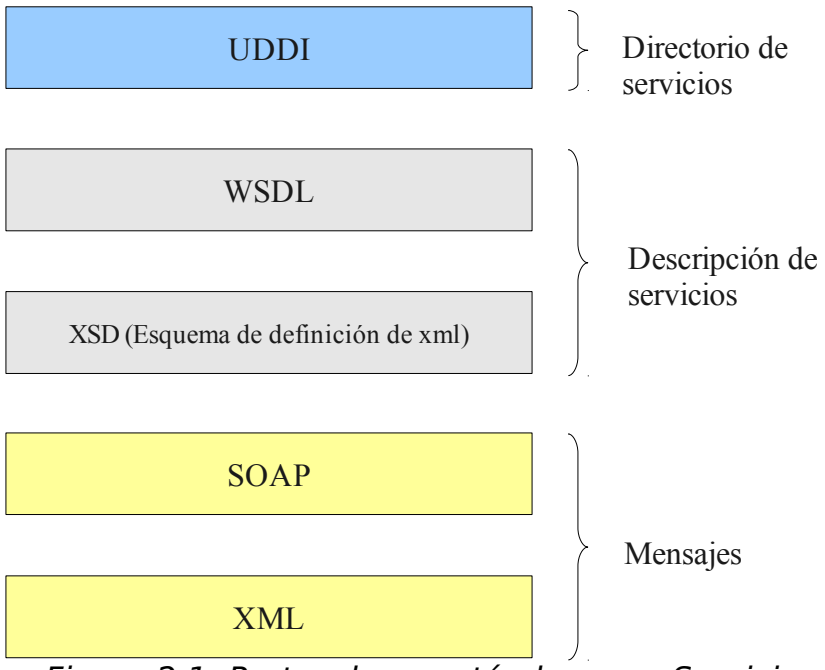


Figura 2.1: Protocolos y estándares en Servicios web.

2.1.2. Integración de Servicios

Actualmente existen varias maneras de organizar los servicios para componer una aplicación:

- ***Orquestación* [36]:** es diseñado para componer y ejecutar una serie de tareas, las cuales requieren de un conductor central, que es el encargado de coordinar la ejecución de los servicios involucrados. Dicho conductor es típicamente implementado como un servicio de integración, el cual verifica la ejecución, plantea eventos (la secuencia en que deben ser invocados los servicios), crea registros, ejecuta las invocaciones de los servicios, y realiza otras funciones para garantizar que el proceso se ejecute según lo esperado. Un ejemplo de este tipo de composición se muestra en la Figura 2.2, en este caso el servicio mediador/integrador es el encargado de invocar

los servicios involucrados en la composición, creando una capa intermedia entre los servicios y los usuarios.

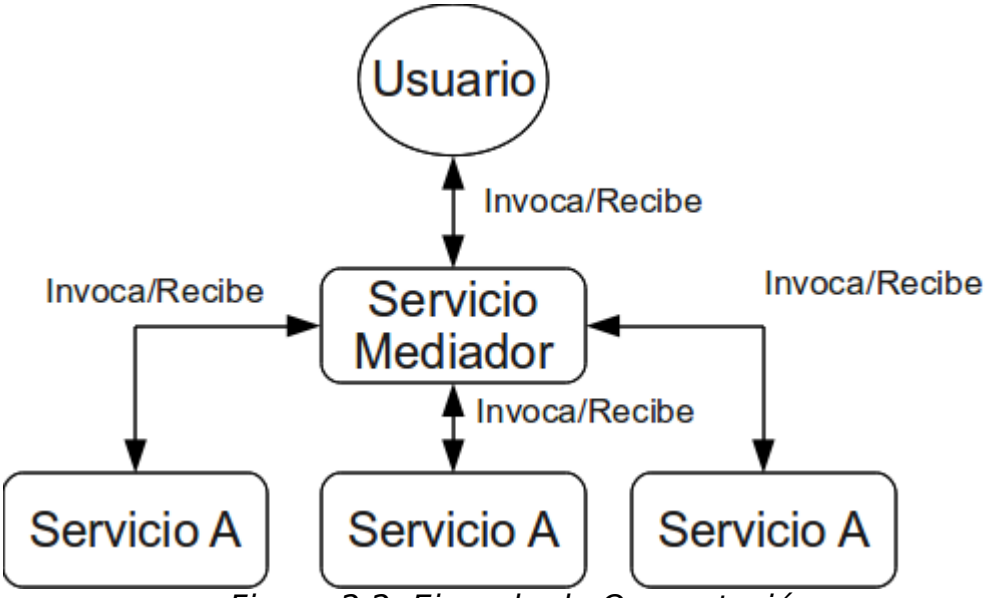


Figura 2.2: Ejemplo de Orquestación

- ***Coreografía* [36, 39]:** es un conjunto de relaciones punto a punto entre servicios individuales. No existe un conductor central, por lo que los servicios participantes interactúan entre si basándose en conjuntos definidos de contratos. Así, la coreografía se refiere a la coordinación distribuida de un conjunto de procesos individuales, para proveer un servicio general dado. Para implementar una coreografía usualmente se usa WS-CDL (Web Services Choreography Description Language) [40], que es un lenguaje basado en XML, que describe la colaboración punto a punto entre participantes (sin intermediarios). Así, el modelo de coreografía describe la colaboración entre una colección de servicios en procura de alcanzar un objetivo común [41]. Una coreografía no describe ninguna acción interna que ocurra dentro de los servicios participantes, sino lo que se observa es el resultado (el producto resultante) derivado de la integración de ellos. En la

Figura 2.3 se muestra un ejemplo de coreografía, en la que la invocación de los servicios se realiza punto a punto, sin contar con la mediación de ningún servicio intermedio. El usuario recibe la respuesta del servicio D, a través de las respuestas que cada servicio emite a quien lo invoco.

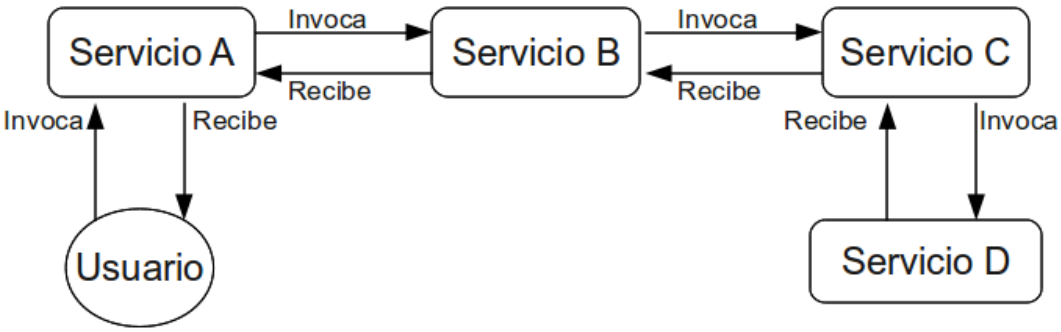


Figura 2.3: Ejemplo de Coreografía

2.1.3. Enterprise Service Bus (ESB)

Una primera plataforma de servicios estuvo basada en servidores de aplicaciones (AS: Application Server), los cuales proveían un middleware mediador (contenedor central) para el alojamiento y despliegue de los componentes de una aplicación (ver Figura 2.4). La complejidad de gestionar las conexiones entre proveedores y consumidores se reducían (integridad), al estar localizados en un solo lugar, pero delegando la gestión de la interoperatividad a los consumidores, al tener que realizar la tarea de adaptar las conexiones a las requeridas por los proveedores, por no estar disponible ningún componente para adaptar los mensajes intercambiados en la aplicación. Al ser una solución centralizada, presentaba los problemas inherentes de estas arquitecturas: escalabilidad y QoS (no podía distribuir la ejecución de las aplicaciones).

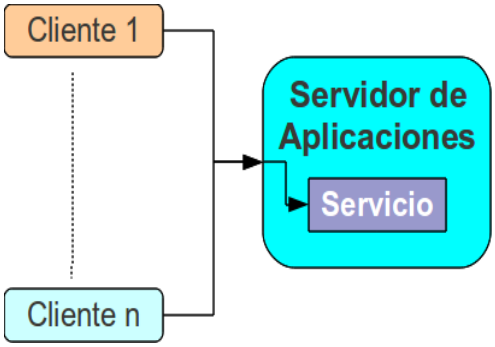


Figura 2.4: Arquitectura de un servidor de aplicaciones

Para mejorar este modelo se crearon los Middleware Orientado a Mensajes (MOM), ofreciendo la comunicación asíncrona entre proveedores y consumidores (ver Figura 2.5), logrando distribuir la ejecución de las aplicaciones (mejora la manera en que se realiza la integración de los servicios). Estos middlewares no proveían mejoras al problema de la interoperatividad de los servicios, ya que no proveían mecanismos para adaptar los mensajes intercambiados, delegando de igual manera la tarea de transformar los mensajes enviados a los proveedores a los consumidores. Lo que si mejoran es el QoS en cuanto a la ejecución distribuida de las aplicaciones.

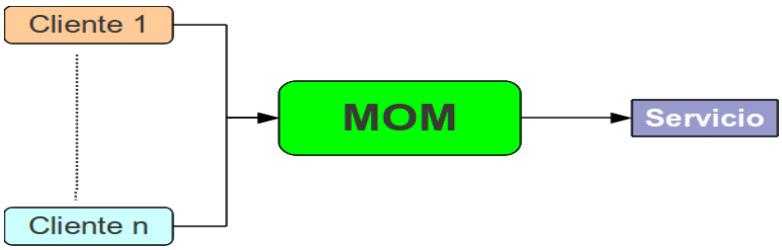


Figura 2.5: Arquitectura de un Middlewares Orientado a Mensajes

Posteriormente se crearon middlewares más sofisticados, conocidos como middlewares de Integración de Aplicaciones Empresariales (EAI). Estos

middlewares funcionan de manera similar a las arquitecturas Hub-and-Spoke[7] presentes en la gestión del tráfico aéreo (ver Figura 2.6). Mejoran la gestión de aplicaciones compuestas por servicios, al contar con los mismos mecanismos de gestión de aplicaciones del MOM, pero añadiéndole adaptadores para proveer la interoperatividad de éstas. Ahora bien, al ser una tecnología propietaria surgieron una gran cantidad de problemas a nivel de estándares.

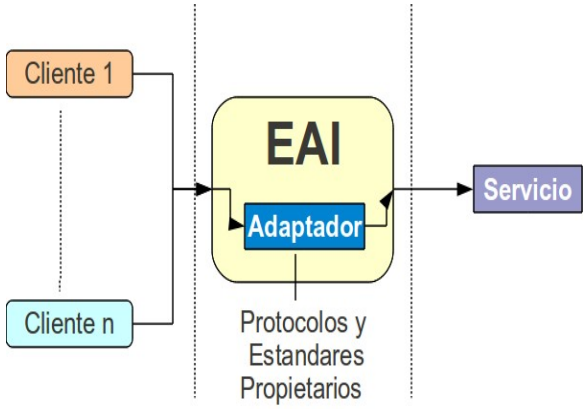


Figura 2.6: Arquitectura de un middleware EAI

Como una evolución de los EAI surgieron los Buses de Servicios Empresariales (ESB), que son el resultado de contar con protocolos estandarizados de enrutamiento, permitiendo tener arquitecturas distribuidas con diferentes tecnologías (ver Figura 2.7). Al romper con el paradigma de las tecnologías propietarias, los ESB facilitan la interoperatividad entre servicios que ofrecen diferentes tecnologías. La diferencia entre las figuras 2.5, 2.6 y 2.7, radica en el componente distribuido que gestiona las interacciones entre los servicios y los clientes.

7 Es un sistema de conexiones que permite reducir el numero de rutas, concentrando el tráfico en determinados aeropuertos de gran capacidad, denominados Hubs, que se encargan de redistribuirlos a otros de menor capacidad.

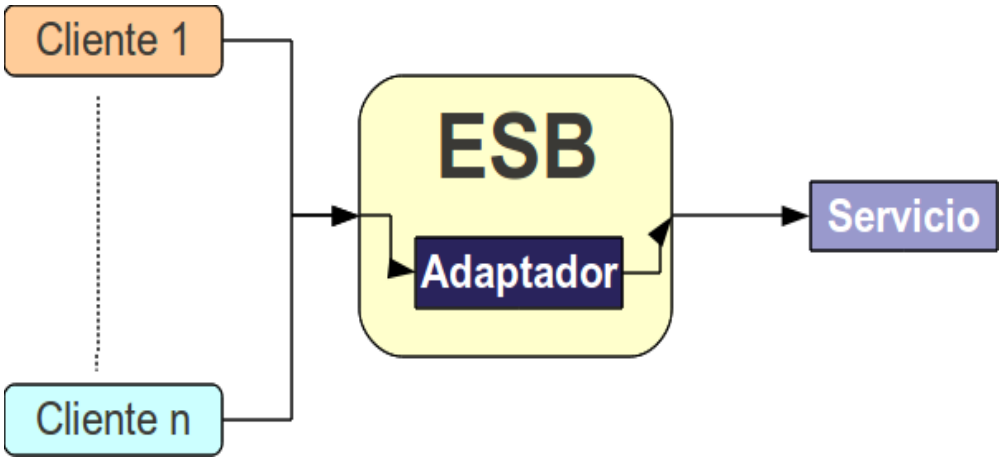


Figura 2.7: Arquitectura de un Buses de servicios Empresariales (ESB)

En particular, un EBS es una infraestructura de capa intermedia (middleware), que permite el despliegue e integración de servicios para aplicaciones SOA. Para esto, se proveen una serie de motores y protocolos que permiten el intercambio de mensajes entre ambientes heterogéneos. La tarea de un ESB es la de hacer transparente al usuario las tareas de conexiones dinámicas, mediación, control de los servicios y sus iteracciones [39]. Los ESB poseen las siguientes funciones [54]:

- Realizar el enrutamiento de los mensajes entre los servicios.
- Proveer el protocolo de transporte entre los consumidores y los servicios.
- Transformar el formato de los mensajes entre los consumidores y los servicios.
- Gestionar los diferentes eventos que se pueden dar en las interacciones entre los consumidores y los servicios.

- Garantizar la calidad de los servicios (seguridad, confiabilidad, entre otros).

Los componentes claves de la arquitectura de ESB se muestran en la Figura 2.8 [39]:

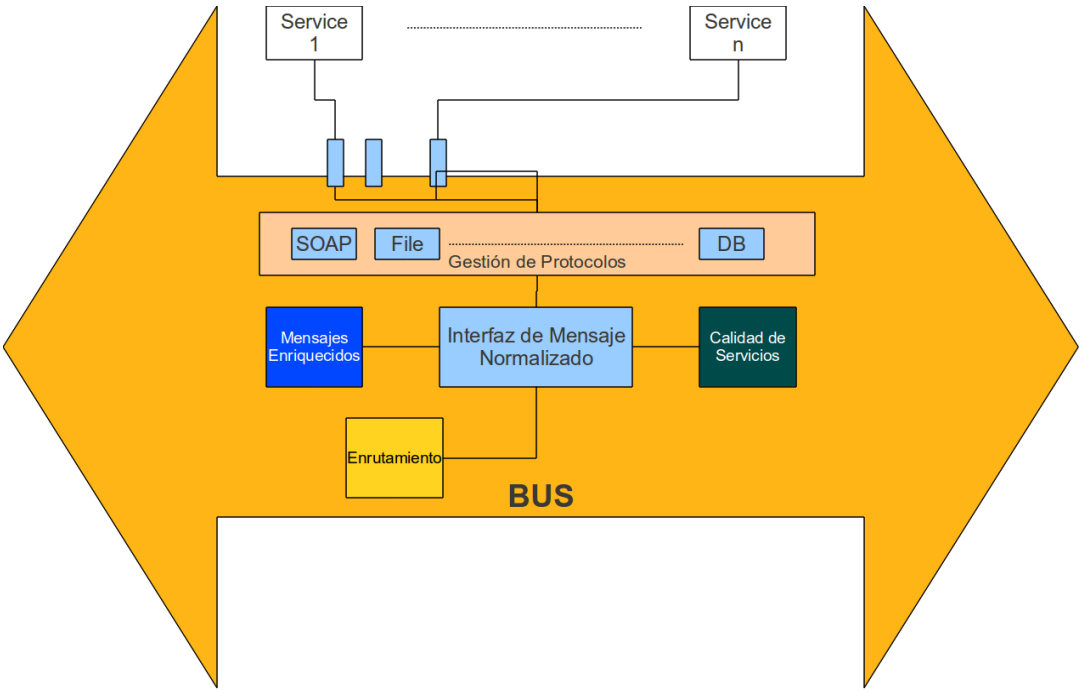


Figura 2.8: Arquitectura de un ESB

El primer componente es el encargado de realizar la **"gestión de los protocolos"**, permitiendo que un servicio que está definido según un protocolo específico (ej. de archivos) pueda ser invocado por un cliente usando otro protocolo (ej. el protocolo SOAP). Los ESB cuentan con una gran gama de protocolos (SOAP, entre otros) para permitir la conexión entre un gran número de servicios. Este componente es la interfaz que tiene el BUS para comunicarse con el mundo exterior.

El BUS convierte todos los mensajes de entrada y salida por mensajes

normalizados, haciendo uso del componente **"Interfaz de Mensaje Normalizado"**, por lo cual usa un lenguaje común para los mensajes dentro del BUS. En caso de ser necesario, los mensajes intercambiados por los servicios dentro del BUS como mensajes normalizados, pueden ser enriquecidos con más información, usando el componente **"Mensajes Enriquecidos"**, el cual permite añadir, sustraer y unir mensajes usando los lenguajes de consulta XSLT y XQuery.

Debido a que el BUS es una capa intermedia entre los clientes y los servicios, los mensajes intercambiados entre ellos, ya en forma normalizada y transformados, son llevados a su ubicación precisa, usando para ello el componente de **"Enrutamiento"**. Su principal función es la de esconder la verdadera ubicación de los servicios. Posibilita la implementación de un enrutamiento inteligente de los servicios, proveyendo rutas alternas para realizar balance de carga. Por último, el BUS garantiza que los criterios de Calidad de Servicio se mantengan en los parámetros establecidos, considerando aspectos de seguridad, detectando errores, entre otras cosas, usando para ello el módulo de **"Calidad de Servicio"**.

En específico, en el momento que un cliente quiere realizar la invocación de un servicio, lo realiza utilizando una interfaz que provee el BUS con un protocolo determinado (**Componente gestión de los protocolos**). Una vez que el BUS recibe la invocación, dentro de su arquitectura se realiza una conversión del mensaje a un mensaje normalizado, que será el lenguaje común de los mensajes que se intercambian dentro del BUS (**componente Interfaz de Mensaje Normalizado**). En caso de ser necesario, el BUS realiza una transformación o enriquecimiento de los datos para colocarlos con información que sea la que espera el servicio a ser invocado (**componente Mensajes Enriquecidos**). El BUS dirige los mensajes hacia la verdadera ubicación del servicio (**componente**

Enrutamiento), lo cual puede ser realizado inteligentemente basado en los criterios definidos por el componente de **Calidad de Servicio**. Finalmente, el BUS desnormaliza el mensaje (**componente Interfaz de Mensaje Normalizado**) e invoca el servicio con el protocolo propio del servicio (**Componente gestión de los protocolos**).

2.2. Aplicaciones SOA Tolerantes a Fallas

El propósito de una aplicación web tolerante a fallas consiste en garantizar que la aplicación se recupere automáticamente de fallas internas, haciendo este proceso transparente al usuario. Básicamente, un sistema tolerante a fallas requiere de dos procesos, el diagnóstico de fallas y la reparación de las mismas. El diagnóstico de fallas consiste en determinar las causas de las fallas [42]. El diagnóstico de fallas consiste en las siguientes tareas [43]:

- **Detección de Fallas:** detección de la ocurrencia de irregularidades (fallas) en las unidades funcionales del sistema, que conducen a un comportamiento indeseable o intolerable de todo el sistema.
- **Aislamiento de fallas:** localización del sitio/componente generador de las fallas.
- **Identificación de Fallas:** especificación de las fallas que están ocurriendo, lo que implica determinar y caracterizar las fallas (clasificación).
- **Análisis de Fallas:** determinación del tipo, magnitud y causa de la falla.

En el diagnóstico de fallas en sistemas a eventos discretos (DES, por sus siglas en inglés), se utiliza un enfoque basado en eventos observables para establecer los estados en el sistema, logrando de esta manera el seguimiento del

comportamiento del sistema en el transcurso del tiempo [16]. En este enfoque, las fallas son vistas como eventos no observables, y sus ocurrencias son determinadas por la ocurrencia de secuencias ordenadas de eventos observables, los cuales son usados para estimar las fallas. Así, para el diagnóstico de DES se requiere observar secuencia de eventos observables.

Cuando el sistema se encuentra incurso en una o varias fallas del mismo tipo, se consideran todas como una sola falla; por el contrario, cuando varios tipos de fallas ocurren se dice que el sistema se encuentra inmerso en múltiples fallas. Se puede denotar al conjunto de todas las posibles fallas que podrían ocurrir en un sistema por P(F), cuyo conjunto de elementos puede ser [8]: $\emptyset$ (ausencia de fallas), $\{f_i\}$ (una sola falla f_i) o una combinación de fallas $\{f_i, f_j, ..., f_n\}$ (múltiples fallas $f_i, f_j, ..., f_n$).

Un modo de falla (F) corresponde a la conducta exhibida por el sistema, cuando tiene una falla o una combinación de ellas (presencia de algunas fallas y ausencia de otras). El diagnóstico de fallas consiste en determinar el conjunto de modos de fallas candidatos. Cuando el diagnóstico contiene un solo modo de falla se dice que este se puede discriminar ($F_i \cap F_j = \emptyset$), por el contrario, si el diagnóstico determina varios modos de fallas se dice que es indeterminado ($F_i \cap F_j \neq \emptyset$) [8]. Por ejemplo, los modos de fallas $F_{SYS} = \{\emptyset, F_1(\{f_k\}), F_2(\{f_i, f_j\})\}$ permiten diagnósticos deterministas, mientras que $F_{SYS} = \{\emptyset, F_1(\{f_i\}), F_2(\{f_j, f_k\}), F_3(\{f_k, f_j\})\}$ no permiten un diagnóstico deterministas (la combinación de las fallas de los modos de falla F_2 y F_3 son iguales).

Por otro lado, para garantizar la recuperación del sistema se deben contar con mecanismos que permitan trasladar el sistema en fallas a un estado que esté libre de error. Así, los mecanismos de recuperación están intrínsecamente vinculados con la dinámica del sistema, abarcando operaciones tales como rehacer, cambiar

configuraciones, utilizar alternativas de duplicación y/o remplazo de componentes del sistema, entre otras.

En el caso de las aplicaciones SOA, el problema se refiere a identificar las fallas que pueden acontecer en los distintos niveles de la aplicación, tanto a nivel del servicio como a nivel de la composición, y aplicar mecanismos de corrección que permitan corregir las fallas. A continuación se muestran las fallas y los mecanismos de corrección en la composición de servicios.

2.2.1. Fallas de Servicios

Una falla es un evento inesperado que ocurre en un sistema o componente, que altera su comportamiento. Un sistema o componente en el que una falla ocurre entonces se dice que se encuentra defectuoso. En la composición de servicios web, las causas de las fallas pueden ser dividas en tres grupos [6, 19]:

- **Fallas Físicas:** estas fallas se deben al entorno en el que opera el servicio (infraestructura), y no están relacionadas con la funcionalidad que ofrece el servicio. Esta falla hace que el servicio sea considerado como no disponible (Servicio o red de conexión del servicio se encuentran desconectados). El síntoma es que no es posible invocar el servicio.

  Para mostrar la secuencia de eventos que ocurren en esta falla, considere dos servicios S_1 y S_2, la ejecución normal de la secuencia de invocación de los servicios es S_1 en el instante T_1, luego S_2 en T_2 (ver Figura 2.9), donde $T_2 > T_1$. En caso de presencia de esta falla, S_1 se invoca normalmente (evento E_1 en T_1), pero S_2 nunca recibe el evento E_2 (ver Figura 2.9).

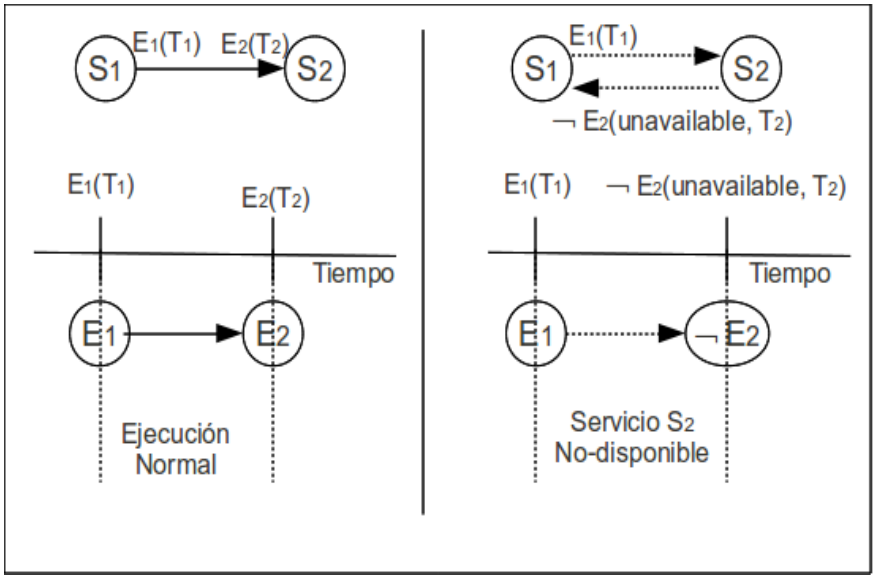


Figura 2.9: Eventos en la falla de servicio no disponible

- **Fallas de Desarrollo:** en el momento de la evolución de los servicios y/o de la composición, pueden emerger fallas que no son consideradas por el desarrollador de los servicios. Entre éstas se tienen:

  - ***Incompatibilidad de Parámetros (Parameter Incompatibility):*** esta falla surge cuando un servicio se invoca con valores y/o tipos de datos de los argumentos incorrectos, con respecto a los tipos y a las restricciones definidas en el documento WSDL. La ejecución normal de los servicios es igual al descrito en el caso de servicio no disponible. La secuencia de eventos en caso de presencia de esta falla es la siguiente (ver Figura 2.10): E_1 en S_1 es invocado normalmente en T_1, pero S_2 recibe el evento E_2 en T_2 con parámetros incompatibles. Es necesario señalar que la invocación del servicio S_2 no se ha realizado previamente (esta es la primera ejecución de S_2 en la composición).

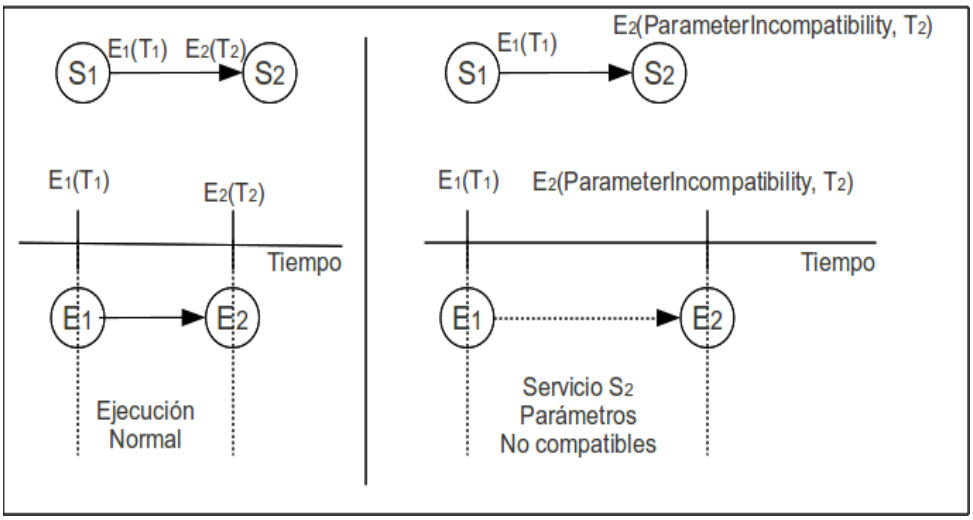


Figura 2.10: Eventos en la falla de Incompatibilidad de Parámetros

- ***Interfaz Podría Haber Cambiado (Interface Might Have Changed):*** la interfaz de algún servicio S_i que es parte de la composición se modifica, de modo que se origina una incompatibilidad de los parámetros cuando el Servicio S_i es nuevamente invocado (en ocasiones anteriores había sido invocado sin problemas). La secuencia de eventos es la siguiente (ver Figura 2.11): Previamente el servicio S_2 fue invocado sin problemas por S_1 (E_1, E_2 ocurren normalmente en el tiempo T_1 y T_2 respectivamente). En una invocación posterior a S_2 una falla de incompatibilidad de parámetros ocurre (E'_1 ocurre en T'_1 sin problemas, pero E'_2 en T'_2 genera un mensaje de Parameters Incompatibility).

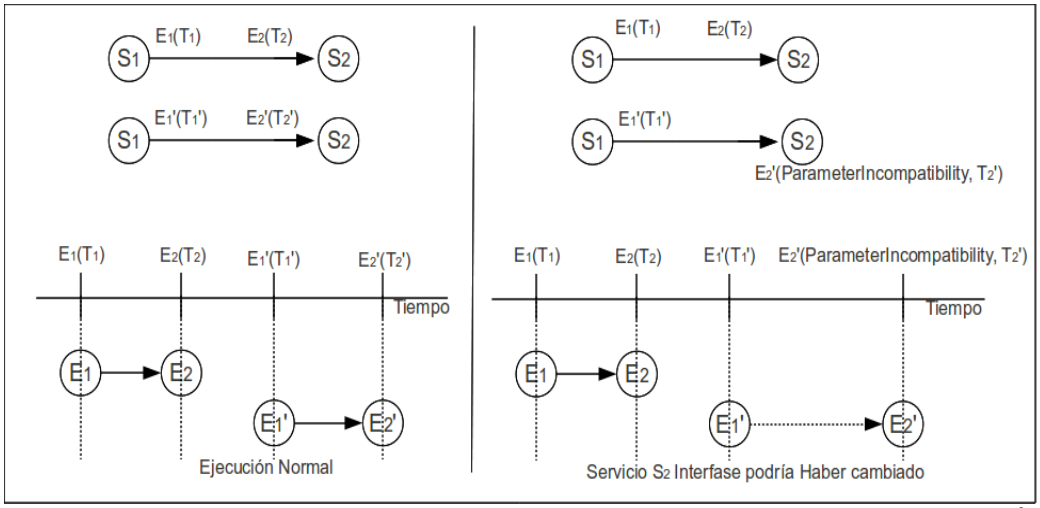


Figura 2.11: Eventos en la falla debido a que la interfaz podría haber cambiado

- ***Flujo de Trabajo Inconsistente (Workflow Inconsistent):*** esta falla ocurre cuando la lógica en el flujo no es correcta (Flujo de trabajo inconsistente). Este tipo de falla se le confunde con la falla física (falla debido a Servicio no disponible, Figura 2.9), ya que el servicio no puede ser invocado debido a que la información de invocación de las operaciones provistas en los WSDL no es correcto.

  Para mostrar la secuencia temporal de los eventos para esta falla, S_1 se invoca normalmente (evento E_1 en el instante T_1), pero el servicio S_2 nunca recibe la invocación (evento E_2 en T_2). Adicionalmente, para poder diferenciar esta falla con la de servicio no-disponible, es necesario utilizar mecanismos con operaciones que permitan tener certeza de que la localización invocada puede ser alcanzada. Para esto se puede utilizar servicios que invoquen la operación PING[8] (la operación PING debe devolver que el sitio está disponible).

8 PING es una instrucción que permite verificar si hay conexión entre dos sitios, enviando paquetes y devolviendo el tiempo que tardan en ir al destino y volver a la fuente.

- **Fallas en la Interacción:** En la composición ocurren interacciones entre los servicios, que pueden provocar fallas. Entre éstas se encuentran:
  - ***Fallas de Contenido:*** Estas fallas se producen debido a que los mensajes intercambiados entre los servicios que forman parte de la aplicación no son correctos. Ellas pueden ser de:
    - *Acción no-Determinada (Non-deterministic Actions):* esta falla se produce cuando el valor de la respuesta de un servicio no siempre genera el valor esperado que debe recibir otro servicio en la composición. Generalmente se debe a cambios en la configuración del servicio, o casos especiales que no fueron contemplados en el momento de desarrollar la operación del servicio, provocando que en algunos caso, por más que el servicio funcione correctamente, no es consonó con lo que espera otro servicio en la composición. Por lo tanto, para mostrar el patrón de eventos de esta falla (ver Figura 2.12), se asume que los servicios S_1 y S_2 anteriormente fueron invocados sin problemas en al menos una ocasión anterior (suceden los eventos E_1 y E_2 en los tiempos T_1 y T_2, donde $T_1 < T_2$). Luego, en una invocación posterior, el servicio S_1 produce una respuesta incoherente (contenido de la respuesta no es la esperada en E_2') en el instante T_1' que es percibida por el servicio S_2 en el evento E_2' en el instante T_2'.

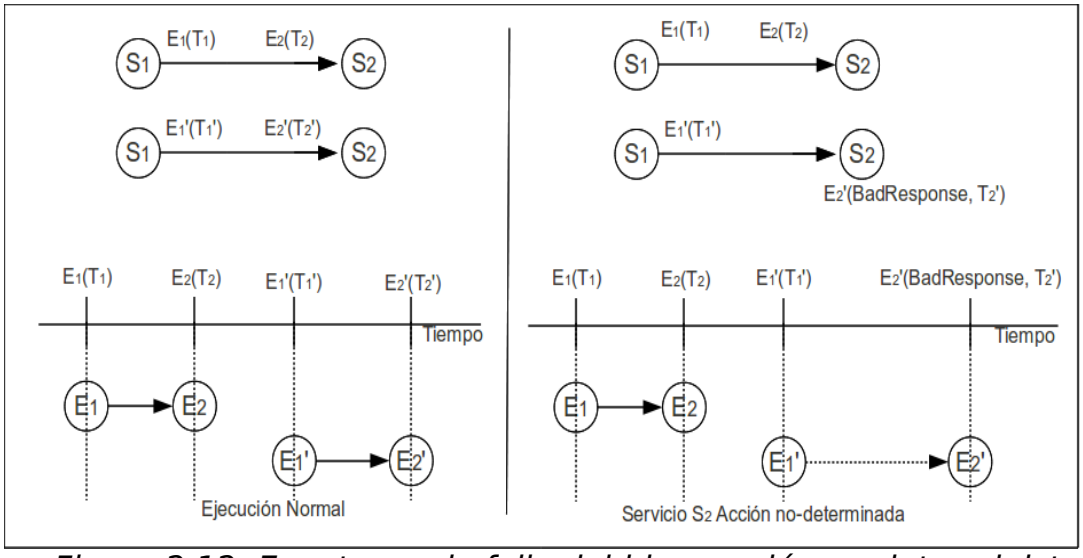


Figura 2.12: Eventos en la falla debido a acción no-determinista

- *Comportamiento Mal Interpretado:* esta falla se produce cuando un sub-flujo o servicio individual en la composición no produce los resultados esperados en su implementación, debido a que no se comprenden los resultados que se van obteniendo. La diferencia con la falla anterior es que los resultados suministrados por la composición nunca se comprenden, en Acción no-Determinada los resultados son correctos o erróneos. Existen varios casos en esta falla:

  - Comportamiento Incomprendido o Servicio Incorrecto (Misunderstood Behavior): uno de los servicios en el flujo de la composición no produce los resultados esperados. Eso no se debe a que el servicio no funciona correctamente (esté podría realizar la operación de la mejor manera posible), pero el resultado no es el esperado. Para mostrar un ejemplo de esta falla, asuma que cuando se invoca un servicio se espera que su respuesta sea la temperatura medida por hora, y el servicio devuelve la temperatura medida cada dos horas. Una vez más, suponemos dos

servicios que componen la composición: S_1 y S_2 (ver Figura 2.13). El servicio S_1 genera la respuesta en el instante T_1, pero la respuesta que se suministra a S_2 en el instante T_2 no es como se esperaba (Bad Response). Esta falla se debe a una mala descripción del servicio (ontología o documento WSDL), o simplemente el servicio pretende ser algo que no es.

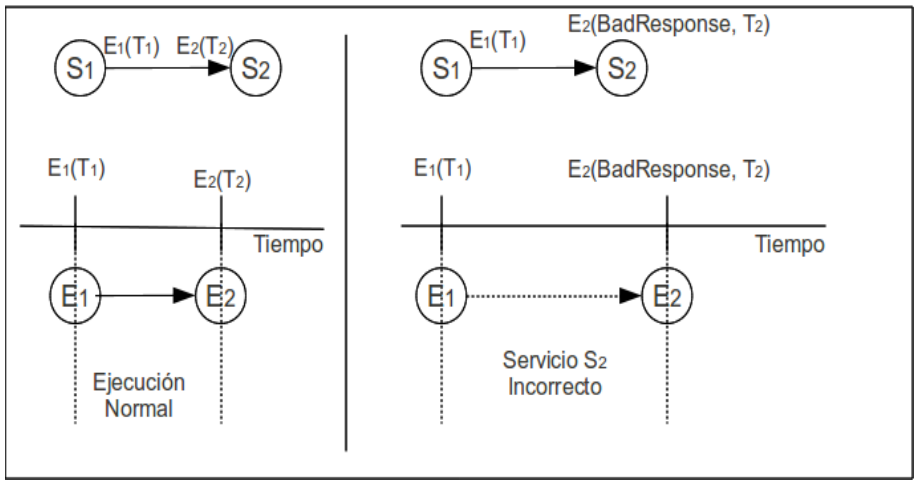


Figura 2.13: Eventos en la falla debido a servicio Incorrecto.

- Mal Comportamiento del Flujo Ejecución (Misbehaving Execution Flow): para lograr diagnósticar esta falla, previamente se debe conocer que el resultado generado por un servicio, que es anterior al sub-flujo de ejecución que genera problemas, funciona correctamente. La diferencia de esta falla con la de "acción no-determinada", consiste en que el resultado de la ejecución del sub-flujo que genera problemas en invocaciones previas, nunca ha generado resultados satisfactorios. Un ejemplo son los servicios S_1, S_2, S_3 y S_4 en la Figura 2.14, donde las respuestas de los servicios S_2 y S_3 son correctas de acuerdo a la forma en que se desarrollaron

(la lógica interna de los servicios), pero no cumplen con las expectativas del resultado esperado en el servicio S_4. Además, se sabe que el resultado generado en la invocación del servicio S_1 en el instante T_1 es correcto y consistente.

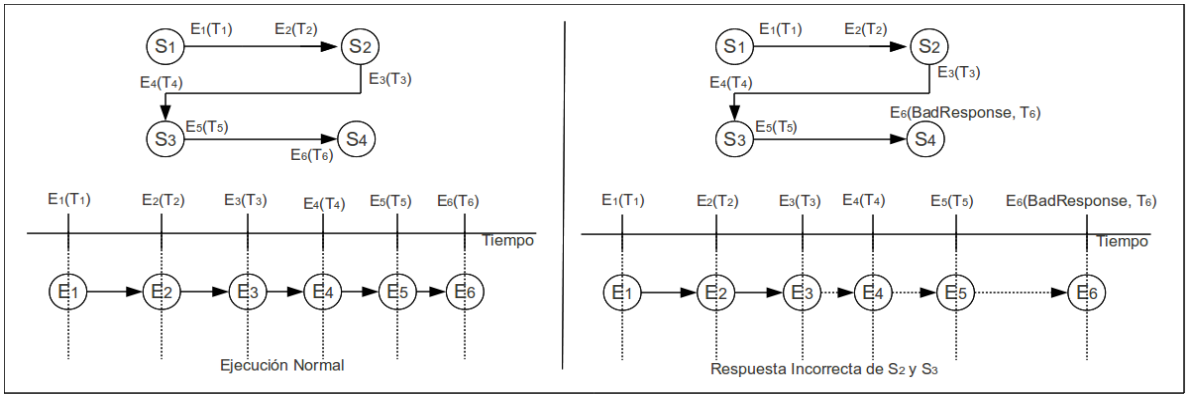


Figura 2.14: Eventos en la falla debido a un mal comportamiento del flujo Ejecución

- *Respuesta de Error (Response Error):* cuando la invocación de un servicio se realiza, se produce un fallo en su funcionamiento; esto puede ser debido a problemas de infraestructura, autenticación, o la lógica interna del servicio genera un mensaje de error. La secuencia de esta falla es simple (ver Figura 2.15): S_2 recibe la invocación del servicio en el evento E_2 en T_2, y produce una respuesta de error del servicio en ese sitio en el instante T_3 (evento E_3).

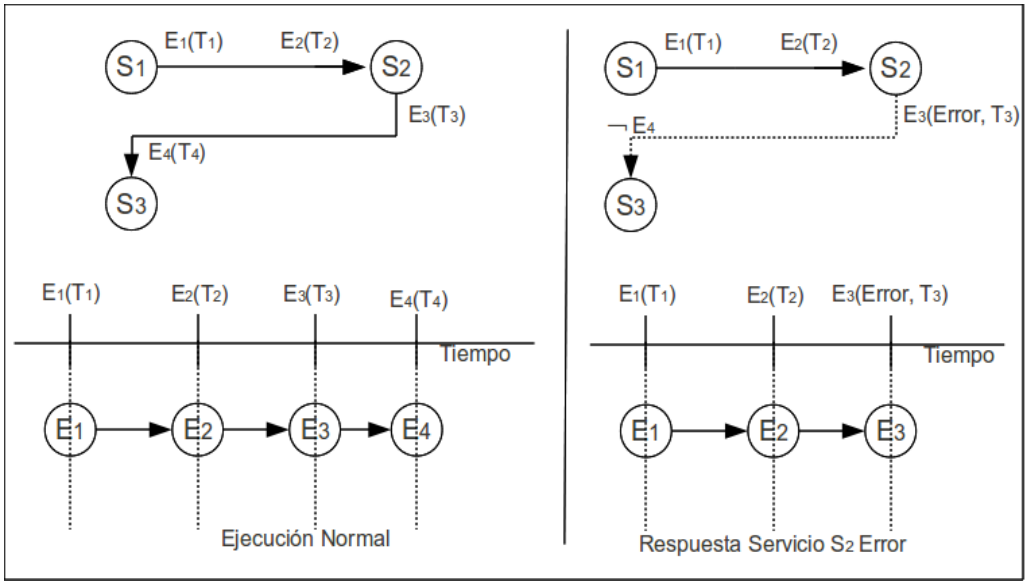


Figura 2.15: Eventos en la falla debido Respuesta de Error

- *Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (QoS):* las propiedades no funcionales de los servicios se expresan en términos de Acuerdos de Nivel de Servicio (SLA) y Calidad de Servicio (QoS). SLA se utiliza para describir las capacidades que deben tener los servicio, y QoS se utiliza para medir la calidad de los servicio basada en la calidad de la información generada. Esta falla se genera cuando alguna restricción de SLA y/o QoS ha sido violada. Para representar estas fallas suponga tres servicios S_1, S_2 y S_3; el servicio S_2 debe garantizar el cumplimiento de cualquier restricción de SLA o QoS. Por lo tanto, el servicio que es capaz de percibir la violación de la restricción es el servicio S_3 (ver Figura 2.16).

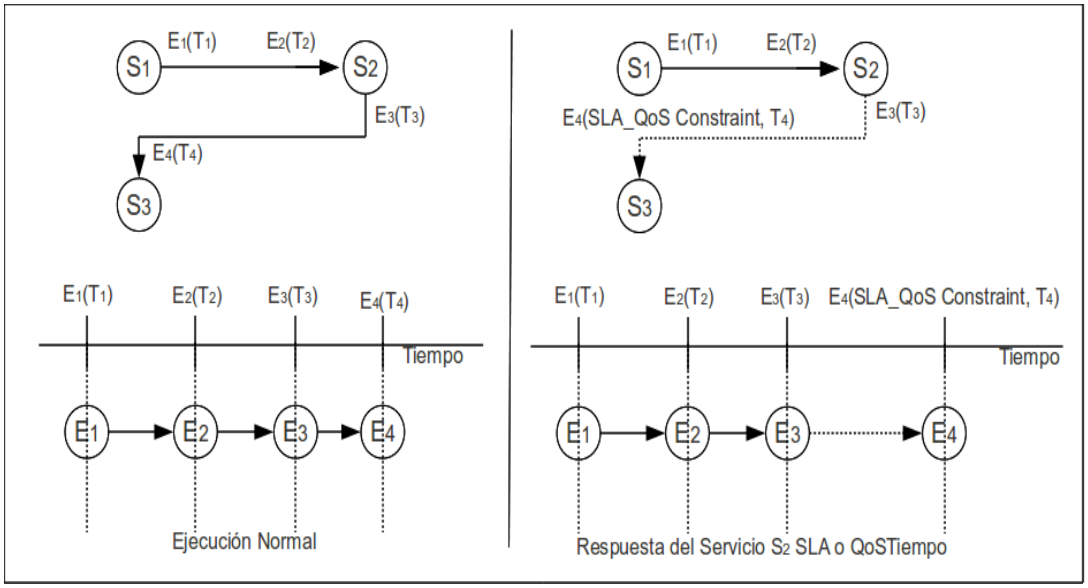


Figura 2.16: Eventos en la falla de Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (QoS)

- ***Fallas de Tiempo:*** estas fallas se producen debido a que el tiempo empleado para generar los mensajes intercambiados entre los servicios que forman parte de la aplicación, generan problemas en el flujo de ejecución. Entre ellas se pueden encontrar:
  - *Incorrecto Orden (Incorrect Order):* esta falla es debido a que los mensajes derivados de la interacción entre los servicios en la composición llegan en un orden diferente a como es definido en la coreografía. Para analizar esta falla, considere los servicios S_1, S_2 y S_3 en la Figura 2.17, tal que el evento E_5 tiene una dependencia de los eventos E_2 y E_4. Si E_2 y/o E_4 no llegan antes de que el evento E_5 suceda ($T_4 > T_5$ y/o $T_2 > T_5$), entonces la coreografía está en la presencia de esta falla.

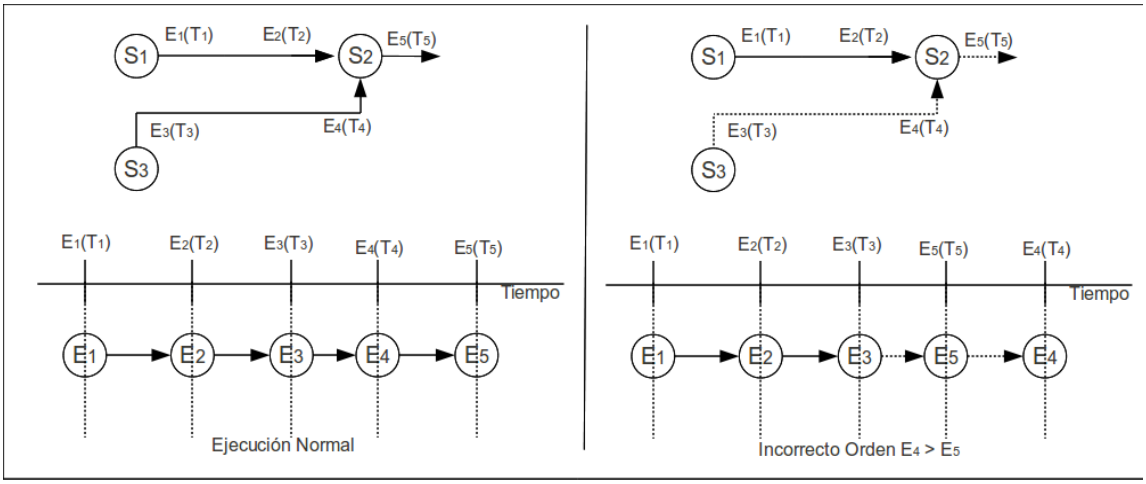


Figura 2.17: Eventos en la falla debido a Incorrecto Orden

- *Tiempo Fuera (Time-out):* cuando se describe la invocación de un servicio en la composición, se especifica un período de tiempo para su respuesta. La secuencia de eventos de esta falla se muestra en la Figura 2.18: suponga que los servicios S_1, S_2 y S_3 forman parte en la composición, el evento E_1 ocurre en S_1 en el instante T_1 (el servicio S_2 es invocado), entonces el evento E_2 ocurre (S_2 recibe el requerimiento en T_2), finalmente, no se recibe ninguna respuesta durante un periodo de tiempo del servicio S_3.

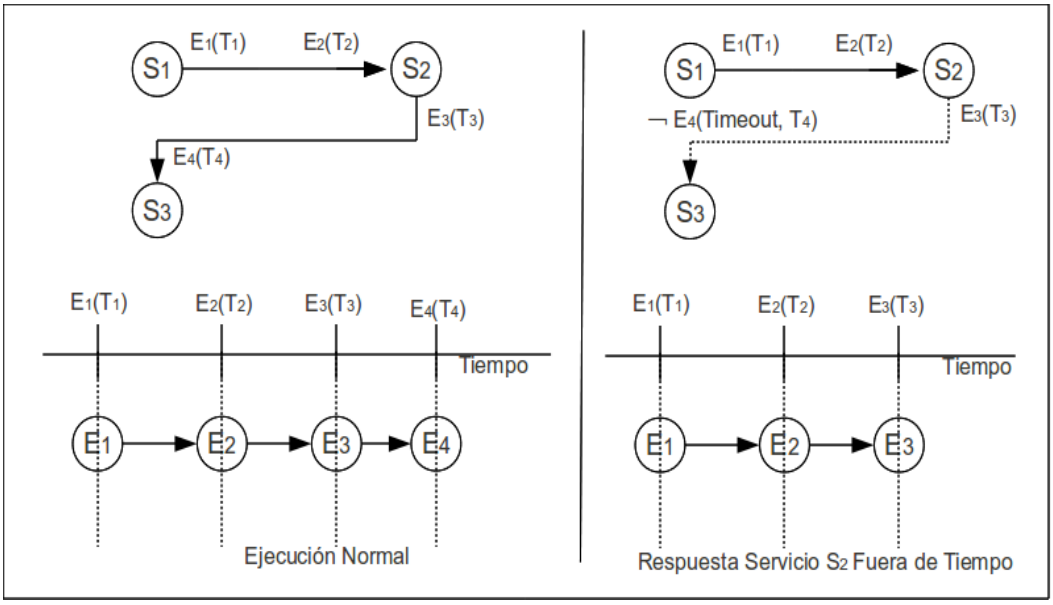


Figura 2.18: Eventos en la falla Fuera de Tiempo

- *Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (QoS):* funciona de la misma forma que las violaciones de SLA y QoS descritos en el aparatado de fallas de contenido, pero en vez de tomar en cuenta restricciones en la calidad de la información intercambiada, se consideran restricciones en base al tiempo de respuesta de los servicios involucrados en la aplicación.

En el contexto de aplicaciones SOA no se usan los términos de fallas débiles o fuertes, todas son importantes, y deben ser reparadas si uno quiere un funcionamiento correcto de la aplicación.

2.2.1. Métodos de Reparación

Una vez que es identificado un problema en una composición de servicios, es necesario realizar un conjunto de acciones sobre los servicios y/o la composición, para que la aplicación SOA pueda volver a funcionar sin problemas. Así, distintos

métodos de reparación han sido propuestos para solventar las fallas en la composición [9, 16, 19], los cuales son aplicados dependiendo del nivel en que la falla ocurre:

- **Servicio:** este método de reparación es aplicado solo a nivel del servicio, y no incluye la composición. Entre los métodos de reparación de este tipo se encuentran:
  - ***Reintentar (Retry):*** es aplicado cuando un servicio se encuentra temporalmente no disponible, y consiste en suspender la ejecución actual del servicio y reintentar la invocación del servicio para ver si vuelva a estar disponible.
  - ***Substituir el Servicio (Substitute a Service):*** es aplicado cuando un servicio es considerado definitivamente no disponible, y consiste en reemplazar el servicio por uno equivalente. La evaluación de compatibilidad se realiza usualmente comparando la interfaz de funcionalidad (WSDL) y los parámetros de calidad (QoS), pero pueden utilizarse otras especificaciones que describen al servicio, como el contrato de servicio SLA u ontologías. Esta solución abarca un gran abanico de fallas como: problemas de interfaces, no disponibilidad, mal comportamiento (acción no determinada, servicio incorrecto), entre otras fallas; debido a que se reemplaza completamente el servicio en la composición.
  - ***Completar Incompatibilidad de Parámetros (Complete Missing Parameters):*** en el momento de invocar un servicio, parte del mensaje o mensajes intercambiados son incompatibles con el establecido en el WSDL. La reparación consiste en colocar un servicio intermedio, que es

el encargado de compensar el mensaje de entrada del servicio. Se aplica cuando no es posible conectar directamente los servicios, requiriendo emplear servicios intermedios adaptadores. Esta reparación es muy usada en las fallas de incompatibilidad de parámetros y cambios de interfaces.

- ***Reasignar el Servicio (Reassign):*** este método de reparación es usado cuando el servicio no cumple con los parámetros de calidad (QoS) o SLA, la acción a realizar consiste en reasignar el servicio a una nueva localización para solventar el problema. A diferencia de la substitución del servicio, este método no consiste en buscar un servicio equivalente, sino invocar el mismo servicio en una nueva ubicación (servidor).

- ***Saltar un Servicio (Skip Service):*** consiste en saltar un servicio que forma parte de la composición, el cual se puede encontrar ejecutándose o aun no ha sido invocado, para continuar con el flujo de ejecución de la composición, y ejecutar la porción del flujo aún no ejecutado en un momento posterior. En los casos en que no es estrictamente necesario su ejecución, es posible extraerla del flujo hasta que se garantice su correcto funcionamiento, o sea reemplazada o retirada permanente de la composición.

- **Flujo:** Consiste en modificar el flujo de ejecución de la composición del servicio:

  - ***Substituir Flujo (Substitute Flow):*** es usado para solventar fallas en algún nivel de la composición. Este método consiste en substituir el flujo de las operaciones por flujos equivalentes, incorporando nuevos servicios o substrayendo servicios.

- ***Rehacer (Redo):*** consiste en repetir la invocación de un trozo del flujo de la composición, usando diferentes parámetros tomados de previas ejecuciones que han funcionado correctamente.

- ***Conductas Alternativas (Alternative Behavior):*** consiste en activar un flujo alternativo que debe seguir la composición, previamente definido, en caso de que una parte de está falle.

- ***Saltar Parte del Flujo (Skip Flow):*** es saltar una parte del flujo en la composición, que se puede estar corriendo en ese momento, o aún no se ha invocado, para continuar el flujo de ejecución de la composición.

- ***Ajustar Configuración (Change Settings):*** consiste en modificar el valor de una variable del proceso. Este método es usado cuando se necesita re-ejecutar una parte del flujo, pero usando diferentes valores de las variables del proceso.

2.3. Middleware Reflexivos

Un middleware es un software de capa intermedia, que permite comunicar sistemas heterogéneos en un sistema distribuido. Su tarea es la de asistir a las aplicaciones y usuarios en sus procesos de comunicación, logrando esconder la heterogeneidad inmersa en la red en que se encuentren. Los Middlewares usualmente residen en una capa intermedia [44], construida sobre el sistema operativo de un gran número de plataformas distribuidas.

En general, un Middleware permite esconder la heterogeneidad de los sistemas distribuidos, soportar modelos avanzados de coordinación entre entidades distribuidas, y realizar la distribución de los cálculos tan transparente como sea posible. Algunas funciones de un Middleware son [45]:

- Esconder la distribución de los usuarios y de las aplicaciones que se ejecutan sobre él.
- Esconder la heterogeneidad de los componentes de hardware, sistemas operativos y protocolos de comunicación.
- Proveer una interfaz estándar de alto nivel para el desarrollo e integración de aplicaciones, de tal manera que una aplicación pueda ser fácil de reusar, portar y mantener comunicación con otras aplicaciones.
- Suministrar un conjunto de servicios comunes para el desarrollo de funciones de propósito general para entornos distribuidos, en orden de evitar la duplicación de esfuerzos y facilitar la colaboración entre las aplicaciones

El papel jugado por un Middleware es hacer el desarrollo de aplicaciones una tarea más fácil, al esconder la complejidad de trabajar con la heterogeneidad y distribución inherentes al hardware, sistemas operativos, e incluso, lenguajes de programación.

El diseño actual de middlewares conciben arquitecturas que deberían ser capaces de adaptarse ellas mismas, basadas en una inferencia de cómo es su conducta actualmente, y las condiciones que presenta el ambiente en el que se desenvuelven. Así, un middleware adaptativo debería estar en la capacidad de modificar su conducta, para responder a cambios que se suceden en el ambiente donde se desenvuelve. Usualmente, la adaptación de los middleware se ha realizado en términos de adaptación paramétrica (variables son usadas para modificar el funcionamiento de variables o políticas del middleware), restandole el potencial de adaptación al ser necesario agregar nuevas funcionalidades para

adaptarlos a nuevos escenarios. Sin embargo, la reflexión ha permitido la construcción de middlewares más flexibles que se adaptan a su ambiente, al poder realizar cambios en su propia estructura y representación.

En cuanto a los Middlewares Reflexivos, partimos por definir el término reflexión. Una definición del término reflexión se refiere al proceso de razonar y actuar sobre uno mismo [46]. En términos computacionales, es la habilidad de un programa para observar y cambiar su propio código, así como aspectos de su lenguaje de programación (sintaxis, semántica o implementación), incluso en tiempo de ejecución. Se dice que un lenguaje es reflexivo cuando proporciona a sus programas la capacidad para realizar reflexión [47].

Las técnicas de reflexión confieren la capacidad de desarrollar sistemas computacionales sensibles a su ambiente, ya que le permiten al programa en que se usa la capacidad de reflexión, de exhibir un comportamiento dinámico.

Un concepto interesante a introducir es la meta-programación, vista como la capacidad de un programa de leer, transformar y escribir otros programas [48]. Así, la reflexión puede verse como un meta-programa que no es realizado sobre un programa externo, sino sobre sí mismo. Se pueden conseguir en la literatura distintos modos de subdividir la reflexión [47]: por el efecto que tiene la reflexión sobre el programa, por el momento en que se aplica la reflexión, entre otras. En esta documento se muestra la que es considerada más importante [47], que ésta basada en el efecto que tiene el meta-programa sobre el programa. En general, en este tipo de reflexión se encuentran dos procesos:

- **Introspección:** la capacidad de un componente para observar y razonar acerca de su propio estado de ejecución [47]. La introspección consiste en observar, y no puede alterar ningún aspecto de su propio código, solamente

examina su propia estructura y comportamiento.

- **Intersección:** es la habilidad de un componente para modificar su propio estado de ejecución (estructura), o alterar su propia interpretación o significado (comportamiento). En esta categoría se encuentran la reflexión estructural, que se refiere a aquella que reifica[9] los aspectos estructurales de un programa, pudiendo modificar jerarquía de herencias, estructura de clases y/o tipos de datos. Por otro lado, en la reflexión de comportamiento al sistema le es posible modificar el programa que se está ejecutando, insertando nuevas líneas de código dentro del intérprete (procesador) del programa que se encuentra en ejecución. Así, usualmente se aplica en lenguajes interpretados, al ser posible realizar modificaciones sobre su propia representación en tiempo de ejecución. Por el contrario, en la mayoría de los lenguajes compilados no es posible accesar o modificar su estructura en tiempo de ejecución; sin embargo, en algunos casos es posible realizarlo usando algunas extensiones, como en el caso de C++ con Open C++, MPC++ e Iguana.

Un Middleware Reflexivo consiste en aplicar las técnicas usadas en el campo de la reflexión, para lograr flexibilidad y adaptabilidad en los middlewares [49]. La reflexión hace al middleware lo que hace a cualquier sistema: ser más adaptable a su ambiente, y mejorar las condiciones para afrontar el cambio [47]. Una definición de Middleware reflexivos fue dada por Coulson [50]:

"*Un middleware reflexivo tiene internamente una representación de su propia conducta (Self-Representation), para permitir la inspección (Introspección) y la adaptación (Intersección) del middleware*"

9 Reificación es el proceso de hacer explícitos procesos o aspectos del sistema que inicialmente se encontraban implícitos, utilizando para ello representaciones del sistema para que estén disponibles.

En términos generales, un middleware reflexivo se refiere a la utilización de una auto-representación para soportar la inspección y adaptación del sistema [44]. Así, las mismas técnicas para aplicar la reflexión en las áreas tradicionales aplican igualmente para los middlewares reflexivos. En particular, la reflexión le permite al middleware la capacidad de realizar una configuración dinámica sobre una aplicación, realizando tareas de conexión y desconexión de componentes, y la modificación de sus propiedades, en tiempo de ejecución [51].

Normalmente, los middlewares reflexivos son usados para la gestión de propiedades no funcionales de los sistemas, por su facilidad y flexibilidad para realizar su reconfiguración en tiempo de ejecución, sin interferir con los usuarios del sistema. Así, los middlewares reflexivos se han aplicado con éxito para mejorar el desempeño de aplicaciones basado en la adaptación de las propiedades no funcionales como distribución, capacidad de respuesta, disponibilidad, fiabilidad, tolerancia a fallos, escalabilidad, seguridad, entre otros.

La arquitectura comúnmente empleada para el desarrollo de middlewares reflexivos se basa en la descomposición del Middleware en dos o más niveles. Así, el middleware reflexivo está compuesto por dos o más niveles (conocido como torre reflectante), en el que cada nivel está conectado con su nivel adyacente:

- El *nivel base* está compuesto por las entidades de bases que definen el comportamiento básico del sistema (realizan la funcionalidad de la aplicación).

- El resto de entidades que trabajan en los niveles superiores de la torre se conocen como *niveles meta*, y son las encargadas de realizar las acciones reflexivas sobre la capa inferior.

En el caso de un sistema reflexivo compuesto por solo 2 niveles (base y meta), el nivel meta está causalmente relacionada con las entidades del nivel base (reificación[10]), tal que cambios en las entidades del nivel base son percibidos por las entidades del nivel meta, y está compuesto por un modelo que refleja las actividades (auto-representación) de dicho nivel. El nivel base usualmente representa la implementación del sistema, y expone una meta-interfaz que se puede acceder desde el nivel meta, pero no tiene la capacidad de inferir la existencia de los niveles meta. Por otro lado, el nivel meta tiene la capacidad de observar y reflexionar acerca del comportamiento del nivel base, puede realizar el proceso de Introspección. Adicionalmente, el nivel meta tiene la capacidad de modificar el estado interno del nivel base, tal que realiza el proceso de Intersección. Esto permite desarrollar sistemas que modifican el comportamiento de las propiedades y/o funcionalidades definidas en el nivel base. En la Figura 2.19 se muestra la arquitectura de un sistema reflexivo compuesto por 2 capas, un nivel base y un nivel meta.

Generalmente, la intersección es realizada en dos fases [44, 52]: la primera es cuando la entidad base realiza una operación (por ejemplo la llamada a un método), que es interceptada por el nivel meta (introspección, al estar causalmente conectados, las operaciones del nivel base son reflejadas en el nivel meta). Luego, en la segunda fase la meta entidad toma el control y aplica la computación asociada a la operación pedida en el nivel base, permitiéndole trasladar la computación desde el nivel base al meta (la entidad meta se ha modificado por el cambio en el nivel base), para posteriormente regresar los resultados al nivel base (intersección, el nivel meta realiza cambio en sus entidades que son reflejadas automáticamente en el nivel base).

10 La reificación es una representación externa del funcionamiento de los componentes internos de un sistema que permiten ser manipulados en tiempo de ejecución.

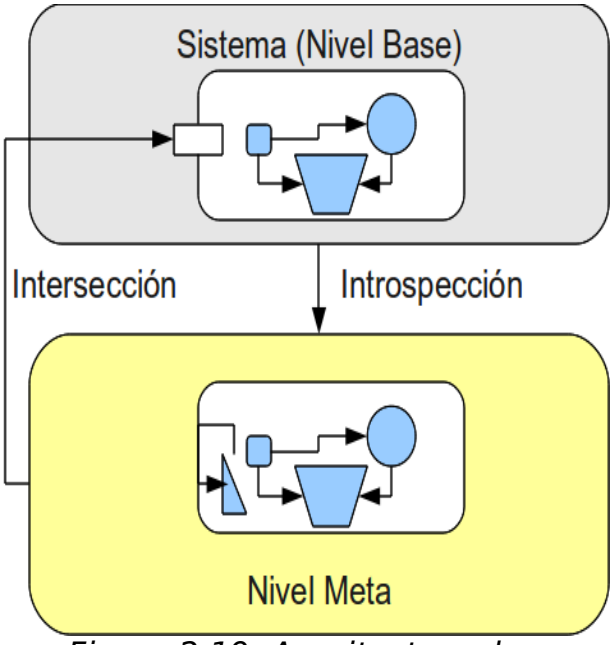


Figura 2.19: Arquitectura de un sistema reflexivo de dos niveles

2.4. Computación Autonómica

La Computación autonómica es un modelo de computación de auto-gestión, inspirado en el sistema nervioso de los seres humanos [53]. Permite el diseño de sistemas que son capaces de ser auto-administrados a alto nivel, manteniendo la complejidad del sistema invisible para el usuario. Incorpora sensores y actuadores en los sistemas, para permitir la recogida de datos sobre el comportamiento del sistema, y actuar en consecuencia.

Las capacidades de auto-gestión en los Sistemas Autonómicos permiten al sistema adaptarse a los cambios en el ambiente sin la intervención humana. La capacidad autonómica se basa en un bucle de control que recolecta datos del sistema, y realiza un conjunto de tareas de análisis para lograr un equilibrio en el comportamiento del sistema. La Computación autonómica ha sido usada para

auto-gestionar capacidades como:

- **Auto-Configuración:** permite al sistema adaptarse dinámicamente a cambios en el ambiente reconfigurándose.
- **Auto-Sanación:** confiere la capacidad de descubrir y diagnosticar fallas en el sistema, y en consecuencia repararse.
- **Auto-Optimización:** posibilita monitorear y ajustar el funcionamiento de los recursos del sistema, para lograr un buen desenvolvimiento del mismo.
- **Auto-Protección:** Permite detectar conductas inadecuadas, e introducir acciones de corrección y protección.

La computación autonómica define una arquitectura compuesta de 6 niveles (ver la Figura 2.20) [17]:

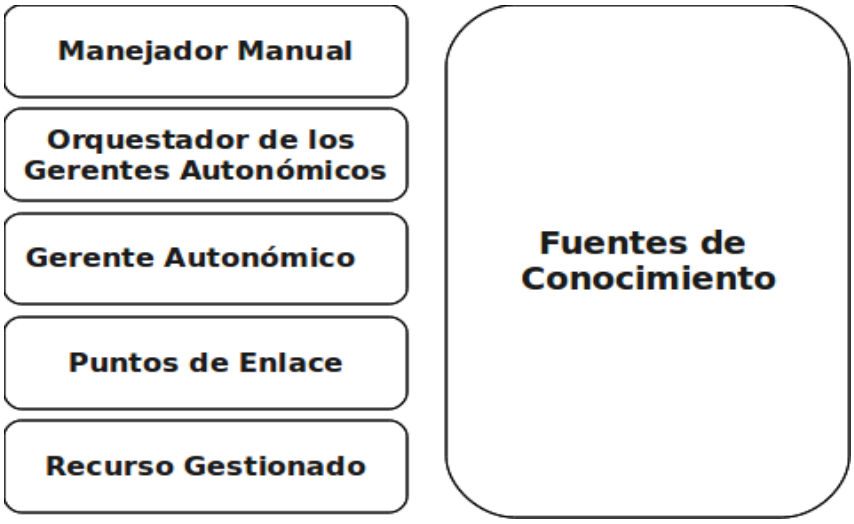


Figura 2.20: Arquitectura de Computación Autonómica

- **Recurso Gestionado:** puede ser cualquier tipo de recursos (hardware o

software) del sistema que pueda ser gestionado. El recurso gestionado se controla a través de sus sensores y actuadores.

- **Puntos de Enlace:** enlaza a los sensores y/o actuadores requeridos para la gestión de los recursos. Los sensores proporcionan mecanismos para recopilar información acerca de los cambios de estado y las transiciones de un recurso, y los actuadores son mecanismos que cambian su estado (configuración).

- **Gerente Autonómico:** implementa los lazos de control inteligentes que automatizan las tareas de auto-regulación/autogestión del sistema. Está compuesto por cuatro módulos que caracterizan el lazo de control autónomo: Monitoreo, Análisis, Planificación y Ejecución (MAPE). A continuación se describen en detalles (Figura 2.21):

  - ***Monitor:*** recolecta los eventos/datos de los sensores sobre lo que está aconteciendo en los recursos gestionados, convirtiéndolos en un conjunto de patrones o síntomas que pueden ser analizados por el componente de Análisis.

  - ***Análisis:*** cuenta con mecanismos para interpretar y analizar los patrones provistos por el monitor, para determinar las causas de los problemas en los recursos gestionados. Al momento de encontrar las causas que están afectando al recurso, se lo comunica al módulo de planificación.

  - ***Planificación:*** genera el conjunto de operaciones a ejecutar sobre el recurso gestionado, a fin de alcanzar las metas y objetivos propuestos; dicho plan podría consistir en crear o seleccionar procedimientos para

lograr la alteración deseada. Este módulo elabora el conjunto de procedimientos a realizar, que envía al componente de ejecución.

- ***Ejecución:*** ejecuta el conjunto de acciones necesarias contenidas en el Plan. Provee una interfaz para la comunicación entre el gestor autonómico y el recurso gestionado (interactúa con los actuadores).

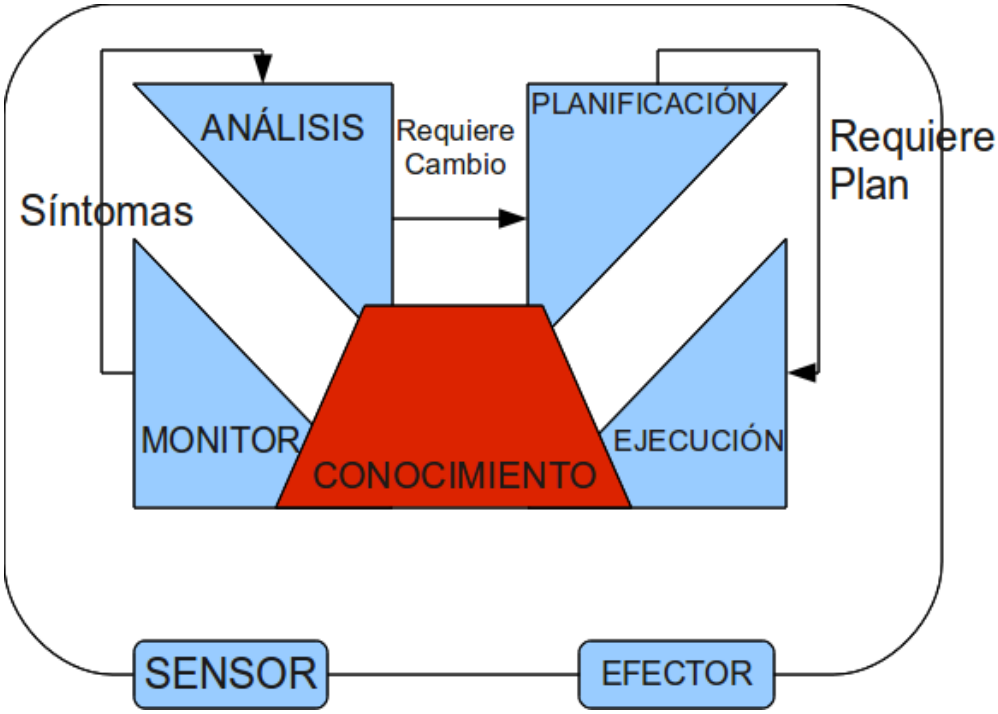


Figura 2.21: El componente MAPE

- **Orquestador de los Gerentes Autonómicos:** debido a que un sistema autonómico puede contar con varios gerentes autonómicos que necesitan trabajar en conjunto para garantizar el funcionamiento correcto de los recursos, este nivel proporciona el canal de comunicación para la coordinación entre los distintos gestores autonómicos.

- **Manejador Manual:** permite a los humanos configurar los gerentes

autonómicos para realizar sus tareas de autogestión, proveyendo para esto una interfaz hombre-máquina que permite conectar al hombre con el gerente autonómico.

- **Fuentes de Conocimiento:** proporciona acceso a los conocimientos requeridos para la gestión autonómica del sistema. Los distintos componentes del gerente autonómico (monitor, analizador, planificador y ejecutor) hacen uso de estas fuentes de conocimientos para realizar sus tareas. Los distintos componentes del MAPE pueden no solo utilizar las fuentes de conocimiento, sino además podrían generar nuevo conocimiento basado en sus interacciones con los recursos gerenciados.

2.5. Crónicas

Las crónicas son patrones temporales que representan la dinámica del sistema observado [55]. Una crónica es un conjunto de eventos, con restricciones temporales entre ellos y con respecto al contexto, que representa una interpretación de lo que está ocurriendo en la dinámica del sistema en estudio en un momento dado [32]. Así, cada crónica representa una situación o escenario de desempeño normal o anormal del sistema, por lo que puede ser vista como el patrón de comportamiento del sistema en esa situación. Como ya se dijo, ella está compuesta por un grupo de eventos observables, vinculados temporalmente según su momento de ocurrencia. Una crónica podría generar nuevos eventos y acciones al momento de reconocerse su ocurrencia, las cuales podrían ser usadas como entradas por otras crónicas (se da un proceso de inferencia entre crónicas [16]).

De esa manera, podemos definir a una crónica C como "un par (E,T), donde E es el conjunto de eventos y T un conjunto de restricciones entre sus tiempos de

ocurrencia” [12].

Un evento en las crónicas define qué es lo que se observa en el sistema en un instante de tiempo dado, y puede ser descrito de diferentes formas [32]:

- El nombre del evento/actividad observada (act).
- El nombre de la actividad enriquecida con el hecho de que ella está comenzando (nombre act^{-}) o terminando (nombre act^{+}).
- El nombre de la actividad enriquecida con algunos parámetros (variables) que deben ser observados cuando ocurra el evento/actividad (act(?var_1 ,... . , ?var_n)).
- Una combinación de las dos últimas:
  - act^{-} (?var_1 , . . . , ?var_n) "La actividad act está comenzando con los parámetros ?var_1 ,. . .?var_n ".
  - act^{+} (?var_1 , . . . , ?var_n) "La actividad act está terminando con los parámetros de retorno ?var_1 , . . .?var_n".

En general, en las crónicas es necesario definir el par (E, ?t), donde E es el nombre del evento (descrito en algunas de las formas antes señaladas) y ?t el tiempo de ocurrencia del evento. Adicionalmente, un conjunto de acciones deberían ser definidas para ser ejecutadas cuando una crónica es reconocida, dichas acciones no solo deberían limitarse a la generación de reportes, sino que podrían generar eventos que servirían para activar nuevas crónicas, como se dijo antes.

Al realizar el diagnóstico de fallas usando crónicas, cada falla es reconocida por

una o más crónicas, permitiendo establecer un vínculo eficiente entre los síntomas de la falla (firma de falla) y la falla en sí.

2.5.1. Representación del Modelo de Crónicas

La representación de las crónicas se basa en el formalismo de lógica temporal [11], donde los términos proposicionales son relacionados con el tiempo u otros proposicionales haciendo uso de predicados. Así, para poder representar las crónicas se hacen las siguientes consideraciones:

- **Representación del Tiempo:** por razones de complejidad algorítmica, el tiempo se considera como conjuntos discretos y linealmente ordenados de instantes, cuya granularidad es lo suficientemente fina para permitir representar la dinámica del sistema en estudio. Los intervalos de tiempo son expresados como el par $I = (t_1, t_2)$, y corresponden a los limites superior e inferior de la distancia entre los puntos de tiempo t_1 y t_2.

- **Atributos de Dominio:** corresponden a las proposiciones atemporales que permiten describir el ambiente. Están representados por la tupla $P (a_1 , . . ., a_n) : v$, donde P es el nombre del atributo, $a_1 , . . ., a_n$ sus argumentos, y v su valor. Un caso particular es el tipo de atributo llamado mensaje, donde al atributo no se le asigna ningún valor.

- **Predicados:** sirven para establecer el vínculo entre el componente temporal y las proposiciones atemporales. La lógica es representada por el lenguaje de predicados de primer orden. Los predicados usados en las crónicas son:

  - ***hold(P : v, (t_1, t_2)):*** establece que un atributo en el dominio P mantiene el valor v en el intervalo $[t_1, t_2]$.

- ***event(P : (v_1 , v_2), t):*** establece que el atributo en el dominio P cambia su valor de v_1 a v_2 en el instante t.
- ***event(P, t):*** establece que el evento P ocurre en el instante t.
- ***noevent(P, (t_1, t_2)):*** establece que no ocurre el evento P en el intervalo [t_1, t_2].
- ***occurs((n_1, n_2), P, (t_1, t_2)):*** corresponde a la restricción de ocurrencia del evento P en el intervalo de tiempo [t_1, t_2] según un número de ocurrencia en el intervalo [n_1, n_2].

- **Acciones:** es posible especificar la ejecución de cualquier acción o efectos externos a generar, cuando el reconocimiento de una instancia de crónica ocurre. El reconocimiento también podría ser usado para producir nuevos eventos que no son los eventos observados por la dinámica del sistema, sino más bien son consecuencia de la inferencia de ellos.

Un modelo para representar las crónicas consta de [11]:

1. Un conjunto de tiempos (componente temporal de la crónica)
2. Un conjunto de eventos que se corresponden con los eventos observados desde el mundo exterior.
3. Un conjunto de restricciones temporales en que deben suceder los eventos.
4. Un conjunto de patrones de afirmaciones, que describen el contexto (eventos, etc.) en el que ocurre una crónica en un momento dado.
5. Acciones a ejecutar durante un reconocimiento de una crónica.

Así, una crónica puede ser escrita de la siguiente forma:

```
1   chronicle FallaServicio() {
2     //tiempos que caracterizan la crónica
3     timepoint tA, tB, tC, tD0, tD1;
4     //Eventos considerados
5     event(EA, tA);
6     event (EB, tB);
7   //las aserciones (El contexto de la crónica)
8     event (EC:(NoFault, Fault), tC);
9     hold (ED: Fault, (tD0, tD1));
10    //- Las restricciones entre los instantes
11    tA < tB;
12    (tC - tB) in [1, 6];
13    (tD0 - tC) in [2, 6];
14    when recognized {
15      //Acciones a tomar cuando el reconocimiento se ha activado
16      generate(eventF)";
17    }
18  }
```

Figura 2.22: Ejemplo de modelo para representar las crónicas

2.5.2. Reconocimiento de Crónicas

El reconocimiento de crónicas consiste en ir analizando el conjunto de eventos que van aconteciendo en el sistema, para lograr el reconocimiento de los patrones temporales que componen una crónica. Para esto, se reciben como

entrada una secuencia de eventos con sus tiempos de ocurrencia, y se van instanciando todas las crónicas que coinciden con el patrón temporal, hasta que una o más crónicas son reconocidas o son descartadas por completo.

Una herramienta para el reconocimiento de crónicas, llamado CRS (Chronicle Recognition System), ha sido desarrollada por Dousson [10, 32]. Esa herramienta consiste en analizar el flujo de eventos y de reconocer, en tiempo real, cualquier patrón de coincidencia con una situación descrita por una crónica. Cuando un nuevo evento es registrado en el sistema, nuevas instancias de las crónicas almacenadas en la base de crónicas son almacenadas en el conjunto de hipótesis. Ese conjunto de hipótesis es evaluado continuamente, a veces reconociéndose un patrón temporal (crónica), o descartándose instancias al violarse restricciones temporales en las crónicas de esas instancias.

En la Figura 2.23 se muestra un ejemplo de reconocimiento de la crónica de la Figura 2.22 (Crónica "FallaServicio"). A medida que van llegando los eventos al reconocedor de crónicas, van creándose instancias de las crónicas que comienzan con esos eventos, o se siguen reconociendo instancias ya existentes en el conjunto hipótesis, o se descartan las instancias en el conjunto hipótesis a las cuales se les violan sus restricciones temporales, o se reconoce un patrón al reconocerse el último evento de una instancia en el conjunto hipótesis.

Para el ejemplo en específico, el reconocedor de eventos detecta el evento E_A en el instante $T_A = 1$, y una instancia I_1 es creada en el reconocedor esperando recibir el evento E_B (restricción $T_B > T_A$). Posteriormente llega el evento E_B en el instante $T_B = 2$, y la instancia I_1 se mantiene en el reconocedor, ahora esperando el evento E_C. Luego, un evento E_x llega en el instante $T_x = 3$ con la variable de no falla (*NO-FAULT*), que el reconocedor no considera porque no corresponde al evento E_C (aún cuando el tiempo de llegada es correcto, en el evento E_C ocurre un

cambio del valor de la variable a *FAULT*). Así, la instancia I_1 sigue esperando el evento E_C (ya que todavía en ese momento, la restricción temporal entre los eventos E_B y E_C no ha sido violada).

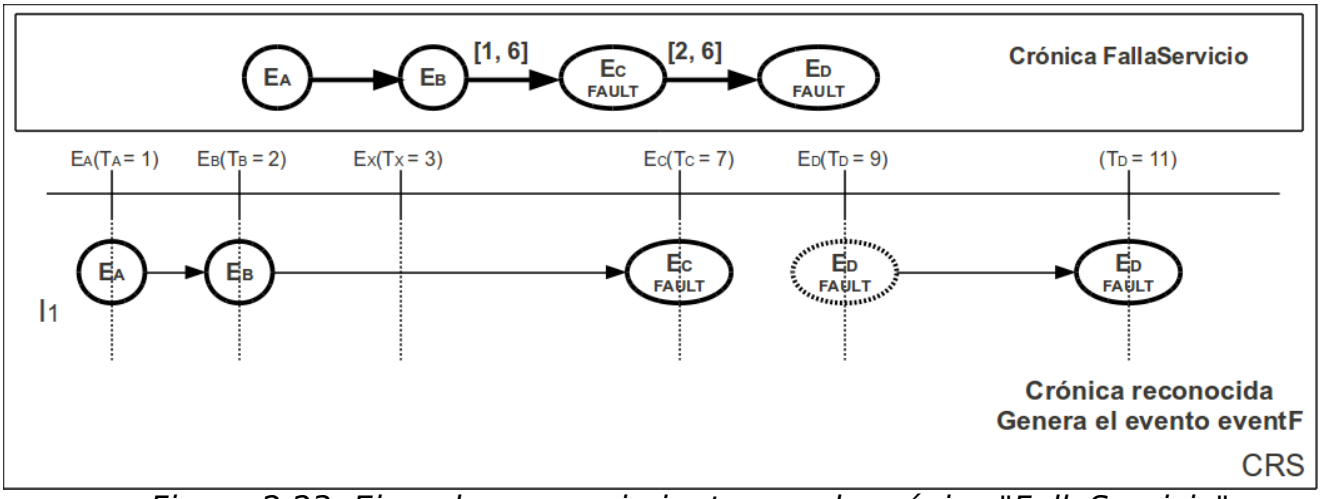


Figura 2.23: Ejemplo reconocimiento para la crónica "FallaServicio"

Continuando con la ejecución del reconocedor, un evento E_C llega con la variable Falla igual a *FAULT* en el instante $T_C = 7$ y la instancia I_1 ahora pasa a esperar el evento E_D. En un instante posterior ($T_D = 9$), llega el evento E_D con el valor *FAULT*, y se mantiene durante 4 unidades de tiempo, por lo que la crónica es reconocida y el evento eventF es generado.

Dos extensiones llamadas CarDeCRS [12] y Matrac [71] fueron desarrolladas para permitir el diagnóstico descentralizado de crónicas en la composición de servicios [12]. Ambas herramientas Matrac y CarDeCRS utilizan el lenguaje CRS de la herramienta desarrollada por Dousson [10, 32].

2.5.3. Firmas de Fallas

Una firma de falla se define como el conjunto de todas las tuplas de estados

observables posibles bajo una falla [8]. El propósito de la firma de falla es la identificación del conjunto de estados de las variables presentes en el momento que ocurre una falla [8]. Si una falla es detectada, las variables del proceso que contribuyen con está son determinadas, como sus contribuciones relativas. Así, cada firma de falla es asociada con al menos una clase de falla, y el monitoreo del sistema consiste en analizar los estados observables del sistema, y reconocer los patrones descritos en las firmas de fallas. Una manera de implementar firmas de fallas consiste en representarlas en formas de Crónicas [32].

En las firmas de fallas, un proceso fundamental es el mapeo de firmas (Signature matching en inglés), el cual es un método para identificar variaciones de los valores medidos de las variables del sistema. Así, en este proceso se determina el conjunto de elementos (variables) que se acoplan con las firmas previamente almacenadas (se realiza una consulta a la base de firmas de fallas). En algunos casos, la mayoría de los elementos no se ajustan exactamente a una firma, pero podrían ser parcialmente similares (esto es conocido como modo de firma de falla parcial) [8].

En ese orden de ideas, el proceso signature matching puede definirse como [8]:

$$\text{Signature Matching: } (q,M,E) = \{e \in E: M(e,q)\} \tag{2.1}$$

Donde q es la consulta (valores reales de las variables observadas en el sistema), M es el predicado de acoplamiento (match) usado (como se dijo anteriormente, se pueden usar diferentes niveles de acoplamiento (match), pudiendo ser matching exactos u otros más relajados en sus criterios (parciales)), y E es el conjunto (base de datos) de firmas de fallas; signature matching retorna el conjunto de elementos (firmas de fallas) que satisfacen el acoplamiento

(mapeo).

Así, las firmas de fallas pueden ser usadas como un mecanismo de detección, que consiste en inferir el estado del sistema (si se encuentra en falla o no) usando el mecanismo de coincidencia de firmas.

Además, como se dijo antes, se pueden tener funciones de acoplamiento (M) con cierto relajamiento. De esta manera, usando firmas de fallas se podría realizar un diagnóstico de fallas (determinando, incluso, la contribución relativa de cada una de las variables en la falla).

Capítulo 3
ARMISCOM

En este Capítulo se presenta una arquitectura para la tolerancia a fallas en aplicaciones SOA, basadas en servicios web, completamente distribuida, llamada ARMISCOM. En particular, ARMISCOM es un middleware reflexivo autonómico que permite la auto-sanación, en el que el diagnóstico de las fallas se realiza a través de la interacción de los diagnósticadores locales presentes en cada servicio y la reparación se realiza a través de un proceso consensuado por los reparadores que están también distribuidos a través de los servicios.

3.1. Arquitectura de ARMISCOM

ARMISCOM (Autonomic Reflective MIddleware for management Service COMposition) es un middleware reflexivo autonómico, cuya capacidad reflexiva le permite controlar y cambiar el comportamiento de las aplicaciones SOA que se encuentran ejecutándose sobre él, cuando las mismas presentan fallas. De esta manera, ARMISCOM tiene un comportamiento dinámico y adaptativo, por ejemplo, puede gestionar aplicaciones SOA que son capaces de cambiar de forma dinámica o evolucionar. En general, los dos procesos clásicos de reflexión, para el caso de ARMISCOM, consisten en [56]:

- **Introspección:** consiste en analizar el intercambio de mensajes entre los servicios que forman parte de la composición y los componentes del sistema SOA. Para alcanzar un nivel adecuado de introspección, ARMISCOM debe observar tanto el sistema SOA, como la aplicación SOA ejecutándose en él. Para esto, se utilizan las especificaciones y estándares para servicios

web: WSDL[11], UDDI[12], y OWL-S[13].

- **Intersección:** es la capacidad de ARMISCOM para modificar la ejecución de la aplicación SOA. Para esto, modifica el flujo de la composición de los servicios, reescribiendo las especificaciones de la aplicación: WC-DL, enlaces dinámicos, entre otros.

Como ya se ha señalado, ARMISCOM está distribuido a través de todos los servicios del sistema, con el fin de tener una visión más cercana (local) de la ocurrencia de los eventos que suceden en los servicios que componen la aplicación. ARMISCOM se divide en los clásicos dos niveles de todo middleware reflexivo (ver Figura 3.1):

- **Nivel Base:** en este nivel se ejecuta la composición de servicios que forman parte de una aplicación SOA, el conjunto de normas y definiciones que rigen esas interacciones (Sistema SOA).

- **Nivel Meta:** en este nivel se encuentra la capacidad reflexiva del middleware. Para ello, el nivel meta hace las tareas de introspección, y determina las operaciones de intersección a realizar. En particular, este nivel despliega gran parte de la arquitectura autonómica del middleware, para analizar el estado actual y determinar las tareas de tolerancia a fallas a realizar.

11 Web Services Description Language (WSDL) Version 2.0 Part 1, http://www.w3.org/TR/wsdl20.

12 Universal Description, Discovery and Integration, http://uddi.xml.org/.

13 OWL-S: El marcado semántico para servicios web, http://www.w3.org/Submission/OWL-S/.

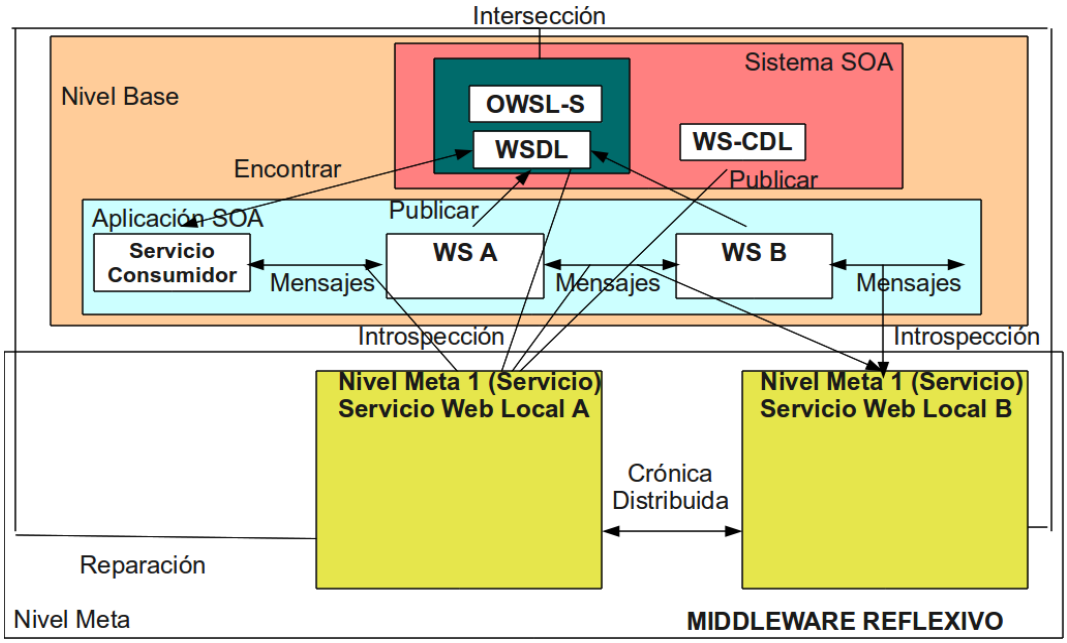


Figura 3.1: Arquitectura de ARMISCOM (Middleware Reflexivo)

3.2. Componente Autonómico de ARMISCOM

En ARMISCOM, los diferentes niveles de una arquitectura autonómica tienen las siguientes características:

- **Managed Resource (recursos administrados):** corresponden a los servicios de la aplicación SOA que se están supervisando.

- **Touch Point (punto de enlace):** permite realizar la conexión con los recursos administrados. Para esto, ARMISCOM intercepta los mensajes intercambiados en formas de eventos (sensores) y modifica el flujo de la aplicación supervisada, teniendo acceso de escritura a las especificaciones de la composición (actuadores).

- **Autonomic Manager (Gestor Autonómico):** automatiza las tareas de

auto-regulación de las aplicación SOA, para esto el gestor autonómico se distribuye en cada servicio que forma parte de la composición (un gestor por cada servicio).

- **Orchestrating autonomic managers (Orquestador gestores autonómicos):** Permite la iteración de los gestores autonómicos, proveyendo los mecanismos para permitir la comunicación de las crónicas distribuidas y la actualización de la metadata.

- **Manual Manager (Administrador manual):** provee la interfaz para modificar las crónicas y la metadata manualmente.

- **Knowledge Sources (Fuentes de Conocimiento):** contiene las fuentes de conocimiento que necesita la arquitectura para su correcto funcionamiento. ARMISCOM utiliza las crónicas para realizar el diagnóstico, una ontología para inferir los mecanismos de reparación a aplicar, y una metadata para ejecutar la reparación cuando la aplicación SOA se encuentra inmersa en una falla.

Por otro lado, ARMISCOM integra en los dos niveles del middleware reflexivo (niveles base y meta) y los seis niveles de la arquitectura de computación autonómica, de la siguiente manera (ver Figura 3.2):

Los manejadores de recursos y los puntos de enlace forman parte del nivel base del middleware; el Manejador Autonómico, la Fuente de Conocimiento y el Orquestador de los Gerentes Autonómicos forman parte del nivel meta. En los puntos de enlace se realiza la interacción con los servicios, con el fin de recuperar los datos de seguimiento (introspección) y ejecutar las decisiones de reparación definidas (intersección). Por otro lado, el Manejador Autonómico está compuesto

por dos componentes un diagnosticador y un reparador, los cuales son equivalentes a la estructura MAPE de la arquitectura clásica de computación autonómica. El diagnosticador observa el sistema y analiza las fallas y el reparador define el plan de reparación y ordena su ejecución. Además, la fuente de conocimiento está compuesta por un ontología que contiene, entre otras cosas, la taxonomía de fallas y mecanismos de reparación para aplicaciones SOA; una base de crónicas distribuida que almacena los patrones genéricos de fallas; y una metadata que describe los mecanismos de reparación disponibles en cada sitio.

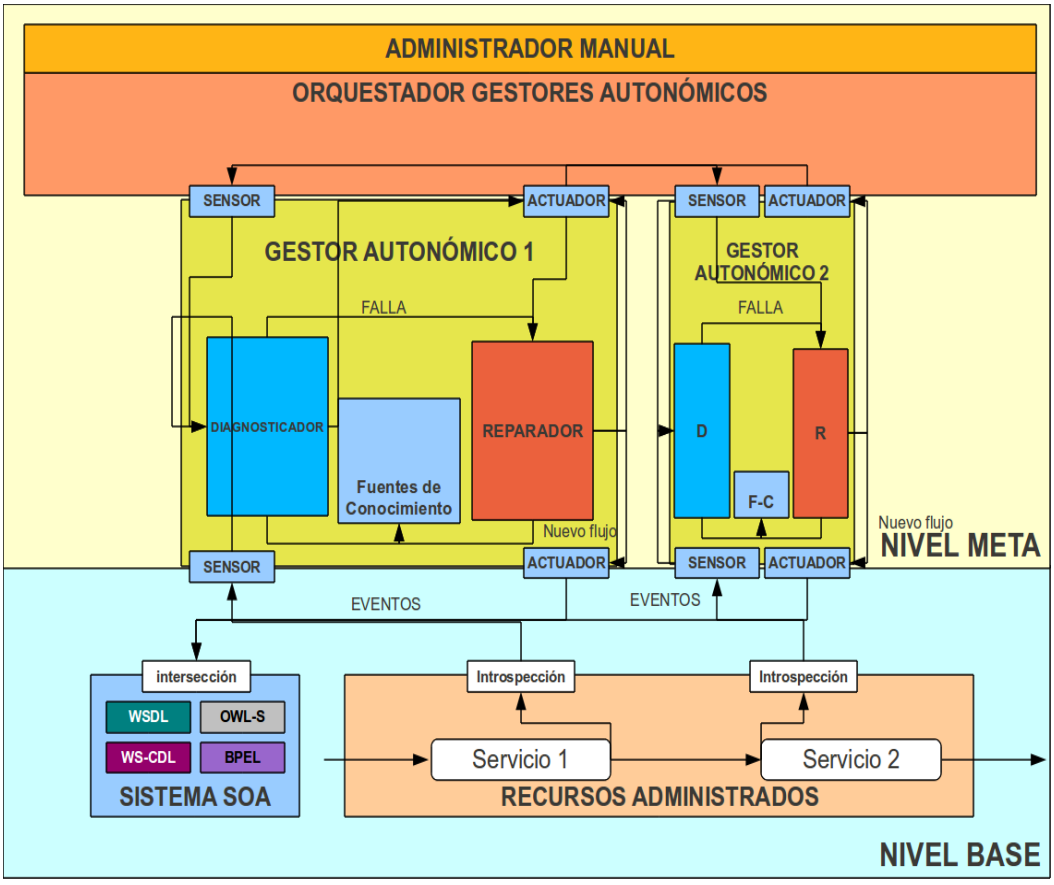


Figura 3.2: Arquitectura del Middleware basada en Computación Autonómica

3.3. MAPE-K de ARMISCOM

Como se comentó en la sección anterior, el gestor autonómico está compuesto por dos módulos del nivel meta: el diagnosticador y el reparador, y una fuente de conocimiento, que corresponden a la estructura MAPE-K de la arquitectura de computación autonómica. Cada gerente autonómico trabaja a nivel local (en cada servicio), para analizar las fallas internas del servicio, y a partir de las interacciones con los otros gestores autonómicos, analiza globalmente las fallas en las aplicaciones basadas en servicios. Por lo tanto, este componente debe comunicarse con el resto de los diagnosticadores de los otros servicios en la composición. Un comportamiento parecido tiene el reparador, para determinar y realizar las tareas de reparación, que solventen la falla en la aplicación SOA.

3.3.1. El Diagnosticador (Monitorea y Analiza)

Es el encargado de realizar el diagnóstico de fallas en los servicios implicados en la composición. Para ello, realiza el análisis de la secuencia de eventos observables en la composición, con el fin de reconocer los patrones temporales que determinan las fallas. En este caso, el patrón temporal es definido por una Crónica.

El diagnosticador verifica el funcionamiento de un servicio web en tiempo de ejecución (conducta observada), basado en los requisitos especificados en la composición (conducta esperada). En particular, cuando se realizan las interacciones de los servicios en la composición, son interceptadas por el diagnosticador para buscar comportamientos contradictorios.

En específico, los dos momentos observados son: cuando se invoca al servicio (call) y cuando el servicio invocado genera su resultado (response). Así, el

intercambio de mensajes es visto como la ocurrencia de dos eventos secuenciales:

- La invocación de un servicio, con el conjunto de variables de entrada respectivas en el tiempo t_0: call(Service, var, t_0).
- La respuesta del servicio, con el conjunto de variables de salida respectivas en el tiempo t_1, que es el resultado de la invocación: response(Service, var, t_1).

Para encontrar anomalías en el intercambio de mensajes, se construyen crónicas que permiten analizar la consistencia entre los mensajes obtenidos (conducta observada) y los especificados en el documento WSDL (conducta esperada). Adicionalmente, las crónicas son enriquecidas con características no funcionales que se utilizan comúnmente en los servicios web, como la calidad de los servicios esperados (QoS), el contrato en los servicios (SLA), entre otros. Así, los eventos a medir en el diagnosticador deben garantizar el análisis para encontrar anomalías. Algunas de las anomalías analizadas son:

- **En la operación Call:**
  - La consistencia entre las variables definidas para la llamada en el WSDL y los que están en la invocación.
  - Los mensajes de error al realizar la invocación de los servicios.
  - Los valores de QoS y SLA que deberían garantizar el servicio.
- **En la operación Response:**

- La consistencia entre las variables definidas para la llamada en el WSDL y los que están en la operación.
- Los mensajes de error de los servicios.
- Los valores de QoS y SLA que deben brindar los servicios en sus respuestas.

3.3.2. El Reparador (Planifica y Ejecuta)

El reparador cuenta con mecanismos para la resolución de fallas presentes en la composición de los servicios. Cuando se diagnóstica una falla, el reparador se invoca para determinar y ejecutar los mecanismos pertinentes para resolver el problema.

Este componente tiene mecanismos que permiten, saltar (skip) y reasignar (reallocating) servicios web. Además, puede modificar la ejecución del flujo de la composición realizando sustituciones y/o añadiendo servicios, entre otras cosas. El reparador tiene 2 sub-componentes:

- **Planificador:** Define el conjunto de estrategias para resolver la falla. Recibe la falla identificada por el diagnosticador e infiere los mecanismos de reparación necesarios usando la ontología Fault-Recovery y la metadata que contiene los métodos de reparación disponibles, ambos forman parte de las fuentes de conocimiento de ARMISCOM.
- **Ejecutor:** Ejecuta los métodos de reparación para subsanar la composición, de acuerdo al plan definido previamente.

3.3.3. La fuente de Conocimiento

ARMISCOM define como fuente de conocimiento un marco ontológico, el cual permite la gestión de todo el conocimiento requerido por el middleware. El marco ontológico está compuesto por (ver Figura 3.3):

- **La plataforma SOA:** en particular, por:
  - ***Web Services Description Language (WSDL):*** Proporciona un modelo para describir cómo los servicios pueden ser llamados, que parámetros son esperados, y qué funcionalidades ofrecen.
  - ***Web Services Choreography Description Language (WS-CDL):*** define un modelo para describir la Coreografía de los Servicios Web. Se centra en la colaboración de igual a igual (peer-to-peer) entre servicios.
  - ***Ontology Web Language - Service (OWL-S):*** es una ontología que describe semánticamente a los servicios Web [59], a partir de la cual se automatizan las tareas de descubrimiento, invocación, composición, y seguimiento de los servicios web.
- **Base de Datos de Crónicas Distribuidas:** almacena al conjunto de crónicas que definen los patrones de fallas de los servicios web. Es utilizada principalmente por el Diagnosticador.
- **La Ontología Fault-Recovery:** define al conjunto de mecanismos de recuperación de fallas, estableciendo las relaciones entre las fallas y los métodos de reparación. Es utilizada por el Reparador.
- **La metadata "Service repair methods":** es usada para almacenar los

métodos de reparación disponibles para un servicio o flujo de la composición.

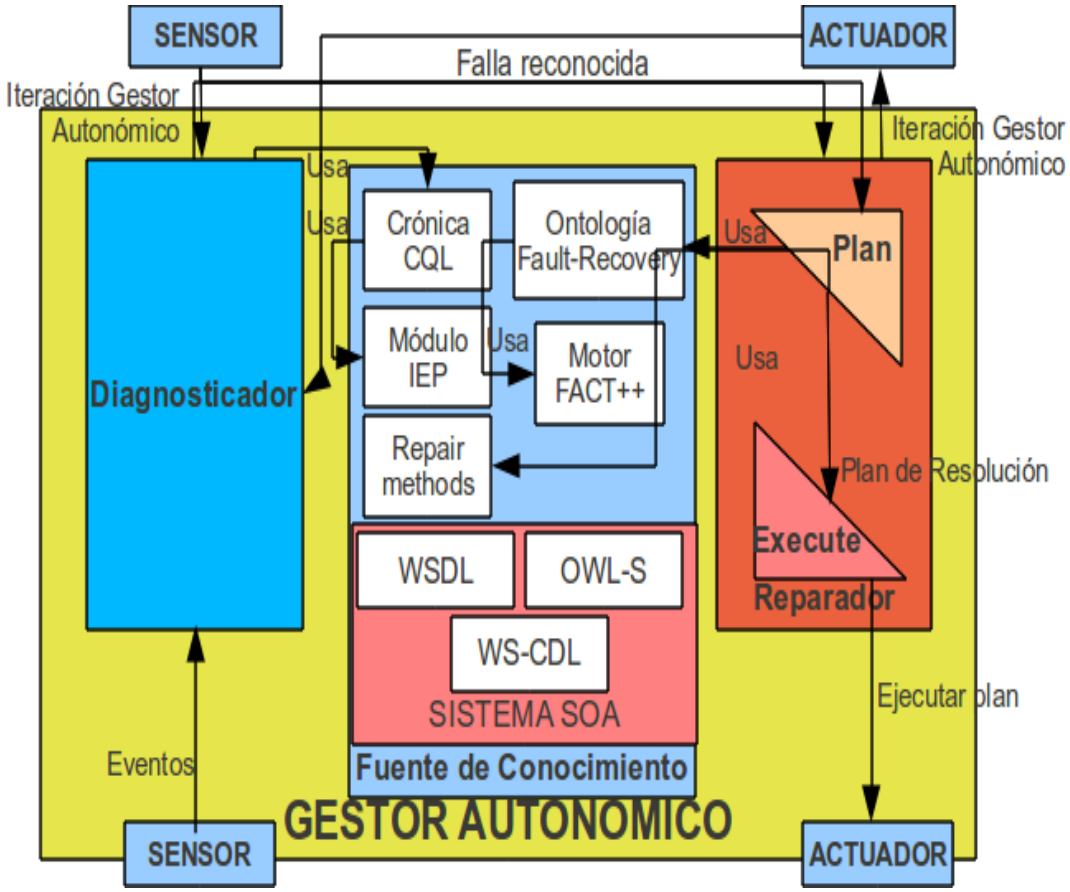


Figura 3.3: Marco ontológico de ARMISCOM

En esta sección describimos los dos últimos componentes del marco ontológico, las crónicas son descritas en el siguiente capítulo.

3.3.3.1 Ontología Fault-Recovery

La ontología Fault-Recovery permite la correlación de fallas en la composición de los servicios, con los métodos disponibles para su corrección en una aplicación SOA. La ontología es el elemento principal del Reparador, al permitirle analizar e

inferir los métodos de corrección de las fallas que han sido previamente diagnosticadas. El Reparador razona acerca de los posibles métodos de corrección de un fallo, utilizando el conocimiento que se describe en la ontología. La ontología sobre los métodos de reparación de cada tipo de falla en una aplicación SOA, se basa en la ontología propuesta en [6], donde ellos construyen una taxonomía de fallas y las causas que las generan. La estructura general de la ontología es mostrada en la Figura 3.4. Pasemos a describir los componentes de la ontología, sus conceptos y relaciones.

3.3.3.1.1 Conceptos Sobre Tipos de Fallas en la Composición de Servicios Web

La ontología contiene dos clases: llamadas Fault (fallas) y Repair Strategies (estrategias de reparación) (ver Figura 3.4), que representan los conceptos de los tipos de fallas y reparación descritos en el Capítulo 2. Por otro lado, la clase fallas es una superclase de las clases Physical, Development y Interaction, así como la clase reparación es una superclase de las clases Service y Flow. Esos son los conceptos de base que caracterizan a nuestra ontología para la recuperación de fallas, ya que permiten inferir para cada tipo de falla (los individuos de la clase fallas) la respectiva estrategia de reparación a usar (los individuos de la clase estrategias de reparación). Las instancias (individuos) en la ontología son mostradas en la Tabla 3.1. En el Capítulo 2 se detallan los posibles individuos (instancias) de cada una de esas clases.

3.3.3.1.2 Relaciones en la Ontología Fault-Recovery

La clase fault tiene una relación llamada *Has_repair_method* que permite asignar elementos de la clase *Repair Strategies* a cada tipo de falla. De esta manera se emparejan las fallas en la composición de los servicios con los

mecanismos para resolverlos. También, la clase Repair Strategies tiene una relación llamada Solve_fault, que realiza la operación inversa a Has_repair_method.

Los métodos de reparación para cada falla son un conjunto de acciones a ejecutar para resolver la falla y su ejecución debe hacerse de forma secuencial entre los métodos disponibles en el sitio (ver tercera columna tabla 3.1). Esto es, el reparador debe tratar de resolver la falla con la primera acción disponible (el mejor caso), y si con éste no le es posible resolver el problema continúa secuencialmente con la siguiente acción, hasta lograr la reparación de la falla o llegar a la última opción. Por ejemplo, para la falla de servicio no disponible (unavailable service), la primera acción es tratar de colocar el servicio de nuevo disponible (redo service, mejor caso), en caso de que no se puede realizar, la segunda acción es tratar de asignar el servicio en otro sitio, si no se puede resolver el problema es necesario intentar la sustitución del servicio por un equivalente, y así hasta que la reparación de la falla sea realizada o probar la última acción (en este caso, skipFlow service).

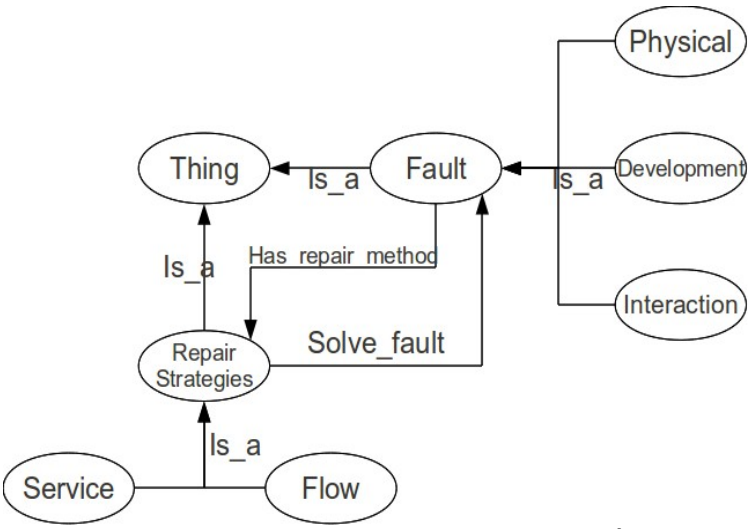


Figura 3.4: Estructura de la ontología Fault-Recovery

Subclase	Instancias de individuos	Has_repair_method
physical	unavailable	• redo • reassign • substitute • substituteflow • skipService • skipFlow
development	parameterIncompatibility	• CompleteMissingParameters • substitute • substituteflow • skipService • skipFlow
	interfaceMightHaveChanged	• CompleteMissingParameters • substitute • substituteflow • skipService • skipFlow
	DueToNonDeterministicActions	• parametersUpdate • substitute • substituteflow • skipService • skipFlow
	workflowinconsistency	• substituteflow • skipFlow
interaction	misunderstoodBehaviourFault	• parametersUpdate • substituteflow • skipFlow
	responseError	• substitute • substituteflow • skipService • skipFlow
	timeout	• reasign • retry • substitute • substituteflow • skipService • skipFlow
	misbehavingExecutionFlow	• redo • substituteflow • skipFlow
	IncorrectOrder	• substituteflow • skipFlow
	QualityOfService QoS	• reasign

		• substitute • substituteflow • parametersUpdate
	ServiceLevelAgreement SLA	• parametersUpdate • reasign • substitute • substituteflow

Tabla 3.1: Instancias de individuos en la ontología Fault-Recovery

3.3.3.2 Metadata Acerca de los Métodos de Reparación de Servicios

Como se mostró en la ontología propuesta anteriormente, una falla en la composición puede tener diferentes métodos de reparación. Aunque algunos mecanismos de resolución se pueden aplicar fácilmente sin necesidad de buscar servicios o flujos equivalentes (redo, parametersUpdate, skip service, entre otros), para todos no es el caso, por lo que se deben realizar en tiempo de ejecución.

Por ejemplo, para implementar el método “substitute a service” se debe previamente identificar los servicios equivalentes, utilizando como base de conocimientos el sistema SOA (UDDI, WSDL, OWL-S), porque la búsqueda de servicios equivalentes lleva algún tiempo (no puede ser implementado en tiempo real). Asimismo, algunos de los mecanismos de corrección no se pueden utilizar en algunos casos/sitios. Por esas razones, es necesario definir una base de conocimientos que permita a ARMISCOM seleccionar los mecanismos de reparación para cada caso/sitio, así como eventualmente personalizarlos.

En particular, en la mayoría de los trabajos dedicados a la sustitución de sub-flujos en la composición de servicios, se refieren a contar con arquitecturas que les permitan encontrar previamente sub-flujos alternos o sustitutos, para dar respuestas a fallas presentes en la composición en tiempo real. Así, los mecanismos consisten en modelar una aplicación SOA como un grafo o camino (path), los cuales pueden ser descompuestos en sub-grafos y conseguir flujos

equivalentes basados en algún criterio de similitud según las propiedades funcionales y no funcionales (p.e. QoS) buscadas [58, 59, 60, 61, 62].

Para este caso, es necesario definir un mecanismo que permita al middleware contar con un almacén de flujos alternativos de mecanismos de reparación, para poder dar una respuesta coherente y rápida a la reparación de una aplicación SOA. Los reparadores distribuidos de ARMISCOM (uno por cada servicio) continuamente están buscando servicios equivalentes para poder remplazar el servicio que se encuentra a su cargo, en el caso de un mal funcionamiento del mismo. Conseguir servicios equivalentes frecuentemente no resulta una tarea sencilla y en muchos casos es necesario modificar el flujo de ejecución de la aplicación SOA (agregar o quitar servicios). Ahora bien, cada reparador continuamente actualiza la metadata con los nuevos servicios y flujos equivalentes que encuentra, lo que le permite contar con un almacén para dar una respuesta rápida. En muchos casos, los reparadores al intentar buscar un servicio equivalente, necesitan modificar servicios que se encuentran a cargo de otros reparadores, e ingresarlos en sus almacenes, realizando de esta manera una actualización/búsqueda distribuida en cada reparador.

De esta manera, es necesario primero analizar el flujo de ejecución de las aplicaciones. Trabajos previos almacenan los sub-flujos modelándolos como un conjunto de servicios que están interconectados entre si, usando para ello redes Petri o grafos de conexiones [58, 59, 60, 61, 62]. Debido a que en ARMISCOM el componente encargado de realizar el análisis de fallas concibe a la composición como un flujo de eventos, es necesario expandir la representación de los sub-flujos como una secuencia de eventos. En este orden de ideas, una aplicación SOA puede ser vista como una secuencia de eventos E, que pueden ser descompuestos en regiones o sub-flujos R_i de eventos Eac_i, que corresponden a

cada instancia del gestor autonómico P_i:

$$Aplicación(E) = UNION_{i=1,\ n}(R_i(Eac_i)) \quad (3.1)$$

donde,

- Eac_i corresponden a un conjunto de eventos del sub-flujo, tal que $Eac_i = \{E_k,,..., E_l\}$ que ocurren en la región/sitio i.
- UNION es un predicado que define la unión del conjunto de eventos (Eac_i) distribuidos en las n regiones.

Para mostrar la descomposición de la aplicación en regiones, suponga la aplicación SOA que se muestra en la Figura 3.5:

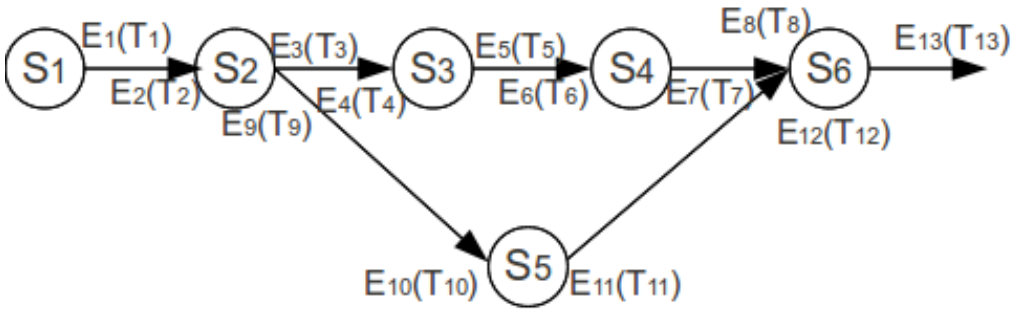


Figura 3.5: Aplicación SOA descompuesta en Regiones de Eventos

La aplicación seria la unión de todos los eventos que suceden en ella:

$$Aplicación = UNION\{E_1,\ E_2,\ E_3,\ E_4,\ E_5,\ E_6,\ E_7,\ E_8,\ E_9,\ E_{10},\ E_{11},\ E_{12},\ E_{13}\} \quad (3.2)$$

Se puede descomponer en regiones los eventos asociados a los servicios en la aplicación, tal que:

$$\text{Aplicación} = \text{UNION}\{R_1(E_1),\ R_2(E_2,\ E_3,\ E_9),\ R_3(E_4,\ E_5),\ R_4(E_6,\ E_7),\ R_5(E_{10},\ E_{11}),\ R_6(E_8,\ E_{12},\ E_{13})\} \mid \forall k,m < 13 \text{ y } \forall i,j < 6,\ i \neq j,\ E_k \in R_i \text{ y } E_m \in R_j, \text{ entonces,}$$

$$R_i(E_k) \cap R_j(E_m) = \varnothing \tag{3.3}$$

En base a las regiones R_1, R_2, R_3, R_4, R_5, R_6, es posible encontrar regiones equivalentes R'_1, R'_2, R'_3, R'_4, R'_5, R'_6. Adicionalmente, es posible encontrar súper-regiones[14], tal que:

$$\text{Aplicación} = \text{UNION}\ \{Rs_1(E_1,\ E_2,\ E_3,\ E_9),\ Rs_2(E_4,\ E_5,\ E_6,\ E_7,\ E_{10},\ E_{11}),\ Rs_3(E_8,\ E_{12},\ E_{13})\} \tag{3.4}$$

En general, una aplicación puede ser expresada como una región:

Definición 3.1. Aplicación expresada en una Región: Una aplicación Apl puede ser expresada como una región R_{Apli} compuesta por la unión de los n eventos que forman parte de ella.

$$\text{Aplicación} = R_{Apli}(E) = \text{UNION}_{i=1,n}\ E_i \mid \forall i=1,\ n\ E_i \in R_{apli} \tag{3.5}$$

Definición 3.2. Una Región puede ser descompuesta en sub-regiones: Una aplicación R_{Apl} compuesta de n eventos puede ser descompuesta en m Regiones, tal que la UNION de los conjuntos de eventos de las regiones sea igual a la región de la aplicación y que la intersección de las regiones sea igual a vacío (∅).

$$R_{Apli} = \text{UNION}_{i=1,n}\ \{R_i(E_i)\} \mid \forall k,m,i,j, \text{ donde } R_i \in R_{Apli},\ R_j \in R_{Apli},\ E_k \in R_i,\ E_m \in R_j, \text{ entonces,}$$

$$R_i(E_k) \cap R_j(E_m) = \varnothing \tag{3.6}$$

14 Una super-región es una región que es compuesta por dos o más regiones de la aplicación

Debido a que las fallas acontecen en un conjunto de eventos específicos delimitados por un evento de inicio E_0 y un evento final E_F, es necesario definir una estructura para representar esas regiones y realizar procesos de inferencias sobre ellas:

Definición 3.3. Región de una aplicación SOA: corresponde a un conjunto o sub-flujo de eventos, entre los cuales están un evento de inicio (E_0), de fin (E_F), y sus eventos de transiciones (E_{trans}), es decir, Región R = {E_0, E_F, E_{trans}}.

Explicación: suponga que una aplicación SOA esta compuesta de n eventos (E_1, E_2, ...E_n), entonces una región para la aplicación SOA se puede escribir como:

$$SOA = R_{SOA} = \{E_0, E_F, E_{trans}\} = \{E_1, E_n, \{E_2, E_3, ..., E_{n-1}\}\} \quad (3.7)$$

En base a esta definición, se pueden escribir las regiones definidas en (3.3) y en (3.4) como (3.8) y (3.9).

$$Aplicación = UNION\{R_1(E_1, E_1, \{\emptyset\}), R_2(E_2, E_9, \{E_3\}), R_3(E_4, E_5, \{\emptyset\}), R_4(E_6, E_7, \{\emptyset\}), R_5(E_{10}, E_{11}, \{\emptyset\}), R_6(E_8, E_{13}, \{E_{12}\})\} \quad (3.8)$$

$$Aplicación = UNION\{Rs_1(E_1, E_9, \{E_2, E_3\}), Rs_2(E_4, E_{11}, \{E_5, E_6, E_7, E_{10}\}), Rs_3(E_8, E_{13}, \{E_{12}\})\} \quad (3.9)$$

Por otro lado, el objetivo de la reparación es encontrar flujos sustitutos de mecanismos de reparación que permitan encontrar una aplicación SOA equivalente. Para realizar esa tarea, se requieren las siguientes definiciones:

Lema 3.1. Región Equivalente: una región R' se considera equivalente a la región R si y solo si R' tiene eventos E_0' y E_F' equivalentes a E_0 y E_F de R.

$$R(E_0, E_F, E_{trans}) \text{ equivalente } R'(E_0', E_F', E_{trans}') \mid E_0 \ \& \ E_F \quad (3.10)$$

equivalentes E_0'& E_F'

Demostración: Considere que la región R esta compuesta por n eventos R = {E_1, E_n, {E_2, E_3,.., E_{n-1}}, entonces si es posible encontrar una región R' con n' eventos tal que: $E_1 \approx E_1$' y $E_n \approx E_n$', la Región R' es equivalente a R.

$$R(E_1, E_n, \{E_2, E_3, \ldots, E_{n-1}\} \text{ equivalente } R'(E_1', E_n', \{E_2', E_3', \ldots, E_{n-1}'\}) \mid E_1 \& E_n \text{ equivalentes } E_1' \& E_n' \quad (3.11)$$

Así, para el ejemplo 3.4, es posible encontrarle regiones equivalencias a las sub-regiones R's_1, R's_2 y R's_3:

$$\text{Aplicación} = \text{UNION}\{R's_1(E_1', E_9', \{E_{transR1}'\}), R's_2(E_4', E_1', \{E_{transR2}'\}), R's_3(E_8', E_{13}', \{E_{transR3}'\})\} \quad (3.12)$$

Donde $E_{transR1}$', $E_{transR2}$' y $E_{transR3}$' pueden ser cualquier flujo de eventos.

Lema 3.2. Super-región: una región R' se considera super-región de la región R si y sólo si los eventos de la región R están contenidos en los eventos de la región R'.

Demostración: Suponga que la región R' está compuesta por n eventos R' = {E_1, E_n, {E_2,..., E_{n-1}}} y la región R está compuesta por los eventos i - k de R = {E_k, E_i, {E_{k+1},..., E_{i-1}}}, tal que $k \geq 0$, $n \geq i$, e $i \geq k$, entonces la región R' es super-región de R si es posible dividir R' en al menos 2 o más sub-regiones R_{si} (i > 1), donde una de las una sub-regiones R_{Si} es igual a R:

$$R' = \{E_1, E_n, \{E_2, \ldots, E_{n-1}\}\} = \text{UNION}\{R_{S1}(E_1, E_{k-2}, \{E_2, E_{k-1}\}), R_{S2}(E_k, E_i, \{E_{k+1}, E_{i-1}\}), R_{S3}(E_{i+1}, E_n, \{E_{i+2}, E_{n-1}\})\} \mid i < n \ \& \ k < i \text{ entonces } R_{S2}=R \quad (3.13)$$

$$R_{S2}(E_k, E_i, \{E_{k+1}, E_{i-1}\}) = R(E_k', E_i', \{E_{k+1}', E_{i-1}'\}) \mid E_k =E_k', E_i = E_i \ \& \ E_{S2trans} = E_{trans}' \quad (3.14)$$

Donde $E_{S2trans} = \{E_{k+1}, E_{i-1}\}$ y $E_{trans}'=\{E_{k+1}', E_{i-1}'\}$.

Lema 3.3: Si la región R' es super-región de R, entonces es posible encontrar una región S que es sub-región de R' y es igual a R.

Demostración: Debido a que la región R' es super-región de R, los eventos de R están contenidos en los eventos de R' (ver Lema 3.2). Entonces, se debe dividir R' en sub-regiones, tal que una S es igual a R:

$$R'(E_0', E_F', E_{trans}') = R_{S1}(E_{S1}) \cup R_{S2}(E_{S2}) \cup \ldots \cup R_{Sn}(E_{Sn}) \quad (3.15)$$

$$R_{2Si}(E_{0Si}, E_{FSi}, E_{transSi}) = R(E_0, E_F, E_{trans}) \mid E_{0Si} = E_0 \ \& \ E_{FSi} = E_F \ \& \ E_{transSi} = E_{trans} \quad (3.16)$$

Note que dos regiones iguales entre si, son por consiguiente equivalentes.

En este orden de ideas, un primer paso a realizar por parte del reparador del servicio S_i para reemplazar la región $R_i(E_{i0}, E_{iF}, E_{itrans})$, que es sub-región de $R_{SOA}(E_{SOA0}, E_{SOAF}, E_{SOAtrans})$, es conseguir una región equivalente $R_i'(E_0, E_F, E_{trans'})$, tal que contenga los mismos estados iniciales, finales, y sustituir R_i por R_i'. En caso contrario, el siguiente paso es conseguir un super-región $R''(E''_0, E''_F, E''_{trans})$ de R_i, y entonces conseguir la sub-región de R'' (R_{Si}) igual a R_i, para sustituir a R_i en R_{SOA}.

Así, para gestionar las regiones equivalentes de la aplicación SOA dentro de ARMISCOM, es necesario definir una metadata para almacenar los mecanismos de reparación en cada caso. La reparación de un flujo de la composición consiste en encontrar una región equivalente que permita mapear los eventos iniciales E_0 y finales E_F, esto es, se deben conocer las regiones equivalentes almacenadas

relacionadas a cada mecanismo de reparación. Una forma es añadirle a cada región equivalente información acerca del mecanismo de reparación donde pudieran ser usadas (RepairMethod), para facilitarle al reparador la tareas de búsqueda de las regiones sustitutas que más se aproximan a la parte de la aplicación donde se encuentra la falla. Es decir, se añade a las regiones equivalentes almacenadas en la metadata, información acerca del mecanismo de reparación (nombre del método) donde podría ser usada, y el orden de importancia de esa región equivalente según algún criterio de optimización como QoS, SLA, proveedor de los servicios, entre otros, que permita discriminarla/priorizarla con respecto a las otras regiones equivalentes disponibles.

De esta manera, en ARMISCOM se define una metadata para almacenar las regiones de eventos equivalentes de la aplicación SOA, que se almacenan en cada reparador, con los métodos de reparación en donde se pueden utilizar:

reparation methods available						
Id	**Weight**	**RepairMethod**	**Transition**	**Event_init**	**Event_end**	**Operations**

Figura 3.6: Estructura de la Metadata

donde:

- **Id:** Identificador único del script de reparación, similar a los utilizados en bases de datos, que permite su indexación para volúmenes grandes de datos, facilitando su búsqueda.
- **Weight:** Representa el orden de importancia de esa región equivalente, permitiendo discriminar entre regiones equivalentes semejantes. El criterio para asignar este valor puede consistir en calcular la cantidad de eventos involucrados en el flujo, o cuales son los valores de QoS, o la similitud de las propiedades funcionales ofrecidas, entre otros. Dos regiones pueden tener

el mismo orden de importancia.

- **RepairMethod:** Representa los métodos de reparación que lo pueden usar.
- **Transition:** Define la secuencia de eventos internos (E_{trans}) de la región. Un reparador puede hacer uso de los eventos que se encuentran en la transición, para encontrar otras regiones equivalentes internamente.
- **Event_init:** Representa el evento inicial de la región (E_0) en el que se debe comenzar la reparación.
- **Event_end:** Representa el evento final de la región (E_f) en el que se debe terminar la reparación.
- **Operations:** Contiene el conjunto de acciones a implementar para agregar la nueva composición de la aplicación SOA (composición corregida).

En general, el conjunto de pasos que debe realizar el reparador S_i para reemplazar la región R_i se muestra en el Algoritmo 3.1.

```
1. IF (R' = searchRegion(Ri(E0, EF, Etrans)), RepairMethod) THEN

2.  repair.replace(Ri, R')

3. ELSE IF (RS = searchSuperRegion(Ri(E0, EF, Etrans)), RepairMethod) THEN

4  repair.replace(Ri, RS)
```

Algoritmo 3.1: Reparador Local S_i

Donde:

searchRegion($R_i(E_0, E_F, E_{trans})$), RepairMethod): Corresponde a la sentencia SQL: "SELECT * FROM methods WHERE RepairMethod='RepairMethod' AND Event_init = E_0 AND Event_end = E_F AND transition = E_{trans} ORDER BY Weight ASC". La consulta SearchRegion busca mecanismos de reparación (RepairMethod es el

nombre del mecanismo a buscar) con regiones con los mismos eventos E_0, y E_F, ordenadas según weight de menor a mayor (ASC, las regiones con menor weight son extraídas antes).

searchSuperRegion($R_i(E_0, E_F, E_{trans})$), RepairMethod): Corresponde a la setencia SQL: "SELECT * FROM methods WHERE RepairMethod='RepairMethod' AND CONCAT[15](Event_init, Event_end, transition) CONTAINS[16](E_0, E_F) ORDER BY Weight ASC". La consulta busca en la Super-regiones R' de R los eventos E_0', E_F' (sub-región R_s), y extrae la sub-región R_s de la región R' igual a R para sustituirlo.

La Tabla 3.2 muestra un ejemplo de implementación de una metadata para una aplicación SOA:

15 la función CONCAT permite unir cadenas de caracteres, en este caso los eventos son almacenados como cadenas y unidos para facilitar la consulta.

16 La función CONTAINS permite buscar coincidencias aproximadas de una cadena de caracteres.

reparation methods available						
Id	**Weight**	**RepairMethod**	**Transition**	**Event_init**	**Event_end**	**Operations**
1	1	substituteflow	{E6, E8}	E5	E9	remFlow({E5, E6, E7, E8, E9}) addFlow({E5,E6, E8,E9})
2	1	parametersUpdate	{Ø}	E5	E5	setting(E5)
3	1	CompleteMissingParameter	{Ø}	E5	E6	remFlow({E5, E6}) addFlow({E5,Compensate,E6)
4	2	substituteflow	{E6, E7, E8}	E5	E9	remFlow({E5, E6, E7, E8, E9}) addFlow({E5,E6, E7,E8,E9})
5	1	Skip Service	{E8}	E7	E10	remFlow({E5, E6, E7, E8, E9}) addFlow({E7,E8, E10)
6	1	substituteflow	{E6, E7, E8}	E5	E9	remFlow({E5, E6, E7, E8, E9}) addFlow({E5,E6, E7,E8,E9})
7	1	substituteflow	{E6, E7, E8}	E5	E9	remFlow({E5, E6, E7, E8, E9}) addFlow({E5,E6, E7,E8,E9})

Tabla 3.2: Ejemplo de implementación de la metadata de las regiones vinculadas, con los métodos disponibles para la reparación de una aplicación SOA

El id de la metadata sirve para diferenciar entre las diferentes regiones. En base a la Tabla 3.2, las regiones con id 1 ($R_{id=1}$), 4 ($R_{id=4}$), 6 ($R_{id=6}$) y 7 ($R_{id=7}$) pueden ser usadas por el mecanismo de reparación de sustituir parte del flujo de la

aplicación SOA (RepairMethod = 'substituteflow') y afectan a las mismas regiones de eventos $E_0 = E_5$ y $E_F = E_9$, pero serán extraídas en el orden $R_{id=1}$, $R_{id=6}$, $R_{id=7}$ y $R_{id=4}$ según la función SearchRegion ($R_{id=1,}$ $R_{id=6}$ y $R_{id=7}$ tienen igual weight, pero $R_{id=1}$ se agrega antes por tener el id menor). Igualmente, las regiones $R_{id=2}$, $R_{id=3}$ y $R_{id=5}$ pueden ser usados por los mecanismos de reparación para actualizar parámetros (parametersUpdate en el evento E_5), completar parámetros (CompleteMissingParameter, en este caso entre los eventos E_5 y E_6), y saltar servicios (Skip Service, salta el servicio E_9), respectivamente.

Para mostrar el funcionamiento del Algoritmo 3.1 y de la metadata de la tabla 3.2, suponga que se desea realizar la reparación de sustituir parte del flujo de ejecución (substituteflow) de una región que tiene los eventos $E_0 = E_5$ y $E_F = E_9$. La región seleccionada por el Algoritmo 3.1 en el paso 1 es $R_{Id = 1}$, ya que coincide con los eventos E_0, E_F, cuenta con menor weight que $R_{Id = 4}$, y ha sido agregada antes del resto de regiones. En otro ejemplo, suponga que se necesita sustituir el flujo R con $E_0 = E_5$, $E_F = E_7$ y $E_{trans} = \{E_6\}$, el Algoritmo 3.1 no consigue una región equivalente en el paso 1, entonces procede a buscar una super-región en el paso 3, retornando la región $R_{Id = 6}$, ya que los eventos E_5, E_7 y E_6, están contenidos en $\{E_5, E_9, \{E_6, E_7, E_8\}\}$. Entonces, según el lema 3.1, es posible extraer un región $R' = \{E_5, E_7, \{E_6\}\}$ de $R_{id=6}$, que es igual a R.

3.4. Conclusiones

En este Capítulo se ha propuesto una arquitectura de middleware reflexivo para la gestión de aplicaciones orientadas a servicios. El middleware está diseñado para ser distribuido a través de todos los servicios de la aplicación SOA, contando para ello con el nivel de base que contiene tanto el sistema SOA como la Aplicación SOA, y el nivel meta con componentes para ejecutar la reflexión. La arquitectura que se utiliza está basada en el modelo de computación autonómica,

que le permite una fácil adaptación para procesos de auto-reparación.

Para apoyar esta arquitectura, se ha diseñado el componente de gestión de conocimiento del middleware. Este conocimiento se compone de la información del sistema SOA, de crónicas distribuidas, que describen el patrón de comportamiento de las fallas de una aplicación SOA, de la ontología Fault-Recovery, que describe los tipos de falla y de mecanismo de reparación, y la metadata que describe los métodos de reparación.

En el caso de la ontología Fault-Recovery, permite correlacionar las fallas presentes en la composición con los mecanismos de reparación, describiendo una taxonomía sobre las fallas y los mecanismos de reparación de fallas para una aplicación SOA. Esta ontología puede ser enriquecida en el futuro para permitir inferencias acerca de situaciones más complejas.

Adicionalmente, en este Capítulo se ha propuesto una metadata para almacenar los métodos de reparación disponibles para la aplicación SOA, que serán utilizados por el componente reparador de cada instancia del gestor autonómico. Para realizar la estructura de esta metadata, se definió el concepto de regiones de eventos para descomponer la aplicación SOA, y usar los conceptos que en este Capítulo hemos definido como regiones equivalentes y super-regiones. Usando esta metadata, para los métodos de reparación que lo requieran, el reparador deduce el flujo equivalente a usar, en base a los eventos de inicio E_0 y fin E_F del flujo que se desea modificar. De esta manera, los reparadores tienen mecanismos de recuperación múltiples para una misma o diferentes fallas.

La Tabla 3.3 muestra una evaluación cualitativa de ARMISCOM con respecto a otros trabajos. Es la única que realiza todas sus tareas de forma distribuida, a

nivel de cada servicio. Las demás realizan algunas de sus fases de forma centralizada o semi-centralizada, lo que dificulta su escalabilidad.

Arquitectura	**Fase de Monitoreo**	**Fase de Diagnóstico**	**Fase de Reparación**
SOAR [21]	Centralizada	Centralizada	Centralizada
Arquitectura de Auto-sanación [22]	Centralizada	Centralizada	Centralizada
Auto-sanación en Servicios Web dinámicos [69]	Centralizada	Centralizada	Centralizada
Arquitectura basada en Crónicas [11, 71]	Distribuida	Semi-Centralizada[17]	Centralizada
Servicios Web con capacidades de Diagnostico [24]	Distribuida	Semi-Centralizada	Centralizada
ARMISCOM	Distribuida	Distribuida	Distribuida

Tabla 3.3: Comparando ARMISCOM con otros trabajos

17 Distribuida pero coordinada por un diagnosticador central.

Capítulo 4
Crónicas Distribuidas para el Diagnóstico de Fallas en Sistemas Distribuidos

En este Capítulo se propone una extensión distribuida al paradigma de crónicas, y su uso en un sistema de diagnóstico de fallas basado en crónicas distribuidas.

4.1. Definición de Crónicas Distribuidas

Antes de definir la extensión de crónicas distribuidas, se parte del hecho de un conjunto de eventos $E=\{E_1, E_2, ..., E_p\}$ distribuidos entre los diferentes n procesos del sistema. En general, dichos eventos son agrupados en las n sub-crónicas distribuidas en los n procesos del sistema. El reconocimiento de las n sub-crónicas resulta en el reconocimiento de la crónica completa. Partiendo de esas premisas, se introducen las siguientes definiciones:

Lema 4.1: Una crónica C puede ser descompuesta en n-sub-crónicas, tal que cada sub-crónica de SC_i es asignada a un sitio/proceso P_i del sistema en estudio, y describe un sub-conjunto de eventos Eac_i, con restricciones temporales Tac_i, que deberían ocurrir en el sitio i con el fin de que se produzca el reconocimiento de la sub-crónica.

$$C(E,T) = UNION_{i=1, n}(SC_i(Eac_i, Tac_i)) \quad (4.1)$$

donde,

- Eac_i y Tac_i corresponden a un conjunto de eventos y restricciones temporales

de la sub-crónica asignada al sitio/componente i, tal que $Eac_i = \{E_k,,..., E_l\}$, $Tac_i = \{T_k,,..., T_l\}$, $E_{k, ..., } E_l \epsilon E$, T_k , ..., $T_l \epsilon T$, y los eventos E_k, ..., E_l ocurren en el sitio i.

- UNION es un predicado que define la unión del conjunto de eventos (Eac_i) y del conjunto de restricciones temporales (Tac_i) distribuidos en las n sub-crónicas.

Demostración: Para demostrar esta afirmación, suponga:

$$C(E,T) = UNION_{j=1,p}(E_j,T_j) \quad (4.2)$$

$$C(E,T) = \{(E_1, T_1), (E_2,T_2) ....(E_p, T_p))\} \quad (4.3)$$

Ahora, suponga que se distribuyen estos p eventos entre los n componentes (Eac_i), tal que $p \geq n$. En el caso de los componentes que no le sean asignados eventos, éstos tendrán sub-crónicas vacías que no influyen en el reconocimiento total de la crónica:

$$C(E,T) = \{(Eac_1, Tac_1), (Eac_2, Tac_2), \ldots, (Eac_n, Tac_n)\} \quad (4.4)$$

Entonces, se puede decir que esos p eventos agrupados (Eac_i) conforman las sub-crónicas, tal que :

$$C(E, T) = \{SC_1(Eac_1, Tac_1), SC_2(Eac_2, Taci_2), \ldots, SC_n(Eac_n, Tac_n)\} = UNION_{i=1, n}(SC_i(Eac_i, Tac_i)) \quad (4.5)$$

Lema 4.2: Debido a que una crónica C se puede descomponer en n sub-crónicas (SC), el reconocimiento de la crónica global puede llevarse a cabo en la sub-crónica SC_i, reconociendo sus eventos (Eac_i, Tac_i), con la unión del reconocimiento parcial de las demás sub-crónicas ($SC_j \ \forall \ j=1,n \mid j \neq i$) de los otros

eventos (en este caso, las otras sub-crónicas deberán remitirle un mensaje para informar el reconocimiento de sus eventos). De este modo, el reconocimiento de la sub-crónica SC_i implica el reconocimiento de la crónica global C(E, T):

$$C(E, T) = \{(Eac_i, Tac_i), UNION_{j=1, n \mid j \neq i}(SC_j(Eac_j, Tac_j))\} \quad (4.6)$$

Demostración: Para demostrar la ecuación (4.6), una crónica se puede descomponer en n SC (ver ecuación 4.1):

$$C(E, T) = \{SC_1(Eac_1, Tac_1), SC_2(Eac_2, Tac_2), \ldots, SC_n(Eac_n, Tac_n)\} \quad (4.7)$$

debido a que el reconocimiento de la crónica C se lleva a cabo en el sitio i, es equivalente a:

$$C(E, T) = \{SC_i(Eac_i, Tac_i), UNION_{j=1, n \mid j \neq i}(SC_j(Eac_j, Tac_j))\} \quad (4.8)$$

En particular, debido a que la crónica se descompone en n sub-crónicas, es necesario extender la representación de las sub-crónicas para poder correlacionar los eventos reconocidos entre las diferentes sub-crónicas, y ser capaz de reconocer la crónica global. Para ello, se extiende el formalismo de crónicas mediante la adición de varios aspectos que permiten gestionar el proceso de sincronización entre las sub-cronicas (ver Figura 4.1):

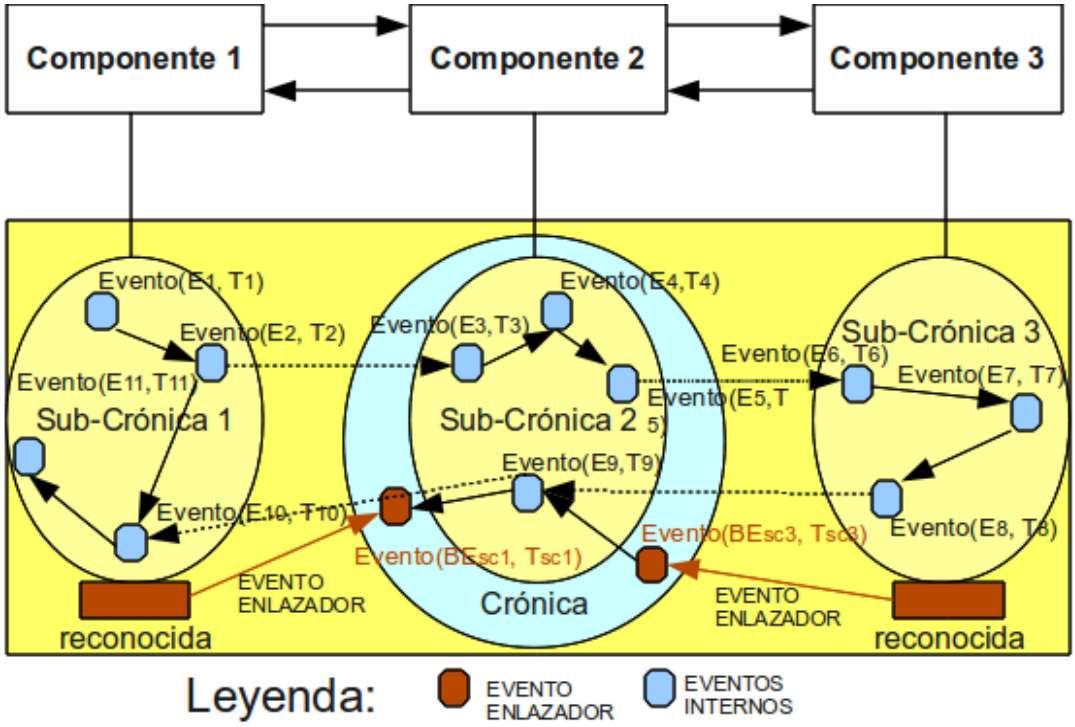


Figura 4.1: Ejemplo de una crónica descompuesta en sub-crónicas

Definición 4.1. Las variables de Estado son variables de un evento que determinan si es normal (¬Err) o anormal (Err). Por lo general, se puede indicar esto usando un valor lógico (VERDADERO y FALSO), pero el estado anormal puede ampliarse para enriquecer la clasificación de este comportamiento.

Definición 4.2 Eventos Enlazadores (Binding Events - BE) son eventos que generan las sub-crónicas para otras sub-crónicas, representando la comunicación entre las sub-crónicas. Un BE_j se instancia cuando una sub-crónica vecina se reconoce, y ese reconocimiento se propaga a las otras sub-crónicas. Por lo tanto, el reconocimiento (evento de salida) de una sub-crónica SC_i puede estar relacionado con el evento BE_k que pertenece a una subcronica Sc_j vecina.

Finalmente, se define una Crónica Distribuida:

Definición 4.3: Una Crónica Distribuida es una crónica C descompuesta en un conjunto de sub-crónicas SC_i, que están unidas entre sí a través de Eventos Enlazadores (BE).

Explicación: Suponga la crónica de la Figura 4.1, la cual está descompuesta en 3 sub-crónicas:

$$SC_1 = \{(E_1, T_1), (E_2, T_2), (E_{10}, T_{10}), (E_{11}, T_{11})\} \quad (4.9)$$

$$SC_2 = \{(E_3, T_3), (E_4, T_4), (BE_{sc3}, T_{sc3}), (E_5, T_5), (BE_{sc1}, T_{sc1}), (E_9, T_9)\} \quad (4.10)$$

$$SC_3 = \{(E_6, T_6), (E_7, T_7), (E_8, T_8)\} \quad (4.11)$$

SC_1 y SC_3 son sub-crónicas con sólo eventos internos. Cuando SC_1 y SC_3 se reconocen entonces emiten los Eventos Enlazadores (BE_{sc3} con tiempo T_{sc3}, y BE_{sc1} con tiempo T_{sc1}) a SC_2. Finalmente, SC_2 puede reconocer la crónica global:

$$SC_2 = \{(E_3, T_3), (E_4, T_4), (BE_{sc3}, T_{sc3}), (E_5, T_5), (BE_{sc1}, T_{sc1}), (E_9, T_9)\} \quad (4.12)$$

$$SC_2 = \{(E_3, T_3), (E_4, T_4), \{(E_6, T_6), (E_7, T_7), (E_8, T_8)\}, (E_5, T_5), \{(E_1, T_1), (E_2, T_2), (E_{10}, T_{10}), (E_{11}, T_{11})\}, (E_9, T_9)\} \quad (4.13)$$

$$SC_2(Eac_2, Tac_2), UNION_{j=1,\ 3\ |\ j \neq 2}(SC_j(Eac_j, Tac_j)) = C(E,T) \quad (4.14)$$

Luego de haberse definido los conceptos alrededor de una Crónica Distribuida, se procede a definir el reconocimiento distribuido de las crónicas.

4.2. Reconocimiento de Crónicas Distribuidas

El proceso de reconocimiento de las crónicas debe ser distribuido entre todos los componentes del sistema. Para ello, se coloca un sistema de reconocimiento por cada componente del sistema, que será el encargado de reconocer la sub-crónica en ese lugar. Como se mencionó anteriormente, cada sub-crónica está vinculada a otras sub-crónicas a través de los Eventos Enlazadores (BE), lo que permite a los sistemas de reconocimiento local inferir información de sus componentes vecinos. Utilizando esa información, el sistema de reconocimiento local puede reconocer sub-crónicas locales, generar eventos a sitios vecinos, y en general, propagar lo inferido. La inclusión de los eventos enlazadores permite que cada sitio tenga una visión parcial-global de todo el sistema, a partir de la información de sus vecinos.

La arquitectura de cada sitio se muestra en la Figura 4.2, y consta de un módulo de reconocimiento de crónica local (llamado CRS), que recibe eventos desde el monitor local, y los eventos generados por otros sitios vecinos (BE). Por ejemplo, la Figura 4.2 muestra como el sitio 2 en un tiempo dado, recibe eventos del monitor 2 y los generados por las sub-crónicas en los sitios 3 (BE_3) y 1 (BE_1).

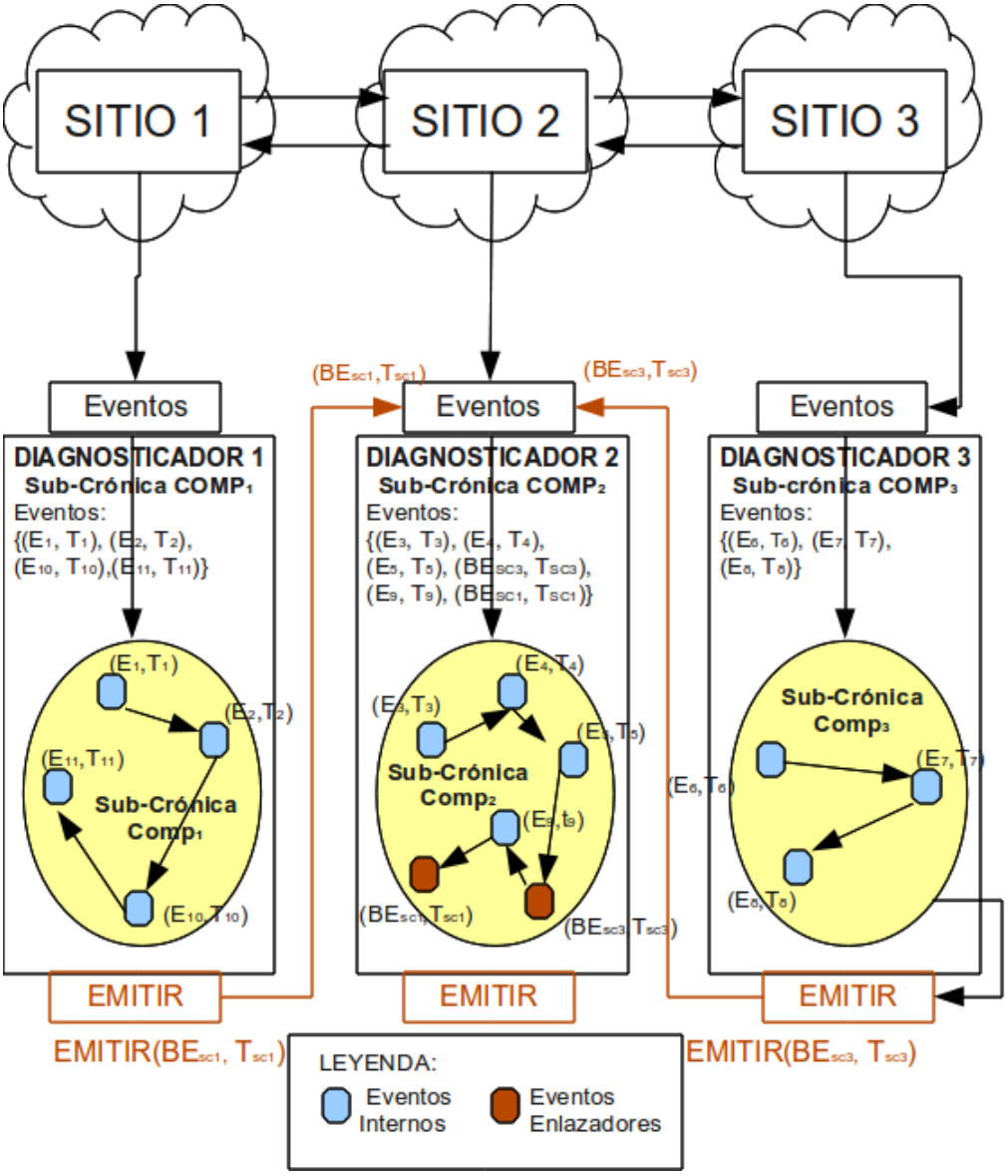


Figura 4.2: CRS distribuidos

Entonces, el modelo de la Crónica de la Figura 4.2 puede escribirse como se muestra en la Figura 4.3.

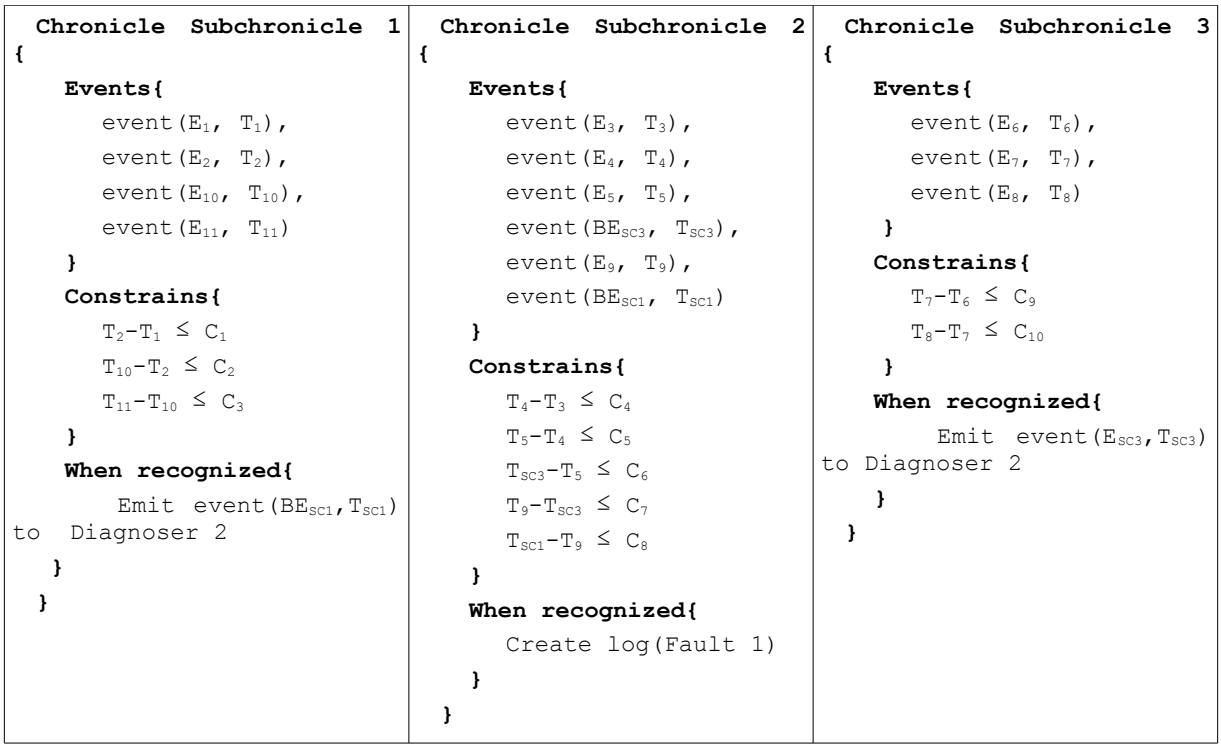


Figura 4.3: Modelo de Crónica

4.2.1. Algoritmo de Reconocimiento de una Crónica Local

El CRS local es responsable de realizar el diagnóstico; el algoritmo para hacer esto se muestra a continuación:

```
1. FOREACH event received DO

2.  diagnoser.addEvents(chronicle c, event)

3.  IF chronicle c is recognized THEN

4    DO c.executeAction()
```

Algoritmo 4.1: CRS local

Para ilustrar el funcionamiento del algoritmo, se va a describir cada paso utilizando el modelo de crónica de la Figura 4.3:

1. Los eventos son la piedra angular de las crónicas, representando un cambio que es observable en el sistema, y están representados por el nombre de la actividad que describe el evento y el tiempo de ocurrencia del evento. Por lo tanto, un diagnosticador obtiene un flujo de eventos que provienen de su entorno local o lo generan otros diagnosticadores, para hacer el reconocimiento de una crónica local. Para ilustrar este punto suponga que el diagnosticador 3 recibe la siguiente secuencia de eventos:

$$\{event(E_6,T_6=2),\ event(E_6,\ T_6=3),\ event(E_7,\ T_7=6),\ event(E_7,\ T_7=7),\ event(E_8,\ E_8=8)\}$$

2. En la medida que se van recibiendo los eventos, el reconocedor local realiza un proceso de agregación de eventos, el cual se detalla en el Algoritmo 4.2:

```
diagnoser.addEvents (chronicle c, event)

1 FOR EACH chronicle c where event is the first event DO

1.1    instances = c.instances();

2    FOR EACH current instances DO

2.1  IF event match instances and temporal constraints are not violated THEN

2.1.1    instances.addEvent(event)

2.2   IF event match instances and temporal constraints are violated THEN

2.2.1   instances.discard()

2.3   IF temporal constraints are violated THEN

2.3.1   instances.discard()
```

Algoritmo 4.2: Procedimiento diagnoser.addEvents

En el momento de la llegada de un evento, se crean tantas instancias de

crónicas, como las distintas crónicas que existan en la base de crónicas que se inicien con ese evento (este evento es el primero de esas crónicas, ver paso 1.1 del Algoritmo 4.2). Además, es necesario comprobar todas las instancias de crónicas creadas (paso 2 del Algoritmo 4.2), con el fin de determinar si algunas de ellas pueden seguir siendo instanciadas (esperan por este evento, ver paso 2.1 del Algoritmo 4.2), o si en algunas de ellas se violan las restricciones temporales (ver pasos 2.2 y 2.3). En estos casos, las instancias son eliminadas del reconocimiento. Por supuesto, una instancia no se afecta si el nuevo evento no es parte del modelo de la crónica.

El comportamiento del algoritmo se muestra en la Figura 4.4 para el CRS del diagnosticador 3. Inicialmente, no hay instancias y llega el evento E_6 al diagnosticador en el tiempo $T_6 = 2$; una instancia I_{31} del modelo de crónica es creada esperando el evento E_7 en el intervalo [3, 5]. Entonces, llega otro evento E_6 en el momento $T_6 = 3$, y una nueva instancia I_{32} es creada esperando el evento E_7 en el rango [4, 6]. En el instante t = 6 la instancia I_{31} tiene una violación de restricción temporal porque el evento E_7 no ha sucedido, y entonces la instancia I_{31} es descartada (si llegase el evento E_7 en, por ejemplo, t = 4, la instancia I_{31} seguiría siendo reconocida). En el momento t = 6 ocurre el evento $E_{7,}$ y la instancia de la Crónica I_{32} es modificada para seguir siendo reconocida, esperando ahora el evento E_8 en el intervalo [7, 8]. Otro evento E_7 llega en el tiempo $T_7 = 7$, pero no hay instancia que lo este esperando, por lo que no es tomado en cuenta. Finalmente, E_8 ocurre en t=8, y la instancia de la crónica I_{32} es reconocida. Cada cierto tiempo, cada CRS local revisa las restricciones de las crónicas instanciadas, para ver si algunas de ellas ya no pueden ser reconocidas (tienen violadas sus restricciones temporales), y así eliminarlas.

3. El reconocimiento de una crónica se realiza cuando dado un flujo de eventos

observables, se alcanza el patrón completo del modelo de la crónica. Es decir, para cada instancia de una crónica en el CRS logra reconocer todos sus eventos. Así, esa instancia se convierte en un reconocimiento de la crónica. Para el ejemplo anterior, se observa en la Figura 4.4 cómo la sub-crónica 3 es reconocida en la instancia I_{32}, y no fue reconocida en la instancia I_{31} (esta instancia fue descartada):

$I_{31} = \{E_6(T_6=2)\}$

$I_{32} = \{E_6(T_6=3),\ E_7(E_7=6),\ E_8(T_8=8)\}$

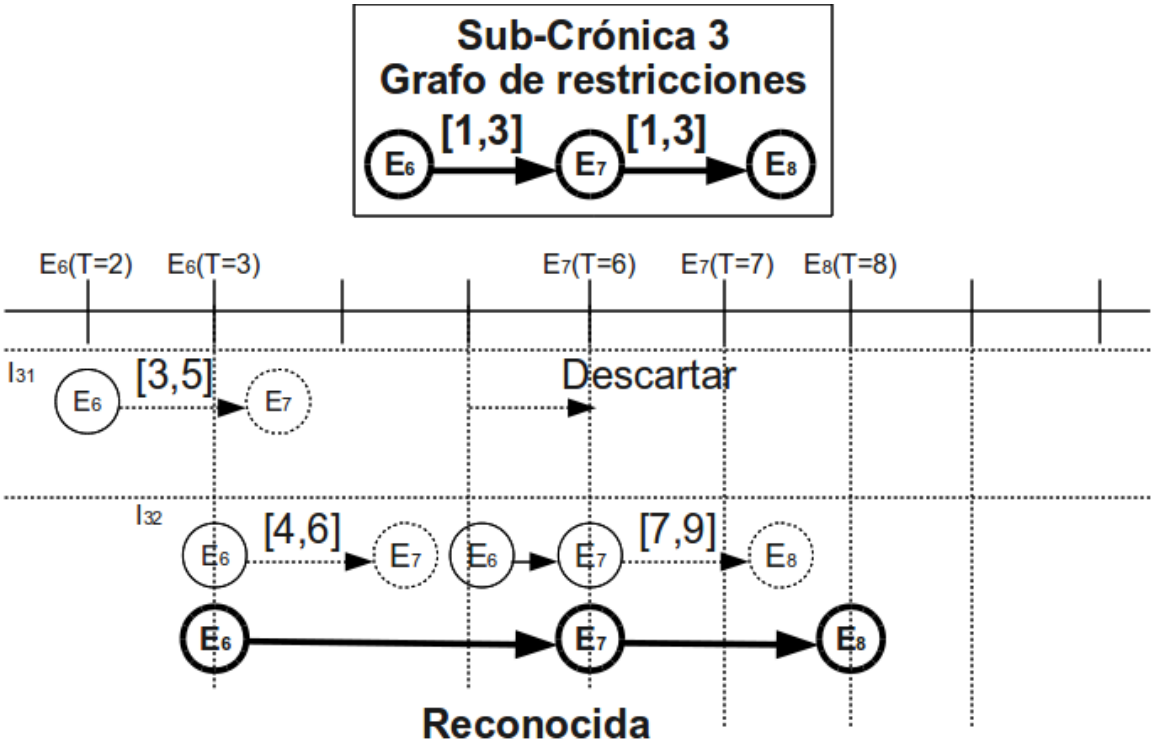


Figura 4.4: Instancias de la sub-crónica 3

4. Normalmente, cuando se reconoce una crónica es porque se identifican comportamientos indeseables en el sistema, por lo que es común la ejecución

de un conjunto de acciones que no se limitan a sólo generar informes de datos. Las mismas pueden consistir en acciones correctivas, o en la generación de nuevos eventos que podrían ser utilizados por otros diagnosticador. En el ejemplo, el modelo de crónica contiene una acción que se ejecuta en el momento de su reconocimiento:

$$\text{Emit event}(e_{SC3}, \text{ Diagnoser 2})$$

Es decir, cuando esa crónica es reconocida en la instancia I_{32} un evento e_{SC3} es generado y enviado al diagnosticador 2.

4.3. Patrones Genéricos de Crónicas Distribuidas para el Diagnóstico de Fallas en la Composición de Servicios Web

Partiendo de la clasificación de fallas en las aplicaciones SOA hecha en el Capítulo 2, se usa el formalismo de crónicas distribuidas para definir en esta sección sus respectivos patrones de fallas.

4.3.1. Fallas Físicas

Estas fallas se deben al entorno en el que opera el servicio (infraestructura) y no están relacionadas con la funcionalidad del servicio, haciendo que el servicio se considere no-disponible. El síntoma es que no es posible invocar el servicio. Para modelar la crónica de esta falla se consideran los eventos generados del ejemplo de la Figura 2.9, la distribución de los eventos por cada diagnosticador seria:

$$D_1 = \{E_1(T_1),\ E_2(\text{status} = \text{'unavailable'},\ T_2)\},\ \text{donde}\ T_2 > T_1 \qquad (4.15)$$

$D_2 = \{\emptyset\}$

El reconocimiento de los eventos E_1 y E_2 con status 'unavailable' en el diagnosticador D_1, permite diagnosticar el síntoma que el servicio S_2 no se puede alcanzar (el servicio S_2 nunca recibe el evento E_2). Sin embargo, no es posible hacer el diagnóstico de que el servicio no está disponible por una falla física, o por una inconsistencia de flujo. Así, para poder diagnosticar esta falla se usa un servicio adicional en el mismo sitio S_1 que permite la operación PING[18] sobre el servidor donde se encuentra alojado el servicio invocador. La operación PING devuelve el estado disponible o no disponible del servidor. El servicio PING es invocado cuando se presenta el síntoma de que el servicio S_2 no es alcanzable. De esta manera, se construye una **sub-crónica Servicio no-disponible 1** con los eventos E_1 y E_2 en el diagnosticador D_1, el cual genera la invocación del servicio PING, que sirve para generar un conjunto de eventos que serán utilizados por una segunda sub-crónica en el mismo sitio S_1. Los eventos generados en la invocación del servicio PING distribuidos entre los diagnosticadores son:

$$D_1 = \{BE_{inv\_PING}(T_{inv\_PING}), E_{PING}(status = 'unavailable', T_{PING})\}, \text{ donde } T_{PING} > T_{inv\_PING} \quad (4.16)$$

$D_2 = \{\emptyset\}$

Donde $Be_{inv\_PING}(T_{inv\_PING})$ corresponde al Evento Enlazador generado por la **sub-crónica Servicio no-disponible 1** cuando es reconocida, y $E_{PING}(T_{PING})$ es el evento que indica la respuesta de la invocación de la operación PING.

El reconocimiento de los eventos $BE_{inv\_PING}(T_{inv\_PING})$ y $E_{PING}(T_{PING})$ permiten el diagnóstico de las falla física de servicio no disponible. Para esto, se construye una segunda sub-crónica **Servicio no-disponible 2** en el sitio S_1 que logre el

18 PING verifica que es posible conectar dos sitios, enviando paquetes y devolviendo el tiempo que tardan en volver a la fuente.

reconocimiento de estos eventos, donde el evento E_{inv_PING} es un evento enlazador que sirve para conectar las dos sub-crónicas. Para generalizar, suponga que los servicios involucrados son S_i y S_j y el patrón de la crónica con las dos sub-crónicas para esta falla cuando el servicio S_i invoca al servicio S_j es:

Diagnoser D_i

```
Subchronicle Si Servicio no-disponible 1 {
  Events{
    event(E1, T1)
    event(E2, Status:'unavailable' T2)
  }
  Constrains{
    T2 > T1
  }
  When recognized{
    PING invoke(Si, hostname: Sj)
    emit event(BEinvPING, TinvPING) to Si diagnoser
  }
}
```

```
Subchronicle Si Servicio no-disponible 2 {
  Events{
    event(BEinvPING, TinvPING)
    event(EPING, status:'NOPING', TPING)
  }
  Constrains{
    TPING > TinvPING + ΔT
  }
  When recognized{
    Repair invoke(Si, 'Unavailable service Sj')
   }
}
```

Figura 4.5: Patrón de Crónica Distribuida para fallas debido a Servicio no disponible

donde:

E_1, E_2: Son los eventos generados en T_1 y T_2 por la aplicación SOA cuando se

invoca el servicio S_j. El evento E_2 tiene el status[19] = unavailable indicando que no se pudo invocar el servicio S_j. Ambos eventos se utilizan para reconocer la sub-crónica Servicio no-disponible 1.

PING invoke: Es la invocación del servicio PING para comprobar si el servicio S_j se encuentran disponible (hostname: S_j).

$BE_{invPING}$: Es el Evento Enlazador generado cuando se reconoce la sub-crónica **"Servicio no-disponible 1"** en D_i.

E_{PING}: Es el evento resultante de la invocación de la operación PING en el tiempo T_{PING} con el status = NOPING (el sitio no es accesible).

ΔT: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_i reciba el evento E_{PING}. En caso de no considerarse retardo se puede colocar en cero ($\Delta T = 0$).

Repair Invoke: Es la invocación de la instancia del reparador en S_i con el diagnóstico de la falla cuando la sub-crónica **"Servicio no-disponible 2"** es reconocida en D_i.

El proceso de reconocimiento de patrones de esta falla se muestra en la Figura 4.6. En un primer momento la aplicación SOA genera los eventos $E_1(T_1)$ y E_2(status: 'unavailable', T_2), los cuales son utilizados por el diagnosticador D_i para el reconocimiento de la sub-crónica "Servicio no-disponible 1", generando el evento enlazador $BE_{invPING}(T_{invPING})$ hacia el mismo (recursivo, Diagnosticador S_i) e invocando el servicio PING para verificar si el servicio Sj es alcanzable. Posteriormente, el servicio PING genera el evento E_{PING}(status='NOPING', T_{PING}) indicando que el servicio S_j no esta disponible. A continuación, se reconoce la sub-

19 Se pueden definir en las crónicas “Variables de Estado” en los eventos (ver definición 4.3)

crónica "Servicio no-disponible 2" en D_i y se diagnóstica la falla de **Servicio no disponible (Unavailable service),** para finalmente invocar su reparador con el diagnóstico.

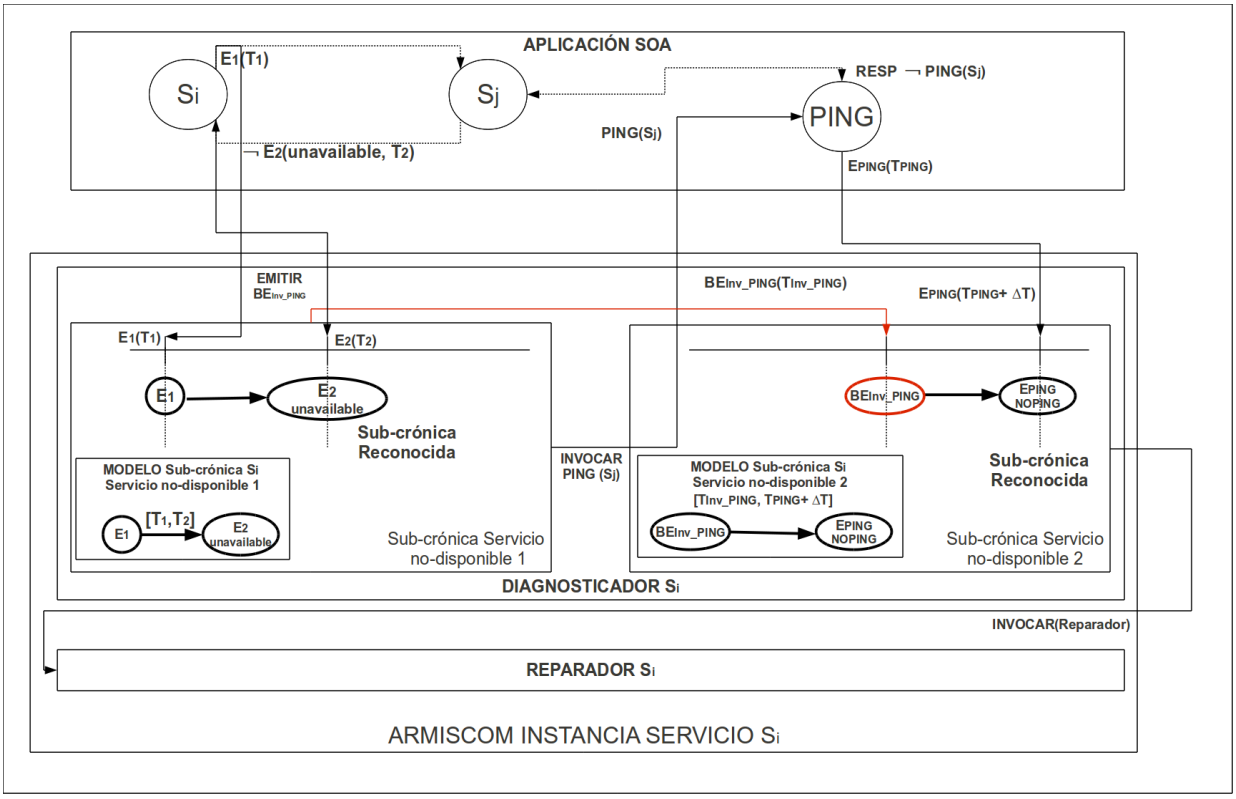


Figura 4.6: Reconocimiento del Patrón de Crónica Distribuida para fallas debido a Servicio no disponible

4.3.2. Fallas de Desarrollo

En el 2.2.1. se definen los siguientes tipos de fallas para este caso:

4.3.2.1 Fallas por Incompatibilidad de Parámetros

Para modelar la crónica de esta falla, se consideran los eventos generados del ejemplo de la Figura 2.10. La distribución de estos por cada diagnosticador sería:

$$D_1 = \{E_1(T_1)\}$$
$$D_2 = \{E_2(\text{status} = \text{'ParameterIncompatibility'}, T_2)\} \quad (4.17)$$

Al estar distribuidos los eventos necesarios para realizar el diagnóstico de la falla, es necesario agregar un evento enlazador que permita al diagnosticador D_1 conocer la falla en el evento $E_2(T_2)$. Para esto, se construye una sub-crónica en el diagnosticador D_2 llamada **"Incompatibilidad de Parámetros"** que reconozca la presencia del evento $E_2(\text{status} = \text{'ParameterIncompatibility'}, T_2)$, añadiendo la condición de que el evento $E_2'([0, T_2 - 1])$ no se haya invocado previamente sin problemas (esta es la primera ejecución de S_2 en la composición, noevent E_2' en el intervalo $[0, T_2 - 1]$) y generando el evento enlazador $BE_{ParamIncom}$ hacia el diagnosticador S_1. Los eventos recibidos finalmente por el diagnosticador D_1 son:

$$D_1 = \{E_1(T_1), BE_{ParamIncom}(T_{ParamIncom})\}, \text{ donde } T_{ParamIncom} > T_1 \quad (4.18)$$

El reconocimiento de los eventos $E_1(T_1)$ y $BE_{ParamIncom}(T_{ParamIncom})$ permite al diagnosticador D_1 el diagnóstico de la falla. Para esto, se construye una sub-crónica **"Incompatibilidad de Parámetros"** en D_1 que logre el reconocimiento de estos eventos, donde el evento $BE_{ParamIncom}$ es un evento enlazador que sirve para conectar las dos sub-crónicas "**Incompatibilidad de Parámetros**" en S_1 y S_2. Para generalizar, suponga que los servicios involucrados son S_i y S_j y el patrón de la crónica con las dos sub-crónicas para esta falla cuando el servicio S_i invoca al servicio S_j es:

Diagnoser D_i	**Diagnoser D_j**
Subchronicle S_i Incompatibilidad de Parámetros { **Events{** event(E_1, T_1) event($BE_{ParamIncom}$, $T_{ParamIncom}$) **}** **Constrains{** $T_{ParamIncom} > T_1 + \Delta T$ **}** **When recognized{** repair invoke(S_i, 'Parameter Incompatibility') **}** **}**	**Subchronicle S_j Incompatibilidad de Parámetros {** **Events{** noevent(E_2, (0, T_2 - 1) event(E_2, Status:'ParameterIncompatibility', T_2) **}** **Constrains{** **}** **When recognized{** emit event($BE_{ParamIncom}$, $T_{ParamIncom}$) to S_i diagnoser **}** **}**

Figura 4.7: Patrón de Crónica Distribuida para fallas de Incompatibilidad de Parámetros

donde:

E_1, E_2: Son los eventos generados en T_1 y T_2 por la aplicación SOA cuando se invoca el servicio S_j. El evento E_2 tiene el status = ParameterIncompatibility indicando que hubo un problema de incompatibilidad de parámetros al invocar a S_j. Los eventos se distribuyen entre los diagnosticadores: $E_1(T_1)$ en el diagnosticador D_1 y $E_2(T_2)$ en el diagnosticador D_j. En el caso concreto de la sub-crónica **"Incompatibilidad de Parámetros"** en D_j, el evento E_2 no debe haberse recibido previamente(noevent) en el intervalo [0, T_2 - 1].

$BE_{ParamIncom}$: Es el evento enlazador generado cuando se reconoce la sub-crónica **"Incompatibilidad de Parámetros"** en D_j.

ΔT: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_i reciba el evento enlazador $BE_{ParamIncom}$. En caso de no considerar retardo se puede colocar en cero (ΔT = 0).

Repair Invoke: Es la invocación del reparador S_i con el diagnostico de la falla cuando la sub-crónica **"Incompatibilidad de Parámetros"** es reconocida en D_i.

El proceso de reconocimiento de patrones de esta falla se muestra en la Figura 4.8. En un primer momento la aplicación SOA genera los eventos $E_1(T_1)$ y E_2(status: 'ParameterIncompatibility ', T_2). El diagnosticador D_j con el evento $E_2(T_2)$ y al no haber ocurrido previamente el evento $E_2(T_2)$, reconoce la sub-crónica **"Incompatibilidad de Parámetros"** y genera el evento enlazador $BE_{ParamIncom}(T_{ParamIncom})$ para el diagnosticador D_i. Con los eventos $E_1(T_1)$ y $BE_{ParamIncom}(T_{ParamIncom})$, el diagnosticador D_i reconoce la sub-crónica **"Incompatibilidad de Parámetros"** y de esta forma, diagnóstica la falla de **Incompatibilidad de Parámetros (Parameter Incompatibility)** e invoca su reparador con el diagnóstico.

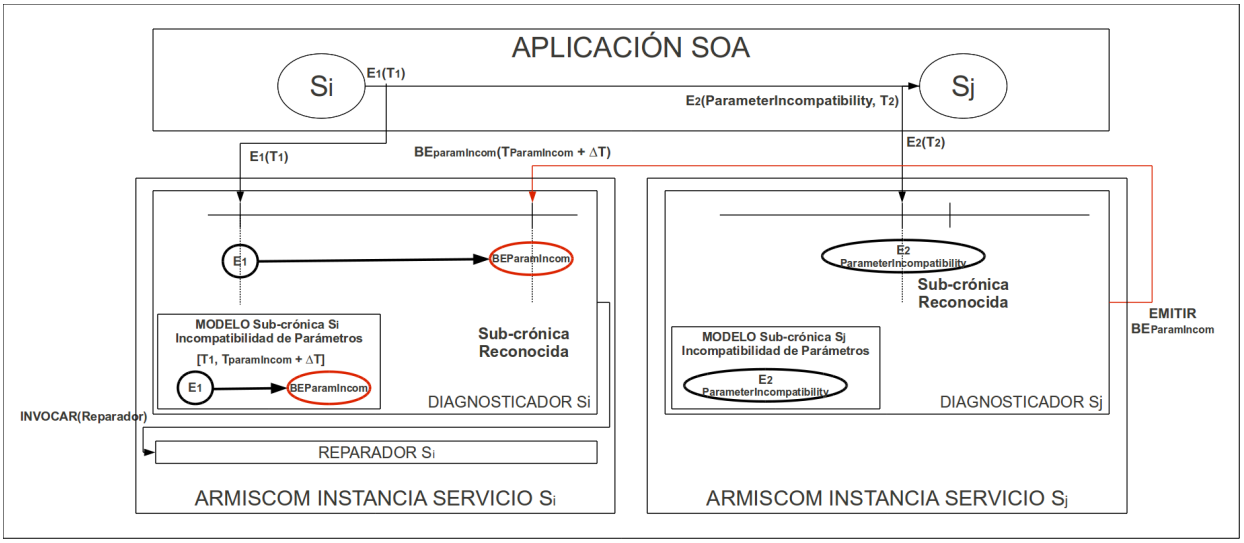


Figura 4.8: Reconocimiento del Patrón de Crónica Distribuida para fallas debido a Incompatibilidad de Parámetros

4.3.2.2 Falla Debido a que la Interfaz Podría Haber Cambiado

La interfaz de algún servicio S_i que es parte de la composición se modifica, de modo que se origina una incompatibilidad de los parámetros cuando el Servicio S_i es nuevamente invocado (en ocasiones anteriores había sido invocado sin problemas). Para modelar la crónica de esta falla se consideran los eventos generados del ejemplo de la Figura 2.11, la distribución de estos por cada diagnosticador seria:

$$D_1 = \{E_1(T_1), E_1'(T_1')\}, \text{ donde } T_1' > T_1$$
$$D_2 = \{E_2(T_2), E_2'(\text{status} = \text{'ParameterIncompatibility'}, T_2')\}, \text{ donde } T_2' > T_2 \quad (4.19)$$

El reconocimiento de los eventos $E_2(T_2)$ y E_2'(status = 'ParameterIncompatibility', T_2') en el diagnosticador D_2 permiten diagnosticar el síntoma que el servicio S_1 está suministrando una entrada no compatible con la interfaz de S_1 en el instante T_2'. Sin embargo, no es posible localmente (diagnosticador D_2) diagnosticar que la falla es debida a que la Interfaz podría haber cambiado en el servicio S_i. E_1-E_2 y E_1'–E_2' están indicando que hay llamadas previas entre los dos servicios, y la última da problemas (por el mensaje de error). Pero se requiere de algo más para poder hacer el diagnóstico. Así, para diagnosticar esta falla se construye una sub-crónica en el diagnosticador D_2, llamada **"Interfaz podría haber Cambiado"**, que reconozca la presencia de los eventos $E_2(T_2)$ y E_2'(status = 'ParameterIncompatibility', T_2'), y genere el evento enlazador $BE_{ParamIncom}$ hacia el diagnosticador D_1. Los eventos recibidos finalmente por el diagnosticador D_1 son:

$$D_1 = \{E_1(T_1), E_1'(T_1'), BE_{ParamIncom}(T_{ParamIncom})\}, \text{ donde } T_{ParamIncom} > T_1' \text{ y } T_1' > T_1 \quad (4.20)$$

Posteriormente, se construye una sub-crónica en D_1 para el diagnóstico de esta falla, para esto se toman los eventos y restricciones temporales mostrados en la ecuación 4.20, pero además, se añade la condición de que el evento enlazador $BE_{ParamIncom}$ no ha ocurrido previamente (el servicio S_2 se ha ejecutado previamente en la composición sin problemas). Para generalizar, suponga que los servicios involucrados son S_i y S_j, el patrón de la crónica con las dos sub-crónicas para esta falla cuando el servicio S_i invoca al servicio S_j es:

Diagnoser D_i	**Diagnoser D_j**
Subchronicle S_i Interfaz podría haber Cambiado { **Events{** event(E_1, T_1) noevent($BE_{ParamIncom}$, (T_1, $T_{ParamIncom}$ + ΔT)) event(E_1', T_1') event($BE_{ParamIncom}$, $T_{ParamIncom}$) **}** **Constrains{** $T_1' > T_1$ $T_{ParamIncom} > T_1' + \Delta T$ **}** **When recognized{** repair invoke(S_i, 'Interface Might Have Changed') **}** **}**	**Subchronicle S_j Interfaz podría haber Cambiado {** **Events{** event(E_2, T_2) event(E_2', status:'ParameterIncompatibility', T_2') **}** **Constrains{** $T_2' > T_2$ **}** **When recognized{** emit event($BE_{ParamIncom}$, $T_{ParamIncom}$) to S_i diagnoser **}** **}**

Figura 4.9: Patrón de Crónica Distribuida para fallas debido a que la interfaz podría haber cambiado

donde

E_1, E_1', E_2, E_2': Son los eventos generados en T_1, T_2, T_1' y T_2' por la aplicación SOA cuando se invoca el servicio S_j. El evento E_2' tiene el status = 'ParameterIncompatibility' indicando que hubo un problema de incompatibilidad de parámetros al invocar a S_j. Previamente, se había invocado sin problemas

según los eventos $E_1(T_1)$ y $E_2(T_2)$. Los eventos se distribuyen entre los diagnosticadores: $E_1(T_1)$ y $E_1(T_1')$ en el diagnosticador D_i y $E_2(T_2)$ y $E_2'(T_2')$ en el diagnosticador D_j.

$BE_{ParamIncom}$: Es el evento enlazador generado cuando se reconoce la sub-crónica **"Interfaz podría haber cambiado"** en D_j. Este evento no debe haberse recibido previamente (noevent) en D_i en el periodo de tiempo $[0, T_{ParamIncom}]$.

ΔT: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_i reciba el evento enlazador $BE_{ParamIncom}$. En caso de no considerar retardo se puede colocar en cero ($\Delta T = 0$).

Repair Invoke: Es la invocación del reparador Si con el diagnóstico de la falla cuando la sub-crónica **"Interfaz podría haber cambiado"** es reconocida en D_i.

Finalmente, el proceso de reconocimiento de patrones de esta falla se muestra en la Figura 4.10. En un primer momento la aplicación SOA genera los eventos $E_1(T_1)$, $E_2(T_2)$, $E_1'(T_1')$ y E_2'(status: 'ParameterIncompatibility', T_2'). El diagnosticador D_j con los eventos $E_2(T_2)$ y E_2'(status: 'ParameterIncompatibility', T_2') reconoce la sub-crónica **"Interfaz podría haber Cambiado"** y genera el evento enlazador $BE_{ParamIncom}(T_{ParamIncom})$ hacia el diagnosticador D_i. Con los eventos $E_1(T_1)$, $E_1'(T_1')$ y $BE_{ParamIncom}(T_{ParamIncom})$, y la clausula de noevent, el diagnosticador D_i reconoce la sub-crónica **"Interface Might Have Changed"** al no haber recibido previamente el evento enlazador $BE_{ParamIncom}$ en el intervalo $[0, T_{paramIncom}]$. Así, puede diagnosticar la falla de **Interfaz podría haber cambiado (Interface Might Have Changed)** e invocar su reparador con el diagnóstico.

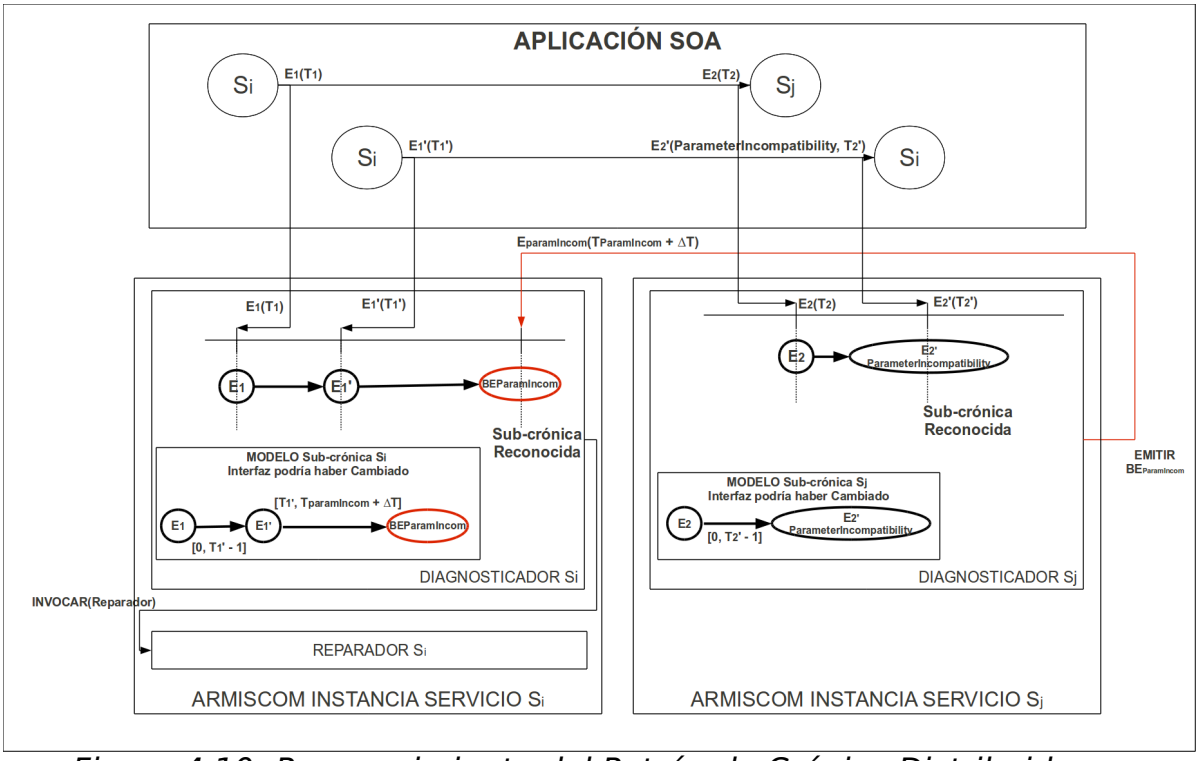


Figura 4.10: Reconocimiento del Patrón de Crónica Distribuida para fallas debido a Interfaz podría haber Cambiado

4.3.2.3 Falla Debido a Flujo de Trabajo Inconsistente

Como se comento en el apartado 2.2.1., El diagnóstico de este tipo de falla es muy complicada, debido a que la información de invocación de las operaciones no es correcto y el servicio no puede ser invocado, confundiéndose con la falla física "Servicio no disponible", por lo que la crónica distribuida para reconocer está falla es muy similar a la de "Servicio no disponible". Para modelar la crónica de esta falla se consideran los eventos generados del ejemplo de la Figura 2.9, la distribución de estos por cada diagnosticador seria:

$D_1 = \{E_1(T_1), E2(status = 'unavailable', T_2)\}$, donde $T_2 > T_1$ (4.21)

$D_2 = \{\emptyset\}$

El reconocimiento de los eventos E_1 y E_2 en el diagnosticador D_1 permite diagnosticar el síntoma que el servicio S_2 no se puede alcanzar (el servicio S_2 nunca recibe el evento E_2). Sin embargo, no es posible deducir si el diagnóstico de que el servicio no está disponible es por una falla física o por una inconsistencia de flujo. Así, para poder diagnosticar esta falla se usa un servicio adicional que permite la operación PING (descrita en la falla de “Servicio no disponible”) sobre el servidor donde se encuentra alojado el servicio invocador (como en el caso anterior). La operación PING devuelve el estado disponible o no disponible del servidor. El servicio PING es invocado cuando se presenta el síntoma de que el servicio S_2 no es alcanzable. Para esto, se construye una sub-crónica **"Flujo de trabajo inconsistente 1"** con los eventos E_1 y E_2 en el diagnosticador D_1, el cual genera la invocación del servicio PING y el evento enlazador BE_{inv_PING} que es usado por una segunda sub-crónica en el diagnosticador D_1 (como en el caso anterior). Los eventos generados en la invocación del servicio PING son:

$$\{BE_{inv\_PING}(T_{inv\_PING}),\ E_{PING}(status = 'available',\ T_{PING})\},\quad \text{donde } T_{PING} > T_{inv\_PING} \tag{4.22}$$

El reconocimiento de los eventos $BE_{inv\_PING}(T_{inv\_PING})$ y $E_{PING}(T_{PING})$ permite el diagnóstico de esta falla. La segunda sub-crónica **"Flujo de trabajo inconsistente 2"** en el diagnosticador D_1 logra el reconocimiento de estos eventos en el diagnosticar T_1, donde el evento BE_{inv_PING} es un evento enlazador que sirve para conectar las dos sub-crónicas en ese sitio. Algo importante a resaltar es la diferencia de la respuesta cuando se invoca a PING aquí (status = 'available'), con respecto a cuando se invoca cuando esta ocurriendo la falla física "Servicio no disponible" (status = 'unavailable'). Para generalizar, suponga que los servicios involucrados son S_i y S_j y el patrón de la crónica con las dos sub-crónicas para esta falla cuando el servicio S_i invoca al servicio S_j es:

Diagnoser D_i

Subchronicle S_i Flujo de trabajo inconsistente 1 {

Events{

event(E_1, T_1)

event(E_2, Status:'unavailable' T_2)

}

Constrains{

$T_2 > T_1$

}

When recognized{

PING invoke(S_i, hostname: S_j)

emit event($BE_{invPING}$, $T_{invPING}$) to S_i diagnoser

}

}

Subchronicle S_i Flujo de trabajo inconsistente 2 {

Events{

event($BE_{invPING}$, $T_{invPING}$)

event(E_{PING}, status:'PING', T_{PING})

}

Constrains{

$T_{PING} > T_{invPING} + \Delta T$

}

When recognized{

Repair invoke(S_i, 'Work-flow Inconsistent Sj')

}

}

Figura 4.11: Patrón de Crónica Distribuida para fallas debido a Flujo de trabajo inconsistente

donde:

E_1, E_2: Son los eventos generados en T_1 y T_2 por la aplicación SOA cuando se invoca el servicio S_j. El evento E_2 tiene el status = unavailable indicando que no se pudo invocar el servicio S_j. Ambos eventos se utilizan para reconocer la sub-crónica **"Flujo de trabajo inconsistente 1"**.

PING invoke: Es la invocación del servicio PING para comprobar si el servicio

S_j se encuentran disponible (hostname: S_j).

$BE_{invPING}$: Es el evento enlazador generado cuando se reconoce la sub-crónica **"Flujo de trabajo inconsistente 1"** en D_i.

E_{PING}: Es el evento resultante de la invocación de la operación PING en el tiempo T_{PING} con el status = PING (el sitio es accesible).

ΔT: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_i reciba el evento E_{PING}. En caso de no considerar retardo se puede colocar en cero (ΔT = 0).

Repair Invoke: Es la invocación de la instancia del reparador en S_i con el diagnostico de la falla cuando la sub-crónica **"Flujo de trabajo inconsistente 2"** es reconocida en D_i.

El proceso de reconocimiento de patrones de esta falla se muestra en la Figura 4.12. En un primer momento la aplicación SOA genera los eventos $E_1(T_1)$ y E_2(status: 'unavailable', T_2), los cuales son utilizados por el diagnosticador D_i para el reconocimiento de la sub-crónica **"Flujo de trabajo inconsistente 1"**, generando el evento enlazador $BE_{invPING}(T_{invPING})$ hacia el mismo (recursivo, Diagnosticador S_i) e invocando el servicio PING para verificar si el servicio S_j es alcanzable. Posteriormente, el servicio PING genera el evento E_{PING}(status='PING', T_{PING}) indicando que el servicio S_j esta disponible, permitiéndole a la sub-crónica **"Flujo de trabajo inconsistente 2"** ser reconocida en D_i y diagnosticar la falla de **Flujo de trabajo inconsistente (Work-flow Inconsistent)**, e invocar su reparador con el diagnóstico.

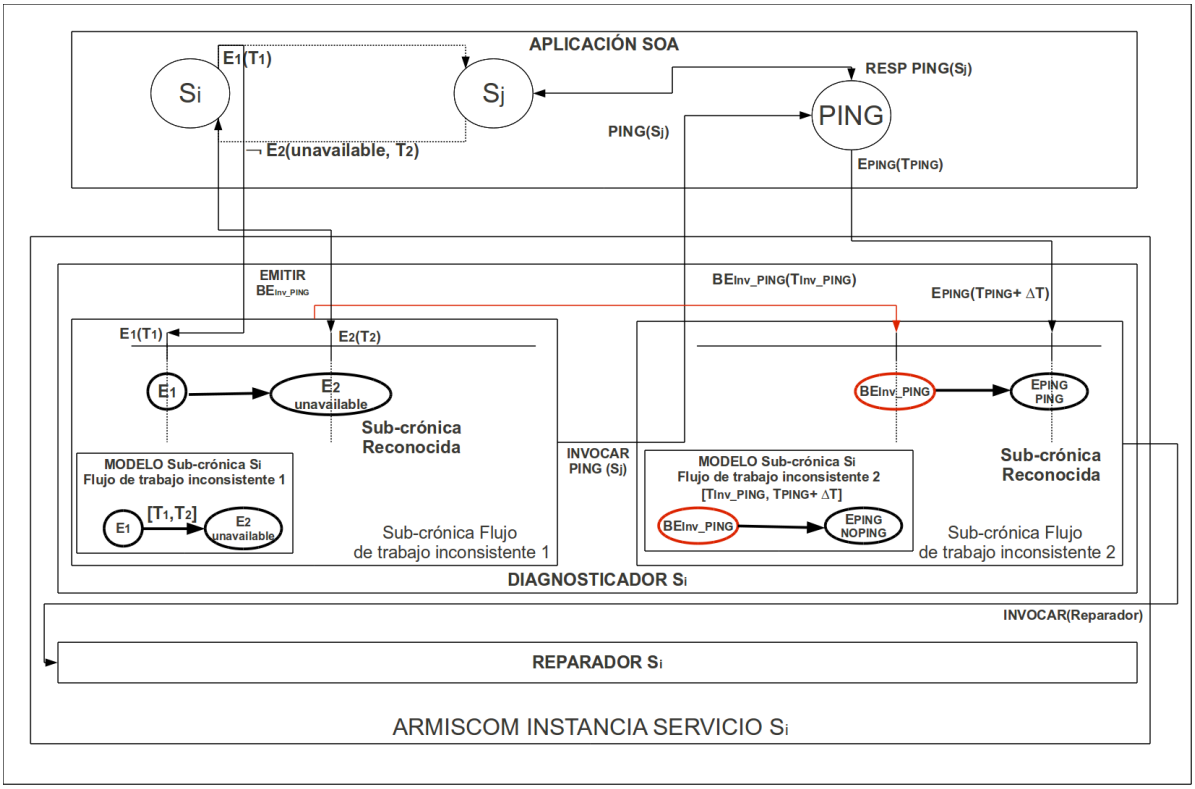


Figura 4.12: Reconocimiento del Patrón de Crónica Distribuida para fallas debido a Flujo de trabajo inconsistente

4.3.3. Fallas de Interacción

En el Capítulo 2 se indico que en este caso los tipos de fallas pueden ser de contenido y tiempo. A continuación se describen sus patrones basado en crónicas.

4.3.3.1 Falla Debido a Acción no-Determinada

Esta falla se produce cuando el valor de la respuesta de un servicio no siempre genera el valor esperado que debe recibir otro servicio en la composición. Para modelar la crónica de esta falla se consideran los eventos generados del ejemplo de la Figura 2.12, la distribución de estos por cada diagnosticador seria:

$$D_1 = \{E_1(T_1), E_1'(T_1')\}, \text{ donde } T_1' > T_1$$
$$D_2 = \{E_2(T_2), E_2'(\text{status} = \text{'Bad response'}, T_2')\}, \text{ donde } T_2' > T_2 \quad (4.22)$$

El reconocimiento de los eventos $E_2(T_2)$ y E_2'(status = 'Bad response', T_2') en el diagnosticador D_2 permite diagnosticar el síntoma que el servicio S_1 esta suministrando una entrada no compatible con la interfaz de S_2 en el instante T_2', pero en un instante T_2 anterior la había suministrado correctamente.

Así, para poder diagnosticar esta falla, se construye una sub-crónica en el diagnosticador D_2 llamada **"Acción no-Determinada"** que reconozca la presencia de los eventos $E_2(T_2)$ y E_2'(status = 'Bad response', T_2'), y que genere el evento enlazador $BE_{IncServ}$ hacia el diagnosticador D_1. Los eventos recibidos finalmente por el diagnosticador D_1 son:

$$D_1 = \{E_1(T_1), E_1'(T_1'), BE_{IncServ}(T_{IncServ})\}, \text{ donde } T_{IncServ} > T_1' \text{ y } T_1' > T_1 \quad (4.23)$$

Posteriormente, se construye una sub-crónica en D_1 para el diagnóstico de esta falla, para lo cual se toman los eventos y restricciones temporales mostrados en la ecuación (4.23), y se añade la condición de que el evento enlazador $BE_{IncServ}$ no ha ocurrido previamente (el servicio S_2 se ha ejecutado previamente en la composición sin problemas). Para generalizar, suponga que los servicios involucrados son S_i y S_j y el patrón de la crónica con las dos sub-crónicas para esta falla cuando el servicio S_i invoca al servicio S_j es:

Diagnoser D_i	**Diagnoser D_j**
Subchronicle S_i Acción no-Determinada { **Events{** event(E_1, T_1) event(E_1', T_1') noevent($BE_{NdetAct}$, (0, $T_{NdetAct}$ + ΔT)) event($BE_{NdetAct}$, $T_{NdetAct}$) **}** **Constrains{** T_1' > T_1 $T_{NdetAct}$ > T_1' + ΔT **}** **When recognized{** repair invoke(S_i, 'Non-deterministic Actions') **}** **}**	**Subchronicle S_j Acción no-Determinada {** **Events{** event(E_2, T_2) event(E_2', status:'Bad response', T_2') **}** **Constrains{** T_2' > T_2 **}** **When recognized{** emit event($BE_{NdetAct}$, $T_{NdetAct}$) to S_i diagnoser **}** **}**

Figura 4.13: Patrón de Crónica Distribuida para fallas debido a acción no-determinista

donde

E_1, E_1', E_2, E_2': Son los eventos generados en T_1, T_2, T_1' y T_2' por la aplicación SOA cuando se invoca el servicio S_j. El evento E_2' tiene el status = 'Bad Response' indicando que el contenido de la respuesta no es correcta al invocar a S_j y ha sido invocado previamente sin problemas según los eventos $E_1(T_1)$ y $E_2(T_2)$. Los eventos se distribuyen entre los diagnosticadores: $E_1(T_1)$ y $E_1(T_1')$ en el diagnosticador D_i y $E_2(T_2)$ y $E_2'(T_2')$ en el diagnosticador D_j.

$BE_{NdetAct}$: Es el evento enlazador generado cuando se reconoce la sub-crónica **"Acción no-Determinada"** en D_j. Este evento no debe haberse recibido previamente(noevent) en D_i en el periodo de tiempo [0, $T_{NdetAct}$].

ΔT: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_i reciba el evento enlazador $BE_{NdetAct}$. En caso de no considerar

retardo se puede colocar en cero ($\Delta T = 0$).

Repair Invoke: Es la invocación del reparador Si con el diagnóstico de la falla cuando la sub-crónica **"Non-deterministic Actions"** es reconocida en D_i.

Finalmente, el proceso de reconocimiento de patrones de esta falla se muestra en la Figura 4.14. En un primer momento, la aplicación SOA genera los eventos $E_1(T_1)$, $E_2(T_2)$, $E_1'(T_1')$ y E_2'(status: 'Bad Response', T_2'). El diagnosticador D_j con los eventos $E_2(T_2)$ y E_2'(status: 'Bad Response', T_2') reconoce la sub-crónica **"Acción no-Determinada"**, y genera el evento enlazador $BE_{NdetAct}(T_{NdetAct})$ hacia el diagnosticador D_i. Con los eventos $E_1(T_1)$, $E_1'(T_1')$ y $BE_{NdetAct}(T_{NdetAct})$, y la clausula que determina no haber recibido previamente el binding event $BE_{NdetAct}$ en el intervalo $[0, T_{ndetAct}]$, el diagnosticador D_i reconoce la sub-crónica "Non-deterministic Actions". Ahora puede diagnosticar la falla de **Acción no-Determinada (Non-deterministic Actions)** e invocar su reparador con el diagnóstico.

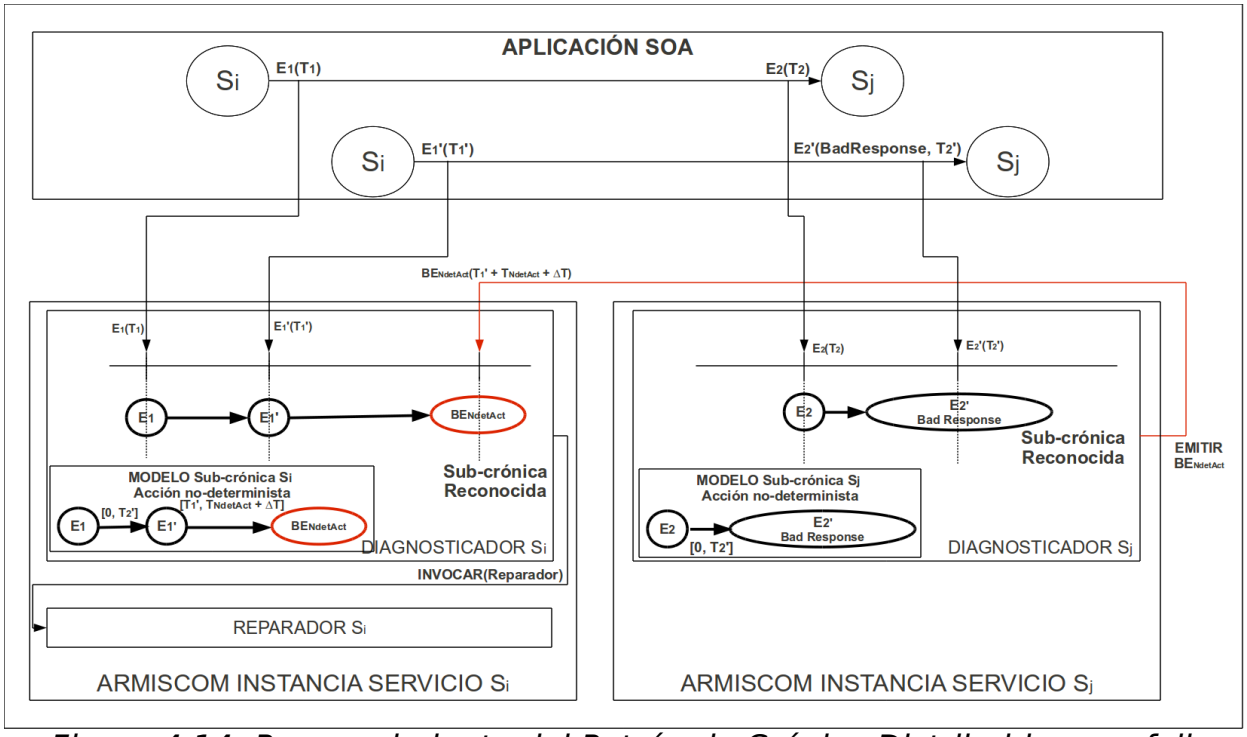


Figura 4.14: Reconocimiento del Patrón de Crónica Distribuida para fallas debido a acción no-determinista

4.3.3.2 Falla Debido a un Comportamiento Incomprendido (Servicio Incorrecto)

Uno de los servicios en el flujo de la composición no produce los resultados esperados, y es la primera vez que se ejecuta la composición (se diferencia de la falla Acción no-determinista, debido a que la falla comportamiento incomprendido nunca ha funcionado). Para modelar la crónica de esta falla, se consideran los eventos generados del ejemplo de la Figura 2.13, la distribución de estos por cada diagnosticador seria:

$$D_1 = \{E_1(T_1)\}$$
$$D_2 = \{E_2(\text{status} = \text{'Bad response'}, T_2)\} \quad (4.24)$$

Al estar distribuidos los eventos necesarios para realizar el diagnóstico de la

falla, es necesario agregar un evento enlazador que permita al diagnosticador D_1 conocer la falla en el evento $E_2(T_2)$. Para esto, se construye una sub-crónica en el diagnosticador D_2 llamada **"Comportamiento incomprendido",** que reconozca la presencia del evento E_2(status = 'Bad response', T_2), añadiendo la condición de que el evento $E_2(T_2)$ no se haya invocado previamente (esta es la primera ejecución de S_2 en la composición, por consiguiente hay un noevent E_2 en el intervalo [0, T_2]), y generando el evento enlazador $BE_{IncServ}$ hacia el diagnosticador D_1. Los eventos recibidos finalmente por el diagnosticador D_1 son:

$$D_1 = \{E_1(T_1), BE_{IncServ}(T_{IncServ})\} \quad (4.25)$$

El reconocimiento de los eventos $E_1(T_1)$ y $BE_{IncServ}(T_{IncServ})$ permiten al diagnosticador D_1 el diagnóstico de la falla. Para esto, se construye una sub-crónica sub-crónica **"Comportamiento incomprendido"** en D_1 que logre el reconocimiento de estos eventos, donde el evento $BE_{IncServ}$ es un evento enlazador que sirve para conectar las dos sub-crónicas **"Comportamiento incomprendido"** en los diagnosticadores D_1 y D_2. Para generalizar, suponga que los servicios involucrados son S_i y S_j, el patrón de la crónica con las dos sub-crónicas para esta falla cuando el servicio S_i invoca al servicio S_j es:

Diagnoser D_i	**Diagnoser D_j**
Subchronicle S_i Comportamiento incomprendido { **Events{** event(E_1, T_1) event($BE_{IncServ}$, $T_{IncServ}$) **}** **Constrains{** $T_{NdetAct} > T_1$ **}** **When recognized{** repair invoke(S_2, 'Misunderstood Behavior') **}** **}**	**Subchronicle S_j Comportamiento incomprendido {** **Events{** noevent(E_2, (0, T_2 - 1)) event(E_2, status:'Bad response' T_2) **}** **Constrains{** **}** **When recognized{** emit event($BE_{IncServ}$, $T_{IncServ}$) to S_i diagnoser **}** **}**

Figura 4.15: Patrón de Crónica Distribuida para fallas debido a comportamiento incomprendido

E_1, E_2: Son los eventos generados en T_1 y T_2 por la aplicación SOA cuando se invoca el servicio S_j. El evento E_2 tiene el status = Bad Response indicando que hubo un problema de incompatibilidad de parámetros al invocar a S_j. Los eventos se distribuyen entre los diagnosticadores: $E_1(T_1)$ en diagnosticador D_i y $E_2(T_2)$ en el diagnosticador D_j. En el caso concreto de la sub-crónica **"Comportamiento incomprendido"** en D_j, el evento E_2 no debe haberse recibido previamente (noevent) en el intervalo [0, T2 - 1].

$BE_{IncServ}$: Es el evento enlazador generado cuando se reconoce la sub-crónica **"Comportamiento incomprendido"** en D_j.

ΔT: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_i reciba el evento enlazador $BE_{IncServ}$. En caso de no considerar retardo se puede colocar en cero (ΔT = 0).

Repair Invoke: Es la invocación del reparador S_i con el diagnóstico de la falla

cuando la sub-crónica **"Comportamiento incomprendido"** es reconocida en D_i.

El proceso de reconocimiento de patrones de esta falla se muestra en la Figura 4.16. En un primer momento la aplicación SOA genera los eventos $E_1(T_1)$ y E_2(status: 'Bad Response', T_2). El diagnosticador D_j con el evento $E_2(T_2)$ y al no haber ocurrido previamente el evento $E_2(T_2)$ (NOEVENT), reconoce la sub-crónica **"Comportamiento incomprendido"**, y genera el evento enlazador $BE_{IncomIncServ}(T_{IncServ})$ hacia el diagnosticador D_i. Con los eventos $E_1(T_1)$ y $BE_{IncServ}(T_{IncServ})$; el diagnosticador D_i reconoce la sub-crónica **"Comportamiento incomprendido"** y diagnóstica la falla de Comportamiento Incomprendido (Misunderstood Behavior), para pasar a invocar su reparador con el diagnóstico.

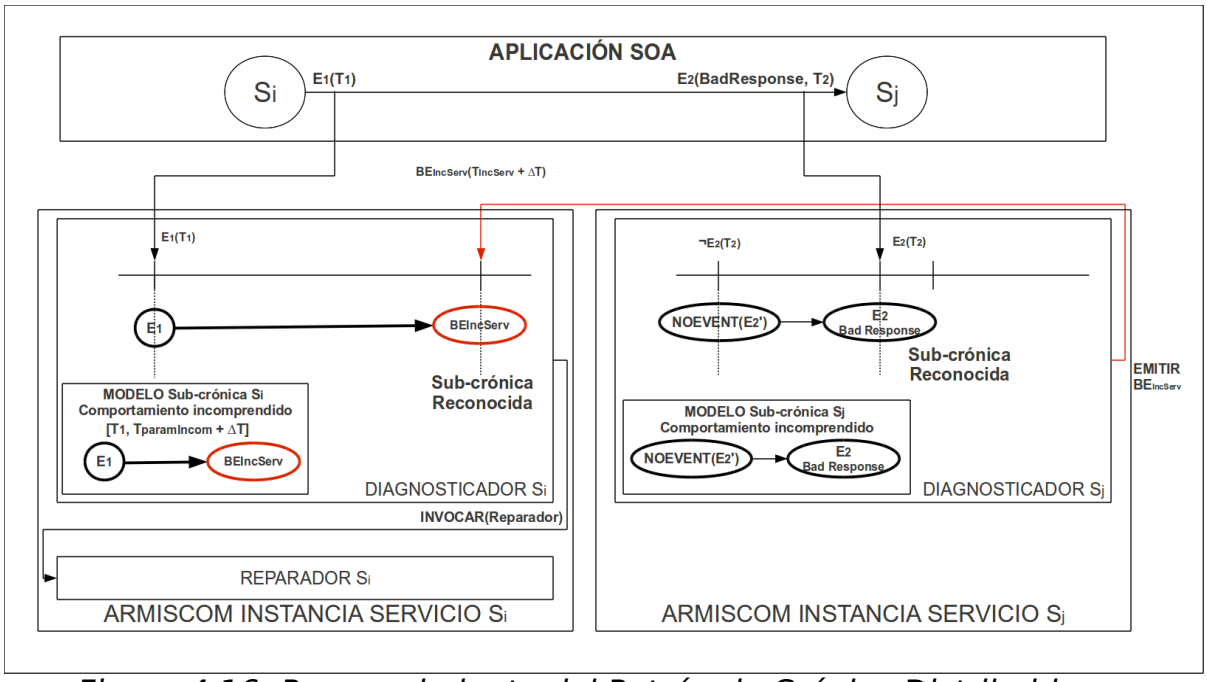


Figura 4.16: Reconocimiento del Patrón de Crónica Distribuida para fallas debido a comportamiento incomprendido

4.3.3.3 Falla Debido a un Mal Comportamiento del Flujo Ejecución

Esta falla es un extensión de la anterior, la previa es de un servicio incorrecto,

y esta es cuando parte del flujo es incorrecto. Para modelar la crónica de esta falla se consideran los eventos generados del ejemplo de la Figura 2.14, la distribución de estos por cada diagnosticador es:

$$
\begin{aligned}
&D_1 = \{E_1(\text{status} = \text{'TrueResponse'}, T_1)\} \\
&D_2 = \{E_2(T_2), E_3(T_3)\}, \text{ donde } T_3 > T_2 \\
&D_3 = \{E_4(T_4), E_5(T_5)\}, \text{ donde } T_5 > T_4 \\
&D_4 = \{E_6(\text{status} = \text{'BadResponse'}, T_6)\}
\end{aligned}
\tag{4.26}
$$

Note que para reconocer esta falla es necesario conocer que el contenido de un evento recibido en algún diagnosticador es correcto (E_1 tiene el status TrueResponse), los eventos E_2, E_3, E_4 y E_5 no se saben si son correctos. Al estar distribuidos los eventos necesarios para realizar el diagnóstico de la falla, es necesario agregar los eventos enlazadores que permitan a los diagosticadores conocer la falla en el evento $E_6(T_6)$. Para esto, se construye una sub-crónica en el diagnosticador D_4 (el diagnosticador donde es detectado por primera vez el síntoma) llamada **"Mal comportamiento del flujo Ejecución"** que reconozca la presencia del evento E_6(status = 'BadResponse', T_6) y genere el evento enlazador BE_{Misbe_Flow} hacia el resto de los diagnosticadores de los servicios que se han ejecutado previamente (D_1, D_2 y D_3), y el evento enlazador BE_{TRUE} con status='S_4' hacia D_3 (el evento E_6 no tiene status = 'TrueResponse'). Los eventos recibidos finalmente por estos diagnosticadores son:

$$
\begin{aligned}
&D_1 = \{E_1(\text{status} = \text{'TrueResponse'}, T_1)), BE_{Misbe\_Flow}(\text{status} = \text{'}S_4\text{'}, T_{Misbe\_Flow})\}, \text{ donde } T_{Misbe\_Flow} > T_1 \\
&D_2 = \{E_2(T_2), E_3(T_3), BE_{Misbe\_Flow}(\text{status} = \text{'}S_4\text{'}, T_{Misbe\_Flow})\}, \\
&\text{donde } T_3 > T_2 \text{ y } T_{Misbe\_Flow} > T_3
\end{aligned}
\tag{4.27}
$$

$D_3 = \{E_4(T_4), E_5(T_5), BE_{Misbe\_Flow}(status = 'S_4', T_{Misbe\_Flow}), BE_{TRUE}(status='S_4', T_{TRUE})\}$, donde $T_5 > T_4$ y $T_{Misbe\_Flow} > T_5$

Con los eventos $E_4(T_4)$, $E_5(T_5)$, $BE_{Misbe\_Flow}(T_{Misbe\_Flow})$ y $BE_{TRUE}(status='S_4', T_{TRUE})$ el diagnosticador D_3 puede inferir que hay un problema con el flujo de la aplicación (el evento E_5 no tiene status = 'TrueResponse'), para esto se construye una sub-crónica llamada **"Mal comportamiento del flujo Ejecución "** en D_3, que genera un evento enlazador BE_{TRUE} con status='S_3' hacia D_2. Los eventos recibidos finalmente para el diagnosticador D_2 son:

$D_2 = \{E_2(T_2), E_3(T_3), BE_{Misbe\_Flow}(status = 'S_4', T_{Misbe\_Flow}), BE_{TRUE}(status='S_3', T_{TRUE})\}$, donde $T_3 > T_2$, $T_{Misbe\_Flow} > T_3$ y $T_{TRUE} > T_3$ (4.28)

Con los eventos definidos en (4.28), el diagnosticador D_2 puede inferir que hay un problema con el flujo de la aplicación (el evento E_3 no tiene status = 'TrueResponse'), para esto se construye una sub-crónica llamada **"Mal comportamiento del flujo Ejecución"** en D_2, que genera un evento enlazador BE_{TRUE} con status='S_2' hacia D_1. Los eventos recibidos finalmente por D_1 son:

$D_1 = \{E_1(status = 'TrueResponse', T_1)), BE_{Misbe\_Flow}(status ='S_4', T_{Misbe\_Flow}), BE_{TRUE}(status='S_2', T_{TRUE})\}$, donde $T_{Misbe\_Flow} > T_1$ y $T_{TRUE} > T_1$ (4.29)

Con los eventos E_1(status = 'TrueResponse', T_1), BE_{Misbe_Flow}(status='S_4', T_{Misbe_Flow}) y BE_{TRUE}((status='S_2', T_{TRUE}) el diagnosticador D_1 puede inferir que hay problemas en el flujo de la aplicación SOA. Para ello se construye una crónica llamada **"Mal comportamiento del flujo Ejecución"** en D_1, que reconoce los evento E_1(status = 'TrueResponse', T_1)), BE_{Misbe_Flow}(status ='S_4', T_{Misbe_Flow}), BE_{TRUE}((status='S_2', T_{TRUE}), e invoca al reparador en S_1.

Así, para generalizar el modelo de la crónica distribuida para la falla **"Mal**

comportamiento del flujo Ejecución", suponga que una aplicación SOA contiene en su flujo de ejecución i servicios: S_1, S_2, .., S_i. Al realizar la invocación de la aplicación sucede un evento $E_{i-1,i}$(status = 'BadResponse', $T_{i-1,i}$)[20] en el servicio S_i y se conoce que la salida generada en algún servicio invocado previamente S_{i-k} es correcta y corresponde con el evento $E_{i-k,i-k+1}$. Para lograr el reconocimiento de la falla es necesario construir una sub-crónica llamada **"Mal comportamiento del flujo Ejecución"** en D_i que reconozca el evento $E_{h,i}$(status = 'Badresponse', $T_{h,i}$) y genere el evento enlazador BE_{Misbe_Flow}(status ='S_i', T_{Misbe_Flow}) que será propagado a los diagnosticadores D_1, .., D_{i-1}, como se describió antes.

En este mismo orden de ideas, para propagar los eventos BE_{TRUE}((status='S_j', T_{TRUE}) a lo largo de los diagnosticadores D_{i-1}, .., D_{i-k} es necesario la construcción de una sub-crónica en cada D_j (D_j | ∀ j > 0 & j < i) para que logre reconocer los eventos BE_{Misbe_Flow}(status='S_i', T_{Misbe_Flow}) y BE_{TRUE}(status='S_j', T_{TRUE}) (los evento BE_{TRUE} se van generando recursivamente a medida que se va avanzando en orden invertido en el flujo de ejecución, p.e. el diagnosticador D_{i-1} genera el evento enlazador BE_{TRUE}(status='S_{i-1}', T_{TRUE}) hacia D_{i-2}). Finalmente, la sub-crónica que contiene al evento que se conoce que su información es correcta ($E_{i-k,\ i-k+1}$ con status TrueResponse), o que contiene el evento que inicio la aplicación ($E_{1,\ 2}$) (no se encontró un evento con status TrueResponse), reconoce la falla. El patrón de la crónicas distribuidas para esta falla se caracteriza por el siguiente programa recursivo, que después es implementado en dos tipos de crónicas en un mismo sitio dado:

20 La nomenclatura para eventos $E_{j,i}$ se usa en este caso de falla para facilitar el desarrollo y comprensión de las crónicas, y define un evento que se genera en el sitio i (servicio i) para el sitio j, donde i>j.

```
1. Algoritmo recursivo_Diagnosticador_i-k{

2.   event(E_{i-k, i-k-1, T_{i-k, i-k-1}) // evento que viene del servicio previo (i-k-1)

3.   event(BE_{Misbe_Flow}, status ='S_i', T_{Misbe_Flow}) //evento generado por el
  diagnosticador i

4.   event(BE_{TRUE}, status ='S_{i-k+1}', T_{TRUEi-k+1}) //evento generado por el diagnosticador
  anterior (i-k-1)

5.  if (E_{i-k,i-k-1}.status≠'TrueResponse'& //no se sabe si el evento es correcto

6.      i-k ≠ 1 & // no es el principio de la composición

7.      T_{Misbe_Flow} > T_{i-k,i-k-1} + ΔT_{Misbe_Flow-k} && //llegada de los eventos

8.       T_{TRUEi-k+1} > T_{i-k,i-k-1} + ΔT_{TRUEi-k} //llegada de los eventos

9.  then

10.     emit event(BE_{TRUE}, status ='S_{i-k+1}', T_{TRUE}) to S_{i-k} diagnoser //llamada otra vez
  a recursivo_Diagnosticador_i-k

11.   else if

12.      (E_{i-k, i-k-1}.status ='TrueResponse' OR // se sabe que es correcta es parada

13.      i-k = 1)& // es el principio de la coreografia se para

14.      T_{Misbe_Flow} > T_{i-k,i-k-1} + ΔT_{Misbe_Flow-k} && //llegada de los eventos l final
  diagnosticador

15.      T_{TRUEi-k+1} > T_{i-k,i-k-1} + ΔT_{TRUEi-k} //llegada de los eventos al final
  diagnosticador

16.   then

17.     repair invoke(S_{i-k}, status='S_{i-k}', 'Misbehaving Execution Flow') // parada
  del algoritmo recursivo

18. }
```

Algoritmo 4.3: Algoritmo recursivo para detectar la falla de Mal comportamiento del flujo Ejecución en el diagnosticador i-k

A partir de eso, se definen dos tipos de crónicas en cada D_{i-k+1}, una llamada "Mal comportamiento del flujo Ejecución" (para hacer la llamada recursiva a la siguiente D_{i-k} generando el evento BE_{TRUE}) y otra llamada "Mal comportamiento del flujo Ejecución 1" (es la parada del proceso de reconocimiento, y llama al reparador).

Diagnoser D_{i-k}	**Diagnoser D_i**
Subchronicle S_{i-k} Mal comportamiento del flujo Ejecución { **Events{** event($E_{i-k,\ i-k-1}$, $T_{i-k,\ i-k-1}$) event(BE_{Misbe_Flow}, T_{vMisbe_Flow}) event(BE_{TRUE}, status ='S_{i-k+1}', $T_{TRUEi-k+1}$) **}** **Constrains{** $E_{i-k,\ i-k-1}$.status ≠'TrueResponse' i-k ≠ 1 $T_{Misbe\_Flow} > T_{i-k,i-k-1} + \Delta T_{Misbe\_Flowi-k}$ $T_{TRUEi-k+1} > T_{i-k,i-k-1} + \Delta T_{TRUEi-k}$ **}** **When recognized{** emit event(BE_{TRUE}, status ='S_{i-k}', $T_{TRUEi-k}$) to S_{i-k-1} diagnoser **}** **}**	**Subchronicle S_i Mal comportamiento del flujo Ejecución {** **Events{** event($E_{i,i-1}$, status:'BadResponse' $T_{i,i-1}$) **}** **Constrains{** **}** **When recognized{** emit event(BE_{Misbe_Flow}, status ='S_i', T_{Misbe_Flow}) to S_i until S_{i-1} diagnoser emit event(BE_{TRUE}, status ='S_i', T_{TRUE}) to S_{i-1} diagnoser **}** **}**
Subchronicle S_{i-k} Mal comportamiento del flujo Ejecución 1 { **Events{** event($E_{i-k,\ i-k-1}$, $T_{i-k,\ i-k-1}$) event(BE_{Misbe_Flow}, T_{vMisbe_Flow}) event(BE_{TRUE}, status ='S_{i-k+1}', $T_{TRUEi-k+1}$) **}** **Constrains{** ($E_{i-k,\ i-k-1}$.status ='TrueResponse' OR i-k = 1) $T_{Misbe\_Flow} > T_{i-k,i-k-1} + \Delta T_{Misbe\_Flowi-k}$ $T_{TRUEi-k+1} > T_{i-k,i-k-1} + \Delta T_{TRUEi-k}$ **}** **When recognized{** repair invoke(S_{i-k}, status='S_{i-k}', 'Misbehaving Execution Flow') **}** **}**	

Figura 4.17: Patrón de Crónica Distribuida para fallas debido a un mal comportamiento del flujo Ejecución

donde:

$E_{i-k,i-k+1}$, …, $E_{i-1,i}$, $E_{i,i-1}$: Son los eventos generados en la aplicación SOA. El evento $E_{i,i-1}$ tiene el status = BadResponse indicando que hubo un problema en la

entrada suministrada en el servicio S_i. Los eventos se distribuyen entre los diagnosticadores previos al servicio con el status de BadResponse (D, ∀ j > 0 & j < i-1) y $E_{i,i-1}$ en el servicio S_i.

BE_{Misbe_Flow}: Es el evento enlazador generado cuando se reconoce la sub-crónica **"Mal comportamiento del flujo Ejecución"** en D_i (llega el evento $E_{i,i-1}$ con status de BadResponse).

BE_{TRUE}: Es el evento enlazador generado cuando se reconoce la sub-crónica **"Mal comportamiento del flujo Ejecución"** en D_j, ∀ j > 0 & j < i. El diagnosticador no tiene certeza que el servicio que tiene encargado para su diagnóstico funciona correctamente en el flujo de la aplicación (no contiene un evento con status TrueResponse), por lo que es necesario propagar el eventos enlazador al diagnosticador antecesor (el D_j genera un evento enlazador BE_{TRUE} hacia D_{j-1}).

$\Delta T_{Misbe\_Flowi-k}$: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_j, ∀ j > 0 & j < i-1, reciba el evento enlazador BE_{Misbe_Flow} desde el D_i. En caso de no considerar retardo se puede colocar en cero ($\Delta T_{Misbe\_Flowi-k} = 0$).

$\Delta T_{TRUEi-k}$: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_j, ∀ j > 0 & j < i reciba el evento enlazador BE_{TRUE} desde D_{j+1}. En caso de no considerar retardo se puede colocar en cero (**$\Delta T_{TRUEi-k} = 0$**).

Repair Invoke: Es la invocación del reparador S_{i-k} con el diagnóstico de la falla cuando la sub-crónica **"Mal comportamiento del flujo Ejecución 1"** es reconocida en D_{i-k}.

Para mostrar el reconocimiento de esta crónica, suponga nuevamente el

ejemplo de la Figura 2.14. El reconocimiento se muestra en la Figura 4.18, el diagnosticador D_4 (se considera el diagnosticador i, i = 4) recibe el evento E_6 en el instante T_6 con status de 'BadResponse' (E_6 corresponde al evento $E_{i,\ i-1}$ que sucede en S_i). Detectando un mal funcionamiento al reconocer la sub-crónica **"S_i Mal comportamiento del flujo Ejecución"**, y genera el evento enlazador BE_{Misbe_Flow} hacia los diagnosticadores previos D_1, D_2 y D_3 (D_j, $\forall$ j > 0 & j < i-1) y el BE_{TRUE} hacia el D_3. Con los eventos E_4 ($E_{i-2,i-1}$, viene de S_{i-2} y se recibe en S_{i-1}), BE_{TRUE} (generado por D_4) y BE_{Misbe_Flow} (generado por D_4), el diagnosticador D_3 (diagnosticador D_{i-1}) reconoce la sub-crónica **"S_{i-k} Mal comportamiento del flujo Ejecución"** (diagnosticador D_{i-k} en el patrón de crónicas de la Figura 4.17) y genera el evento BE_{TRUE} hacia el D_2. En el mismo orden de ideas, el diagnosticador D_2 con los eventos E_2 ($E_{i-k-1,\ i-k}$), BE_{Misbe_Flow} (generado por D_4) y BE_{TRUE} (generado por D_3) reconoce la sub-crónica **"S_{i-k} Mal comportamiento del flujo Ejecución"** (diagnosticador D_{i-k} en el patrón de crónicas de la Figura 4.17) y genera el evento BE_{TRUE} hacia D_1. Finalmente, el diagnosticador D_1 con los eventos E_1 (con status = 'TrueResponse'), BE_{Misbe_Flow} y BE_{TRUE} (generado por D_2) reconoce la sub-crónica **"S_{i-k} Mal comportamiento del flujo Ejecución 1"** (diagnosticador D_{i-k} en el patrón de crónicas de la Figura 4.17). la restricción $E_{i-k,\ i-k+1}$.status ='TrueResponse' se ha cumplido, aunque también podría ser reconocida por la restricción i-k = 1, al ser el ultimo servicio en la aplicación. Asi, reconoce la **falla debido a un mal comportamiento del flujo Ejecución (Misunderstood Behavior Flow)**.

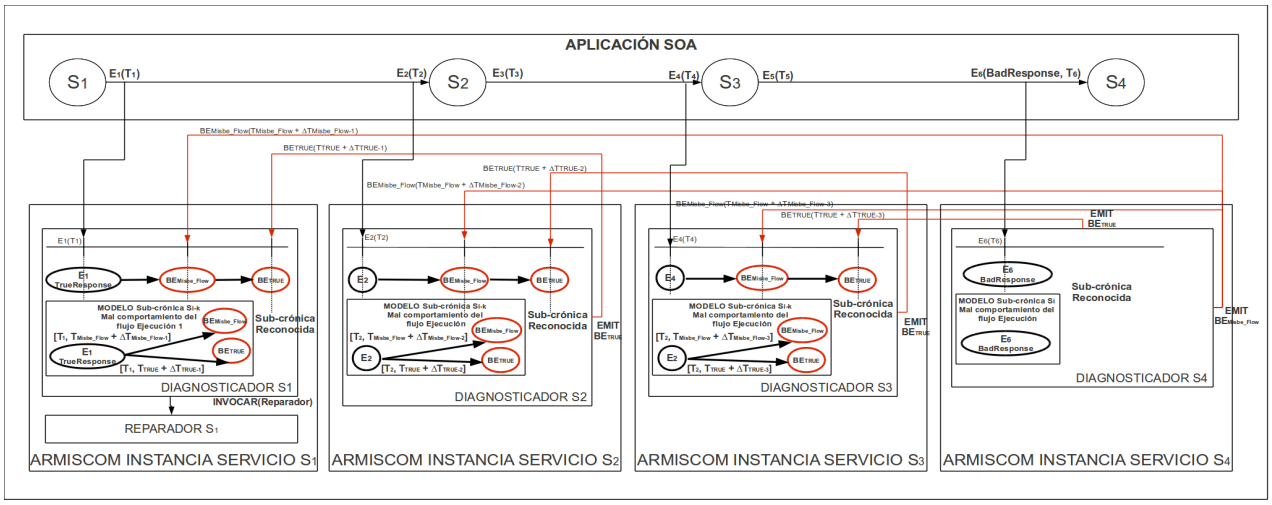


Figura 4.18: Reconocimiento del Patrón de Crónica para fallas de un mal comportamiento del flujo Ejecución

Adicionalmente, para mostrar la generalidad del patrón de crónicas propuesto en la Figura 4.17, la cual no solo reconoce cuando es el ultimo servicio en la composición, considere ahora que la aplicación mostrada en la Figura 2.14 suministra los eventos $E_1(T_1)$, $E_2(T_2)$, $E_3(T_3)$, E_4(status='TrueResponse', T_4), $E_5(T_5)$ y E_6(status='BadResponse', T_6). La ejecución de la crónica se muestra en la Figura 4.19, el diagnosticador D_4 recibe el evento E_6(status='BadResponse', T_6), reconoce la sub-crónica **"S_i Mal comportamiento del flujo Ejecución"** (diagnosticador D_i), y genera el evento enlazador BE_{Misbe_Flow} hacia los diagnosticadores D_1, D_2 y D_3 (D_j, $\forall$ j > 0 & j < i-1) y BE_{TRUE} hacia el D_3. D_3 con los eventos E_4(status='TrueResponse', T_4) y BE_{Misbe_Flow} reconoce la sub-crónica **"S_{i-k} Mal comportamiento del flujo Ejecución 1"** (diagnosticador D_{i-1}), logrando detectar la **falla debido a un mal comportamiento del flujo Ejecución (Misunderstood Behavior Flow)** e invocar su reparador con su diagnóstico.

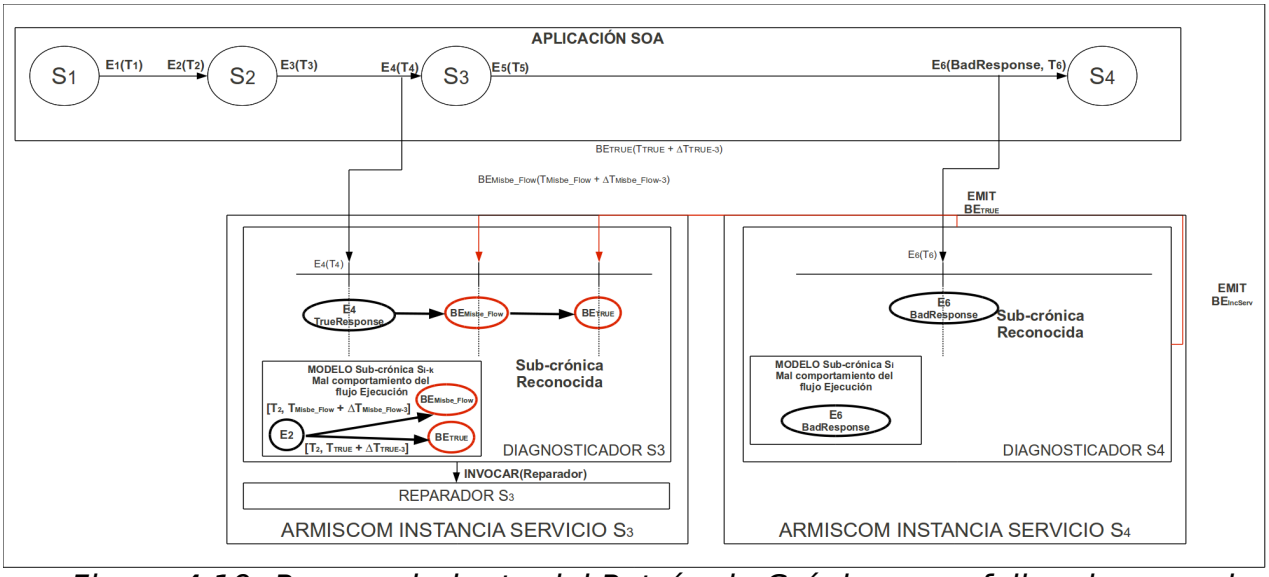


Figura 4.19: Reconocimiento del Patrón de Crónica para fallas de un mal comportamiento del flujo Ejecución 1

4.3.3.4 Falla de Servicio con un Mensaje de Error

Es cuando se realiza la invocación de un servicio dentro de la composición, y se produce una falla en su funcionamiento interno (genera un mensaje de error como salida). Para modelar la crónica de esta falla se consideran los eventos generados del ejemplo de la Figura 2.15, la distribución de estos por cada diagnosticador seria:

$$D_1 = \{E_1(T_1)\}$$
$$D_2 = \{E_2(T_2),\ E_3(status = 'error',\ T_3)\},\ \text{donde}\ T_3 > T_2. \quad (4.30)$$

El reconocimiento de los eventos $E_2(T_2)$ y $E_3(status = 'error', T_3)$ en el diagnosticador D_2 permite diagnosticar la falla. El patrón de la crónica en este caso no es distribuida, es una crónica simple que recibe los eventos E_2 en el instante T_2 y E_3 en T_3 con el mensaje "error". Generalizando, suponga que los eventos E_2 y E_3 suceden en el servicio S_j, el patrón de crónica para esta falla es la

siguiente:

Diagnoser D_j

```
Chronicle Sj Response Error {
  Events{
    event(E2, T2)
    event(E3, 'error', T3)
  }
  Constrains{
     T3 ≤ T2
  }
  When recognized{
    repair invoke(Sj, 'Response faults')
   }
}
```

Figura 4.20: Patrón de Crónica Distribuida para fallas debido Respuesta de Falla

El proceso de reconocimiento de patrones de esta falla utilizando las crónicas se muestra en la Figura 4.21: el diagnosticador D_j recibe el evento E_2 (invocación desde el servicio S_i con su contenido en un formato correcto y sin problemas), pero entonces recibe un evento E_3 local que indica un error en la respuesta del servicio. Entonces, el diagnosticador D_j reconoce la **falla debido Respuesta de Error (Response error)** e invoca el componente de reparación para corregir la falla.

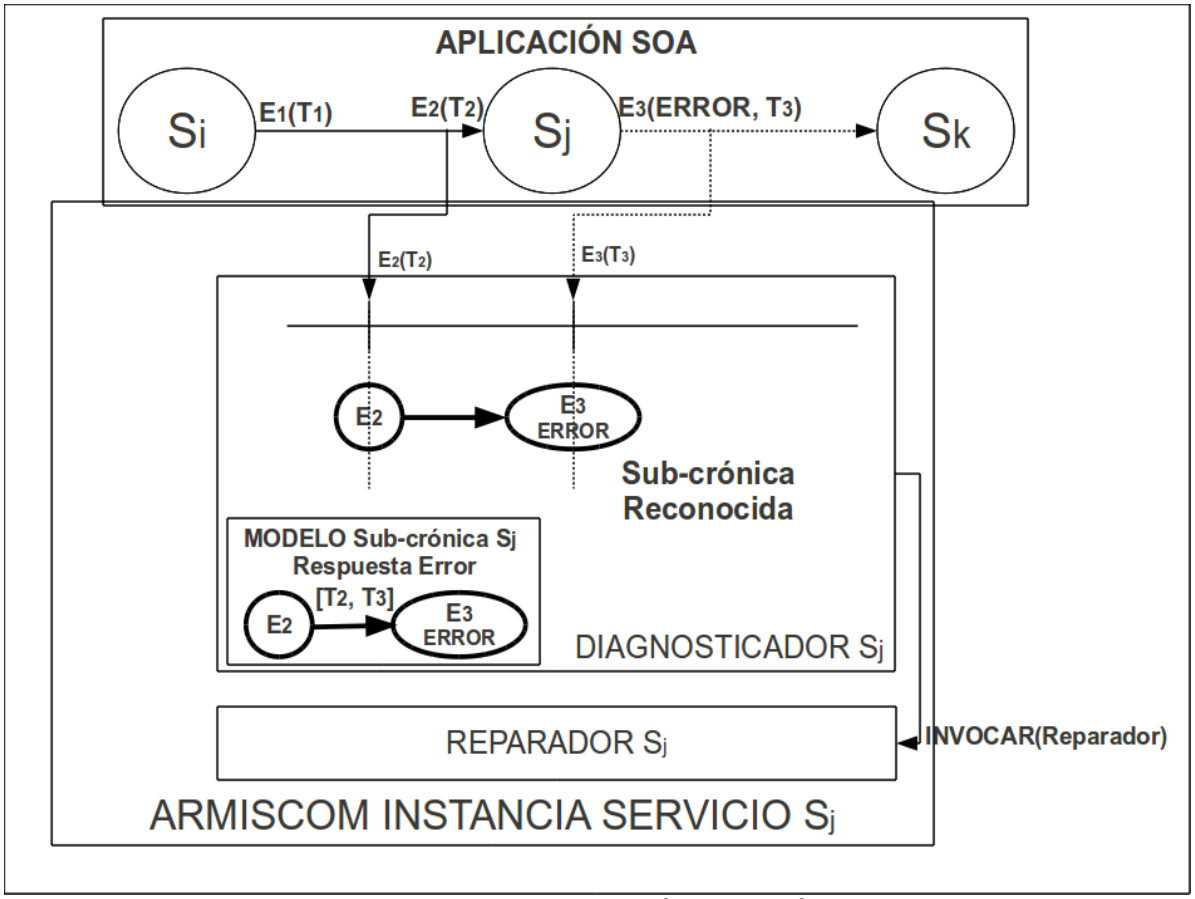


Figura 4.21: Reconocimiento del Patrón de Crónica para fallas debido Respuesta de Error

4.3.3.5 Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (QoS)

Esta falla se genera cuando alguna restricción de SLA y/o QoS han sido violados. Para modelar la crónica de esta falla se consideran los eventos generados del ejemplo de la Figura 2.16, la distribución de estos por cada diagnosticador es:

$$D_1 = \{E_1(T_1)\} \quad (4.31)$$

$D_2 = \{E_2(T_2), E_3(T_3)\}$, donde $T_3 > T_2$

$D_3 = \{E_4(T_4)\}$

Al estar distribuidos los eventos necesarios para realizar el diagnóstico de la falla, es necesario agregar un evento enlazador que permita al diagnosticador D_2 conocer la falla de violación en el evento $E_4(T_4)$. Para esto, se construye una sub-crónica en el diagnosticador D_3 llamada **"SLA OR QoS"**, que reconoce la presencia del evento $E_4(T_4)$ con alguna violación de las restricciones de las propiedades no funcionales del evento E_4 (la violación se detecta sobre atributos presentes en el evento E_4), y genera un evento enlazador $BE_{SLAORQoS}$ hacia el diagnosticador S_2. Los eventos recibidos finalmente por el diagnosticador D_2 son:

$$D_2 = \{E_2(T_2), E_3(T_3), BE_{SLAORQoS}(T_{SLAORQoS})\} \quad (4.32)$$

El reconocimiento de los eventos $E_2(T_2)$, $E_3(T_3)$ y $BE_{SLAORQoS}(T_{SLAORQoS})$ permite al diagnosticador D_2 el diagnóstico de la falla. Para esto, se construye una sub-crónica sub-crónica **"SLA OR QoS"** en D_2 que logre el reconocimiento de estos eventos, donde el evento $BE_{SLAORQoS}$ es un evento enlazador que sirve para conectar las dos sub-crónicas **"SLA OR QoS"** en los diagnosticadores D_2 y D_3. Para generalizar, suponga que los servicios involucrados son S_i, S_j y S_k, y se cuenta con una función llamada noFunctional compuesta por el conjunto de expresiones lógicas que permiten detectar la violación de las propiedades no funcionales del evento E_4 recibido por S_k (un ejemplo de la función noFunctional podría ser las clausulas: E4.attribute > 0 AND E4.attribute < n, para definir que el valor de la propiedad no funcional representada por el atributo attribute debe estar comprendida en el intervalo [0, n], donde n es un número entero mayor que 0), entonces el patrón de la crónica con las dos sub-crónicas para esta falla cuando el servicio S_j invoca al servicio S_k es:

Diagnoser D_j	**Diagnoser D_k**
Subchronicle S_j SLA OR QoS { **Events{** event(E_2, T_2) event(E_3, T_3) event($BE_{SLAORQoS}$, $T_{SLAORQoS}$) **}** **Constrains{** $T_3 > T_2$ $T_{SLAORQoS} > T_3 + \Delta T$ **}** **When recognized{** repair invoke(S_j, 'SLA OR QoS') **}** **}**	**Subchronicle S_k SLA OR QoS {** **Events{** event(E_4, T_4) **}** **Constrains{** noFunctional(E_4 attributes) **}** **When recognized{** emit event($BE_{SLAORQoS}$, $T_{SLAORQoS}$) to S_j diagnoser **}** **}**

Figura 4.22: Patrón de Crónica Distribuida para la falla de Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (QoS)

donde

E_2, E_3, E_4: Son los eventos generados en T_2, T_3 y T_4 por la aplicación SOA cuando se invocan los servicios S_j y S_k.

$BE_{SLAORQoS}$: Es el evento enlazador generado cuando se reconoce la sub-crónica en el diagnosticador D_k **"SLA OR QoS"**.

noFunctional: es una función que representa el conjunto de restricciones de SLA o QoS del servicio S_j, la cual es evaluada en base a un atributo que contiene el evento E_4 en T_4. Esa función regresa un valor lógico "verdad" cuando alguna de las restricciones de SLA o QoS son violadas, de tal manera de que la crónica se reconozca.

ΔT: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_j reciba el evento enlazador $BE_{SLAORQoS}$. En caso de no considerar retardo se puede colocar en cero ($\Delta T = 0$).

Repair Invoke: Es la invocación del reparador S_j con el diagnóstico de la falla cuando la sub-crónica **"SLA OR QoS "** es reconocida en D_j.

El proceso de reconocimiento de patrones de esta falla utilizando las crónicas se muestra en la Figura 4.23: el diagnosticador D_k recibe el evento E_4 en T_4 (invocación de S_k desde el servicio S_j) con una violación de una restricción de SLA O QoS, reconociendo la sub-crónica de **"SLA OR QoS"** y generando el evento enlazador $BE_{SLAORQoS}$ hacia D_j. Con el evento $BE_{SLAORQoS}$ y los eventos $E_2(T_2)$ y $E_3(T_3)$, el diagnosticador D_j reconoce la sub-crónica y emite un mensaje al reparador en S_j con el diagnóstico de falla: **"Violation of the Service Level Agreement o Quality of Service"**.

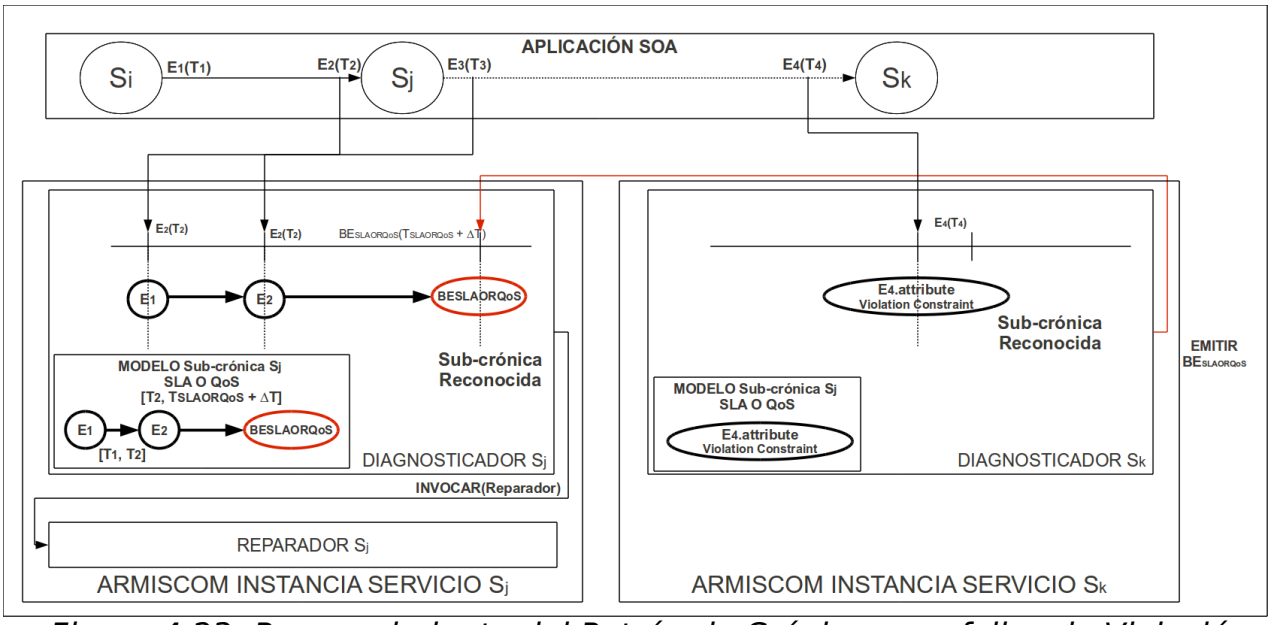


Figura 4.23: Reconocimiento del Patrón de Crónica para fallas de Violación delAcuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (QoS)

En algunas ocasiones, una violación se basa en frecuencias de sucesos (n cantidad de veces que un servicio debe repetir una violación para ser considerada una falla). Así que se puede usar esta idea, para indicar que la frecuencia de

ocurrencia de una violación de las restricciones (n cantidad de veces) en el intervalo [T_{11}, T_{12}], determina una falla. En la Figura 4.24 se muestra la crónica para ese caso, para detectar la frecuencia de ocurrencia de violaciones de SLAs y QoS.

Diagnoser D_j	**Diagnoser D_k**
Subchronicle S_j SLA OR QoS 1 { **Events{** event(E_2, T_2) event(E_3, T_3) occurs((n_1 , n_2), $BE_{SLAORQoS}$, (T_{11} , T_{12})) **}** **Constrains{** T_3 > T_2 T_{12} > T_3 T_{12} > T_{11} $n_2 \geq n_1$ $n_1 \geq 0$ **}** **When recognized{** repair invoke(S_j, 'SLA OR QoS') **}** **}**	**Subchronicle S_k SLA OR QoS 1 {** **Events{** event(E_4, T_4) **}** **Constrains{** noFunctional(E_4 attributes) **}** **When recognized{** emit event($BE_{SLAORQoS}$, $T_{SLAORQoS}$) to S_j diagnoser **}** **}**

Figura 4.24: Patrón de Crónica Distribuida para la falla de Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (QoS) 1

donde los parámetros de la Sub-crónica Sj SLA OR QoS son:

n_1 y n_2: representa el rango de la cantidad de veces que la violación ha sucedido ($n_1 \leq$ frecuencia $\leq n_2$). Note que si $n_1 = 0$ y $n_2 = 1$ la crónica resultante es similar a la mostrada en la Figura 4.22.

T_{11} y T_{21}: es el intervalo de tiempo en que se repiten la violación.

El reconocimiento de esta crónica distribuida se muestra en la Figura 4.25. El

proceso de reconocimiento de las sub-crónicas **"SLA OR QoS 1"** en D_j y D_k es similar al mostrado en la Figura 4.23, con la diferencia que cuando se produce su reconocimiento en D_k se genera un evento enlazador $BE_{SLAORQoS}$ hacia el diagnosticador D_j, repitiendo este proceso cada vez que suceda la violación de restricción, permitiendo poder reconocer la sub-crónica **"SLA OR QoS 1"** cuando la violación ha sucedido n_2 - n_1 veces en el intervalo $[T_{11}, T_{12}]$.

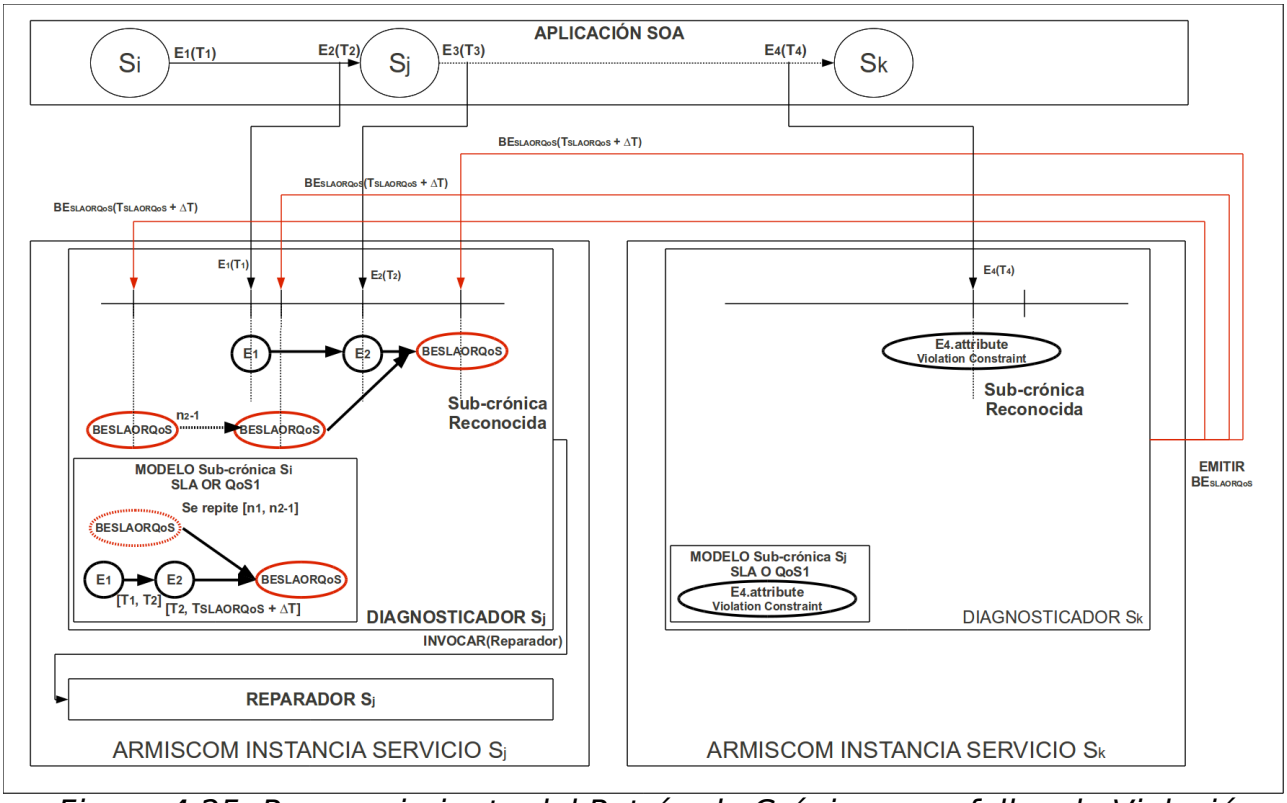


Figura 4.25: Reconocimiento del Patrón de Crónica para fallas de Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (QoS) 1

Es pertinente señalar que los dos crónicas mostradas anteriormente se pueden ampliar fácilmente para medir violaciones de QoS y SLA no solo en servicios, sino en secciones (sub-flujos) de la aplicación SOA (que involucran varios servicios). Para eso se tiene que añadir en la sub-crónica S_k SLA OR QoS de la Figura 4.22 y Figura 4.24 los eventos que son parte de la sección, y sus restricciones

temporales. Por lo tanto, si se quiere detectar violaciones de QoS o SLA en un sub-flujo (sección) de la composición, se tiene que volver a escribir la sub-crónica en S_k **"SLA OR QoS"** agregando eventos que forman parte del sub-flujo ($E_{4.1}$, $E_{4.2}$,..., $E_{4.z}$) y sus restricciones temporales, expresadas en una función llamada TemporalConstraint (compuesta por el conjunto de expresiones lógicas para expresar las restricciones de tiempo de los eventos del flujo, p.e. $T_{4.2} > T_{4.1}$), como sigue:

```
Subchronicle Sk SLA OR QoS {
 Events{
   event(E4.1, T4.1)
   event(E4.2, T4.2)
   .
   .
   event(E4.z, T4.z)
 }
 Constrains{
    TemporalConstraint (T4.1, T4.2, .., T4.z)
    noFunctional(E4.1, E4.2, ..,E4.z attributes)
 }
 When recognized{
   emit event(BESLAORQoS, TSLAORQoS) to Sj diagnoser
 }
}
```

Figura 4.26: Extensión del Patrón Crónica para la falla de Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (QoS) para estudiar un sub-flujo de la composición.

Donde:

$E_{4.z}$: son eventos que deberían ocurrir en el sub-flow seleccionado.

$T_{4.z}$: son el tiempo de ocurrencia de los eventos $E_{4.z}$.

TemporalConstraint: es una función que representa el conjunto de

restricciones temporales de los eventos del sub-flujo ($E_{4,z}$). Esa función regresa un valor lógico de verdad cuando todas las las restricciones temporales se cumplen.

Finalmente, los patrones de crónica de las 169 (Subchronicle Sj SLA OR QoS) y Figura 4.24 (Subchronicle Sj SLA OR QoS 1) se pueden extender para detectar violaciones de SLA en los atributos de los eventos en la sub-crónica (puede definir restricciones en los atributos de los eventos E_2 y E_3, para detectar cuando son violados).

4.3.3.6 Incorrecto Orden

La falla por Incorrecto Orden es debido a que los mensajes utilizados para interactuar con los servicios en la composición llegan en un orden diferente del tiempo esperado. Para modelar la crónica de esta falla se consideran los eventos generados del ejemplo de la Figura 2.17, la distribución de estos por cada diagnosticador seria:

$$\begin{aligned} D_1 &= \{E_1(T_1)\} \\ D_2 &= \{E_2(T_2),\ E_4(T_4),\ E_5(T_5)\},\ \text{donde}\ T_5 > T_4,\ T_5 > T_2 \\ D_3 &= \{E_3(T_3)\} \end{aligned} \tag{4.33}$$

El evento E_5 depende de los eventos E_2 y E_4, si E_2 y E_4 no llegan antes de que el evento E_5 suceda ($T_4 > T_5$ o $T_2 > T_5$), entonces la aplicación SOA está en la presencia de esa falla. Para realizar la construcción del patrón de crónicas se consideran solo el orden incorrecto del servicio S_1 (se considera que E_4 llega en un orden correcto), entonces la llegada del evento $E_2(T_2)$ en el diagnosticador D_2 después de $E_5(T_5)$ es necesaria propagarla al diagnosticador D_1 con un evento enlazador llamado BE_{IncOrd}. Además, se construye una sub-crónica en el diagnosticador D_2 llamada **"Incorrecto Orden S_1"** que reconoce el orden de los evento $E_2(T_2)$ y $E_5(T_5)$, la cual genera el evento enlazador BE_{IncOrd} hacia el

diagnosticador D_1 cuando es reconocida. Los eventos recibidos finalmente por el diagnosticador D_1 son:

$$D_1 = \{E_1(T_1),\ BE_{IncOrd}(T_{IncOrd})\},\ \text{donde}\ T_{IncOrd} > E_1(T_1) \quad (4.34)$$

El reconocimiento de los eventos $E_1(T_1)$ y $BE_{IncOrd}(T_{IncOrd})$ en el diagnosticador D_1 permite diagnosticar la falla de orden incorrecto. En este mismo orden de ideas, ahora suponga que es el evento $E_4(T_4)$ que llega en el diagnosticador D_2 después de $E_5(T_5)$, provocando la falla de orden incorrecto. Para reconocer este síntoma se construye otra crónica en D_2 llamada **"Incorrecto Orden S_3"** que genera un evento enlazador llamado BE_{IncOrd} cuando el evento $E_4(T_4)$ llega después del evento $E_5(T_5)$. Los eventos recibidos finalmente por el diagnosticador D_3 son:

$$D_3 = \{E_3(T_3),\ BE_{IncOrd}(T_{IncOrd})\},\ \text{donde}\ T_{IncOrd} > E_3(T_3) \quad (4.35)$$

El reconocimiento de los eventos $E_3(T_3)$ y $BE_{IncOrd}(T_{IncOrd})$ en el diagnosticador D_3 permite diagnosticar la falla de orden incorrecto en ese sitio. Las 2 subcrónicas **"Incorrecto Orden S_1"** y **"Incorrecto Orden S_3"** permiten reconocer de forma distribuida e independiente, el orden incorrecto de ejecución de los servicios S_1, S_2 y S_3. Así, para generalizar suponga que se tiene el servicio S_j, el cual es dependiente de los eventos $E_{j,i}$ generados por los n servicios S_i, donde i = 1, n servicios[21]. La crónica distribuida para reconocer la llegada posterior de cualquier evento $E_{i,j}$ en D_j es:

21 El evento $E_{j,i}$ corresponde a un evento en el servicio j que depende del servicio i

Subchronicle S_i Incorrecto Orden { **Events{** event(E_i, T_i) event(BE_{IncOrd}, T_{IncOrd}) **}** **Constrains{** ....T_{IncOrd} > T_i + ΔT **}** **When recognized{** repair invoke(S_i, 'Incorrect Order') **}** **}**	**Subchronicle S_j Incorrecto Orden {** **Events{** event(E_{jSi}, T_{jSi}) event(E_j, T_j) **}** **Constrains{** T_{jsi} > T_j **}** **When recognized{** emit event(BE_{IncOrd}, T_{IncOrd}) to S_i diagnoser **}** **}**

Figura 4.27: Patrón de Crónica Distribuida para la falla Incorrecto Orden en S_i

donde

E_i: Son los eventos generados en T_i por la aplicación SOA en el servicio S_i, que indican que el orden de ejecución en ese sitio es correcto.

E_{jsi}, E_j: Son los eventos del servicio S_j, donde el evento E_{jsi} es un evento que se genera en el servicio j cuando llega al diagnosticador D_j un evento de S_i. Se diagnóstica la falla porque E_{jsi} llega después del evento E_j ($T_{jsi} > T_j$).

BE_{IncOrd}: Es el evento enlazador generado cuando se reconoce la sub-crónica del diagnosticador D_j **"Incorrecto Orden S_i"**.

ΔT: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_i reciba el evento enlazador BE_{IncOrd}. En caso de no considerar retardo se puede colocar en cero (ΔT = 0).

Repair Invoke: Es la invocación del reparador S_i con el diagnóstico de la falla cuando la sub-crónica **"Incorrecto Orden S_i"** es reconocida en D_i.

El proceso de reconocimiento de patrones de esta falla utilizando las crónicas, se muestra en la Figura 4.28: el diagnosticador D_j recibe el evento E_j en T_j y luego E_{si} en T_{si} (por ejemplo, podría ser la invocación de S_j después de su ejecución). Al estar en un orden incorrecto el evento E_{si}, se reconoce la sub-crónica **"Incorrecto Orden S_i"** y se genera el evento enlazador BE_{IncOrd} hacia D_i. Con los eventos recibidos de la ejecución de S_i (E_{i1},.., E_{ik}) y el evento enlazador BE_{IncOrd}, al diagnosticador D_i le es posible reconocer la sub-crónica de **"Incorrecto Orden S_i"**, e invocar su reparador con la descripción de la falla.

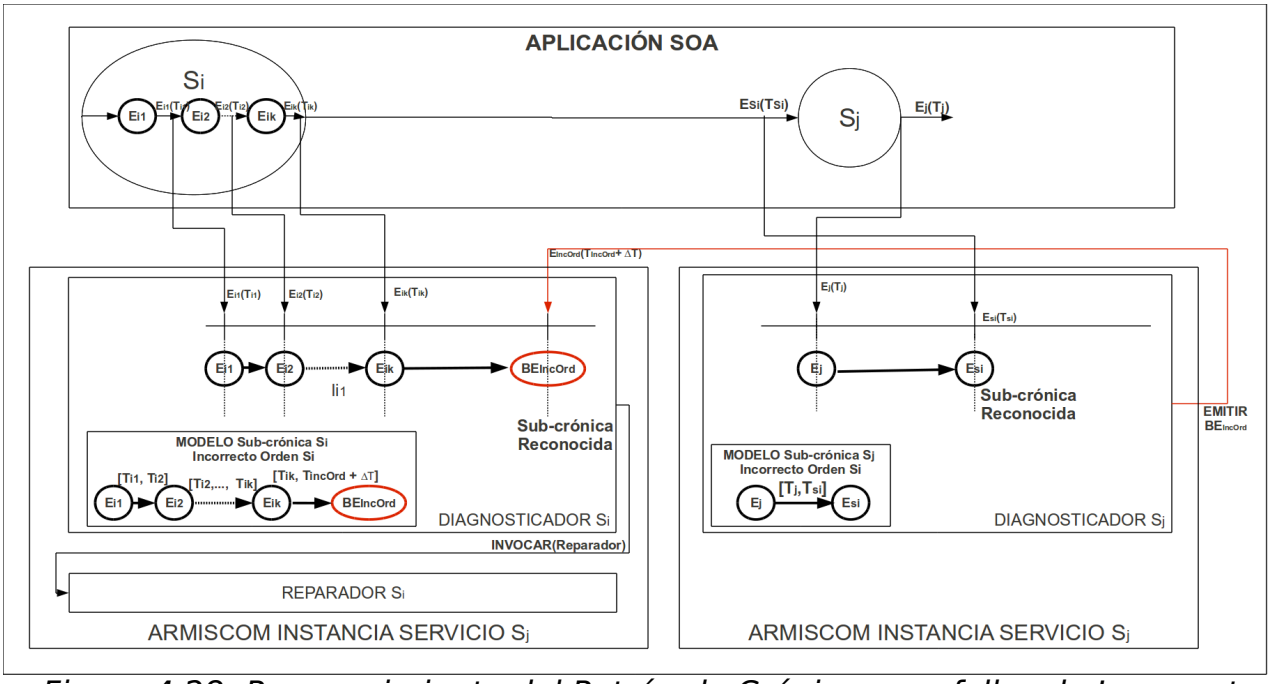


Figura 4.28: Reconocimiento del Patrón de Crónica para fallas de Incorrecto Orden en S_i

4.3.3.7 Fuera de Tiempo (Time-out)

La secuencia de eventos de esta falla se muestra en la Figura 2.18: suponga que los servicios S_1, S_2 y S_3 forman parte de la composición, el evento E_1 ocurre en S_1 en el instante T_1 (el servicio S_2 es invocado), entonces el evento E_2 ocurre (S_2

recibe el requerimiento) en T_2, finalmente, no se recibe ninguna respuesta durante un periodo de tiempo del servicio S_3. Para modelar la crónica de esta falla se consideran los eventos generados del ejemplo de la Figura 2.18, la distribución de estos por cada diagnosticador es:

$$D_1 = \{E_1(T_1)\}$$
$$D_2 = \{E_2(T_2)\} \quad (4.36)$$
$$D_3 = \{\emptyset\}$$

La llegada del evento $E_1(T_1)$ en el diagnosticador D_1 es necesaria propagarla al diagnosticador D_3 para permitirle conocer que la invocación de la aplicación ha comenzado, y se ha invocado un servicio que se encuentra previa a su ejecución. Para esto, el middleware propaga hacia el diagnosticador D_3 el evento $E_2(T_2)$ interceptando los mensajes de la aplicación SOA, después de una espera de ΔT_{espera} sin que la aplicación avance (eso lo puede hacer el middleware a partir de la observación que hace sobre la composición de servicios que guarda de la aplicación SOA supervisada). En caso contrario, es necesario la construcción de una sub-crónica en el diagnosticador D_1 que genere un evento enlazador para notificarle a D_3 que se invoco el servicio S_2, lo cual generaría un gran volumen de mensajes). Los eventos recibidos finalmente por el diagnosticador D_3 son:

$$D_3 = \{INVE_2(T_{INV2}+\Delta T_{espera})\} \quad (4.37)$$

Con el evento $INVE_2$ y al no recibir un evento $E_4(T_4)$ invocándolo desde S_2, en D_3 se puede construir la subcrónica Time-out, que al ser reconocida genera un evento enlazador $BE_{timeout}$ hacia D_2, informando del fuera de tiempo (Time-out) en la invocación. Los eventos recibidos finalmente en el diagnosticador D_2 son:

$$D_2 = \{E_2(T_2),\ BE_{timeout}(T_{timeout})\},\ \text{donde}\ T_{timeout} > T_2\} \quad (4.38)$$

Para generalizar, suponga que los servicios involucrados son S_i, S_j S_k (i = 1, j = 2, k = 3), el patrón de la crónica con las dos sub-crónicas que se requieren para diagnosticar esta falla es (en S_i no hay crónica, porque la generación del evento $INVE_2$ es generada internamente por el middleware, después de una espera de ΔT_{espera} sin que avance la aplicación SOA supervisada por él):

Subchronicle S_j Fuera de Tiempo { **Events{** event(E_2, T_2) event($BE_{Timeout}$, $T_{Timeout}$) **}** **Constrains{** $T_{Timeout} > T_2 + \Delta T$ **}** **When recognized{** repair invoke(S_j, 'Time-out') **}** **}**	**Subchronicle S_k Fuera de Tiempo {** **Events{** event($INVE_2$, $T_{INV2}+\Delta T_{espera}$) noevent($E_4$, ($T_{INV2} - \Delta T_{Timeout}$, $T_{INV2} + \Delta T_{Timeout}$)) **}** **Constrains{** **}** **When recognized{** emit event($BE_{Timeout}$, $T_{Timeout}$) to S_j diagnoser **}** **}**

Figura 4.29: Patrón de Crónica Distribuida para la falla Fuera de Tiempo

donde

E_1, E_2, E_3: Son los eventos generados en T_1, T_2, T_3 por la aplicación SOA cuando se invocan los servicios.

$BE_{timeout}$: Es el evento enlazador generado cuando se reconoce la sub-crónica "Fuera de Tiempo " en D_k.

$\Delta T_{timeout}$: Considera el tiempo que el diagnosticador D_j debe esperar para considerar que el evento E_4 se encuentra fuera de tiempo.

ΔT_{espera}: tiempo de espera del middleware para determinar que hay problemas

con la ejecución de la aplicación SOA supervisada.

ΔT: Considera el tiempo de retardo inherente en las comunicaciones para que el diagnosticador D_i reciba el evento enlazador $BE_{Timeout}$. En caso de no considerar retardo se puede colocar en cero ($\Delta T = 0$).

Repair Invoke: Es la invocación del reparador S_j con el diagnóstico de la falla cuando la sub-crónica **"Fuera de Tiempo"** es reconocida en D_j.

Note que la sub-crónica en D_j no toma en consideración el evento $E_3(T_3)$, ya que el diagnosticador D_k ha considerado que se ha incurrido en un fuera de tiempo al recibir el evento enlazador $BE_{Timeout}$.

El proceso de reconocimiento de patrones de esta falla utilizando las crónicas, se muestra en la Figura 4.30: el diagnosticador D_k recibe el evento $INVE_2$ (invocación del servicio S2 desde el servicio S_1) después que el middleware espera ΔT_{espera} porque avance la aplicación supervisada, y al no recibir el evento E_4 en el intervalo $[T_{NV2} - \Delta T_{Timeout}, T_{INV2} + \Delta T_{timeout}]$, el diagnosticador D_k reconoce la sub-crónica y genera el evento enlazador $BE_{Tiemout}$ hacia el diagnosticador D_j. Finalmente, el diagnosticador D_j recibe el evento $E_2(T_2)$ de la aplicación SOA, y con la llegada del evento $BE_{Timeout,}$ le es posible reconocer la sub-crónica y reconocer la falla.

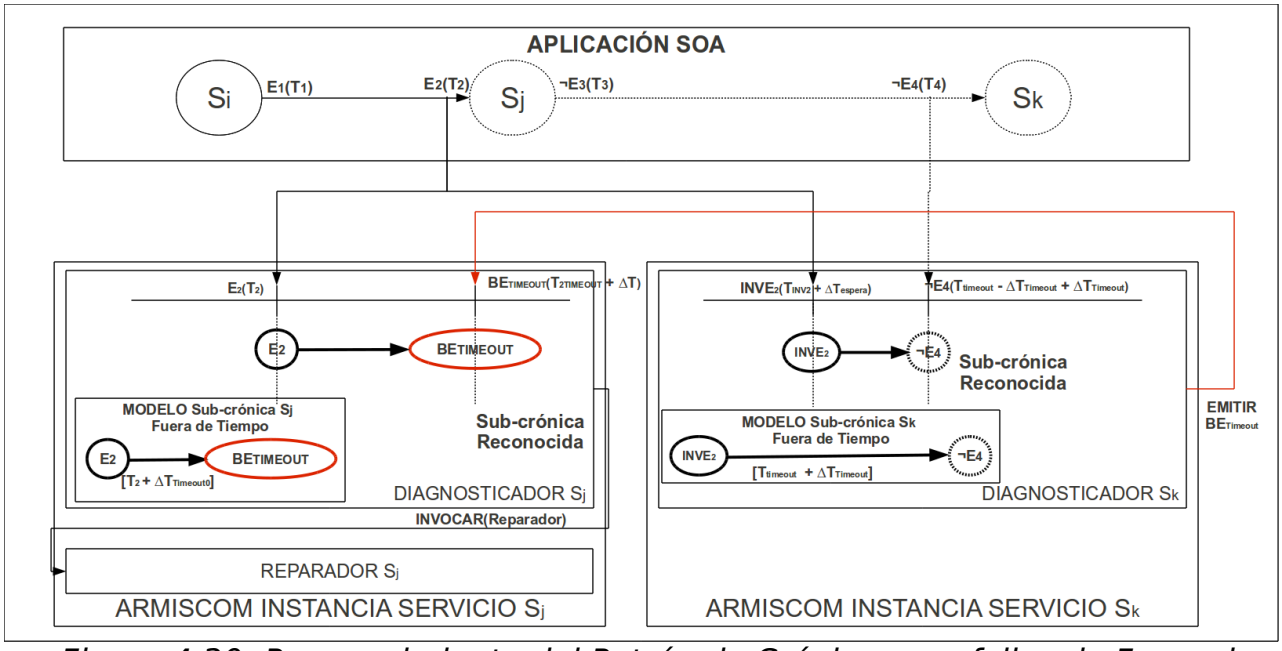


Figura 4.30: Reconocimiento del Patrón de Crónica para fallas de Fuera de Tiempo

En este ejemplo, se ve que la capacidad reflexiva del middleware le permite resolver el reconocimiento de esta falla sin la necesidad de usar muchos mensajes, solo esperando un tiempo dado ΔT_{espera} para determinar que algo esta pasando, dejándole a las crónicas la tarea de definir que esta ocurriendo. En este caso, el usar ΔT_{espera} permite determinar que hay restricciones que no se cumplen dentro de la aplicación SOA, al no recibir el evento E_4 en el tiempo previsto.

4.4. Conclusiones

Para apoyar la arquitectura propuesta de ARMISCOM, en este Capítulo se propuso un mecanismo distribuido basado en crónicas, que permite repartir el reconocimiento de las posibles fallas en las aplicaciones SOA, lo que favorece su aplicación en sistemas con gran número de servicios. Para esto, se ha ampliado el formalismo de crónicas, introduciendo el concepto de sub-crónicas, eventos

enlazadores (binding events), etc. Además, se ha descrito el proceso de reconocimiento del modelo de crónica totalmente distribuida.

Las crónicas distribuida permiten el reconocimiento distribuido de situaciones anormales (fallas) usando el conjunto de eventos observables en el sistema. Para el caso de fallas en la composición de servicios web, se han desarrollado un conjunto de patrones temporales basados en crónicas distribuidas para lograr la detección de las fallas definidos en la taxonomía propuesta en [10], teniendo en cuenta los eventos que son las causas de las fallas y de sus consecuencias (propagación de falla).

La extensión del formalismo de las crónicas y la distribución de los CRS entre los servicios que forman parte de la composición, facilita la interacción de los diagnosticadores locales, haciendo posible el reconocimiento de la crónica global sin necesitar de un coordinador para gestionar sus interacciones. A nivel de comunicación, esto representa una mejora notable sobre los mecanismos mostrados en otros estudios [12, 71], al procesar los eventos generados en él diagnosticador de cada componente y no en un diagnosticador central (ver la tabla Tabla 4.1, donde se especifican los eventos que deben ser intercambiados entre los componentes en cada sistema, siendo mucho menor para el caso de nuestra propuesta). Además, al ser un enfoque distribuido, el problema de escalabilidad de la aplicación distribuida puede ser manejada adecuadamente por el mecanismo propuesto en este trabajo.

Para comparar el mecanismo de reconocimiento de crónicas distribuidos con las propuestas en [12, 31, 71], se calculan teóricamente la cantidad de eventos de intercambio entre las diferentes diagnosticadores, y la cantidad de eventos procesados para el reconocimiento global de la crónica. Para ello, se supone que la crónica global se compone de m eventos distribuidos entre n diagnosticadores

(cada diagnosticador tiene en promedio m/n eventos que son necesarios para el reconocimiento de sus crónicas), donde m ≥ n. La Tabla 4.1 muestra el cálculo de ambas métricas en las diferentes propuestas, mostrando que el mecanismo de crónicas propuesto en este trabajo requiere sólo intercambiar n - 1 eventos para el reconocimiento de una crónica, a diferencia de las otras que requieren más eventos. Por otra parte, la arquitectura tiene que procesar menos eventos para el diagnóstico global (m + n - 1 eventos), contra los eventos procesados por las otras arquitecturas. La razón de que el mecanismo de reconocimiento requiere intercambiar menos eventos, y procesa menos eventos, es porque en la arquitectura el cálculo se distribuye totalmente, minimizando la cantidad de eventos necesarios para el diagnóstico global, y por lo tanto, el número de eventos a procesar. En ese sentido, es más escalable, e ideal para el diagnóstico en sistema distribuido con un gran número de componentes.

Arquitectura	**Eventos Intercambiados**	**Eventos Procesados** **(Restricciones Globales + Restricciones Locales)**
CarDeCRS [12] y Matrac [71]	$n * \left(\frac{m}{n}\right) = m$	$\left(n * \left(\frac{m}{n}\right)\right) + (m) = 2\mathrm{m}$
Distributed Architecture "Event Spread" [31]	$(n-1) * \left(\frac{m}{n}\right)$	$(m) + \left((n-1) * \left(\frac{m}{n}\right)\right) = \frac{(m * (2\mathrm{n}-1))}{n}$
Propuesta de Crónicas distribuidas	$n - 1$	$\left(\left(\frac{m}{n}\right) + (n-1)\right) + \left((n-1) * \left(\frac{m}{n}\right)\right) = m + n - 1$

Tabla 4.1: Comparando los mecanismos de reconocimiento de Crónicas.

Capítulo 5
Implementación de ARMISCOM

En los Capítulos previos se ha propuesto una arquitectura distribuida para el diagnóstico de fallas en la composición de servicios, en el que el diagnóstico de las fallas se realiza a través de la interacción de diagnosticadores presentes en cada servicio. Del mismo modo, las estrategias de reparación se desarrollan a través de un consenso entre los mecanismos de reparación distribuidos entre los servicios. Por otro lado, OpenESB es una arquitectura orientada a servicios utilizados para construir aplicaciones SOA [64], y en particular, apoyar en sus ejecuciones. En esta sección se presenta la implementación de ARMISCOM usando OpenESB, tal que cada componente de su gestor autonómico es implementado como un servicio web. Adicionalmente, se presenta la implementación del componente de gestión del conocimiento de ARMISCOM usando el lenguaje CQL y Protege [66], en particular, el motor de inferencia implementado en él (FACT++).

5.1. ARMISCOM en OpenESB

ARMISCOM, esta diseñado como un middleware reflexivo distribuido, que permite la gestión de fallas en la composición de servicios. Para la implementación de ARMISCOM, se seleccionó un middleware como punto de partida. En particular, un ESB es un tipo de middleware que hace posible la integración de servicios, haciendo posible contar con una arquitectura distribuida que funciona en ambientes heterogéneos, para desplegar aplicaciones SOA. Actualmente existen una gran cantidad de ESB basados en software libre, como Mule ESB [79], Apache ServiceMix [80], openESB [64], entre otros. Sin embargo, solo OpenESB cuenta con un componente llamado IEP SE (más adelante se

describe), que permite procesar eventos complejos, base fundamental del motor de inferencia para el reconocimiento de las crónicas distribuidas.

En general, OpenESB es una arquitectura que permite el despliegue de aplicaciones basadas en SOA, facilitando el desarrollo de aplicaciones flexibles con bajo acoplamiento [65]. Es “Open Source”, y esta implementado conforme a la especificación JSR208[22]. Particularmente, OpenESB esta compuesto por los siguientes componentes:

- **Marco de Trabajo (Framework):** Implementa un bus virtual conocido como Normalised Message Router (NMR), que permite el envío de mensajes entre servicios débilmente acoplados. NRM es responsable de tomar los mensajes desde los clientes y enrutarlos hacia los “Service Engines” apropiados para ser procesados, logrando proveer un modelo flexible en el cual un “Service Engine” no tiene conocimiento de la existencia de otros “Services Engines”.

- **Componentes** basados en JBI[23] de 2 tipos:

  - ***Motores de servicio (Service Engines - SE):*** son componentes que llevan a cabo la lógica de negocio de los clientes. Permite recibir y enviar mensajes sobre el bus, y esta compuesto por:

    - *BPEL SE:* permite integrar diferentes servicios en una sola aplicación. Es usado para ejecutar procesos de negocio en WS-BPEL 2.0[24] (Web Services Business Process Execution Language). Típicamente, se

22 Java Specification Request (JSR), dirección: http://www.jcp.org/en/jsr/detail?id=208

23 JBI es un conjunto de componentes para la comunicación asíncrona de servicios web vía a Normalized Message Router (NMR).

24 WS-BPEL es un lenguaje basado en XML usado para programar procesos de negocios.

encarga del intercambio de mensajes entre una aplicación y los servicios invocados por ella, conocidos como “partner services”.

- *XSLT SE:* permite transformar un documento XML en otro usando el XSL stylesheets.
- *IEP SE:* permite el procesamiento complejo de eventos basado en el lenguaje de consulta CQL. Consiste en un procesador que realiza la búsqueda de patrones de eventos usando el lenguaje CQL. El IEP permite editar, desplegar y ejecutar el procesador de eventos, y puede trabajar sobre múltiples flujos de eventos en tiempo real.
- *POJO SE:* transforma cualquier clase escrita en el lenguaje de programación JAVA en un componente OpenESB que puede ser invocado como un servicio.
- *Sun Java EE SE:* transforma cualquier EJB o Servlet en un componente OpenESB que puede ser invocado como un servicio.

◦ ***Componentes de vinculación (Binding Components - BC):*** se usan para enviar y recibir mensajes desde el exterior hacia el bus usando protocolos particulares. Corresponde al componente de **"gestión de protocolos"** para Buses de Servicios descrito en el Capítulo 2. Gestiona una gran cantidad de protocolos como SOAP, FTP, entre otros. Aísla al entorno JBI del ambiente exterior heterogéneo, normalizando[25] los mensajes entrantes y desnormalizando los mensajes salientes. Para ello, se compone de:

25 Un mensaje normalizado se encuentra en un formato que pueda ser entendido por OpenESB internamente (formato en que se manejan los mensajes dentro del BUS). Por el contrario, un mensaje des-normalizado se encuentra en otro formato externo que necesita traducción, como SOAP, JMS. entre otros.

- *File BC:* permite la comunicación del componente JBI con el sistema de archivos, haciendo posible obtener y enviar información a archivos que se encuentran accesibles localmente.
- *SOAP BC:* permite a los mensajes ser enviados o recibidos usando el protocolo SOAP sobre HTTP y HTTPS.
- *FTP BC:* permite al componente JBI recibir y enviar mensajes usando el protocolo FTP.
- *SMTP BC:* permite al componente JBI recibir y enviar mensajes usando el protocolo de correo electrónico (Mensajes SMTP).
- *JMS BC:* permite a las aplicaciones comunicarse usando Java Message Service (JMS).

- **Servidor de Aplicaciones:** OpenESB utiliza como servidor de aplicaciones el GlassFish[26].
- **Entorno de desarrollo integrado:** Provee un Ambiente de desarrollo integrado (IDE: Integrated Development Environment) que facilita el desarrollo de aplicaciones, compuesto por: un editor XSD (permite describir gráficamente las estructuras de datos utilizadas, y sus restricciones, en los mensajes intercambiados por los servicios), un editor WSDL (permite de forma gráfica realizar cambios en los documentos WSDL que definen los servicios), un editorBPEL (permite gráficamente desarrollar composiciones de servicios), entre otros

En la Figura 5.1 se muestra la arquitectura implementada por

26 GlassFish es un servidor de aplicaciones de software libre desarrollado por Sun Microsystems

openESB/Glassfish.

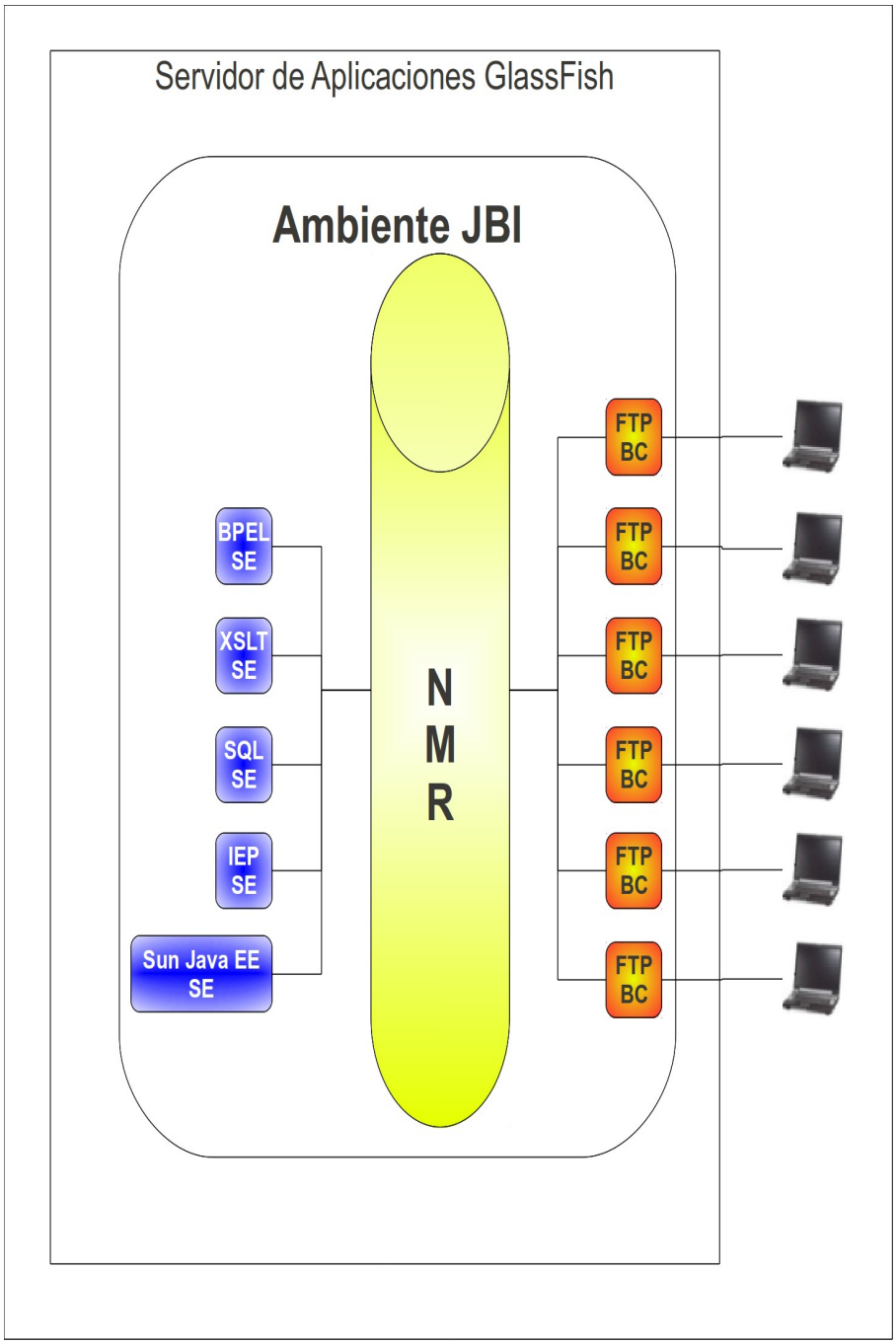


Figura 5.1: Arquitectura OpenESB - Glassfish

En el caso de ARMISCOM, el desarrollo de los componentes que conforman su nivel meta como una composición de servicios facilita su implementación en entornos distribuidos heterogéneos, y al ser débilmente acoplados, permite el desarrollo de los componentes de manera independiente (reusabilidad). Esto último también facilita sus usos en otras aplicaciones no desplegadas en el middleware. A continuación se presenta la implementación de ARMISCOM en OpenESB.

La Figura 5.2 muestra la implementación de ARMISCOM con OpenESB. El nivel meta de ARMISCOM es una composición de servicios gestionada por OpenESB, compuesta por los servicios Diagnosticador, Reparador y las Fuentes de Conocimiento, que corresponden a la arquitectura MAPE del gestor autonómico. El gestor se encuentra distribuido en cada servicio de la aplicación SOA , recibiendo los mensajes intercambiados en la aplicación SOA en forma de eventos (introspección de las entidades en el nivel base), que son provistos por OpenESB, y los eventos enlazadores suministrados por los otros gestores del middleware. ARMISCOM realiza la intersección, realizando la reparación de la aplicación SOA que se encuentra en el nivel Base. La composición de los servicios que forman parte del gestor autonómico funciona de la siguiente manera:

(I) El servicio diagnosticador recibe como entrada los mensajes intercambiados por la aplicación SOA del nivel Base (introspección).

(II) Cuando se produce el reconocimiento parcial de una crónica en un diagnosticador de uno de los servicios de la aplicación SOA bajo supervisión, se produce un evento enlazador, que es usado como entrada por otro servicio Diagnosticador de uno de los servicios de la aplicación SOA.

(III) Cuando uno de los servicios Diagnosticador reconoce una falla en la

aplicación, construye el diagnóstico de la falla en un archivo, e invoca al servicio reparador ubicado en ese sitio.

(IV) El servicio reparador determina los posibles métodos de reparación para solventar la falla en la aplicación SOA. Finalmente, genera el plan de reparación y lo ejecuta, modificando así la aplicación SOA para solventar la falla (intersección).

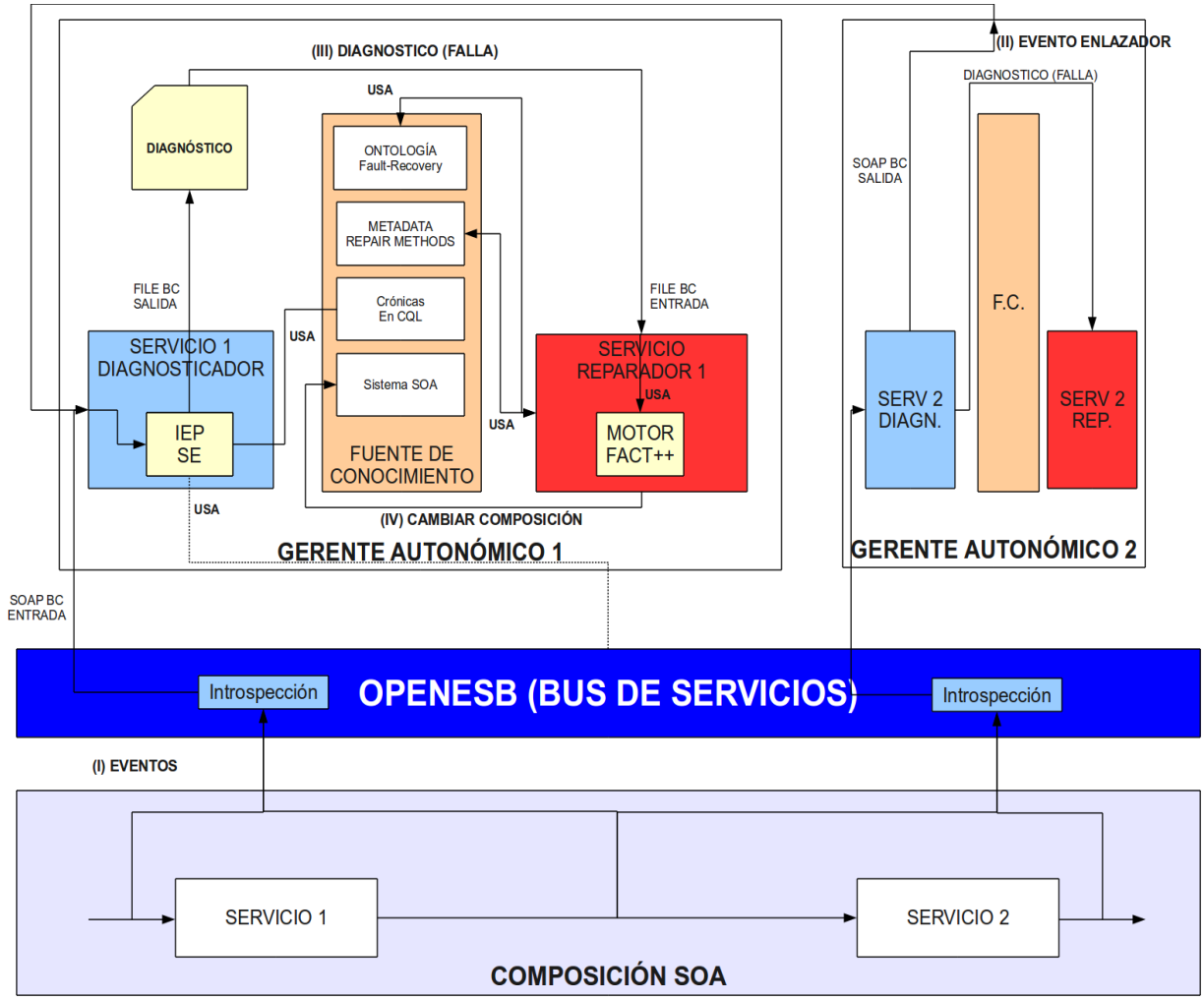


Figura 5.2: ARMISCOM implementado en OpenESB

5.1.1. El componente Diagnosticador

La tarea principal del componente Diagnosticador es realizar el diagnóstico de situaciones anormales que acontecen en la composición de los distintos servicios presentes en una aplicación. Para ello, el Diagnosticador es el encargado de verificar el conjunto de llamadas y respuesta de los servicios, los cuales serán las fuentes de eventos que utilizarán las crónicas para el reconocimiento de los patrones de fallas (ver). Así, el Diagnosticador usa el paradigma de crónicas para caracterizar los patrones de fallas a detectar y para reconocerlos. En particular, usa la extensión del formalismo de crónicas presentado en el Capítulo 4, tal que son definidos un conjunto de patrones distribuidos para el diagnóstico de fallas en la composición de servicios.

Actualmente, el CRS (Chronicle Recognition System) usado comúnmente para reconocer crónicas fue desarrollado por Dousson [10, 32], y dos extensiones, llamadas CarDeCRS [12] y Matrac [71], fueron desarrolladas para permitir el reconocimiento semi-distribuido de crónicas. Las herramientas CRS, CarDeCRS y Matrac, utilizan el lenguaje CRS, que no permite el uso de operadores matemáticos para definir las restricciones de las variables atemporales [33].

Por otro lado, OpenESB cuenta dentro de sus motores de servicio (Service Engines-SE) con un procesador inteligente de eventos, llamado (IEP), con un enfoque para el procesamiento de eventos complejos (CEP) basado en el lenguaje continuo de consulta (CQL) [67, 69], el cual es un lenguaje declarativo utilizado para realizar consultas continuas en un flujo de eventos. La implementación del Diagnosticador se realiza utilizando el lenguaje CQL.

El servicio Diagnosticador cuenta con una interfaz de entrada que capta los eventos que ocurren en la composición. Adicionalmente, posee dos interfaces de

salida, que permiten la comunicación con los otros Diagnosticadores y lograr comunicar eventos, y con los reparadores para informar el diagnóstico de las fallas. A continuación se presenta la descripción del servicio Diagnosticador y de sus interfaces:

<table>
<tr><th rowspan="2">Servicio</th><th rowspan="2">Operación</th><th colspan="3">Interfaz de Entrada</th><th colspan="3">Interfaz de Salida</th></tr>
<tr><th>Protocolo</th><th>Variables</th><th>Descripción</th><th>Protocolo</th><th>Variables</th><th>Descripción</th></tr>
<tr><td rowspan="15">Servicio Diagnosticador</td><td rowspan="15">Diagnoser events</td><td rowspan="15">Protocolo de Servicios (SOAP), BC:SOAP</td><td colspan="2"><u>Event</u></td><td rowspan="6">Protocolo de Servicios (SOAP), BC:SOAP</td><td colspan="2"><u>Evento Enlazador</u></td></tr>
<tr><td rowspan="2"><u>id</u></td><td rowspan="2">Identificador único que permite diferenciar eventos que no forman parte de la misma operación.</td><td><u>id</u></td><td>Identificador único que permite diferenciar eventos que no forman parte de la misma operación.</td></tr>
<tr><td><u>event</u></td><td>Nombre del evento.</td></tr>
<tr><td><u>event</u></td><td>Nombre del evento.</td><td><u>time</u></td><td>Tiempo de ocurrencia del evento.</td></tr>
<tr><td><u>time</u></td><td>Tiempo de ocurrencia del evento.</td><td><u>extras</u></td><td>Otros atributos que se consideren necesarios para realizar el reconocimiento de los eventos.</td></tr>
<tr><td rowspan="9"><u>extras</u></td><td rowspan="9">Otros atributos que se consideren necesarios para realizar el reconocimiento del evento.</td><td rowspan="9">Protocolo de archivos BC:FILE</td><td colspan="2"><u>Fault</u></td></tr>
<tr><td><u>id</u></td><td>Identificador único que permite diferenciar eventos que no forman parte de la misma operación.</td></tr>
<tr><td><u>fault</u></td><td>Nombre de la falla.</td></tr>
<tr><td><u>faulttype</u></td><td>El tipo de la falla. Aplicable en algunos casos en el que se necesita obtener más información acerca de la falla, tal como QoS, SLA, entre otros.</td></tr>
<tr><td><u>time</u></td><td>Tiempo de ocurrencia de la falla.</td></tr>
<tr><td><u>extras</u></td><td>Otros atributos que se consideren necesarios para realizar el reconocimiento de los eventos.</td></tr>
</table>

Tabla 5.1: Operaciones del servicio Diagnosticador

El servicio Diagnosticador, en la operación llamada “Diagnoser events”, recibe los eventos usando la interfaz de entrada Event. En el momento que se produce el arribo de un evento, se pone en funcionamiento el motor de inferencia para el reconocimiento de las crónicas distribuidas y al momento de realizarse el reconocimiento de una crónica, el Diagnosticador expresa ese reconocimiento. Para esto, el servicio cuenta con una interfaz de salida compuesta por 2 mensajes: *Evento Enlazador* en el caso que necesite comunicarse con otro Diagnosticador, y *Fault* para informar a su reparador local que ha sido detectada una falla. En la Figura 5.3 se presenta la implementación del servicio Diagnosticador en el middleware ARMISCOM.

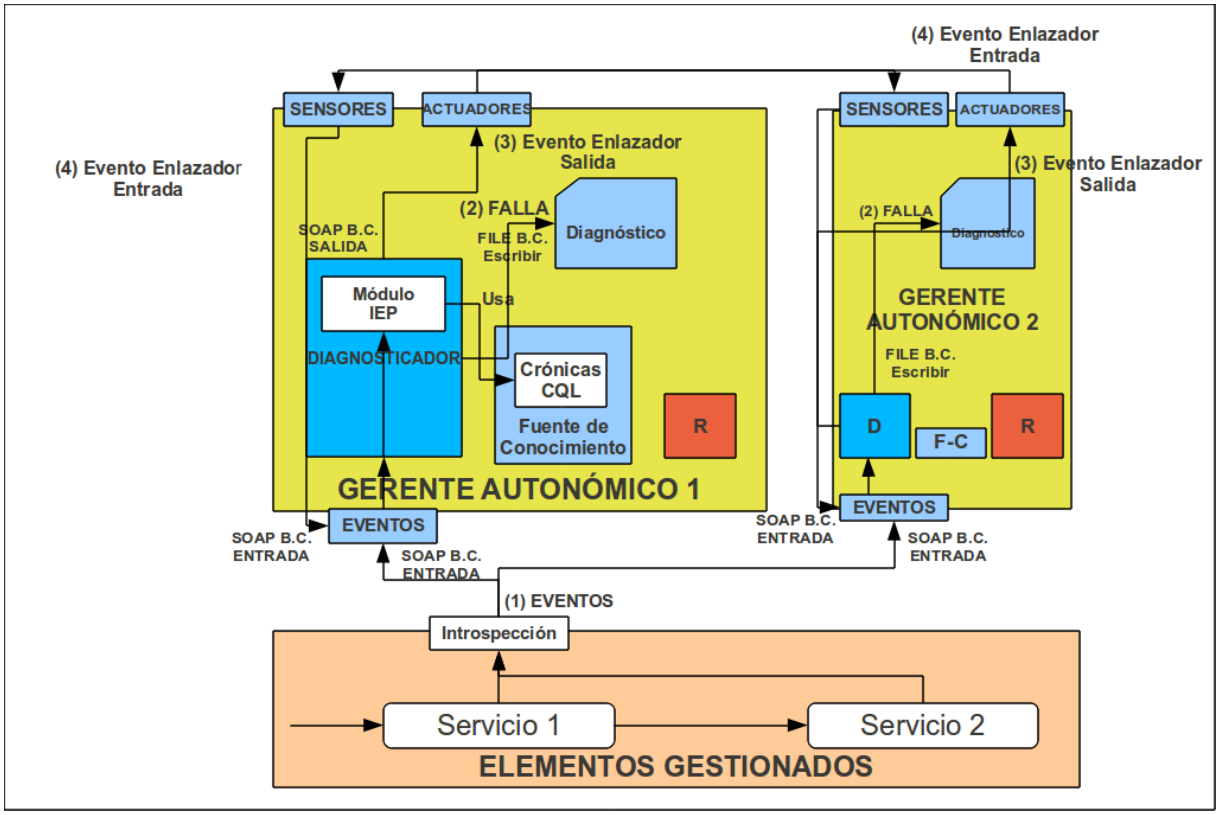


Figura 5.3: Implementación del servicio Diagnosticador en ARMISCOM

Como se muestra en la Figura 5.3, el Diagnosticador se coloca en cada

instancia del gestor autonómico de ARMISCOM (servicio de la aplicación SOA desplegada sobre ARMISCOM), recibe como entradas el conjunto de evento que están aconteciendo en el servicio local de la composición (paso **(1) Eventos**) y los "eventos enlazadores" generados por otros Diagnosticadores (paso **(4) Evento Enlazador**), utilizando para todo esto la interfaz de entrada con el mensaje "event". Cada vez que un nuevo evento llega, el Diagnosticador activa el motor de inferencia de crónicas usando el módulo IEP. Cada vez que una crónica es reconocida, el Diagnosticador puede tener dos salidas: un mensaje "evento enlazador" (ver paso **(3) Evento Enlazador**), que corresponde a los eventos que utilizan otros Diagnosticador para poder realizar un reconocimiento local de las fallas (este se ha desarrollado utilizando el protocolo SOAP); o un mensaje "fault" (ver paso **(2) Falla**), que permite informar a los Reparadores que ha ocurrido una falla (este se ha desarrollado con el protocolo para la gestión de archivos FILE). En la sección 5.2 de este Capítulo se presenta la forma de representar las crónicas en lenguaje CQL.

5.1.2. El Componente Reparador

El componente Reparador es el encargado de realizar los ajustes en la composición cuando acontece una falla. El Reparador tiene 2 sub-servicios:

5.1.2.1 Sub-servicio Planificador

En Capítulos previos se ha definido, una ontología llamada "Fault Recovery", cuya implementación se muestra en la sección 4.2, que permite relacionar las fallas que pueden ocurrir en la composición de servicios con los posibles mecanismos de corrección que puedan aplicarse según la falla. Posteriormente, con el resultado generado por la ontología, se realiza una búsqueda por una metadata basada en regiones (mostrada también en el 3.3.3.2), que permiten al

middleware seleccionar los métodos de corrección a aplicar en las regiones problemáticas, que han sido almacenados previamente. Así, el sub-servicio Planificador contiene 2 operaciones:

Servicio	Operación	Interfaz de Entrada			Interfaz de Salida		
		Protocolo	Variables	Descripción	Protocolo	Variables	Descripción
Sub-Servicio Plan	***Repair Method***	*Protocolo de archivos, BC:FILE*	***Fault***	Nombre de la falla	*Protocolo de Servicios (SOAP), BC:SOAP*	***Possible Methods***	Contiene los distintos métodos que se podrían aplicar para corregir la falla suministrada por la interfaz de entrada.
	Resolution plan	*Protocolo de Servicios (SOAP), BC:SOAP*	***Possible Methods***	Los posibles métodos de corrección que se pueden aplicar a una falla.	*Protocolo de Servicios (SOAP), BC: SOAP*	***Resolution plan***	Contiene el conjunto de operaciones a realizar para solventar la falla.
			event_init	El lugar de inicio del flujo que se desea modificar en la composición.			
			event_end	El ultimo servicio que se desea modificar en la composición.			

Tabla 5.2: Operaciones del sub-servicio Planificador

El sub-servicio planificador en la operación **“Repair Method”** recibe el nombre de la falla que se encuentra en un archivo, con esta falla el sub-servicio infiere el conjunto de métodos de solución que pueden ser aplicados para resolver la falla. La operación **“Repair Method”** utiliza la ontología “*Fault-Recovery*”, diseñada en la sección 3.3.3.1 , e implementada en la sección 5.2, para correlacionar las fallas y los métodos de reparación. Para la implementación del servicio web en OpenESB se realizó un programa en JAVA (apéndice A.1),

utilizando el motor de inferencia FACT++[27], encapsulándolo como un servicio web con el BC "Sun Java EE SE".

Por otro lado, la operación **“Resolution plan”** recibe como entrada el conjunto de métodos que se pueden implementar (*Possible Methods*) y la parte del flujo de la composición que se encuentra afectada por la falla; una vez invocado, la operación realiza la búsqueda de los métodos que se encuentran disponibles haciendo uso del campo *operations* (operaciones) de la *metadata* diseñada en el 3.3.3.2 , y devuelve la acción que mejor se ajuste al requerimiento (*Resolution plan*). En la Figura 5.4 se muestra la implementación del sub-servicio planificador en ARMISCOM.

Como se muestra en la Figura 5.4, la operación **“Repair Method”** recibe como entrada el diagnóstico del diagnosticador (ver paso **(1) Falla (Fault)**), que se encuentra en forma de archivo. Para esto, extrae el nombre de la falla (***fault***) del mensaje de salida Fault de la operación “*Diagnoser events*” del diagnosticador (es el mensaje de entrada en la operación **“Repair Method”**). Después, realiza la inferencia de los métodos que se pueden implementar para solventar este tipo de falla, usando el motor de ontologías FACT++ y retornando los métodos posibles a implementar (ver paso **(2) Posibles métodos**). El resultado de la operación **“Repair Method”** (***Posibles Métodos***) se suministra como entrada a la operación **“Resolution plan”**, agregando los atributos *Flow_init* y *Flow_end* tomados del mensaje generado del diagnóstico de la falla por el Diagnosticador. Con esta información, la operación busca en la *metadata* el mecanismo de solución que mejor se ajusta a la falla y responde con el mensaje de salida *Resolution plan* (ver paso **(3) Plan de Resolución**).

27 FaCT++ es un razonador basado en tablas para las expresiones de descripción Lógicas (DL), desarrollado por la Universidad de Manchester, Cubre los lenguajes OWL y OWL2.

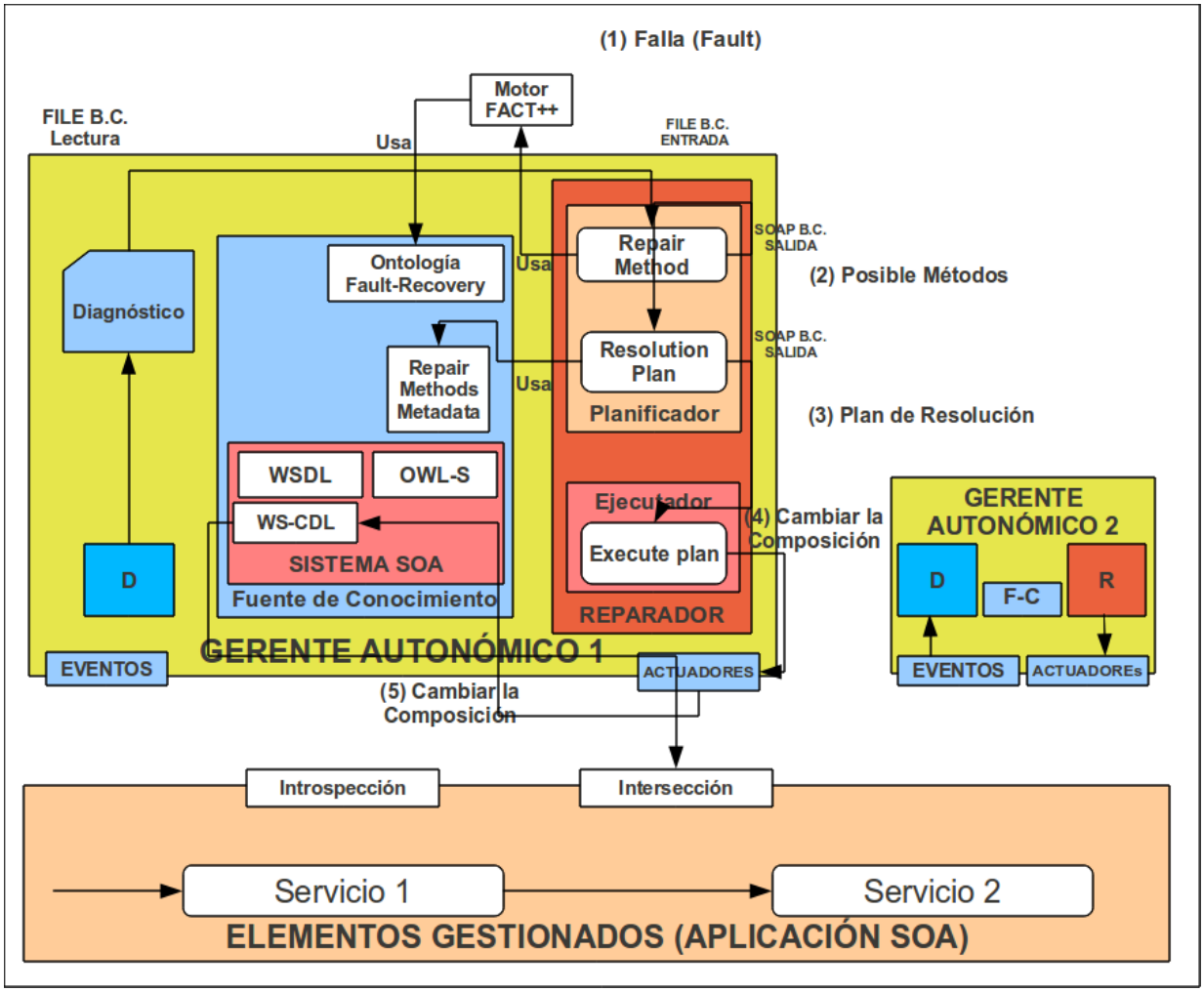


Figura 5.4: Implementación del Reparador en ARMISCOM

5.1.2.2 Sub-servicio Ejecutador

Ejecutador es el encargado de realizar los cambios en la composición. Para esto, ARMISCOM modifica la aplicación SOA usando el contenido de la nueva composición. Particularmente, la metadata utilizada por el reparador contiene un campo llamado operations (Figura 3.6), donde se encuentran almacenados el conjunto de pasos a realizar para cambiar la aplicación SOA que se encuentre ejecutando. Para ello, el sub-servicio Ejecutador cuenta con una sola operación,

llamada execute plan:

Servicio	Operación	Interfaz de Entrada			Interfaz de Salida		
		Protocolo	Variables	Descripción	Protocolo	Variables	Descripción
Sub-servicio Ejecutador	***Execute plan***	*Protocolo de Servicios (SOAP), BC:SOAP*	***Resolution plan***	Contiene el conjunto de operaciones ha realizar para modificar la composición de servicio, esta información es derivada de la metadata (ver Figura 3.6).	*Protocolo de Servicios (SOAP), BC:SOAP*	***previuosstate***	Describe el estado en que se encontraba el flujo antes de realizar la modificación.
						currentstate	Representa el estado final de la composición luego de cambiar su flujo.

Tabla 5.3: Operaciones del sub-servicio Ejecutador

La operación “Execute plan” se encarga de realizar las modificaciones en el flujo de la composición de los servicios, para esto recibe como entrada el conjunto de operaciones (plan) que se desea ejecutar, modifica la composición de los servicios según lo establecido en las operaciones de la metadata, y luego retorna un mensaje con el contenido previuosstate y currentstate, describiendo las modificaciones realizadas. En la Figura 5.4 se muestra que la operación “Execute plan” recibe como entrada el mensaje obtenido previamente por la operación “Resolution plan” del sub-servicio Planificador (ver paso **(3) Plan de Resolución**), y ejecuta las operaciones para modificar el flujo de ejecución de la composición (modificar el documento WS-CDL que se encuentra en , ver pasos **(4) y (5) Cambiar la Composición**).

5.2. Componente Fuente de Conocimiento

Como se indicó en el Capítulo 3, el componente Fuente de Conocimiento caracteriza todas las fuentes de conocimiento que utiliza ARMISCOM. En lo que se refiere a su implementación, se usa OpenESB usando el lenguaje CQL para representar las crónicas y Protege para presentar la ontología “Fault Recovery”.

5.2.1. Representación de Crónicas Distribuidas Utilizando el Lenguaje CQL

Sintácticamente, CQL es muy similar a la instrucción SELECT del lenguaje SQL, pero la ejecución de consultas son diferentes a las consultas de bases de datos convencionales en SQL, cuyas consultas se ejecutan por demanda hasta que se completen todos los datos solicitados. Por el contrario, en CQL las consultas son continuas sobre los streams corriendo indefinidamente (infinitas tuplas de streams), o hasta que se terminan. Antes de profundizar con la sintaxis del CQL, es necesario definir lo que se conoce como streams y relations [67, 69]:

Streams (Flujo de eventos): Un Stream S es una serie (posiblemente infinita) de elementos <s, τ>, donde s es una tupla perteneciente al esquema de S y $\tau \in T$ es la marca de tiempo del elemento (series de eventos en el tiempo que tienen el mismo esquema).

Relations (Relaciones): Una relación R es una condición en un intervalo de T que cumplen un conjunto de tuplas pertenecientes al esquema S (son colecciones de eventos que cumplen con una condición R dada en un momento dado).

La sintaxis CQL contiene muchos operadores que se encuentran en SQL, como projection, selection, aggregation, joining, grouping, etc., pero CQL tiene otros

operadores para convertir streams en relations [68, 69]:

- **Stream to Relation Operators (Operadores para convertir Stream a Relation):** Se basan en el concepto de una ventana deslizante sobre un stream. Estos se dividen en:
    - ***Time-based:*** convierte un stream en una relation en un período de tiempo determinado. EJ: Event 1 [Range 30 Seconds], el stream Event 1 se convertirá en una relation que estará disponible por 30 segundos desde su ocurrencia.
    - ***Tuple-based:*** convierte un stream en una relation una vez se cumpla un número determinado de eventos. EJ: Event 1 [Rows 3], el stream Event 1 sera convertido en relation cuando se obtenga un tercer valor del stream.
    - ***Attribute based:*** convierte un stream en una relation cuando un atributo especifico del stream toma un valor dado. EJ: Event 1 [Partition By Status = TRUE], el stream Event 1 se convertirá en una relation cuando el valor de su atributo Status tenga valor de TRUE.
    - ***Partitioned:*** convierte un stream en una relation cuando ocurre un valor específico en un atributo del stream y un número determinado de eventos. p.e.: Event 1 [Partition By Status = TRUE Rows 2], el stream Event 1 sera convertido a una relation cuando se obtenga el segundo stream con el atributo Status igual a TRUE.
- **Relation-to-Relation Operator (Operador para convertir Relation a Relation):** se utiliza para mapear relations semánticas de variables temporales, utilizando consultas similares a las usadas en las sentencias

SQL:

SELECT event1 .id

FROM event1 [Range 30 Seconds]

WHERE event1.id > 10

La sentencia CQL mostrada previamente corresponde a la generación de una nueva relation con atributo id, la cual es generada del stream event 1 que es convertida en relation usando el operador Time-based por 30 segundos, restringida por el operador de Relation-to-Relation del atributo de la relation con valores superiores a 10.

- **Relation to Stream Operators (Operadores para convertir Relation a Stream):** Se utilizan cuando se requiere convertir una relation en un stream:
  - ***ISTREAM:*** convierte una relation en un stream que se encuentra disponible en el momento T y no en momento T-1 (la relation se acaba de producir en el instante T, previamente no existía).
  - ***DSTREAM:*** convierte una relation en un stream que existía en el momento T - 1 y no en T (la relation ya no se encuentra disponible en el instante T, pero existía previamente en T - 1).
  - ***RSTREAM:*** convierte a un stream a una relation que se encuentra en el momento T.

Particularmente, la lógica temporal se puede utilizar en la representación de las crónicas, permitiendo términos proposicionales con objetos temporales (predicados reificados). La Reificación de predicados usados en las crónicas son

descritos en [70]. A continuación se presentan la estructura necesaria para representar a los predicados reificados usados por las crónicas en el lenguaje CQL:

Temporal Predicate of the Chronicles	Descripción	Temporal Predicate of the Chronicles in CQL	Query CQL description
`hold(P : v, (t1 , t2 ))`	El dominio del atributo P debería mantener el valor v en el el intervalo [t1 , t2].	**SELECT** ISTREAM('newevent') **FROM** Event0[t2 - t1], Event1[now] **WHERE** Event0.P = 'v' AND NOT(Event0.P <> 'v') AND Event1.P = 'v'	Para estar seguro que el atributo P mantiene el valor de v sobre el intervalo [t1 , t2], El atributo del stream entrante Event1 (time = NOW) debería tener el valor de v. Además,las relation previas no deben tener ningún valor diferente de v (NOT(Event0.P <> 'v')). Esto garantiza que en el intervalo se mantiene el valor anterior de Event0 en el momento NOW - t2 + t1 igual a v.
`event(P : (v1 , v2 ), t)`	El atributo P cambia su valor de v1 a v2 en el tiempo t.	**SELECT** ISTREAM('newevent') **FROM** Event0[ΔT], Event1[NOW] **WHERE** Event0.P = 'v1' AND NOT(Event0.P = 'v2') AND Event1.P = 'v2'	El actual stream Event1 tiene el atributo P (time = NOW) con valor v2. Además, existe al menos una relation de Event1 (llamada Event0 en la sentencia para diferenciarlas) con el atributo P igual a v1 en el momento ΔT (ΔT tiene un valor que asegure que el atributo P puede tener el valor v1 previamente, ej: 1 segundo).
`event(P, t)`	Mensaje P ocurre en el tiempo t.	**SELECT** ISTREAM('newevent') **FROM** event[NOW] **WHERE** event.msg = 'P'	El actual stream de Event tiene un Mensaje P en el tiempo NOW (Ahora).
`noevent(P, (t1 , t2 ))`	No se recibe el dominio del atributo P entre los momentos t1 y t2.	**SELECT** ISTREAM('newevent') **FROM** Event0[t2 - t1], Event1[NOW] **WHERE** NOT (event0.msg = 'P') AND NOT (event1.msg = 'P')	El actual stream de Event1 no tiene un Mensaje P en el tiempo NOW y no existe una relation de Event1 (llamada Event0) con el Mensaje P que ha ocurrido previamente: NOW - t2 + t1.
`occurs((n1, n2), P, (t1, t2))`	El evento que coincide con el patrón P ocurre N veces, tal que (n1≤N≤n2), entre el intervalo de tiempo t1 y t2 . El valor de ∞ puede ser usado para n2.	**SELECT** ISTREAM('newevent') **FROM** event[NOW] **WHERE** event.msg = 'P' AND (***SELECT*** count(event) ***FROM*** event[t2 - t1] ***WHERE***	El actual stream de Event tiene un Mensaje P en el tiempo NOW y ha ocurrido un numero N - 1 de veces en forma de relation (entre n1 - 1 y n2 - 1 en el tiempo t2 - t1). Note que en el caso de que n1 = 0 puede eliminarse la relation generada de n1. En el caso de n2 = ∞ se elimina la relation de n2.

Temporal Predicate of the Chronicles	Descripción	Temporal Predicate of the Chronicles in CQL	Query CQL description
		event.msg = 'P') >= n1 - 1 AND (***SELECT*** count(event) ***FROM*** event[t2 - t1] ***WHERE*** event.msg = 'P') <= n2 - 1	

Tabla 5.4: Predicados reificados en el lenguaje CQL

Entonces, un modelo de crónicas que utiliza los eventos E_1, E_2 y E_3 con las restricciones temporales ($T_3 > T_1 + C_1$ y $T_3 > T_2 + C_2$), puede ser escrito en lenguaje CQL usando los operadores Relation-to-Stream Operators como:

SELECT ISTREAM(E_3.id, E_3.time, 'Situación de interes')

FROM E_1 [Range C_1 Seconds], E_2 [Range C_2 Seconds], E_3 [now]

WHERE Constrains

donde:

- ***SELECT*** define el resultado de una crónica reconocida (equivale a "**When is recognized**"), generando un Stream que puede ser usada por otra crónica o componente del sistema.
    - *ISTREAM* describe el contenido de la relation generada por la crónica, que puede tener una mezcla de diferentes atributos de las relations y los streams involucrados en la crónica (debido a que la información de los atributos presentes en los eventos se encuentran disponibles, estos pueden ser usados para enriquecer el stream generado). En este caso se utiliza el operador ISTREAM para convertir relation a stream (se podría

utilizar DSTREAM o RSTREAM). Adicionalmente, los atributos id y time se utilizan en la generación de los stream:

- E_i.id representa el atributo que hace referencia al identificador de los eventos en la crónica.
- E_i.time representa atributo para representar el punto de tiempo de ocurrencia de los eventos.
- El último atributo es el nombre del evento que se está generando. En este caso se ha establecido un nombre genérico llamado 'Situación de interés".

- ***FROM*** representa las fuentes de eventos que se usaran en la crónica, y están representadas por el conjunto de streams y relations que se utilizan en las crónicas para extraer los eventos (los eventos E_1 y E_2 se encuentran en forma de relation y E_3 es un stream). Esta sección tiene todos los eventos que forman parte de la crónica, expresados como streams y relations:
  - C_i son constantes usadas para representar la diferencia entre los puntos de tiempo de ocurrencia de dos eventos.
  - E_i *[Range* C_i *Seconds]* describe las relations que fueron convertidas desde un stream e_i usando el operador time-based (Stream to Relation), y son almacenados por C_i segundos.
  - E_j *[now]* define el stream E_j que esta ocurriendo en este momento (now).

- ***WHERE:*** *son el conjunto de restricciones entre* T_i (variables temporales, Ej: $E_i.T_i > E_{i-1}.T_{i-1}$) y/o otros atributos de los eventos (variables atemporales, Ej: E_i.status = FALSE).

Así, cuando se reconoce la crónica, este emite un stream con algunos atributos y otra información utilizando la sentencia select. Por ejemplo, la crónica distribuida de la Figura 4.3 implementada en CQL se muestra en la Figura 5.5.

```
  Chronicle Subchronicle 1
{
    SELECT
      ISTREAM(
        'BESC1',
        'Diagnoser 2'
      )
  FROM
      E1[C1 + C2 +C3],
      E2[C2 + C3],
      E10[C3],
      E11[now]
  WHERE
      E2.time > E1.time AND
      E10.time > E2.time
AND
      E11.time > E10.time
  }
```

```
  Chronicle Subchronicle 2
{
    SELECT
      ISTREAM(
        'log',
        'Fault 1'
      )
  FROM
      E3[C4 + C5 + C8],
      E4[C5 + C8],
      E5[C8]
      BESC3[C8 - C7]
      BESC1[now]
  WHERE
      E4.time > E3.time AND
      E5.time > E4.time AND
      BESC3.time > E5.time
AND
      BESC1.time >
BESC3.time
  }
```

```
  Chronicle Subchronicle 3
{
    SELECT
      ISTREAM(
        'BESC3',
        'Diagnoser 2'
      )
  FROM
      E6[C9 + C10],
      E7[C10],
      E8[now]
  WHERE
      E7.time > E6.time AND
      E8.time > E7.time
  }
```

Figura 5.5: Modelo de Crónica en CQL del ejemplo de la Figura 4.3

Como se muestra en la Figura 5.5, se puede expresar la crónica de la Figura 4.3 en sentencias CQL. La explicación de la conversión de la sub-chronicle 1 en sentencia CQL se muestra a continuación:

- **SELECT:** En el momento cuando sub-chronicle 1 es reconocida, es

necesario emitir el evento enlazador BE_{SC1} a Diagnoser 2, para esto se define la producción de un stream utilizando el operador de relation a stream ISTREAM con event name BE_{SC1} y destination Diagnoser 2.

- **FROM:** El último evento que alcanza sub-chronicle es E_{11}, este es un stream que ocurre en este momento (now). Además, los eventos E_1, E_2 y E_{10} son convertidos en relations usando el operador Time-based con un período de tiempo constante de C_3 para E_{10} (E10[C3]), $C_2 + C_3$ para E_2 (el tiempo de ocurrencia de E_2 debería ser antes de E_{10}, así que debería ocurrir en el periodo de E_{10} más el periodo de E_2: $C_2 + C_3$) y $C_1 + C_2 + C_3$ para E_1 (tiempo de ocurrencia de E_1 debería ser antes de E_2, así que el evento E_1 debería considerar el periodo de E_2 más el periodo de E_1: $C_1 + C_2 + C_3$) .

- **WHERE:** Con solo definir los eventos como relations en las sub-crónicas mostradas en la Figura 5.5 no garantiza el orden correcto de llegada de los eventos (p.e. el evento E_1 podría ocurrir después de E_2), es necesario definir un conjunto de operaciones adicionales sobre los eventos para establecer el orden de ellos (p.e es necesario colocar una operación que garantice que E_1 se produce antes de E_2, tal que se genere un stream que será usado por la sub-crónica). Así, para facilitar el proceso de escritura de las crónicas, es necesario establecer restricciones en el orden en que los eventos deben llegar a la sub-crónica usando el atributo time, de esta manera se establece que el evento E_1 debería ocurrir antes que E_2, entonces el evento E_{10}, y finalmente el evento E_{11}.

La estructura de las sub-crónicas "Subchronicle 2 y 3" son similares a la "Subchronicle 1", en el caso de Subchronicle 3 se establecen las relations E6[C9 + C10] y E7[C9] y el stream E_8, y para Subcronicle 2 las relations E3[C4 + C5 + C8], E4[C5 + C8], E5[C8] y BESC3[C8 - C7] y el stream BESC1. Además, se ha añadido

el atributo de tiempo (time) a los eventos stream para facilitar la inferencia acerca de la secuencia de ocurrencia de estos y expresar su ocurrencia exacta en el sistema real, que no son necesariamente los tiempos de llegadas en el reconocedor de crónicas: los tiempos de comunicación pueden afectar los tiempos de llegadas de los eventos desde el sistema real hacia el reconocedor, cuando se produce el evento es enriquecido con su tiempo de ocurrencia en el atributo time (p.e. cuando se produce el evento E_1 en la composición se agrega el atributo time con su estampilla de tiempo en formato Unix. E_1.time es igual a 1412630719054, el cual es equivalente a las 04:55pm del 6 de agosto de 2014). Finalmente, las 3 sub-crónicas emiten los eventos enlazadores correspondientes BE_{SC1}, BE_{SC3} y el log, los cuales son enrutados a sus destinos (p.e. a diagnoser 2 desde las Sub-cronicas 1 y 3).

5.2.2. Ontología Fault Recovery

Se ha implementado la ontología utilizando la herramienta Protégé[28], que se basa en la Web Ontology Language (OWL). La ontología es utilizada con el motor de inferencia FACT++ (apéndice A.1). Los conceptos de fallas y métodos de reparación, y la relación entre ellos Has_repair_method, generados en Protegé, son mostrados en Figura 5.6 (fallas físicas), Figura 5.7 (fallas de desarrollo) y Figura 5.8 (falla de interacción).

28 Protégé es un editor de ontologías de código abierto. Proporciona una interfaz gráfica de usuario para definir ontologías. Esta aplicación está escrita en el lenguaje Java.

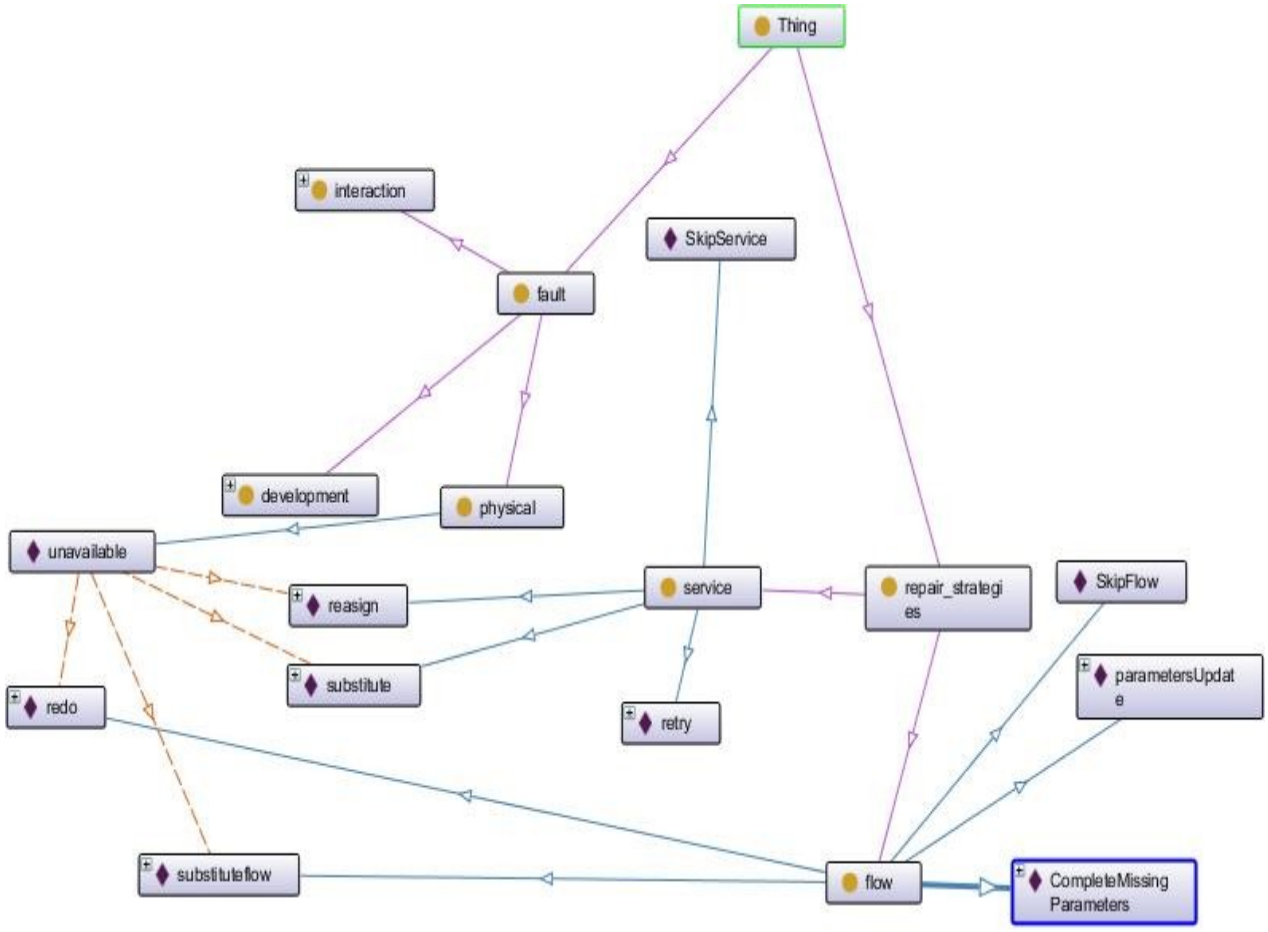


Figura 5.6: Relaciones entre los conceptos para las fallas físicas en la ontología Fault-Recovery

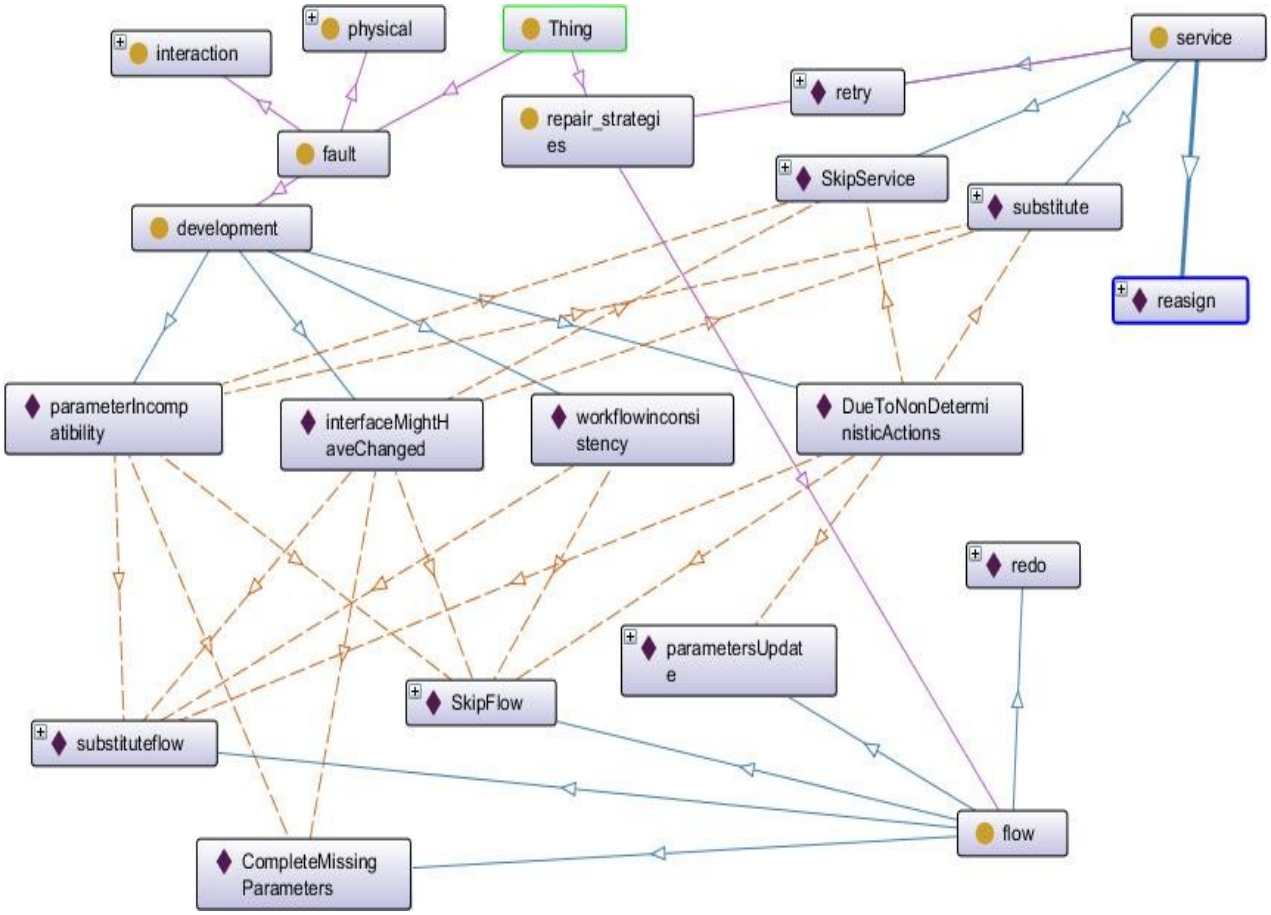


Figura 5.7: Relaciones entre los conceptos para las fallas de desarrollo (development) en la ontología Fault-Recovery

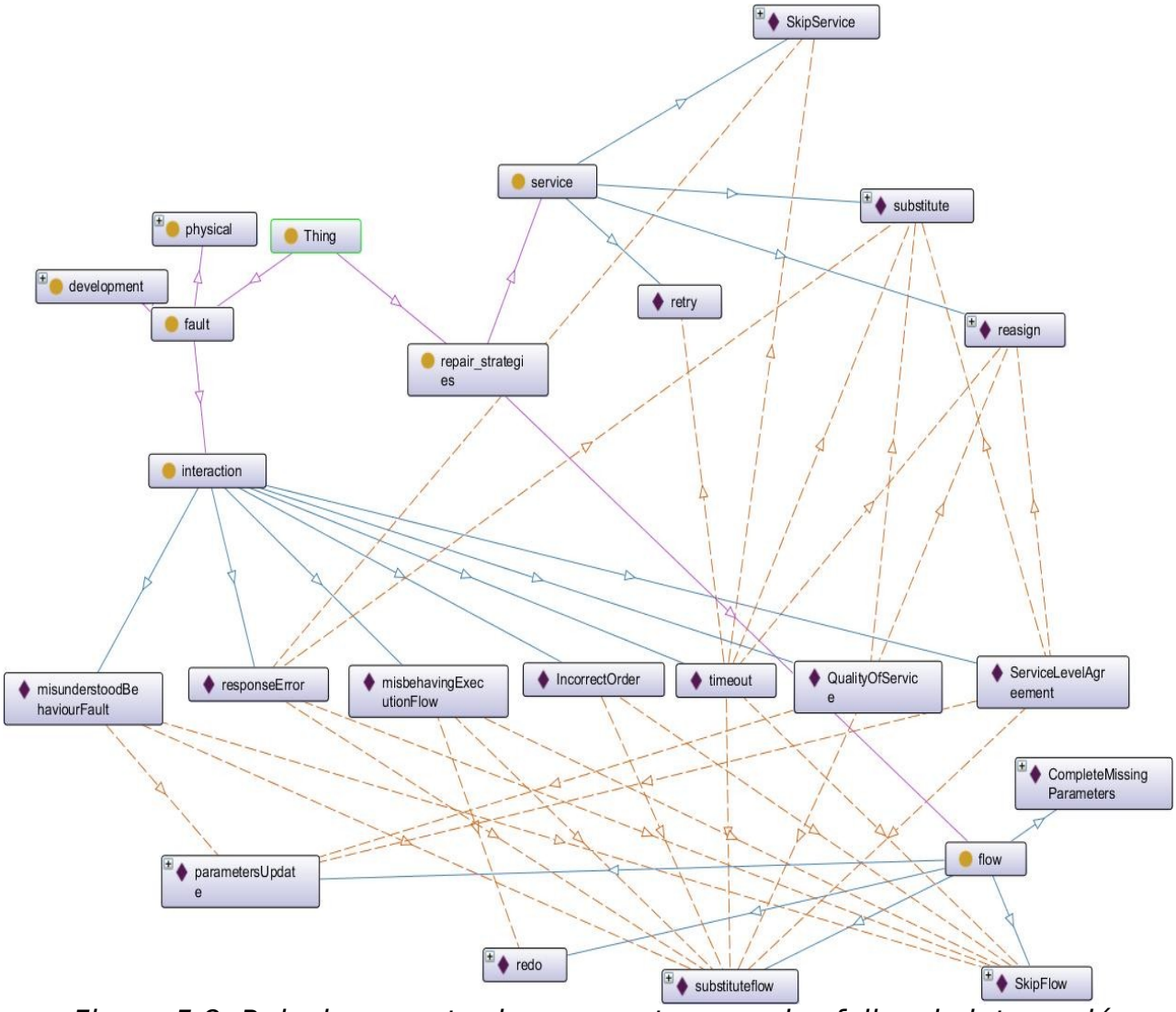


Figura 5.8: Relaciones entre los conceptos para las fallas de interacción (interaction) en la ontología Fault-Recovery

Como se evidencia en las Figura 5.6, Figura 5.7, Figura 5.8, la ontología es capaz de categorizar los mecanismos de reparación por tipos de fallas. Por ejemplo, para las fallas físicas (ver Figura 5.6), cuando se tiene el diagnóstico de *servicio no disponible* (unavailable) se consulta la ontología con la falla y se obtienen los métodos de reparación: redo, reassign, substitute y substituteFlow. En este mismo orden de ideas, cuando se tiene el diagnóstico de la falla de

desarrollo parámetros incompatibles (parameterIncompatibility) (ver Figura 5.7), la ontología responde con los métodos: CompleteMissingParameters, SkipFlow y substituteFlow. De manera similar funciona la ontología para las fallas de interacción (ver Figura 5.8), cuando por ejemplo se presenta la falla *responseError,* los métodos derivados de la ontología son: SkipService, SubstituteSkipFlow y substituteFlow.

5.3. Conclusiones

En este Capítulo se ha mostrado la implementación de ARMISCOM en openESB y Protege. Los distintos componentes del middleware fueron construidos como servicios independientes y autónomos, lo que facilita su reuso y la dan una gran flexibilidad para poder ser construido al ser definido como una composición de servicios, permitiendo que en un futuro pueda ser modificada y extendida su estructura, principalmente a nivel de los componentes del gestor autonómico. También, al ser reusable los distintos servicios/componentes, pueden ser utilizados separadamente, como por ejemplo el servicio Diagnosticador que podría usarse como una herramienta de reconocimiento de crónicas.

En la Tabla 5.5 se realiza una comparación cualitativa del beneficio de implementar las crónicas y su reconocimiento usando el lenguaje CQL y el componente IEP de OpenESB para nuestro middleware ARMISCOM vs usar las herramientas CRS [10, 32], CarDeCRS [12] y Matrac [71].

Los criterios usados para la comparación fueron: se necesita que los componentes de ARMICOM estén en **software libre**, para que puedan ser usados libremente y mejorados continuamente. En ese mismo orden de ideas, debido a que el middleware realiza el diagnóstico de fallas en aplicaciones SOA y debe interactuar con el resto de componentes de nuestro MAPE, el diagnosticador debe

permitir la **interoperatividad** y composición propia de una aplicación SOA, para esto debe permitir que el flujo de eventos y el diagnóstico puedan estar conectados utilizando un gran abanico de protocolos (como SOAP, archivos, entre otros). Por otro lado, debido a que las distintas instancias de ARMISCOM se distribuyen en cada servicio de la composición, es necesario que el diagnosticador permita un funcionamiento que exhiba un comportamiento de la **arquitectura distribuido**. Por último, como se mostró en el Capítulo 4, realizar el diagnóstico de fallas en servicios web frecuentemente necesita realizar inferencia sobre predicados construidos en base a restricciones de los valores de los esquemas que componen los eventos, para detectar violación de valores de SLA, QoS, entre otros. Para esto, el motor de inferencia del diagnosticador debe permitir una gran **expresividad de las restricciones de los eventos**, tanto a nivel temporal como atemporal. Usando esos criterios es que se realiza la comparación cualitativa en la Tabla 5.5. Estos criterios definen las características que debe tener el motor de reconocimiento de eventos a usar en ARMISCOM.

Herramienta	Software libre	Interoperatividad	Arquitectura	Expresividad de las Restricciónes de eventos
CRS [10, 32]	**NO**	NO (Es necesario encapsular el CRS como un servicio web)	Monolítica	Predicados Temporales
CarDeCRS [12] y Matrac [71]	**NO**	Protocolo Interno de comunicación para permitir la comunicación jerarquica de los diagnosticadores locales con el global.	Semi-descentralizado	Predicados Temporales
ARMISCOM	**SI**	Implementa una gran variedad de protocolos para entrada y salida	Completamente Distribuida.	Predicados Temporales Variables Atemporales

Tabla 5.5: IEP vs CRS y CarDeCRS

Las herramientas CRS, CarDeCRS y Matrac no están liberadas en **software libre** y el acceso a las fuentes de la aplicación, como su uso, esta restringido a la

concesión de un permiso. Asimismo, la **interoperatividad** de las herramientas no permiten la comunicación con otros componentes propios de una aplicación SOA. En el caso concreto, CRS al estar escrita en el lenguaje JAVA y ser una arquitectura monolítica, podría encapsularse como un servicio web utilizando el B.C. Sun Java EE SE de OpenESB, posibilitando así su completa **interoperatividad.** Esto también permitiría su completa distribución entre cada servicio de la composición, al utilizar nuestra extensión al formalismo de crónicas mostrada en el Capítulo 4. Ahora bien, las herramientas CarDeCRS y Matrac al contar con una **arquitectura** concreta descentralizadas, el protocolo de comunicación (**interoperatividad**) entre los diagnosticadores locales y globales (interacción con los diagnosticadores) ha sido previamente establecido. Eso implica que para modificar su estructura a un enfoque distribuido su topología de comunicaciones debe ser modificada manipulando su código fuente. Además, expandir CarDeCRS y Matrac para permitir la interoperatividad solo seria posible al realizar el encapsulamiento de cada CRS como servicio, lo que implicaría prácticamente la reconstrucción de las herramientas. Finalmente modificar la arquitectura descentralizada de CarDeCRS y Matrac a una completamente distribuida, modificarían las extensiones de detección de crónicas propuestas en esos trabajos.

Por otro lado, las herramientas CRS, CarDeCRS y Matrac, al estar en el lenguaje CRS, no permiten la aplicación de restricciones sobre variables atemporales[29] (p.e. E_i.rate + E_j.rate < 50), quitándole expresividad a las crónicas. Además, con CQL se pueden implementar variables temporales, y todos los predicados temporales clásicos utilizados comúnmente en las crónicas (holds, events, no events y occurs).

29 variables en los eventos que no están atadas a las leyes del tiempo

Capítulo 6
Crónicas - Experimentación y Análisis de Resultados

En Capítulos anteriores se ha propuesto una extensión del paradigma de crónicas, para poder caracterizar y reconocer patrones distribuidos de eventos. Adicionalmente, se han propuesto un conjunto de patrones de crónicas distribuidas para la detección de fallas en la composición de servicios y el empleo del lenguaje CQL en la construcción de reconocedores de crónicas. En este Capítulo se presenta el conjunto de experimentos realizados para comprobar el funcionamiento de las crónicas para el diagnóstico de fallas en la composición de servicios web usando el lenguaje CQL.

6.1. Casos de Estudio

Para verificar el funcionamiento del mecanismo para el reconocimiento distribuido y los patrones para detectar fallas en aplicaciones SOA utilizando crónicas, se han seleccionado dos casos de estudio: Industria del mueble (Furniture Manufacturing) y Comercio Electrónico (E-Commerce).

6.1.1. Caso Industria del Mueble (Furniture Manufacturing)

Uno de los escenarios seleccionados para poner a prueba la propuesta es la industria del mueble (Furniture Manufacturing), esta industria corresponde a uno de los cinco escenarios estudiados en el proyecto IMAGINE Living Labs[30] [72]. La industria del mueble tiene como objetivo la optimización de las redes de

30 IMAGINE es un proyecto de Investigación y Desarrollo, financiado por la Comisión Europea en el marco del “Virtual Factories and Enterprises” theme of the 7th Framework Programme (FoF-ICT-2011.7.3, Grant Agreement No: 285132)

suministro y producción bajo demanda para el sector de la decoración [73]. La industria del mueble no funciona de una manera única, y en cada nivel de la cadena de suministros hay empresas que trabajan en los diferentes procesos implícitos en esa cadena. La descripción de los actores que intervienen son:

- **Comprador Final (End Customer):** cualquier persona que tiene la intención de amoblar una habitación o una casa entera.

- **Tienda (Retail):** es comúnmente representado por tiendas de muebles, que reciben solicitudes del cliente final y envían solicitudes a uno o más Fabricante de muebles.

- **Fabricante de Muebles (Furniture Manufacturer):** es el fabricante de muebles, recibe una orden de un minorista (tienda de muebles) y no vende directamente al cliente final. La Tienda puede tener varios Fabricante de muebles que reciben sus requisitos, y selecciona uno de acuerdo con la calidad de las respuestas emitidas de acuerdo con los criterios de calidad de los productos, costo y tiempo de entrega solicitados. En este caso particular, hemos elegido sólo 2 Fabricantes, pero podrían haber muchos más.

Las interacciones derivadas de la composición de servicios para el caso de estudio, se muestran en la Figura 6.1:

(**1) Enviar Requerimiento:** el comprador envía la lista de productos requerida a la tienda.

(**2) Petición de Productos:** la tienda envía la petición de productos al fabricante. Esto involucra las órdenes de todos los clientes.

(3) Recibir Productos: la tienda recibe los productos del fabricante.

(**4) Enviar Productos:** la tienda envía los productos requeridos al comprador.

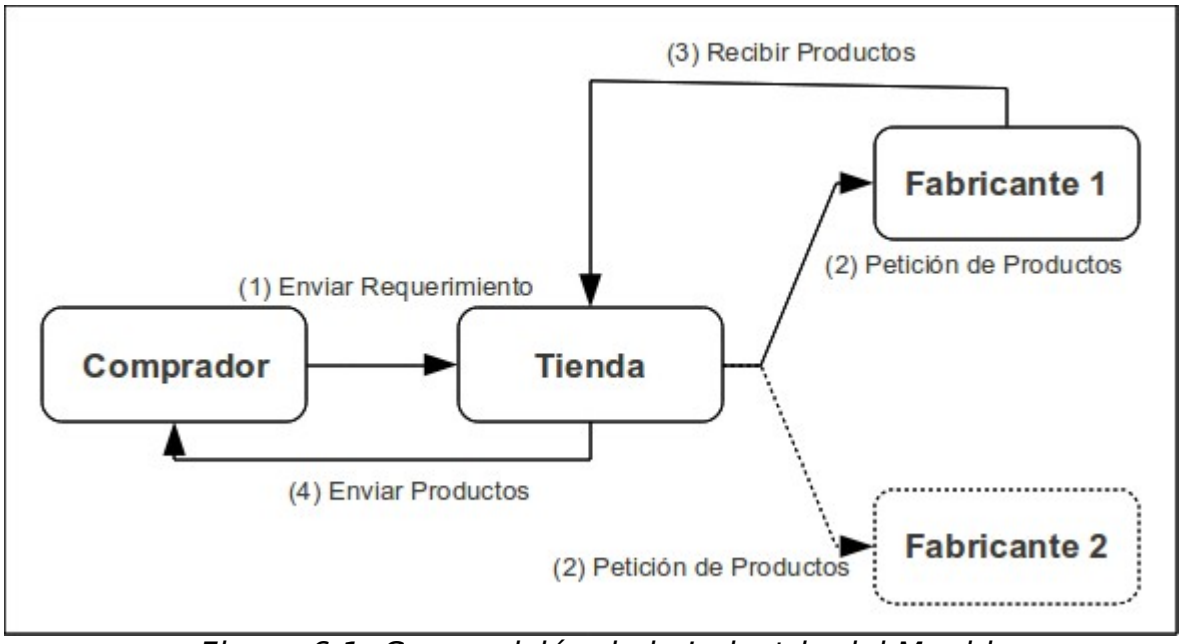
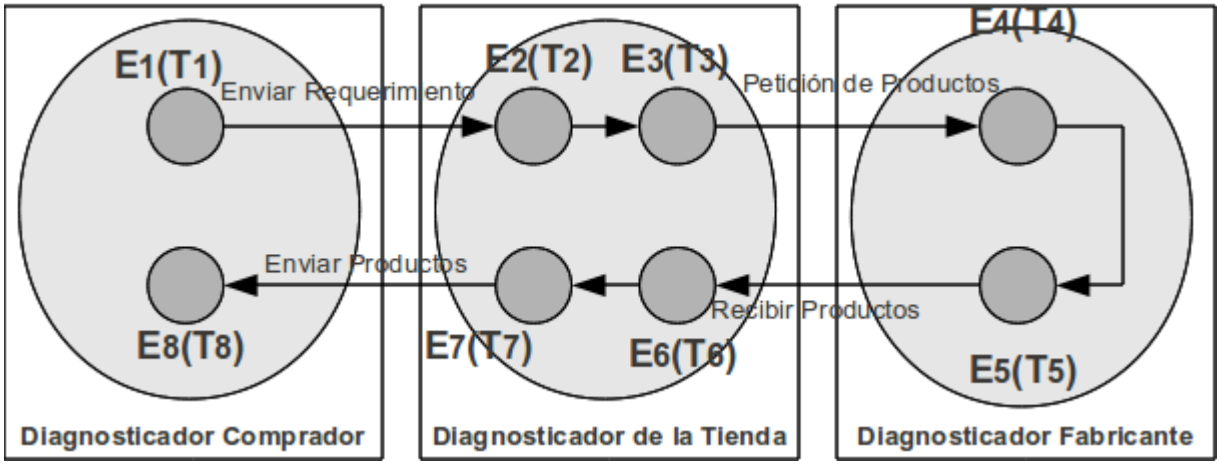


Figura 6.1: Composición de la Industria del Mueble

Ahora se caracterizan la distribución de los eventos entre los diferentes diagnosticadores (sitios) que forman parte de la composición, para construir una crónica genérica para esta aplicación (conectando todos los eventos que pueden ocurrir). Desde esta crónica genérica podemos derivar cada crónica específica, que describa cualquier situación anormal que se quiera detectar. Además, por razones prácticas, se considera que el tiempo es medido en segundos, y el retraso en las comunicaciones y el tiempo de reconocimiento de crónicas son insignificantes. La secuencia de eventos de esta crónica genérica se muestran en la Figura 6.2.



Comprador	Tienda	Fabricante
● **E_1:** Envía el requerimiento de productos a la Tienda. ● **E_8:** Recibe los productos de la Tienda.	● **E_2:** Recibe el requerimiento de los productos del Comprador. ● **E_3:** Envía la solicitud de los productos al Fabricante. ● **E_6:** Recibe los productos terminados del Fabricante. ● **E_7:** Envía los productos al Comprador,	● **E_4:** Recibe la lista de productos requeridos de la Tienda. ● **E_5:** Una vez que los productos son terminados son enviados a la Tienda.

Figura 6.2: Eventos de la Composición de la Industria del Mueble por diagnosticador

6.1.1.1 Implementación de la Aplicación de la Industria del Mueble

Para verificar el funcionamiento del middleware, en el servicio Fabricante se agregaron las siguientes operaciones adicionales para inducir fallas de SLA de Calidad (Quality):

- **tuneQualityBehavior:** Usada para generar la falla en la operación E_5. Si está activado (tuneQualityBehavior = TRUE), la operación E_5 produce menos productos de calidad que los que se necesitan (PRvar = PSvar - 1);

en caso contrario, la operación E_5 funciona normalmente.

- **turnprovider:** Operación propia del servicio Tienda para definir el servicio Fabricante a invocar. Si la operación turnprovider es invocada con un valor FALSE, la composición es invocada con Fabricante 1, en caso contrario (valor TRUE) se invoca al Fabricante 2. El valor inicial de esta operación es FALSE (invoca, por defecto, al Fabricante 1).

6.1.2. Caso Comercio Electrónico

Este caso es un ejemplo común de implementación de aplicaciones SOA en el contexto de comercio electrónico (ver Figura 6.3), el cual comprende tres procesos de negocio (que constituirá los servicios en la composición):

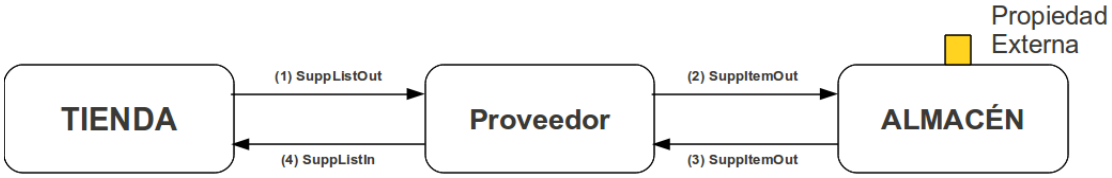


Figura 6.3: Ejemplo de composición de servicios de la aplicación de Comercio Electrónico

- **Tienda:** es la tienda donde los usuarios compran los productos.

- **Proveedor:** ofrece productos a la tienda; necesita comprobar en su almacén las disponibilidades, antes de emitir una respuesta a la tienda.

- **Almacén:** es donde los productos se almacenan. Este proceso tiene un acuerdo de servicio (SLA) con el Proveedor, que consiste en que al menos un producto de la lista debe ser devuelto cuando se realiza una solicitud al Almacén. Puede realizar búsquedas en sitios externos para comprar productos utilizando la propiedad externa ExternalSearch (búsqueda

Externa).

A continuación, se describen las interacciones derivadas de la composición de servicios, para esta aplicación:

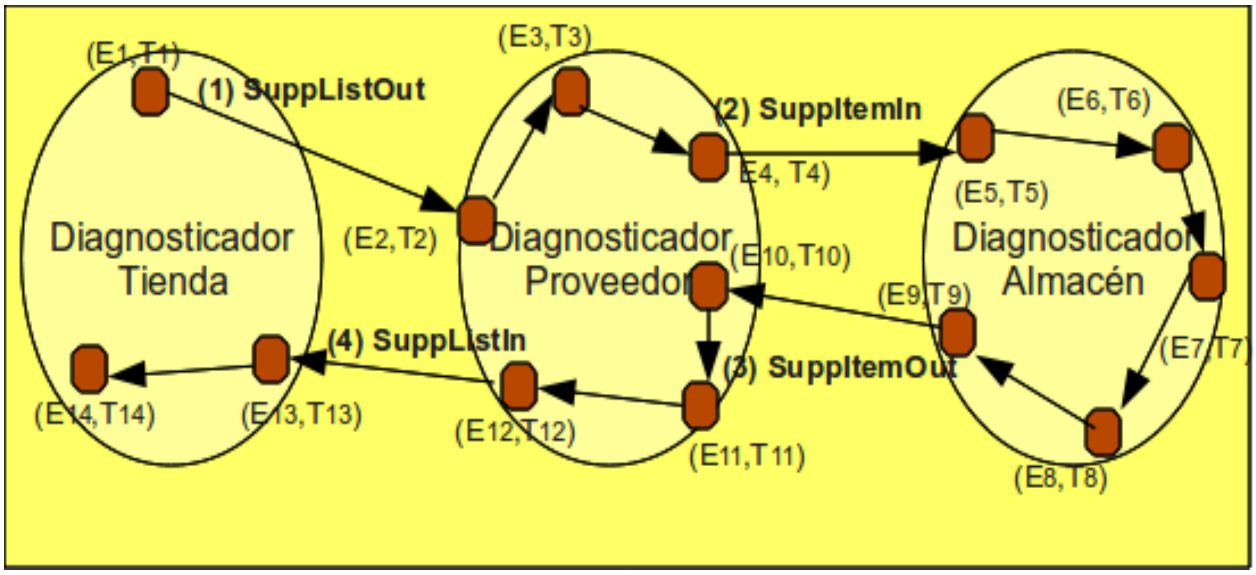
(1) SuppListOut: La tienda envía la lista de los productos que requiere al Proveedor.

(2) SuppItemIn: El Proveedor comprueba la disponibilidad de los productos en su depósito invocando al Almacén.

(3) SuppItemOut: El Almacén proporciona la respuesta sobre la lista de productos en el depósito al Proveedor, el cual debe contener al menos un producto.

(4) SuppListIn: El Proveedor notifica a la Tienda de los productos que puede ofrecer.

Como en el caso anterior, nuevamente se caracterizan la distribución de los eventos entre los diferentes diagnosticadores (sitios) que forman parte de la composición, para construir una crónica genérica para esta aplicación en particular. Además, por razones prácticas, se considera que el tiempo es medido en segundos, y el retraso en las comunicaciones y el tiempo de reconocimiento de crónicas son insignificantes. La secuencia de eventos de esta crónica genérica es:



Tienda	Proveedor	Almacén
● **E_1:** La Tienda provee la orden de productos al Proveedor. ● **E_{13}:** La Tienda recibe la lista de los productos. ● **E_{14}:** La Tienda realiza el pago de los productos.	● **E_2:** El Proveedor recibe la orden de los productos. ● **E_3:** El Proveedor verifica los productos se encuentran en el catalogo, si se encuentran en el catalogo se invoca al Almacén, de lo contrario invoca a la Tienda con una lista vacía de productos. ● **E_4:** El Proveedor provee la orden al Almacén. ● **E_{10}:** El Proveedor recibe la respuesta de los productos. ● **E_{11}:** El Proveedor hace la factura. ● **E_{12}:** El Proveedor responde a la Tienda con los productos enviados.	● **E_5:** El Almacén recibe el requerimiento del Proveedor. ● **E_6:** El Almacén busca los productos. (para esta operación puede invocar otros almacenes). ● **E_7:** El Almacén actualiza el inventario. ● **E_8:** El Almacén empaqueta los productos y envía (packs y ships) los productos al comprador. ● **E_9:** El Almacén responde el requerimiento al Proveedor de los productos solicitados que se encuentran en el almacén.

Figura 6.4: Eventos de la Composición de Comercio Electrónico por diagnosticador

6.1.2.1 Implementación de la Aplicación de Comercio Electrónico en OpenESB

Para verificar el funcionamiento del middleware, en el servicio Almacén se añadieron cuatro operaciones adicionales para inducir fácilmente fallas de violación de SLA (Almacén operaciones **"*Buscar Productos*"** y **"*empaqueta y envía los productos*"**) y de retardo (Delay):

- **tuneSearchBehavior:** es usada para inducir la falla en el evento de la operación Buscar Productos (E_6: Search product). Si está activado (**tuneSearchBehavior = TRUE**), la operación **"Buscar Productos"** no produce resultados en la búsqueda (retorna lp = 0). En caso contrario, la operación **"Buscar Productos"** funciona normalmente. Por ejemplo, si **tuneSearchBehavior = FALSE** y recibe el requerimiento de buscar 20 productos (lpin = 20), la operación **"Buscar Productos"** retorna todos los productos encontrados (lpout = 20); por el contrario, si **tuneSearchBehavior = TRUE** y recibe el mismo requerimiento de buscar 20 productos (lpin = 20), la operación **"Buscar Productos"** no encuentra ningún producto (lpout = 0).

- **tuneShip:** es utilizado para inducir la falla en el evento de la operación **"empaqueta y envía los productos"** (E_8: ship and pack). Si está activado (**tuneShip = TRUE**), la operación deja un producto sin empaquetar de todos los requeridos. Para mostrar el funcionamiento de esta operación suponga que se coloca **tuneShip = FALSE** y recibe la solicitud de empacar 20 productos (lpin = 20), la operación retorna los 20 productos empaquetados (lpout = 20). De lo contrario, suponga que **tuneShip** es cambiado a **TRUE** y recibe el mismo requerimiento de

empacar 20 productos (lpin = 20), la operación retorna solo 19 productos empaquetados, dejando un producto sin empacar (lpout = 19).

- **setTuneDelay:** Se utiliza para inducir un tiempo de retardo en el servicio Almacén. La operación recibe como entrada la variable delay (retardo), que es un entero positivo que define la cantidad de tiempo que debe esperar el servicio Almacén para dar su respuesta. El valor inicial de delay es 0 (retardo inicial es 0 ms, sin retardo). Si la operación **setTuneDelay** es invocada con la variable delay igual a 6000, el servicio Almacén esperara 6000 ms para generar su respuesta.

- **ExternalSearch:** Se utiliza para indicarle al Almacén que puede realizar búsqueda de productos en otros Almacenes. El funcionamiento de esta operación funciona de la siguiente manera: Si el valor de **ExternalSearch = FALSE**, el actor almacén solo realiza la búsqueda de los productos internamente, y en el caso de que no consiga ninguno de los productos requeridos devuelve el valor de lp = 0. En el caso contrario, si **ExternalSearch = TRUE** el actor Almacén puede realizar la búsqueda de los productos en los almacenes vecinos de la misma compañía, permitiéndole mayor probabilidad de éxito de encontrar los productos requeridos.

- **SetShip:** Se utiliza para modificar la operación de empaqueta y envía los productos (E_8) provista por la empresa 1 (empresa encargada de realizar los envíos) por otra operación equivalente (E_8') provista por la empresa 2. El funcionamiento de esta operación funciona de la siguiente manera: Si el valor de **ship = TRUE** se utiliza la operación por defecto E_8 de la empresa 1. En el caso contrario, **ship = FALSE**, se utiliza la operación E_8'. Cuando recién comienza la operación de comercio electrónico, la

operación que funciona por defecto es la provista por la empresa 1 (E_8).

6.2. Crónicas Distribuidas para el Reconocimiento de Fallas

Para verificar el funcionamiento de la extensión del paradigma de crónicas para el reconocimiento distribuido, se ha seleccionado el caso de estudio de Comercio Electrónico definido en la sección 6.1.2.

6.2.1. Especificación de las Crónicas para Detectar Fallas

A Continuación se definen la crónicas específicas que describen las fallas que se quieren diagnosticar (los patrones de fallas). Se considera los siguientes escenarios de fallas:

- La falla ocasionada por la violación del contrato del servicio Almacén (violación de SLA), la cual es una falla en el servicio web.
- La falla debido al retraso de tiempo (time delay) para proveer una respuesta a la solicitud del servicio Almacén (Almacén: Retardo del Servicio), la cual es una falla en el flujo de la composición.

La detección de estas dos fallas son interesantes, ya que permiten evaluar la capacidad del sistema para detectar fallas, tanto de servicio (local) como de la composición (global).

Desde los eventos de la composición, se construyen las crónicas específicas para las dos fallas (los patrones de las dos fallas). En principio, cada crónica específica se descompone en las mismos tres sub-crónicas por cada servicio definidas anteriormente (sección 6.1.2). En algunas ocasiones, para una situación dada, se puede necesitar menos sub-crónicas, como en el caso de las fallas que

se están estudiando (ver Figura 6.5, que muestra la estructura de las crónicas especificas para las fallas de violación de SLA por parte del servicio Almacén y retardo en el servicio Almacén).

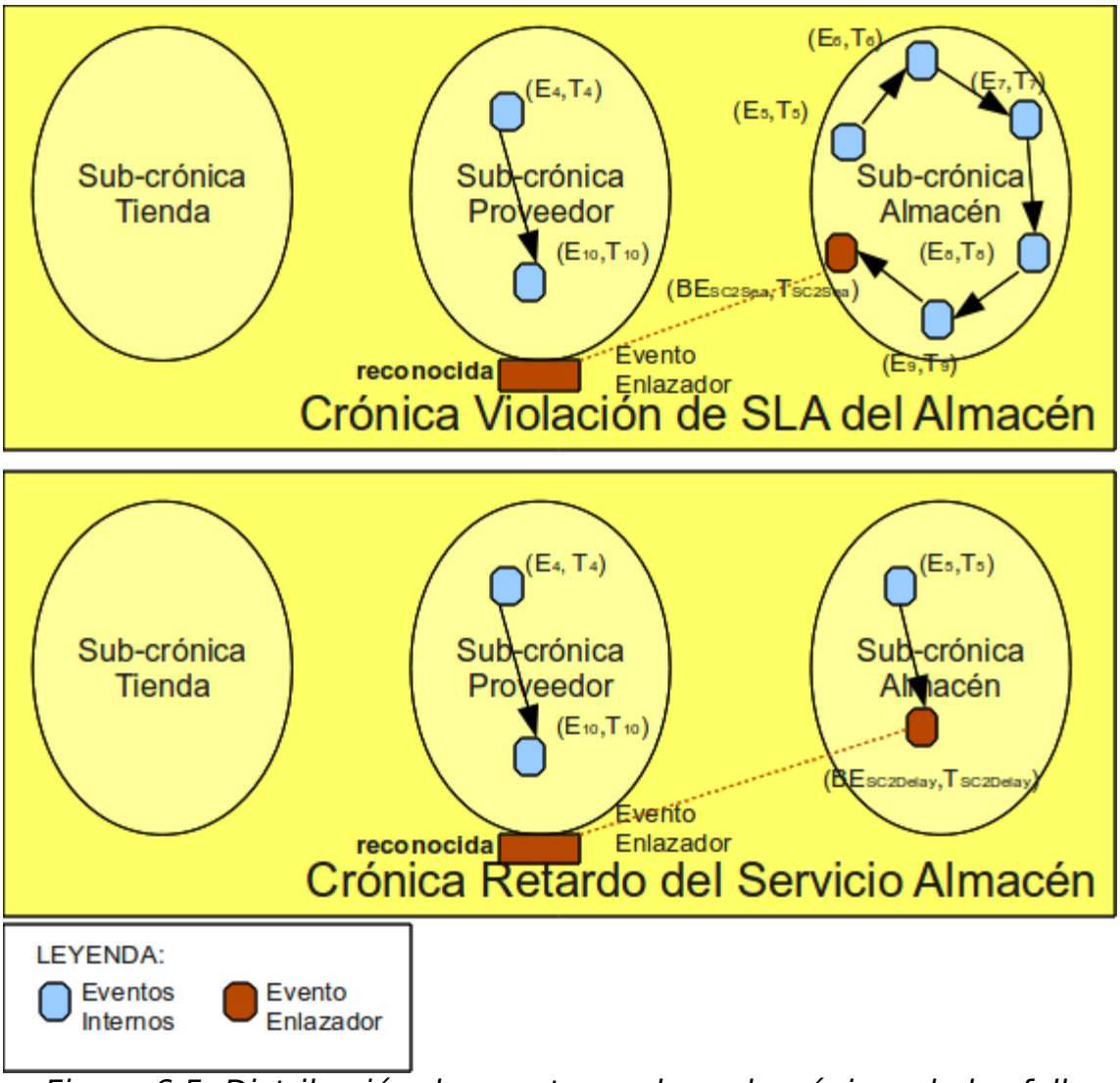


Figura 6.5: Distribución de eventos en las sub-crónicas de las fallas violación de SLA y retardo del servicio del Almacén

Para diseñar el modelo de la Crónica para la violación de SLA por parte del servicio Almacén, se analizan los eventos en el Almacén: el problema de violación

de SLA ocurre cuando el servicio lleva a cabo la búsqueda de productos en el evento E_6 (Buscar Productos), no consigue nada y no invoca otros Almacenes; o el servicio falla en el empaquetado y envío (empaqueta y envía) de productos al vendedor (E_8). En todos esos casos, emite la respuesta al Proveedor con una lista vacía de productos. Entonces, el servicio Proveedor detecta la falla en el evento E_{10} (Proveedor recibe la respuesta de los productos). Por esta razón, la sub-crónica en el Proveedor cuando es reconocida envía el evento enlazador BE_{SC2Sea} al diagnosticador en el servicio Almacén (realmente, el CRS en el diagnosticador del Proveedor envía el mensaje), entonces ese diagnosticador puede reconocer la falla de violación de SLA. Cuando el diagnosticador en el Almacén reconoce la sub-crónica, invoca al reparador en ese sitio con la falla encontrada.

Para el caso de la falla de violación de SLA (caso de Buscar Productos), la crónica se muestra en la Figura 6.6, donde pl es la lista de productos. La sub-crónica en el proveedor detecta que existe una violación de SLA cuando recibe la respuesta de la lista de productos vacía (lp = 0) del Almacén en la operación E_{10}, y el tiempo máximo esperado para recibir el evento E_{10}, luego de haber sucedido E_4, es de 9 seg.

Al reconocer la sub-crónica, el Proveedor genera el evento enlazador BE_{SC2Sea} hacia el diagnosticador en el Almacén, permitiendo reconocer la falla e invocar el reparador. En esta crónica, se han establecido las siguientes restricciones temporales:

Sub-chronicle Tienda-BuscarProductos { **Events{ }** **Constraints{ }** **When recognized{ }** **}**	**Sub-chronicle Proveedor-BuscarProductos {** **Events{** event(E_4: pl > 0, T_4), event(E_{10}: pl = 0, T_{10}) **}** **Constraints{** T_{10}-T_4 ≤ 9 **}** **When recognized{** Emit event(BE_{SC2Sea},T_{SC2Sea}, Diagnoser 3) **}** **}**	**Sub-chronicle Almacén-BuscarProductos {** **Events{** event(E_5; pl > 0, T_5), event(E_6 : pl = 0, T_6), event(E_7: pl = 0, T_7), event(E_8: pl = 0, T_8), event(E_9: pl = 0, T_9), event(BE_{SC2Sea}, T_{SC2Sea}) **}** **Constraints{** T_6-T_5 ≤ 1 T_7-T_6 ≤ 2 T_8-T_7 ≤ 1 T_9-T_8 ≤ 1 T_{SC2Sea}-T_9 ≤ 1 **}** **When recognized{** repairer Invoke(Warehouse, 'SLA-Violation' **}** **}**

Figura 6.6: Modelo de Crónica distribuida para violación de SLA por parte del servicio Almacén (caso de la operación Buscar Productos)

- El evento E_6 debe suceder en un tiempo menor o igual a 1 seg después de ocurrido E_5 (T_6 - T_5 ≤ 1).
- El evento E_7 debe suceder en un tiempo menor o igual a 2 seg después de ocurrido E_6 (T_7 - T_6 ≤ 2).
- El evento E_8 debe suceder en un tiempo menor o igual a 1 seg después de ocurrido E_7 (T_8 - T_7 ≤ 1).

- El evento E_9 debe suceder en un tiempo menor o igual a 1 seg después de ocurrido E_8 ($T_9 - T_8 \leq 1$).

- El evento enlazador E_{SC2Sea} debe suceder en un tiempo menor o igual a 1 seg después de ocurrido E_9 ($T_{SC2Sea} - T_9 \leq 1$).

En el caso de la segunda falla, el diseño de la crónica “Almacén: Retardo del Servicio” es más compleja (ver Figura 6.7). Se considera que el servicio Almacén presenta un retardo (delay) cuando este emite la respuesta al Proveedor después de transcurridos 10 segundos (se permite que el tiempo transcurrido entre invocar el almacén con la lista de productos E_4 y recibir su respuesta en el evento E_{10} sea menor a 10 seg; y se considera retardo cuando se excede el tiempo de la operación normal (mayor a 10 seg), y este comportamiento se repite más de 2 veces en 150 seg. Así, la sub-crónica en el proveedor detecta cuando el evento E_{10} no ocurre a tiempo (mayor a 10 seg) después de ocurrido el evento E_4, y envía el evento $BE_{SC2SeaDelay}$ al diagnosticador del Almacén cada vez que detecta el retardo. En cuanto al diagnosticador en el Almacén, este reconoce la falla cuando recibe más de 2 veces el evento $BE_{SC2SeaDelay}$, y previamente ha recibido el evento E_5 (Almacén recibe el requerimiento del Proveedor). Cuando el Almacén reconoce la sub-crónica, invoca al reparador local con la falla encontrada.

En la crónica "Almacén: Retardo del Servicio" se establecen las restricciones temporales en la sub-crónica del Proveedor en los eventos E_4 y E_{10}, que deben tener un tiempo de ocurrencia mayor o igual 10 seg (T_{10}-$T_4 \geq 10$). Por consiguiente, la restricción que se viola en “Retardo del Servicio” es caracterizada por T_{10}-$T_4 > 10$, que es la que debe reconocer la crónica. Adicionalmente, se ha establecido que el evento enlazador $E_{SC2SeaDelay}$ debe ser recibido en un tiempo mayor a E_5 ($T_{SC2SeaDelay} > T_5$).

Sub-chronicle Tienda-Retardo {	Sub-chronicle Proveedor-Retardo {	Sub-chronicle Almacén-Retardo {
Events{ }	Events{	Events{
Constraints{ }	event(E_4; pl > 0, T_4),	occurs((3,100), {event(E_5; pl = *, T_5), event($BE_{SC2SeaDelay}$, $T_{SC2SeaDelay}$)}, (T_5, T_5+150))
When recognized{ }	event(E_{10}; pl > 0, T_{10})	}
}	}	Constraints{
	Constraints{	$T_{SC2SeaDelay}$ > T_5
	T_{10}-T_4 > 10	}
	}	When recognized{
	When recognized{	repairer Invoke(Warehouse, 'Delay'
	Emit event($BE_{SC2SeaDelay}$,$T_{SC2SeaDelay}$, Diagnoser 3)	}
	}	}
	}	

Figura 6.7: Modelo de Crónica para el retardo del servicio Almacén

6.2.2. Descripción del Reconocimiento de la Crónica para Detectar la Falla de Violación de SLA por parte del Almacén

En el primer caso, suponga que ocurren la siguiente secuencia de eventos en la aplicación comercio electrónico:

Eventos para el servicio Tienda: $E_1(T_1 = 1, Ip=3)$

Eventos para el servicio Proveedor: $E_2(T_2 = 2, Ip=3)$, $E_3(T_3 = 3, Ip=3)$, $E_4(T_4 = 5, Ip=3)$, $E_{10}(T_{10} = 12, Ip=0)$

Eventos para el servicio Almacén: $E_5(T_5 = 6, Ip= 3)$, $E_6(T_6 = 7, Ip=0)$, $E_7(T_7 = 9, Ip=0)$, $E_8(T_8 = 10, Ip=0)$, $E_9(T_9 = 11, Ip=0)$, $E_{SC2Sea}(T_{SC2Ses} = 12)$

Para este flujo de eventos, el Algoritmo 4.1 (Capítulo 4) reconoce la falla de violación de SLA en el diagnosticador del servicio Almacén usando la crónica para violación de SLA. La secuencia de eventos que son detectados y las instancias reconocidas de crónicas son mostradas en la Figura 6.8. El CRS en el diagnosticador Proveedor reconoce la sub-crónica "Proveedor-BuscarProductos"

porque es instanciada en {$E_4(T_4 = 5, pl=3)$ y $E_{10}(T_{10}=12, pl=0)$}, es decir el Almacén provee una lista vacía de productos. Entonces, el diagnosticador Proveedor emite el evento E_{SC2Sea} al diagnosticador en el Almacén, el cual reconoce la instancia de la sub-crónica "Almacén-BuscarProductos" {$E_5(T_5 = 6, lp= 3)$, $E_6(T_6 = 7, lp=0)$, $E_7(T_7 = 9, lp=0)$, $E_8(T_8 = 10, lp=0)$, $E_9(T_9 = 11, lp=0)$, $E_{SC2Sea}(T_{SC2Ses} = 12)$}.

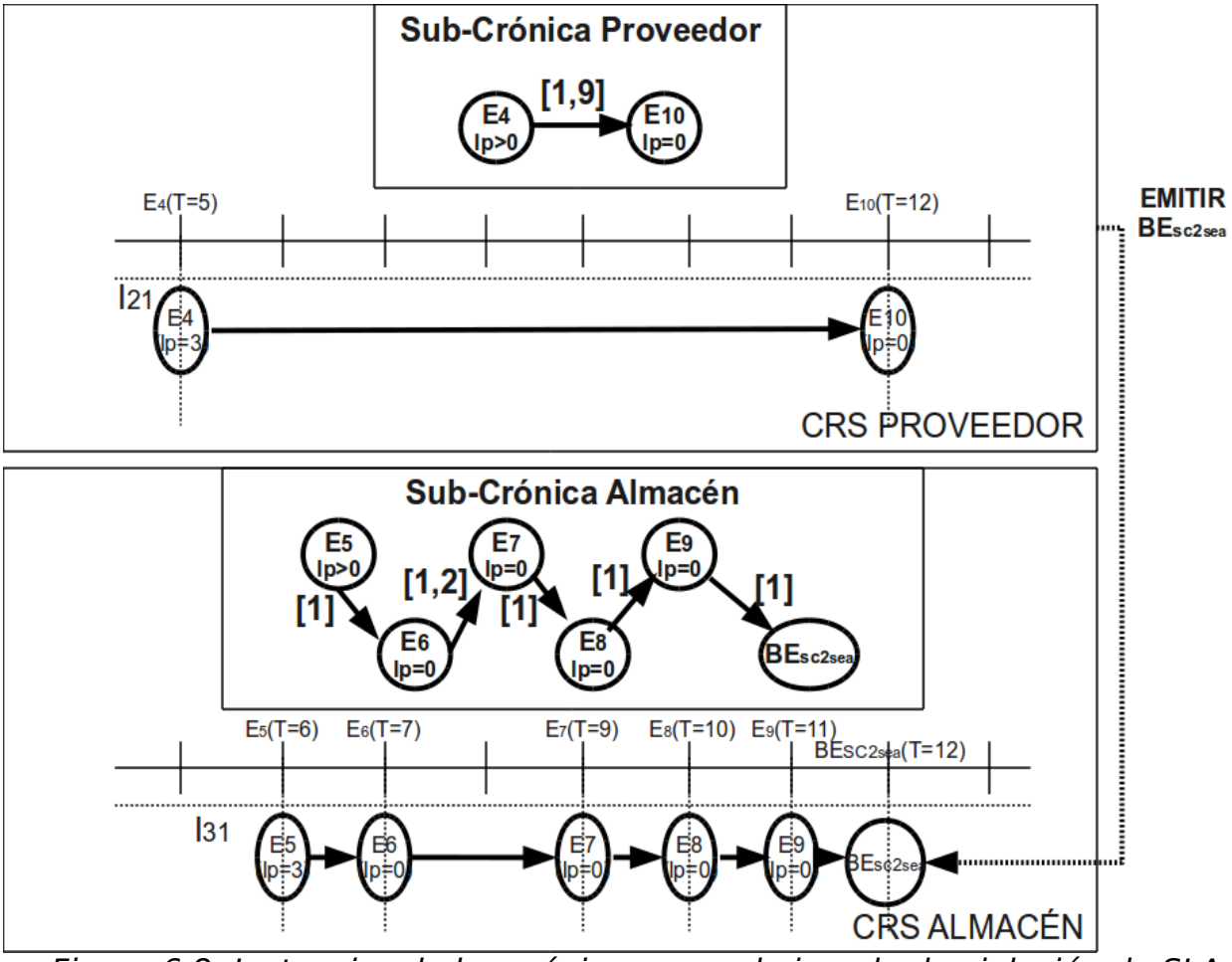


Figura 6.8: Instancias de las crónicas para el ejemplo de violación de SLA por parte del Almacén.

6.2.3. Descripción del Reconocimiento de la Crónica para Detectar la Falla de Retardo en el Servicio Almacén

Ahora se considera la secuencia de eventos en la aplicación de comercio electrónico:

Eventos para el servicio Tienda: $E_1(T_1 = 1, Ip=1)$, $E_1(T_{11} = 15, Ip=2)$, $E_1(T_1 = 26, Ip=4)$.

Eventos para el servicio Proveedor: $E_2(T_2 = 2, Ip=1)$, $E_3(T_3 = 3, Ip=1)$, $E_4(T_4 = 5, Ip=1)$, $E_{10}(T_{10} = 15, Ip=1)$, $E_2(T_2 = 16, Ip=2)$, $E_3(T_3 = 17, Ip=2)$, $E_4(T_4 = 19, Ip=2)$, $E_{10}(T_{10} = 29, Ip=2)$, $E_2(T_2 = 32, Ip=4)$, $E_3(T_3 = 33, Ip=4)$, $E_4(T_4 = 34, Ip=4)$, $E_{10}(T_{10} = 45, Ip=4)$.

Eventos para el servicio Almacén: $E_5(T_5 = 6, Ip= 1)$, $E_6(T_6 = 9, Ip=1)$, $E_7(T_7 = 11, Ip= 1)$, $E_8(T_8 = 13, Ip=1)$, $E_9(T_9 = 14, Ip=1)$, $E_{SC2Delay}(T_{SC2Delay} = 15)$, $E_5(T_5 = 20, Ip= 2)$, $E_6(T_6 = 21, Ip=2)$, $E_7(T_7 = 26, Ip= 1)$, $E_8(T_8 = 27, Ip=2)$, $E_9(T_9 = 28, Ip=2)$, $E_{SC2Delay}(T_{SC2Delay} = 29)$, $E_5(T_5 = 34, Ip= 4)$, $E_6(T_6 = 40, Ip=4)$, $E_7(T_7 = 41, Ip= 4)$, $E_8(T_8 = 43, Ip=4)$, $E_9(T_9 = 44, Ip=4)$, $E_{SC2Delay}(T_{SC2Delay} = 45)$.

La secuencia de eventos que son detectados y las instancias reconocidas de crónicas son mostradas en la Figura 6.9. El diagnosticador Proveedor reconoce 3 instancias de la sub-crónica "Proveedor-Delay"; I_{21}: $\{E_4(T_4 = 5), E_{10}, (T_{10} = 15)\}$, I_{22}: $\{E_4(T_4=19), E_{10}, (T_{10} = 29)\}$ y I_{23}: $\{E_4(T_4 = 34), E_{10}, (T_{10} = 45)\}$. Entonces, el diagnosticador Proveedor emite el evento enlazador $EB_{SC2SeaDelay}$ 3 veces al diagnosticador en el Almacén para informarle de que hay un problema de retardo de respuesta. De este modo, el diagnosticador Almacén puede reconocer la sub-crónica "Almacén-Delay" debido a la siguiente secuencia de eventos $\{E_5(T_5 = 6), E_{SC2Delay}(T_{SC2Delay} = 15), E_5(T_5 = 20),,E_{SC2Delay}(T_{SC2Delay} = 29), E_5(T_5 = 35), E_{SC2Delay}(T_{SC2Delay} = 45)\}$, ya que se produce más de 2 veces en menos de 150 segundos.

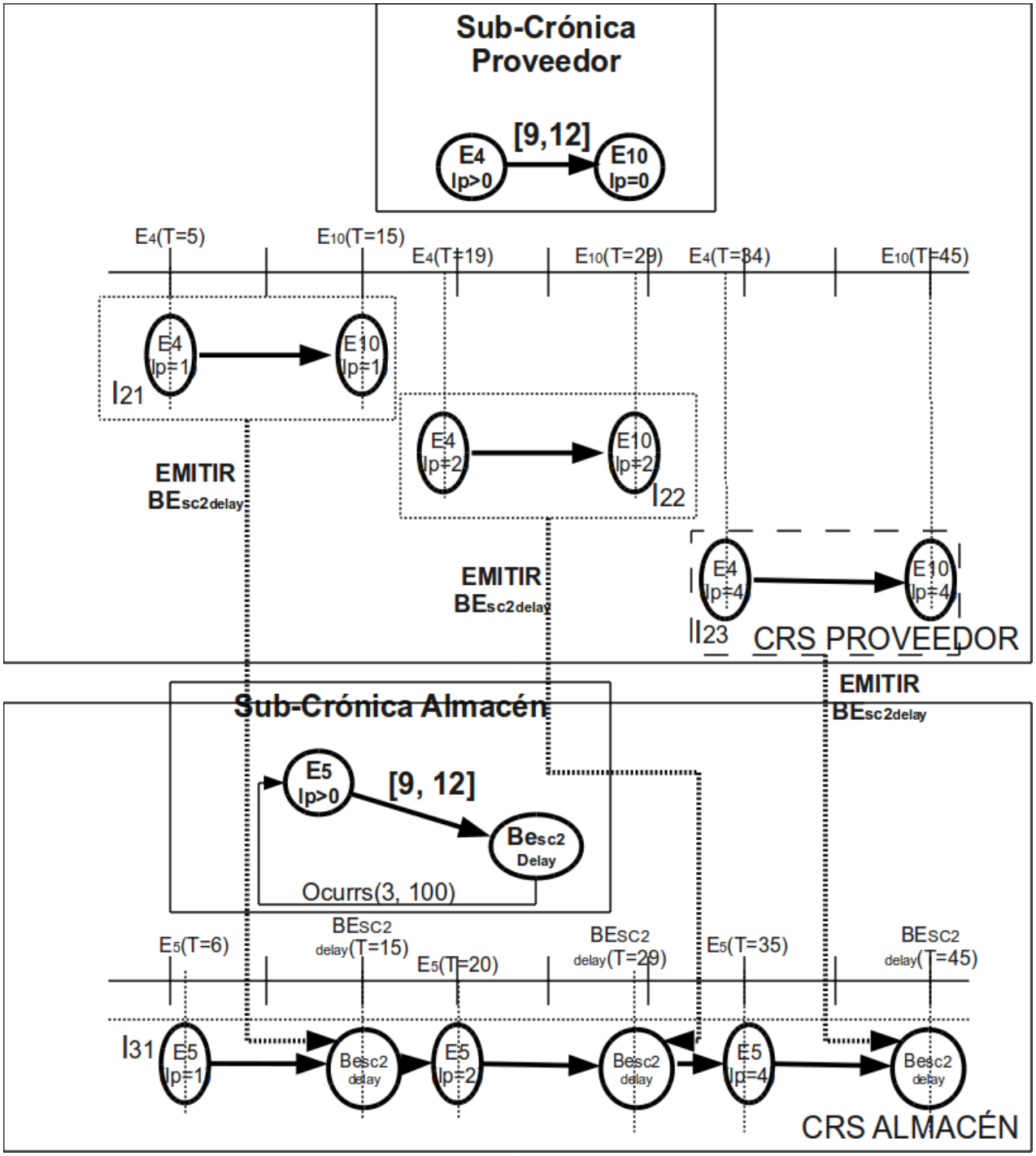


Figura 6.9: Instancias de las crónicas para el ejemplo de retardo del servicio Almacén

6.2.4. Análisis de Resultados

En este experimento se ha puesto a prueba la capacidad de detección de fallas del Algoritmo 4.1, usando crónicas distribuidas. En el primer caso, a nivel de una falla en el servicio (problemas en el mismo servicio, lo que se traduce en una falla local); y en el segundo caso una falla en la composición (característico de entornos distribuidos). Los eventos fueron inducidos, para probar la capacidad de reconocimiento de las crónicas. Esto demuestra la versatilidad del enfoque para abordar los diferentes aspectos de una aplicación distribuida.

La extensión del formalismo de las crónicas y la distribución de los CRS (Chronicle Recognition System) entre los servicios que forman parte de la composición facilita la interacción de los diagnosticadores locales, haciendo posible el reconocimiento de la crónica global sin necesidad de un coordinador para gestionar sus interacciones. A nivel de comunicación, esto representa una mejora notable sobre los mecanismos mostrados en otros estudios [10, 12, 15, 16, 32, 71]. Además, la aplicación del mecanismo en los casos estudiados es natural (un reconocedor por servicio). Asimismo, al ser un enfoque distribuido el problema de escalabilidad de la aplicación puede ser manejada adecuadamente.

6.3. Construcción de Crónicas Distribuidas para el Diagnóstico de Fallas en Sistemas Distribuidos Utilizando el Lenguaje CQL

Para comprobar el uso del lenguaje CQL en la implementación del motor de reconocimiento de crónicas, nuevamente se toma el caso de prueba de Comercio electrónico. Se consideran los siguientes escenarios de fallas para la violación del contrato del servicio SLA por parte del Almacén:

- Búsqueda de Productos (E_6).
- Empaqueta y envía (evento E_8).

Nuevamente, se utilizan las crónicas específicas de SLA violation (Warehouse search products y packs and ships) diseñadas en la sección anterior. Para la falla de violación de SLA en el Almacén (caso Búsqueda de Productos), la crónica se muestra en la Figura 6.10 (la figura muestra en la parte superior la notación clásica de crónicas definida por Dousson y luego reescrita en el lenguaje CQL).

Crónica de falla de violación de SLA (caso de Búsqueda de Productos) en predicados reificados (CRS [10, 32], CarDeCRS [12] y Matrac [71])

```
Subchronicle Proveedor SLA-BuscarProductos {
  Events{
     event(E4, T4, lp > 0),
    event(E10, T10, lp = 0)
  }
  Constrains{
     T10 ≥ T4
  }
  When recognized{
     Emit event(BESLA,TBESLA) to
   Diagnoser Warehouse
  }
}
```

```
Sub-chronicle Almacén SLA-BuscarProductos {
  Events{
    event(E5; pl > 0, T5 ),
    event(E6 : pl = 0, T6),
    event(E7: pl = 0, T7 ),
    event(E8: pl = 0, T8 ),
    event(E9: pl = 0, T9),
    event(BESC2Sea, TSC2Sea)
  }
  Constraints{
    T6 ≥ T5
    T7 ≥ T6
    T8 ≥ T7
    T9 ≥ T8
    TSC2Sea ≥ T9
  }
  When recognized{
    repairer Invoke(Warehouse, 'SLA-Violation')
  }
}
```

Crónica de falla de violación de SLA (caso de Búsqueda de Productos) en el lenguaje CQL

```
Subchronicle Proveedor SLA {
  SELECT
     ISTREAM(id => E4.id, event => 'BESLA', time
=> E10.time, lpsupplier => E10.lp, lp => E4.lp, to
=> 'Diagnoser Almacén')
  FROM
     E4[10], E10[now]
  WHERE
     E10.time >= E4.time AND E10.id = E4.id AND
E4.lp - E10.lp > 0
  }
```

```
Subchronicle Almacén SLA Búsqueda de Productos {
  SELECT
     ISTREAM(id => E5.id, fault =>
'ServiceLevelAgreement', faulttype =>
'SearchProducts', time => EBESLA.time, lp =>
E6.lp, to => 'Repair Almacén')
  FROM
     E5[10], E6[8], E7[7], E8[5], E9[3],
EBESLA[now]
  WHERE
     E6.time >= E5.time AND E7.time >= E6.time AND
E8.time >= E7.time AND E9.time >= E8.time AND
EBESLA.time >= E9.time AND E5.lp > 0 AND E6.lp = 0
AND E6.lp = E7.lp AND E8.lp = E7.lp AND E6.id =
E5.id AND E7.id = E6.id AND E8.id = E7.id AND
EBESLA.id = E7.id
  }
```

Figura 6.10: Modelo de Crónica para la falla de violación de SLA en el Almacén (caso Búsqueda de Productos)

Donde, el atributo pl en los eventos corresponde a la lista de los productos.

Analicemos el modelo crónica de la figura 6.10 para el caso de estudio. Este ejemplo nos permite mostrar la versatilidad del lenguaje CQL para manejar

restricciones directamente en las variables atemporales. En concreto, se nota la diferencia en la forma en que el valor del atributo lp en la sub-crónica del Proveedor para el caso de SLA es representado en predicados temporales (es necesario colocar un sensor previo para indicar que el valor de lp en el evento E_{10} es diferente (pl = 0) con respecto al del evento E_4 (pl > 0), y en CQL (se representa muy fácil, como la diferencia del atributo lp para los eventos E_4 y E_{10}, E_4.lp – E_{10}.lp > 0). Los motores de reconocimiento de crónicas CRS [10, 32], CarDeCRS [13] y Matrac [71] no permiten el uso de operadores matemáticos en las restricciones de las variables atemporales (por ejemplo, operadores matemáticos sobre un atributo s como E.s ≥ E_1.s * 5), haciendo muy complejo la gestión de valores de atributos tipo lp en diferentes eventos (para esto, se necesita indicar cuando el valor de lp es correcto (pl > 0) y cuando es erróneo (pl = 0)). Usando CQL se enriquece la manera como se representan las crónicas, ya que permite el uso de los operadores matemáticos en las restricciones de las variables atemporales, como en el ejemplo anterior.

El modelo de Crónica para la falla de violación de SLA en el Almacén (caso Empaqueta y envía) es muy similar al caso anterior, y se muestra en el lenguaje CQL en la Figura 6.11, allí se ve que el atributo pl=0 en el evento E_8 (el Almacén falla al realizar el empaquetado y envío de los productos) es menor que en el evento E_7 (el Almacén).

```
Subchronicle Proveedor SLA {
  SELECT
    ISTREAM(id => E4.id, event => 'BESLA', time => E10.time, lpsupplier => E10.lp, lp => E4.lp, to => 'Diagnoser 3')
  FROM
    E4[10], E10[now]
  WHERE
    E10.time > E4.time AND E10.id = E4.id AND E4.lp - E10.lp > 0
}
```

```
Subchronicle Almacén SLA Empaqueta y envía {
  SELECT
    ISTREAM(id => E5.id, fault => 'ServiceLevelAgreement', faulttype => 'Ship and Packed', time => EBESLA.time, lp => E8.lp, to => 'Repair 3')
  FROM
    E5[10], E6[8], E7[7], E8[5], EBESLA[now]
  WHERE
    E6.time > E5.time AND E7.time > E6.time AND E8.time > E7.time AND EBESLA.time > E8.time AND E5.lp > 0 AND E5.lp = E6.lp AND E6.lp = E7.lp AND E8.lp < E7.lp AND E6.id = E5.id AND E7.id = E6.id AND E8.id = E7.id AND EBESLA.id = E7.id
}
```

Figura 6.11: Modelo de Crónica para la falla de violación de SLA en el Almacén (caso de Empaqueta y envía)

En general, según las sub-crónicas, existe un acuerdo de que la lista de los productos suministrada por el proveedor al almacén debe ser igual a la cantidad de los productos recibidas por el Proveedor (E_8.lp = E_7.lp), de lo contrario se considera una violación del acuerdo de SLA (E_8.lp < E_7.lp). En este orden de ideas, note que la sub-crónica en el Proveedor para la falla de violación de SLA en ambos casos es idéntica: "Búsqueda de Productos" y "Empaqueta y envía", debido a que el servicio Almacén esta cometiendo una violación de SLA. Al informar el diagnosticador Proveedor al Almacén con el evento enlazador EB_{SLA}, es que el diagnosticador puede reconocer cual falla es la que esta aconteciendo en realidad ("Búsqueda de Productos" o "Empaqueta y envía"), al revisar los diferentes valores de lp en los eventos.

Para facilitar la lectura de la información generada por los eventos enlazadores

y el diagnóstico de las fallas, el contenido de los archivos de las Figura 6.10 y Figura 6.11 se describe a continuación:

- **Evento enlazador:**
  - ***id:*** Corresponde al identificador de la invocación de la aplicación comercio electrónico (ej: id = 4).
  - ***event:*** Indica el evento enlazador que es generado cuando la sub-crónica es reconocida.
  - ***time:*** Corresponde al tiempo en milisegundos cuando es generado el evento enlazador.
  - ***lp:*** La lista de productos (lp) cuando el evento enlazador es generado.
  - ***lpsupplier:*** La lista de productos (lp) del evento E_{10}.
  - ***Timestamp:*** corresponde al tiempo en formato de tiempo entendido por los humanos.
- **Diagnóstico de fallas:**
  - ***id:*** corresponde al identificador de la invocación de la aplicación comercio electrónico (ej: id = 5).
  - ***fault:*** Nombre de la falla que es diagnosticada (ej: SLA).
  - ***faulttype:*** corresponde al tipo de falla, es usado generalmente para agregar información en el diagnóstico (ej: SearchProducts).
  - ***time:*** corresponde al tiempo en milisegundos cuando es generado el

diagnóstico.

- ***lp:*** La lista de productos (lp) que tiene el evento enlazador que genero el diagnóstico.
- ***Timestamp:*** corresponde al tiempo cuando el diagnóstico de la falla es realizado, se presenta en formato de tiempo entendido por los humanos.

6.3.1. Implementación de las Crónicas Utilizando el Lenguaje CQL

Para probar el mecanismo de reconocimiento de crónicas distribuidas desarrollada en este trabajo, se implementaron los modelos de crónicas para las fallas de violación de SLA del Almacén desarrolladas en las Figura 6.10 y Figura 6.11, usando el lenguaje CQL. Así, la composición de la aplicación de comercio electrónico fue implementada en OpenESB. En el servicio Almacén se han utilizado las operaciones **tuneSearchBehavior** y **tuneShip** para generar las fallas de violación de SLA, y poder verificar el reconocimiento de las crónicas. Se ejecuta la aplicación de comercio electrónico 9 veces (con diferentes patrones de pruebas, en distintos escenarios de funcionamiento: normal y fallas de SLA), añadiendo un atributo id para cada invocación y poder diferenciarlas. En la Tabla 6.1 se muestran los resultados de las invocaciones con los diferentes valores usados para tuneSearchBehavior y tuneShip.

id	tuneSearchBehavior	tuneShip	Resultado de las Operaciones (lp)												
			E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13
1	FALSE	FALSE	15	15	15	15	15	15	15	15	15	15	15	15	15
2	FALSE	FALSE	30	30	30	30	30	30	30	30	30	30	30	30	30
3	FALSE	FALSE	10	10	10	10	10	10	10	10	10	10	10	10	10
4	***TRUE***	***FALSE***	***8***	***8***	***8***	***8***	**8**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
5	FALSE	FALSE	2	2	2	2	2	2	2	2	2	2	2	2	2
6	FALSE	FALSE	13	13	13	13	13	13	13	13	13	13	13	13	13
7	FALSE	FALSE	14	14	14	14	14	14	14	14	14	14	14	14	14
8	***FALSE***	***TRUE***	***22***	***22***	***22***	***22***	***22***	***22***	***22***	**21**	***21***	***21***	***21***	***21***	***21***
9	FALSE	FALSE	5	5	5	5	5	5	5	5	5	5	5	5	5

Tabla 6.1: Resultado de las invocaciones en la aplicación Comercio Electrónico

Cuando cada sub-crónica es reconocida los eventos generados son almacenados en un archivo. En el caso cuando la sub-crónica SLA se reconoce en el Proveedor (Supplier), el evento generado BE_{SLA} es almacenado en el archivo Supplierdiagnoser.xml, resultando el mismo archivo para ambos casos en que suceden la fallas de SLA (Búsqueda de Productos o Empaqueta y envía) debido a que utilizan la misma sub-crónica, y emite el evento BE_{SLA} a las sub-crónicas del Almacén: *SLA Búsqueda de Productos y SLA Empaqueta y envía*. La sub-crónica *Almacén SLA Búsqueda de Productos* almacena los eventos generados en el archivo Warehousediagnoser.xml, y emite un evento con el contenido de la falla que pueda ser leído por el reparador del Almacén. Igualmente lo hace la sub-crónica *Almacén SLA Empaqueta y envía,* almacena los eventos generados en el archivo Warehousediagnoser1.xml, y emite un evento con el contenido de la falla al reparador en el Almacén. El contenido de los archivos derivados de cada uno de los reconocimiento de las sub-crónicas se muestran en la Figura 6.12:

Figura 6.12: Detección de la Crónica Distribuida para la violación de SLA en los casos de falla en "Búsqueda de Productos" y "Empaqueta y envía" en la aplicación de comercio electrónico

Como se muestra en la Figura 6.12, cuando el funcionamiento de la aplicación de comercio electrónico es normal (id = 1, 2, 3, 5, 6, 7 9) ninguna sub-crónica es activada. De lo contrario, en la invocación de la aplicación en id = 4 se genera el reconocimiento de la falla de violación de SLA en el Almacén (E_{10}.lp = 0), y los resultados son almacenados en el archivo Supplierdiagnoser.xml (primer elemento del árbol XML de la Figura 6.12). Adicionalmente, emite el evento BE_{SLA}, que es leído por el diagnosticador en el Almacén y permite seguir instanciando la sub-crónica *Almacén SLA Búsqueda de Productos*. Esta última sub-crónica emite

el contenido de la falla en forma de evento al reparador en el Almacén (archivo Warehousediagnoser.xml). Del mismo modo, para la falla de violación de SLA Empaqueta y envía (id = 8) la falla es parcialmente reconocida por el diagnosticador en el proveedor (ver contenido del archivo Supplierdiagnoser.xml en la Figura 6.12, segundo elemento del árbol XML), y entonces emite el evento BE_{SLA} al diagnosticador en el Almacén. Con este evento, el diagnosticador puede reconocer la falla utilizando la sub-crónica *Almacén SLA Empaqueta y envía* (ver el archivo Warehousediagnoser1.xml de la Figura 6.12).

6.3.2. Análisis de Resultados

La implementación de las crónicas utilizando el lenguaje CQL, ha proporcionado una mayor expresividad al añadir restricciones en las variables atemporales (para el ejemplo de esta capítulo, no es necesario implementar un traductor que indique si lp es correcto). El lenguaje permite definirlas, de forma natural, desplegando los operadores matemáticos en las restricciones de los atributos (restricciones atemporales). Por ejemplo, en el caso del diagnosticador del Proveedor, la restricción en la lista de productos (lp) entre los eventos E_4 y E_{10} se describe con una expresión matemática como $E_4.lp - E_{10}.lp > 0$. En el caso del Almacén, la restricción de lp para detectar los problemas de "Búsqueda de Productos" es $E_6.lp = 0$, y de "Empaqueta y envía" es $E_8.lp < E_7.lp$. Vemos así que las crónicas distribuidas implementadas con sentencias CQL, detectan las fallas en la composición por la violación de SLA del Almacén en ambos casos ("*Búsqueda de Productos*" y "*Empaqueta y envía*").

6.4. Patrones de Crónicas Distribuidas Genéricos para el Diagnóstico de Fallas en la Composición de Servicios Web

Para verificar los patrones de crónicas distribuidas propuestos en este trabajo

para el diagnóstico de fallas en la composición de los servicios, se han seleccionado el patrón de falla para el diagnóstico de la violación de SLA. Recordemos que esta falla se refiere a detectar problemas en las propiedades no funcionales de la información intercambiada por los servicios. Al realizar la detección de problemas sobre los atributos de los eventos intercambiados por los servicios, los patrones de crónicas para las fallas de SLA definidos en el Capítulo 4, permiten un gran abanico de configuraciones para cada caso (ajustes a nivel de configuración de las restricciones de SLA y QoS, como calidad de los productos, costos, cantidad de productos enviados, entre otros). Esa es la razón por la que se seleccionó ese patrón. Para ello, se han seleccionado los dos casos de estudio (industria del mueble y comercio electrónico), y se han ajustado cuatro patrones de crónicas para detectar problemas en diferentes situaciones (caracterizadas en los atributos de los patrones genéricos que miden variables físicas intercambiados en los mensajes de la aplicación SOA).

6.4.1. Caso Industria del Mueble

Para realizar el ajuste de las crónicas para las fallas de SLA, es necesario definir el conjunto de restricciones que se utilizarán, en base a las propiedades no funcionales expresadas en los atributos presentes en los eventos (mensajes intercambiados). En el caso concreto de la industria del mueble, se han establecido las siguientes propiedades no funcionales:

Calidad (Quality): Todos los productos solicitados por la Tienda deben tener una calidad establecida y debe ser igual a la calidad de los productos recibidos por parte del fabricante (PSvar = PRvar). Para detectar esta falla, los servicios implicados en la crónica son la Tienda y la Fábrica. La crónica para detectar esta falla se deriva a partir del patrón de la Figura 4.22, ya que no se necesita medir la frecuencia de la cantidad de veces que ocurre la violación (es medida en el patrón

de Crónicas de la Figura 4.23). Debido a que la violación de SLA solo puede ser calculada utilizando los eventos E_4 (PSvar) y E_5 (PRvar), la restricción no involucra un solo evento como estipula el patrón de la Figura 4.22, sino un flujo de eventos, por lo que es necesario expandir la crónica con el patrón de la Figura 4.26, para poder agregar la restricción sobre los atributos cuando se comete la violación (E_4.PSvar > E_5. PRvar), como se muestra en la Figura 6.13:

```
Subchronicle Tienda calidad(SLA) {
   Events{
     event(E3, T3)
     event(E6, T6)
     event(BESLAQuality, TSLAQuality)
   }
   Constrains{
     T6 - T3 > 0
     T6 ≤ TSLAORQoS
   }
   When recognized{
      repair invoke(Tienda, 'Service Level Agreement')
   }
}
```

```
Subchronicle Fabrica calidad(SLA) {
   Events{
     event(E4, PSvar T4)
     event(E5, PRvar, T4)
   }
   Constrains{
     T5 - T4 > 0
     E4.PSvar > E5.PRvar
   }
   When recognized{
     emit event(BESLAQuality, TSLAQuality) to Tienda diagnoser
   }
}
```

Figura 6.13: Patrón de Crónica distribuida para la aplicación Fabrica de Muebles con la violación de SLA de la calidad de los productos

donde:

- **Subchronicle Fabrica calidad (SLA)** es generada usando el patrón de la Figura 4.26 (Capítulo 4) con:
  - ***Sub-flow:*** es compuesta por los eventos de la fábrica de muebles E_4 y E_5.
  - ***Temporal Constraint:*** Corresponden a las restricciones temporales del patrón de crónicas de la Figura 4.26 y se muestra en formato de *cursiva* en la Figura 6.13. El flujo de la coreografía estudiado implica el

flujo de los eventos E_4 y E_5. En este caso, es necesario expresar una restricción temporal para asegurar que el evento E_4 se produce en un instante previo que el evento E_5 (T_5 - T_4) en la sub-crónica Fabrica calidad (SLA).

- ***SLA OR QoS Constraint:*** Corresponden a las restricciones de SLA de los atributos, y se muestra en formato ***negrita y cursiva*** en la sub-crónica de Fabrica calidad (SLA) de la Figura 6.13, expresando la violación de la diferencia en la calidad de los productos entre los eventos E_4 y E_5, utilizando para ello los atributos PRvar y PSvar presentes en los eventos (E_5.PSvar < E_4.Prvar) en la sub-crónica Fabrica calidad (SLA).

- **Subchronicle Tienda calidad (SLA)** es generada usando el patrón de Subchronicle S_j en la Figura 4.23 con los eventos E_3, E_6 y $BE_{SLAQuality}$ (emitido como un evento enlazador desde la Subchronicle Fabrica calidad (SLA) de la Figura 6.13, cuando es reconocida).

Supongamos el mismo patrón de crónica, pero ahora para evaluar una restricción de costos de producción. Se añade a **Subchronicle Fabrica calidad (SLA)**:

- Cost (Costo) Constraint: Los costos de los productos suministrados por la Fábrica deben ser menor o igual a un costo preestablecido C_0.

Una vez más, los servicios implicados en la crónica son la Tienda y la Fábrica. Debido a que los valores de las variables no funcionales involucrados corresponde a un solo evento (E_5), la crónica para detectar esta falla se deriva a partir del patrón de la Figura 4.23, y se muestra en la Figura 6.14:

Subchronicle Tienda Costo (SLA) { **Events{** event(E_3, T_3) event(E_6, T_6) event($BE_{SLACost}$, $T_{SLACost}$) **}** **Constrains{** $T_6 - T_3 > 0$ $T_6 \leq T_{SLAORQoS}$ **}** **When recognized{** repair invoke(Tienda, 'Service Level Agreement') **}** **}**	**Subchronicle Fabrica Costo (SLA) {** **Events{** event(E_5, Cost, T_4) **}** **Constrains{** ***E_5.Cost > C_0*** **}** **When recognized{** emit event($BE_{SLACost}$, $T_{SLACost}$) to Tienda diagnoser **}** **}**

Figura 6.14: Patrón de Crónica distribuida para la aplicación Fabrica de Muebles con la violación de SLA del costo de los productos

donde:

- **Subchronicle Fabrica Costo (SLA)** es generada usando el patrón de la Figura 4.23 (Subchronicle S_k) con el evento E_5.
  - ***SLA OR QoS Constraint:*** Son las restricciones de SLA, y se muestra en ***negrita y cursiva*** en la Figura 6.14 en Subchronicle Fabrica Costo (SLA), expresando la violación de que el costo debe ser menor o igual a C_0 en el evento E_5 (E_5.Cost > C_0).
- **Subchronicle Tienda Cost (SLA)** es generada usando el patrón de la Figura 4.23 (Subchronicle S_j) con los eventos E_3, E_6 y $BE_{SLACost}$ (emitido como evento enlazador desde la Subchronicle Fabrica Costo (SLA) de la FFigura 6.14.

Suponga ahora la crónica para evaluar el tiempo de entrega de los productos a la Tienda. La restricción es:

Tiempo de Entrega (Delivery Time): El tiempo de entrega de los productos

a la Tienda debe ser menor que o igual a un tiempo establecido ($TimeDelivery_0$). Esta crónica es similar a la construida previamente para el costo (Figura 6.14), la única modificación es: cambiar el atributo Cost en el evento E_5 por fecha de entrega (E_5.entrega), ajustar la constante C_0 al valor en tiempo $TimeDelivery_0$ en la restricción temporal del patrón de crónica de la Figura 6.14 (E_5.entrega > $TimeDelivery_0$) y generar el evento enlazador $BE_{SLAdelivery}$ hacia la sub-crónica en Tienda.

6.4.1.1 Comprobar la Capacidad de Detección de la Crónica

Como antes, para verificar el reconocimiento de la crónica se implementó el modelo de crónicas usando el lenguaje CQL. Por otro lado, la composición de la aplicación SOA Industria del Mueble fue implementada en OpenESB. Adicionalmente, se utilizó la operación **tuneQualityBehavior** en el servicio Fábrica para generar la falla de violación de SLA de calidad, y poder verificar el reconocimiento del patrón de crónica de la Figura 6.13. En este orden de ideas, se ejecuta la aplicación de la industria del mueble 3 veces, para comprobar el reconocimiento de la falla de SLA con patrones de prueba en distintos escenarios de funcionamiento: normal y fallas de SLA, añadiendo un atributo id que diferencia cada invocación, en la primera invocación (id =1) se produce una falla de SLA, en los demás casos la aplicación funciona sin problemas (id = 2 y id = 3). En la Tabla 6.2 se muestra el resultado de las invocaciones con los diferentes valores utilizados para tuneQualityBehavior.

Id	tuneQualityBehavior	Operation result (Quality)							
		E1	E2	E3	E4 (PSvar)	E5 (PRvar)	E6	E7	E8
1	**TRUE**	**9**	**9**	**9**	***9***	***8***	***8***	***8***	***8***
2	FALSE	23	23	23	23	23	23	23	23
3	FALSE	9	9	9	9	9	9	9	9

Tabla 6.2: Invocaciones de la aplicación de la industria del mueble

Para cada sub-crónica reconocida, los eventos generados son almacenados en un archivo (ver Figura 6.15). En el caso de la Fábrica el evento enlazador $BE_{SLAQuality}$ generado en la Subchronicle Fabrica Costo (SLA) es almacenado en el archivo Manufacturerdiagnoser.xml (almacena cada vez que el evento enlazador $BESLA_{Quality}$ es generado hacia el diagnosticador de la Tienda), y cuando Subchronicle Tienda calidad (SLA) es reconocida en el diagnosticador de la tienda es almacenado en el archivo Retaildiagnoser.xml (emite el evento con el contenido de falla al reparador de la Tienda cada vez que la sub-crónica es reconocida). Así, el contenido de los archivos derivados del reconocimiento de cada sub-crónica al invocar la aplicación de la industria de Muebles se muestran en la Figura 6.15.

Para facilitar la lectura de la información generada por los eventos enlazadores y el diagnóstico de las fallas en la Figura 6.15, el contenido de los archivos se describe a continuación:

- **Evento enlazador:**
  - ***id:*** Corresponde al identificador de la invocación de la aplicación Industria del Mueble (ej: id = 4).
  - ***event:*** Indica el evento enlazador que es generado cuando la sub-crónica es reconocida.

- ***time:*** Corresponde al tiempo en milisegundos cuando es generado el evento enlazador.
- ***lp:*** La cantidad de productos de calidad en el evento E_5.

- **Diagnóstico de fallas:**
  - ***id:*** corresponde al identificador de la invocación de la aplicación Industria del Mueble (ej: id = 4).
  - ***fault:*** Nombre de la falla que es diagnosticada (ej: SLA).
  - ***faulttype:*** corresponde al tipo de falla, es usado generalmente para agregar información en el diagnostico (ej: SearchProducts).
  - ***lpretail:*** La cantidad de productos de calidad (lp) terminados recibidos del Fabricante (E_6).
  - ***lp:*** La cantidad de productos de calidad solicitado por la tienda (E_3).
  - ***time:*** corresponde al tiempo en milisegundos cuando es generado el diagnóstico.
  - ***Timestamp:*** corresponde al tiempo cuando el diagnóstico de la falla es realizado, se presenta en formato de tiempo entendido por los humanos.

Figura 6.15: Detección de la Crónica Distribuida para la violación de SLA en el caso de falla en la calidad de los productos en la aplicación de la industria del mueble

Como se muestra en la Figura 6.15, cuando el funcionamiento de la aplicación de la industria del mueble es correcto (id = 2, 3) ninguna sub-crónica es activada. En caso contrario, en id = 1 genera el reconocimiento de la falla de violación de SLA para la calidad de productos, reconociendo la sub-crónica Fabrica calidad(SLA) cuando la calidad suministrada por la fábrica (E_5.PSvar ≠ E_4.PSvar) es distinta a la esperada (9 ≠ 8), generando el evento enlazador $BE_{SLAQuality}$ hacia el diagnosticador de la Tienda y el contenido del evento es almacenado el archivo Manufacturerdiagnoser.xml de la Figura 6.15. Posteriormente, con el evento enlazador generado por la crónica en el diagnosticador de la Fabrica, el diagnosticador en la Tienda le es posible reconocer la sub-crónica Tienda calidad(SLA) y obtener el diagnóstico correcto de la falla de SLA para la calidad de los productos provistos por la Fábrica (archivo Retaildiagnoser.xml en la Figura 6.15).

6.4.2. Caso Comercio Electrónico

Para este caso de estudio se construyen dos modelos de crónicas para detectar fallas de violación de SLA para los casos de "Búsqueda de Productos" y "Empaqueta y envía". Para el caso de "Búsqueda de Productos", el conjunto de eventos involucrados para detectar la violación son E_4, E_5, E_6, E_7, E_8, E_9 y E_{10} con el atributo que contiene la lista de productos (lp). Ese es el flujo de eventos que reconoce el patrón de crónicas de la Figura 4.22, con la extensión para flujos de la Figura 4.26, considerando las restricciones no temporales de los eventos. En el caso del Proveedor, la violación se refiere a la diferencia en la cantidad de los productos requeridos con los recibidos por el Almacén (E_4.lp > E_{10}.lp), y son expresados usando la sub-cronica de la Figura 4.26 y la expresión atemporal anterior de lp. Asimismo, para el caso del Almacén cuando recibe el evento enlazador generado por la sub-crónica del Proveedor, involucra el flujo de eventos E_5, E_6, E_7, E_8, E_9, y la sub-crónica S_j SLA OR QoS de la Figura 4.22 con una extensión para sub-flujos para los eventos E_5, E_6, E_7 y E_8 (más de 2 eventos) y usando las restricciones atemporales de que la cantidad de productos solicitadas por el Proveedor sea mayor a cero (E_5.lp > 0), y que la cantidad de productos retornados en la búsqueda es cero (E_6.lp = 0), manteniendo el valor de la cantidad de productos hasta que el servicio Almacén retorna su respuesta al Proveedor (eventos E_7, E_8 y E_9). El patrón de crónicas obtenido, luego de añadir las restricciones temporales y atemporales, se muestra en la Figura 6.16:

Subchronicle Tienda { **Events{** **}** **Constrains{** **}** **When recognized{** **}** **}**	**Subchronicle Proveedor SLA {** **Events{** event(E_4: pl > 0, T_4) event(E_{10}: pl = 0, T_{10}) **}** **Constrains{** $T_{10} - T_4 > 0$ ***E_{10}.pl < E_4.pl*** **}** **When recognized{** Emit event(BE_{SC2Sea}, T_{SC2Sea}, Diagnoser 3) **}** **}**	**Subchronicle Almacén SLA Búsqueda{** **Events{** event(E_5; pl > 0, T_5), event(E_6 : pl = 0, T_6), event(E_7: pl = 0, T_7), event(E_8: pl = 0, T_8), event(BE_{SC2Sea}, T_{SC2Sea}) **}** **Constrains{** $T_6 - T_5 > 0$ $T_7 - T_6 > 0$ $T_8 - T_7 > 0$ $T_{SC2Sea} - T_8 > 0$ ***E_5.pl > 0*** ***E_6.pl = 0*** ***E_7.pl = E_6.pl*** ***E_8.pl = E_7.pl*** ***E_9.pl = E_8.pl*** **}** **When recognized{** Invoke(Warehouse, repairer 'SLA-Violation') **}** **}**

Figura 6.16: Modelo de Crónica para la violación de SLA en la cantidad de los Productos en el Almacén (caso Búsqueda de Productos)

donde:

- **Subchronicle Proveedor SLA** es generado usando el patrón de la Figura 4.26 con:
  - ***Sub-flow:*** es compuesto por los eventos de Supplier E_4 y E_{10}.
  - ***Temporal Constraint:*** Es mostrado en formato *cursiva* en la Figura 6.16. El flujo de la composición implica los eventos E_4 y E_{10}. En este caso, es necesario expresar una restricción temporal para asegurar que el evento E_4 ocurre previamente al evento E_{10} (T_{10} - T_4 > 0).
  - ***SLA OR QoS Constraint:*** Se muestra en formato ***negrita y cursiva***

en la Figura 6.16, expresando la violación de la diferencia en el atributo de la lista de productos (pl) entre los eventos E_4 y E_{10} (E_4.pl - E_{10}.pl > 0).

- **Subchronicle Almacén SLA Búsqueda** Es generada usando el patrón de la Figura 4.22 (S_j SLA OR QoS), con la extensión para flujos de la Figura 4.26:

  - ***Sub-flow:*** Es compuesta por los eventos del Almacén E_5, E_6, E_7, E_8, E_9 y BE_{SC2Sea}.
  - ***Temporal Constraint:*** Se muestran en formato *cursiva* en la Figura 6.16, al involucrar un sub-flujo de eventos en la composición, es necesario expresar una restricción temporal para asegurar que los eventos ocurren en la secuencia correcta (T_6 - T_5 > 0, T_7 - T_6 > 0. T_8 - T_7 > 0 y T_{SC2Sea} - T_8 > 0).
  - ***SLA OR QoS Constraint:*** Es mostrado en formato ***negrita y cursiva*** en la Figura 6.16, expresando la violación de la diferencia en el atributo que expresa la lista de productos (pl) entre los eventos E_5 y E_6 (E_5.pl > 0 y E_6.pl = 0), y los valores previos de lp en E_6 son mantenidos en los eventos E_7 y E_8 (E_7.pl = E_6.pl y E_8.pl = E_7.pl).

El modelo de la Crónica para detectar la falla de violación de SLA en la cantidad de productos en el caso de "Empaqueta y envía" es muy similar al modelo de crónica obtenido de la Figura 6.16 para la "Búsqueda de Productos", manteniendo la lista de los productos (lp) igual y mayor que cero para los eventos E_5, E_6, E_7, y menor en los eventos E_8, y E_9 en la sub-crónica del Almacén. Además, la misma es idéntica a la de la Figura 6.16 para el Proveedor:

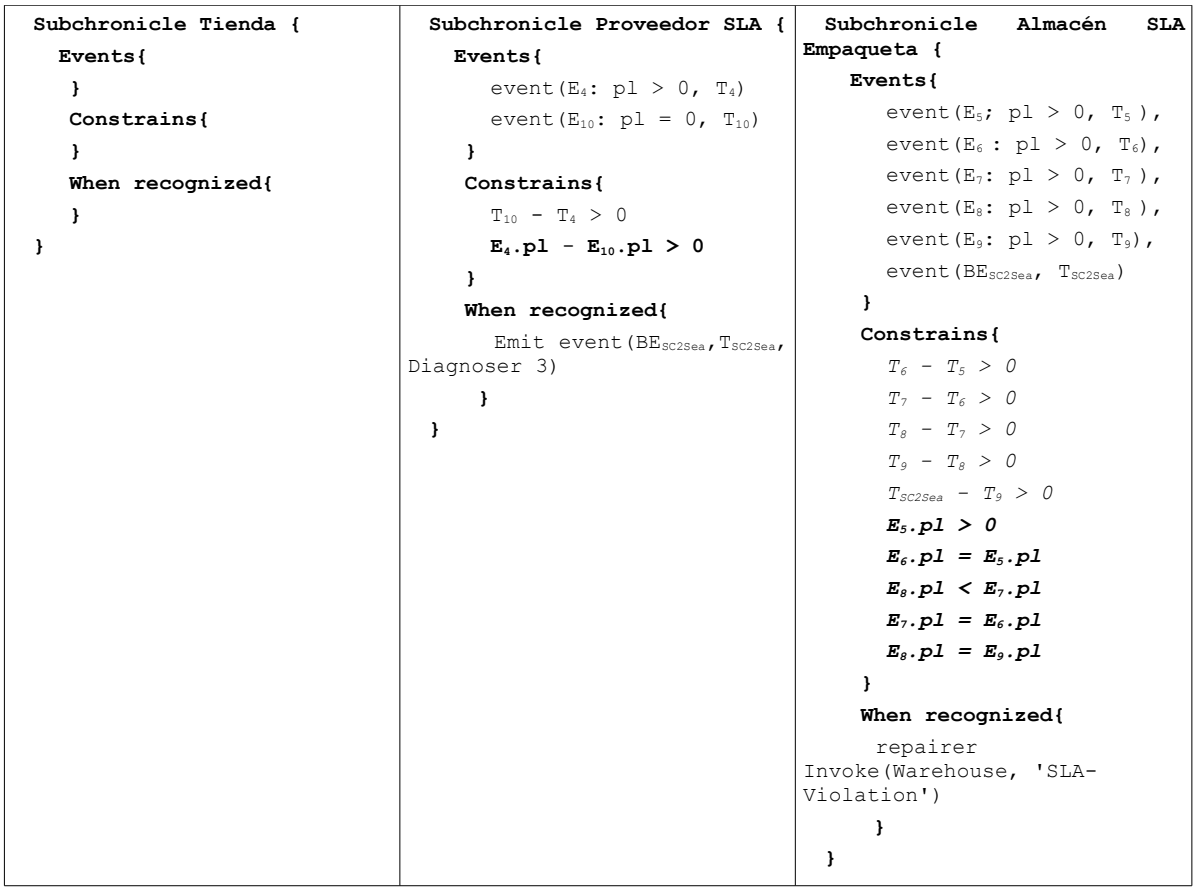


Figura 6.17: Modelo de Crónica para la violación de SLA para la cantidad de los productos en el Almacén (caso Empaqueta y envía)

En general, según las crónicas, existe un acuerdo de que lista de los productos debe ser igual a la cantidad de los productos solicitados, de lo contrario es una violación del acuerdo (una falla). La información generada en los atributos de los eventos enlazadores y el diagnóstico de las fallas es la misma que la mostrada en la sección 6.3.

6.4.2.1 Comprobar la Capacidad de Detección de la Crónica

Para verificar el reconocimiento de las crónicas propuestas en las Figura 6.16 y Figura 6.17, se implementaron utilizando el lenguaje CQL. La composición de la aplicación SOA comercio electrónico fue implementada en OpenESB.

Adicionalmente, se han utilizado las operaciones **tuneSearchBehavior** y **tuneShip** en el servicio Almacén para generar las fallas de violación de SLA de la cantidad de productos ("Búsqueda de Productos" y "Empaqueta y envía"), para poder verificar el reconocimiento de la crónica. Se ejecuta la aplicación de comercio electrónico 9 veces (con diferentes patrones de pruebas, en distintos escenarios de funcionamiento: normal y fallas de SLA), añadiendo un atributo id para diferenciar cada invocación. En la Tabla 6.3 se muestran las diferentes invocaciones, con los diferentes valores utilizados para id, lp, tuneSearchBehavior y tuneShip.

id	tuneSearchBehavior	tuneShip	Operation result (lp)													
			E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14
1	FALSE	FALSE	15	15	15	15	15	15	15	15	15	15	15	15	15	15
2	FALSE	FALSE	30	30	30	30	30	30	30	30	30	30	30	30	30	30
3	FALSE	FALSE	10	10	10	10	10	10	10	10	10	10	10	10	10	10
4	***TRUE***	***FALSE***	***8***	***8***	***8***	***8***	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
5	FALSE	FALSE	2	2	2	2	2	2	2	2	2	2	2	2	2	2
6	FALSE	FALSE	13	13	13	13	13	13	13	13	13	13	13	13	13	13
7	FALSE	FALSE	14	14	14	14	14	14	14	14	14	14	14	14	14	14
8	***FALSE***	***TRUE***	***22***	***22***	***22***	***22***	***22***	***22***	***22***	**21**	***21***	***21***	***21***	***21***	***21***	***21***
9	FALSE	FALSE	5	5	5	5	5	5	5	5	5	5	5	5	5	5

Tabla 6.3: El resultado de las invocaciones en la aplicación de Comercio Electrónico

Para cada sub-crónica que es reconocida, los eventos generados se almacenan en un archivo (ver Figura 6.18). En el caso del diagnosticador del Proveedor, la sub-crónica Proveedor SLA genera el evento enlazador BE_{SLA} hacia el diagnosticador del Almacén cada vez que la sub-crónica es reconocida, y es almacenado en el archivo Supplierdiagnoser.xml (es idéntico en ambos casos de fallas de SLA). Al emitir el evento BE_{SLA} a la sub-crónica Almacén SLA Búsqueda, se almacena el diagnóstico cuando la falla es identificada en el archivo Warehousediagnoser.xml (emite un evento con el contenido de la falla). De igual

manera, cuando la sub-crónica Almacén SLA Empaqueta es reconocida se almacena el diagnóstico de la falla en el archivo Warehousediagnoser1.xml. Así, el contenido de los archivos generados cuando se invoca la aplicación de comercio electrónico y cada sub-crónica es reconocida se muestra en la Figura 6.18:

```
-<case>
 -<msgns:StreamOutput0_MsgObj>
    <id>4</id>
    <event>BESLA</event>
    <time>1408573191762</time>
    <lpsupplier>0</lpsupplier>
    <lp>8</lp>
    <Timestamp>2014-08-20T17:49:57.375-04:30</Timestamp>
  </msgns:StreamOutput0_MsgObj>
 -<msgns:StreamOutput0_MsgObj>
    <id>8</id>
    <event>BESLA</event>
    <time>1408573528762</time>
    <lpsupplier>21</lpsupplier>
    <lp>22</lp>
    <Timestamp>2014-08-20T17:55:34.418-04:30</Timestamp>
  </msgns:StreamOutput0_MsgObj>
</case>
```

Archivo: Supplierdiagnoser.xml Almacena el evento generado BE_{SLA} (evento enlazador) cuando se reconoce la sub-crónica Proveedor SLA

```
-<msgns:RepairOutput0_MsgObj>
   <id>4</id>
   <fault>ServiceLevelAgreement</fault>
   <faulttype>SearchProducts</faulttype>
   <time>1408573191762</time>
   <lp>0</lp>
   <Timestamp>2014-08-20T17:49:59.647-04:30</Timestamp>
 </msgns:RepairOutput0_MsgObj>
```

Archivo: Warehousediagnoser.xml Contenido del evento generado cuando se reconoce la Falla de SLA de la cantidad de los productos en el caso de Búsqueda de Productos en el Almacén (sub-crónica Almacén SLA Búsqueda)

```
-<msgns:RepairOutput1_MsgObj>
   <id>8</id>
   <fault>ServiceLevelAgreement</fault>
   <faulttype>Ship and packed</faulttype>
   <time>1408573528762</time>
   <lp>21</lp>
   <Timestamp>2014-08-20T17:55:36.908-04:30</Timestamp>
 </msgns:RepairOutput1_MsgObj>
```

Archivo: Warehousediagnoser1.xml Contenido del evento generado cuando se reconoce la Falla de SLA de la cantidad de los productos en el caso de Empaqueta y envía en el Almacén (sub-crónica Almacén SLA Empaqueta)

Figura 6.18: Archivos generados por la detección del patrón de crónica distribuida para las fallas de violación de SLA para los casos "Búsqueda de Productos" y "Empaqueta y envía" en la aplicación de comercio electrónico.

Como se muestra en la Figura 6.18, cuando el funcionamiento de la aplicación de comercio electrónico es normal (id = 1, 2, 3, 5, 6, 7 9) ninguna sub-crónica es

activada. De lo contrario, en la invocación id = 4 se genera el reconocimiento de la falla violación de SLA para la Búsqueda de Productos. En este caso, los eventos generados son almacenados en Supplierdiagnoser.xml (emite el evento BE_{SLA} en el instante 1408573191762), que es leído por el diagnosticador del Almacén en la sub-crónica Almacén SLA Búsqueda, y emite el evento con el reconocimiento de la falla (archivo Warehousediagnoser.xml). Del mismo modo, para el caso de "Empaqueta y envía" (id = 8), la falla es parcialmente reconocida en la sub-crónica del Proveedor (archivo Supplierdiagnoser.xml en el instante 1408573528762), y emite el evento BE_{SLA} al diagnosticador del Almacén. Con este evento es posible reconocer la falla (emite el evento con el contenido de la falla en Warehousediagnoser1.xml).

6.4.3. Análisis de Resultados

En los casos de estudio se han configurado diferentes patrones genéricos de crónicas distribuidas para diagnosticar las fallas de violación de SLA en diferentes niveles de la composición (servicio y flujo). En el primer caso (Industria de Muebles) se ha diagnosticado la falla de SLA a nivel del servicio, inclusive se han desarrollado tres modelos diferentes de crónicas para la falla de SLA para reconocer problemas en diferentes variables físicas: Calidad, Costo y Tiempo de Envío. En el segundo caso se ha utilizado el mismo patrón genérico para diagnosticar el mismo fallo a nivel del flujo de la composición, que implica diferentes operaciones ("Búsqueda de Productos" y "Empaqueta y envía"). Así, se ha mostrado lo fácil que es ajustar los patrones genéricos para caracterizar las fallas para diferentes escenarios, y cómo el funcionamiento de las crónicas distribuidas es distribuido.

Para el escenario de la Fábrica de Muebles se ha modificado la crónica de la Figura 4.22 para generar crónicas distribuidas que permitan el diagnóstico la

violación de SLA en Costo y Tiempo de Envío (Figura 6.14). Además, se utilizó la extensión de la Figura 4.26 para lograr el diagnóstico de SLA en el sub-flujo de la composición (caso Calidad SLA, Figura 6.13). En este caso, se definieron diagnosticadores distribuidos (Tienda y Fabricante) para lograr la detección distribuida de la falla.

Por otra parte, el escenario de comercio electrónico expande aún más el patrón genérico de las crónicas de la Figura 4.22 para las fallas de violación de SLA. En este caso, es necesario diferenciar el evento responsable de la violación de SLA en la aplicación (E_6 o E_7). Así, se define una primera crónica basada en la crónica genérica de la Figura 4.22, y se expande utilizando el patrón genérico de la Figura 4.26 para definir la primera detección de la violación en el diagnosticador del Proveedor (sub-crónica Proveedor SLA en las Figura 6.16 y Figura 6.17), para luego emitir un evento enlazador, llamado BE_{SLA}, al diagnosticador del Almacén. Entonces, se amplia la crónica del Almacén utilizando de nuevo la extensión definida en la Figura 4.26 sobre la crónica de la Figura 4.22 (Sub-crónica **S_j SLA OR QoS**), para detectar los problemas en el evento Búsqueda de Productos (E_6 Figura 6.16, sub-crónica Almacén SLA Búsqueda) y en el evento Empaqueta y envía (E_8 Figura 6.17, sub-crónica Almacén SLA Empaqueta).

En la Tabla 6.4 se muestra las distintas configuraciones (ajustes) que fueron necesarios realizar a los patrones de crónicas para detectar las fallas de SLA, tanto a nivel de un solo servicio como en el sub-flujo de la aplicación.

Caso de Estudio	Propiedad no Funcional	Nivel de Aplicación	Configuraciones	Restricciones Atemporales
Industria del Mueble	Calidad	Flujo	**Tienda:** Subchronicle Tienda calidad(SLA) generada de Subchronicle S_j SLA O QoS de la Figura 4.22:	Fabrica: • E_4.PSvar > E_5.PRvar

Caso de Estudio	Propiedad no Funcional	Nivel de Aplicación	Configuraciones	Restricciones Atemporales
			• ***Equivalencia de eventos:*** ◦ $E_2 \approx E_3$ (Tienda) . ◦ $E_3 \approx E_6$ (Tienda). ◦ $BE_{SLAORQoS} \approx BE_{SLAQuality}$ (generado en Fabrica). • ***Acción:*** Invocar reparador Tienda. **Fabrica:** Subchronicle Fabrica calidad(SLA) generada de Subchronicle S_k SLA O QoS de la Figura 4.26: • ***Equivalencia de eventos (Sub-flujo):*** ◦ $E_{4.1} \approx E_4$ (Fabrica). ◦ $E_{4.2} \approx E_6$ (Fabrica). • ***Acción:*** Emitir el evento enlazador ($BE_{SLAQuality}$) a Diagnosticador Tienda.	
	Costo	Servicio	**Tienda:** Subchronicle Tienda Costo (SLA) generada de Subchronicle S_j SLA O QoS de la Figura 4.22: • ***Equivalencia de eventos:*** ◦ $E_2 \approx E_3$ (Tienda) . ◦ $E_3 \approx E_6$ (Tienda). ◦ $BE_{SLAORQoS} \approx B_{ESLACost}$ (generado en Fabrica) . • ***Acción:*** Invocar reparador Tienda. **Fabrica:** Subchronicle Fabrica Costo (SLA) generada de Subchronicle S_k SLA O QoS la Figura 4.22. • ***Equivalencia de eventos:*** ◦ $E_4 \approx E_5$ (Fabrica) • ***Acción:*** Emitir el evento enlazador ($BE_{SLACost}$) a Diagnosticador Tienda.	Fabrica: • E_5.Cost > C_0
	Tiempo de Envío	Servicio	**Tienda:** Subchronicle Tienda entrega (SLA) generada de Subchronicle S_k SLA O QoS de la Figura 4.22: • ***Equivalencia de eventos:*** ◦ $E_2 \approx E_3$ (Tienda). ◦ $E_3 \approx E_6$ (Tienda). ◦ $BE_{SLAORQoS} = BE_{SLAdelivery}$ (generado en Fabrica). • ***Acción:*** Invocar reparador	Fabrica: • E_5.entrega > TimeDelivery$_0$

Caso de Estudio	Propiedad no Funcional	Nivel de Aplicación	Configuraciones	Restricciones Atemporales
			Tienda. **Fabrica:** Subchronicle Fabrica entrega (SLA) generada de Subchronicle S_j SLA O QoS la Figura 4.22: • ***Equivalencia de eventos:*** ◦ $E_4 \approx E_5$ (Fabrica). • ***Acción:*** Emitir el evento enlazador ($BE_{SLAdelivery}$) a Diagnosticador Tienda	
Comercio Electrónico	Lista de Productos (Búsqueda de Productos)	Flujo	**Proveedor:** Subchronicle Proveedor SLA generada de Subchronicle S_k SLA O QoS de la Figura 4.26: • ***Equivalencia de eventos (Sub-flujo):*** ◦ $E_{4.1} = E_4$ (Proveedor). ◦ $E_{4.2} = E_{10}$ (Proveedor). • ***Acción:*** Emitir el evento enlazador (BE_{SC2Sea}) a Diagnosticador Almacén. **Almacén:**Subchronicle Almacén SLA Búsqueda generada de Subchronicle S_j SLA OR QoS la Figura 4.22 con sub-flujo: • ***Equivalencia de eventos (Sub-flujo):*** ◦ $E_{4.1} = E_5$ (Almacén).$E_{4.2} = E_6$ (Almacén). ◦ $E_{4.3} = E_7$ (Almacén). ◦ $E_{4.4} = E_8$ (Almacén). ◦ $BE_{SLAORQoS} = BE_{SC2Sea}$ (generado en Proveedor). • ***Acción:*** Invocar reparador Almacén.	Proveedor: • $E_{10}.pl < E_4.pl$ Almacén: • $E_5.pl > 0$ • $E_6.pl = 0$
	Lista de Productos (Empaqueta y envía)	Flujo	**Proveedor:** Subchronicle Proveedor SLA generada de Subchronicle S_k SLA O QoS de la Figura 4.26: • ***Equivalencia de eventos (Sub-flujo):*** ◦ $E_{4.1} = E_4$ (Proveedor). ◦ $E_{4.2} = E_{10}$ (Proveedor). • ***Acción:*** Emitir el evento enlazador (BE_{SC2Sea}) a Diagnosticador Almacén. **Almacén:**Subchronicle Almacén Almacén SLA Empaqueta generada de	Proveedor: • $E_{10}.pl < E_4.pl$ Almacén: • $E_5.pl > 0$ • $E_6.pl = E_5.pl$ • $E_7.pl = E_6.pl$ • $E_8.pl < E_7.pl$

Caso de Estudio	Propiedad no Funcional	Nivel de Aplicación	Configuraciones	Restricciones Atemporales
			Subchronicle S_j SLA O QoS la 2Figura 4.22 con sub-flujo: • ***Equivalencia de eventos (Sub-flujo):*** ◦ $E_{4.1}$ = E_5 (Almacén).$E_{4.2}$ = E_6 (Almacén). ◦ $E_{4.3}$ = E_7 (Almacén). ◦ $E_{4.4}$ = E_8 (Almacén). ◦ $E_{4.5}$ = E_9 (Almacén). ◦ $BE_{SLAORQoS}$ = BE_{SC2Sea} (generado en Proveedor). • ***Acción:*** Invocar reparador Almacén.	

Tabla 6.4: Resumen de las configuraciones hechas a los patrones de crónicas para los casos de estudio

En base a los pasos necesarios para configurar las crónicas a las distintas fallas de SLA en los experimentos realizados, es posible la construcción de un nuevo patrón de crónicas que permita reunir todas las configuraciones realizadas en los distintos casos sin realizar tantos ajustes, y de esta manera facilitar su configuración e implementación, al no tener que realizar combinaciones de los patrones de crónicas de la Figura 4.22 y Figura 4.26. En general, para permitir al patrón de crónicas detectar las fallas de violación de SLA, tanto a nivel de un servicio (Figura 4.22) como de sub-flujo (Figura 4.26), es posible generar un nuevo patrón de crónica, aun más genérico, que reúna las distintas configuraciones (Figura 4.22 y Figura 4.26) que fueron necesarias para adaptar las crónicas a los casos de estudio. Estas pruebas fueron necesarias para determinar eso. Así, se puede escribir un nuevo patrón de crónicas mucho más genérico de la forma:

Diagnosticador D_j	**Diagnosticador D_k**
Subchronicle S_j SLA OR QoS { **Events{** event($E_{j.1}$, $T_{j.1}$) . . . event($E_{j.m1}$, $T_{j.m1}$) occurs((n_1 , n_2), $BE_{SLAORQoS}$, ($T_{SLAORQoS1}$, $T_{SLAORQoS2}$)) **}** **Constrains{** $T_{j.2}$ > $T_{j.1}$ . . . $T_{j.m1}$ > $T_{j.(m1 - 1)}$ $T_{SLAORQoS2}$ > $T_{j.m1}$ + ΔT $T_{SLAORQoS2}$ > $T_{SLAORQoS1}$ noFunctional (Flow({$E_{j.1}$,..$E_{j.m1}$}) attributes) **}** **When recognized{** repair invoke(S_j, 'SLA OR QoS') **}** **}**	**Subchronicle S_k SLA OR QoS {** **Events{** event($E_{k.1}$, $T_{k.1}$) . . . event($E_{k.m2}$, $T_{k.m2}$) **}** **Constrains{** $T_{k.2}$ > $T_{k.1}$ . . . $T_{k.m2}$ > $T_{k.(m2 - 1)}$ noFunctional (Flow({$E_{k.1}$,..$E_{k.m2}$}) attributes) **}** **When recognized{** emit event($BE_{SLAORQoS}$, $T_{SLAORQoS}$) to S_j diagnoser **}** **}**

Figura 6.19: Patrón de Crónica Distribuida para la falla de Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (QoS) 3

donde los parámetros de la Sub-crónica Sj SLA OR QoS 2 son:

$E_{j.1}$, ..., $E_{j.m1}$: Son los eventos generados en los instantes $T_{j.1}$, ..., $T_{j.m1}$ por la aplicación SOA cuando se invoca el flujo del diagnosticador de S_j y (el valor de m1 representa la cantidad de eventos involucrados).

$E_{k.1}$, ..., $E_{k.m2}$: Son los eventos generados en los instantes $T_{k.1}$, ..., $T_{k.m1}$ por la aplicación SOA cuando se invoca el flujo del diagnosticador de S_k y (el valor de m2

representa la cantidad de eventos involucrados).

noFunctional (Flow): Representan las restricciones de SLA o QoS de los flujos S_j y S_k, la cual es evaluada en base a la propiedad no funcional que contiene el flujo de la restricción.

$BE_{SLAORQoS}$: Es el evento enlazador generado cuando se reconoce la sub-crónica del diagnosticador D_k **"SLA OR QoS"**.

n_1 y n_2: representa el rango de la cantidad de veces que la violación ha sucedido ($n_1 \leq$ frecuencia $\leq n_2$). Si no tenemos una crónica que detecta la ocurrencia de este fenómeno, entonces n_2 debe ajustarse al valor ∞. En este mismo orden de ideas, note que si $n_1 = 1$ y $n_2 = 1$ la crónica resultante se activa solo cuando el fenómeno ocurre una vez.

$T_{SLAORQoS2} > T_{SLAORQoS1}$: Representa el rango de tiempo en que se considera la cantidad de veces en que la violación se sucede. Para considerar la cantidad de veces en que ocurre la violación de SLA desde que empezó a operar la aplicación se considera $T_{SLAORQoS1} = 0$.

Es posible obtener las crónicas de los casos de estudio con el patrón de crónicas de la Figura 6.19, y no tomando en cuenta múltiples ocurrencias ($n_1 = 1$ y $n_2 = 1$):

- **Caso de Industria del mueble:**
  - ***Calidad:***
    - *Fabricante:* La crónica se construye tomando la sub-crónica S_k con el flujo de Eventos E_4 y E_5, y detectando la violación de la propiedad no

funcional E_4.PSvar > E_5.PRvar.

- *Tienda:* La crónica se construye tomando la sub-crónica S_j con el flujo de Eventos E_3 y E_6, recibiendo el evento enlazador BE_{SLA}, y no colocando ninguna restricción para propiedades no funcionales.

◦ ***Costo:***

- *Fabricante:* La crónica se construye tomando la sub-crónica S_k con el flujo de Eventos E_5 (un solo evento), y detectando la violación de la propiedad no funcional E_5.Cost > C_0.

- *Tienda:* La crónica se construye tomando la sub-crónica S_j con el flujo de Eventos E_3 y E_6, recibiendo el evento enlazador BE_{SLA}, y no colocando ninguna restricción para propiedades no funcionales.

◦ ***Tiempo de Envió:***

- *Fabricante:* Similar a la sub-crónica de costo pero tomando la propiedad funcional del tiempo de envío (E_5.entrega > $TimeDelivery_0$) para la violación de SLA.

- *Tienda:* Idéntica a la construcción de la crónica para el caso de costo.

- **Comercio Electrónico:**

◦ ***Caso Búsqueda de Productos:***

- *Proveedor:* La crónica se construye tomando la sub-crónica S_k con el flujo de Eventos E_4 y E_{10}, y detectando la violación de la propiedad no funcional E_4.lp > E_5.lp.

- *Almacén:* La crónica se construye tomando la sub-crónica S_j con el flujo de Eventos E_5, E_6, E_7, E_8 y E_9, recibiendo el evento enlazador BE_{SLA}, y agregando la restricción para propiedades no funcionales de lp a los eventos (E_5.pl > 0 y E_6.pl = 0).

◦ ***Caso Empaqueta y envía:***

- *Proveedor:* La crónica es igual a la descrita para el caso de Búsqueda de Productos.

- *Almacén:* La crónica se construye tomando la sub-crónica S_j con el flujo de Eventos E_5, E_6, E_7, E_8 y E_9, recibiendo el evento enlazador BE_{SLA}, y agregando la restricción para las propiedades no funcionales de lp a los eventos (E_5.pl > 0, E_6.pl = E_5.pl, , E_7.pl = E_6.pl y E_8.pl < E_7.pl).

6.5. Conclusiones

En este Capítulo se ha mostrado, mediante el empleo de diferentes experimentos, el funcionamiento del motor de reconocimiento de crónicas distribuidas del componente diagnosticador. En una primera experiencia se comprobó la extensión del formalismo de Crónicas propuesto en este trabajo, para permitir el diagnóstico distribuido de las fallas, Adicionalmente, en este experimento se mostró el uso de los eventos enlazadores, los cuales permiten sincronizar los reconocimientos parciales y lograr el reconocimiento global.

Seguidamente, se ha comprobado la capacidad del lenguaje CQL para el reconocimiento de crónicas, permitiendo dar mayor expresividad a las restricciones no temporales usadas en las crónicas, al permitir añadir expresiones matemáticas, tal como se ha mostrado en los casos de estudio. Adicionalmente, la

posibilidad de implementar el reconocedor de crónicas en una arquitectura SOA, permite la implementación de las distintas instancias del Gestor Autonómico de ARMISCOM como componentes distribuidos débilmente acoplado, lo que le confiere al MAPE una gran flexibilidad.

En cuanto a los patrones Distribuidos de crónicas para el diagnóstico de fallas en la composición de servicios, se ha mostrado su funcionamiento en un gran número de escenarios a lo largo de este Capítulo, para el caso de fallas de servicios no funcionales, resultando fácil la implementación y ajuste. En una primera instancia se mostró el ajuste del patrón de fallas de SLA para medir distintas propiedades no funcionales en múltiples casos y escenarios. En los casos de estudios se adaptó el patrón de crónicas de la Figura 4.22 para detectar violaciones de SLA tanto a nivel de un solo servicio (caso de Costo y Tiempo de entrega para la Industria del Mueble) como de un flujo de la composición, con la anexión del patrón de la crónica de la Figura 4.26 (caso Calidad en la industria de muebles y lista de productos en la Aplicación de Comercio Electrónico). Con los resultados obtenidos de las distintas configuraciones, en la sección de Análisis de Resultados de los experimentos realizados sobre los patrones se diseño un nuevo patrón de crónicas para la detección de violaciones de SLA (Figura 6.19), que une la extensión propuesta en el patrón de la Figura 4.26 con los patrones de las Figura 4.22 y Figura 4.23, haciéndola mucho más genérica y de fácil configuración para cualquier caso en que se quiera implementar, incluso considerando la frecuencia de ocurrencia de la la violación de la Figura 4.23.

En el próximo Capítulo se usarán otros patrones de crónica considerando aspectos funcionales y no funcionales, y se probará el funcionamiento de todos los componentes de ARMISCOM, para la gestión automática de fallas en la composición de servicios.

Capítulo 7
ARMISCOM - Experimentación y Análisis de Resultados

En Capítulos anteriores se ha mostrado la arquitectura de ARMISCOM, la cual está basada en los paradigmas de Middlewares Reflexivos y Computación Autonómica. Adicionalmente, se ha propuesto una extensión del paradigma de crónicas, para poder caracterizar y reconocer patrones distribuidos, el cual es usado por ARMISCOM para caracterizar su conocimiento. Finalmente, se ha presentado la implementación de ARMISCOM en OpenESB y Protege. En este Capítulo se presenta el conjunto de experimentos realizados para comprobar el funcionamiento de ARMISCOM.

7.1. Componente de Representación del Conocimiento de ARMISCOM

Para comprobar el funcionamiento del componente de representación del conocimiento de ARMISCOM, nuevamente se toma el caso de prueba de Comercio Electrónico, y se consideran los siguientes escenarios de fallas:

- Timeout (Tiempo Fuera).
- Quality Of Service (Calidad de Servicio).

Con base a los patrones de las crónicas genéricas para los diferentes tipos de fallas de una aplicación SOA propuestas en el Capítulo 4, se construye el componente de conocimiento de la crónica distribuida específica para cada falla:

Calidad de Servicio (patrón de crónicas de las Figura 6.19), Fuera de tiempo (patrón de crónicas de la Figura 4.29). La adecuación de los patrones de crónicas distribuida para estos casos de fallas se muestran en la Figura 7.1:

Crónica Distribuida: Fuera de Tiempo (Timeout)

Subchronicle Proveedor Fuera-Tiempo {

```
SELECT
          ISTREAM(id => E4.id, event => 'EBTimeout', time => ENOEVENT.time, lpsupplier => 0, lp => E4.lp, to => 'Diagnoser Almacén')
FROM
     E4[15000], ENOEVENT[now]
WHERE
     ENOEVENT.time >= E4.time + 1000 AND
     ENOEVENT.id = E4.id
```

}

Subchronicle Almacén Fuera-Tiempo {

```
SELECT
          ISTREAM(id => E5.id, fault => 'timeout', faulttype => 'N/A', time => EBTimeout.time, lp => E5.lp, event_init => E5, event_end => E9, to => 'Repair Almacén')
FROM
     E5[15010], EBTimeout[now]
WHERE
     EBTimeout.id = E5.id AND EBTimeout.time > E5.time
```

}

Crónica Distribuida: Calidad de Servicio-QoS (Delay: Retardo)

Subchronicle Proveedor Retardo {

```
SELECT
         ISTREAM(id => E4.id, event => 'EBDelay', time => E10.time, lp => E4.lp, to => 'Diagnoser Proveedor')
FROM
     E4[5500], E10[now]
WHERE
     E10.time - E4.time >= 2000 AND E10.id = E4.id
```

}

Subchronicle Almacén Retardo{

```
SELECT
       ISTREAM(id => EBDELAY1.id, fault => 'QualityOfService', faulttype => 'Delay', time => EBDELAY1.time, lp => E5.lp, event_init => E5, event_end => E9, to => 'Repair Almacén')
FROM
     EBDELAY[15500], EBDELAY1[now],
WHERE
          count(EBDELAY.id) + 1 > 2 AND EBDELAY.id <> EBDELAY1.id
```

}

Figura 7.1: Crónicas distribuidas para las fallas fuera de Tiempo y Calidad de Servicio (Delay) en CQL para la aplicación de Comercio Electrónico

Donde:

- **Crónica Fuera de Tiempo (timeout):**
  - ***Subchronicle Proveedor Fuera-Tiempo:*** Es generada usando el patrón de la Figura 4.29 (Subchronicle S_k Fuera de Tiempo):
    - *Entradas:*

- ***E₄*** es un evento que es mantenido por ***15000 ms*** ($INVE_2$) y ***$E_{NOEVENT}$*** es un stream producido por la no respuesta del Almacén. Ambos, **E_4** y **$E_{NOEVENT}$** tienen el atributo atemporal ***id*** (es un identificador usado para diferenciar los eventos generados en las distintas invocaciones de la aplicación), ***time*** (se genera cuando se produce el evento) y ***lp*** (la lista de productos).

▪ *Restricciones:*

- Los eventos deben tener el mismo id, el tiempo que el diagnosticador D_j debe esperar para considerar que el evento E_4 se encuentra fuera de tiempo es de ***5000*** *ms* ($\Delta T_{timeout}$), y el valor de tiempo que ha tenido que esperar el middleware para producir el evento E_4 ($INVE_2$ considerar que se ha producido un posible retardo en la ejecución del Almacén) es de 4000 ms (ΔT_{espera}). Así, finalmente la diferencia de tiempo entre ***$E_{NOEVENT}$*** y ***E_4*** debe ser 1000 *ms* ($\Delta T_{timeout}$ - ΔT_{espera}).

▪ *Salida*

- emite un evento enlazador llamado ***$E_{BTimeout}$*** al diagnosticador Almacén cuando la sub-crónica es reconocida**.**

◦ ***Subchronicle Almacén Fuera-Tiempo:*** Es generada usando el patrón de la Figura 4.29 (Subchronicle S_j Fuera de Tiempo):

▪ *Entradas:*

- ***E_5*** es mantenido por ***15010 ms*** (relation), y ***$EB_{Timeout}$*** es el evento enlazador generado por el diagnosticador Proveedor cuando

reconoce la sub-crónica Proveedor Fuera-Tiempo, y se encuentra en forma de un stream (sucede ahora "now"). Todos los eventos tienen los mismos atributos id, tiempo y lp, como en la subcrónica del Proveedor.

- *Restricciones:*
  - Los eventos deben tener el mismo id y respetar la secuencia de llegada (el evento E_5 debe suceder antes que el evento enlazador EB_{DELAY}).
- *Salida:*
  - Emite el evento al Reparador del Almacén con la información de la falla de tiempo fuera. Para ello se ha añadido información adicional en el evento para indicarle al Reparador el nombre y tipo de falla (name= timeout, type=N/A), y la parte del flujo de la aplicación afectada (event_init = E_5 y event_end = E_9, el flujo afectado esta compuesto por cinco eventos E_5, E_6, E_7, E_8 y E_9). La información suministrada en event_init y event_end facilita la búsqueda de mecanismos de reparación en la metadata para esta falla.

- **Crónica Calidad de Servicio- QoS (Delay):**
  - ***Subchronicle Proveedor Retardo:*** Es generada usando el patrón de la Figura 6.19 (S_j SLA OR QoS):
    - *Entradas:*
      - ***E_4*** es un evento que es mantenido por ***55000 ms*** y ***E_{10}*** es un

stream, y es generada por la aplicación de comercio electrónico. Tienen los atributos ***id***, ***time*** y ***lp***.

- *Restricciones:*
  - La diferencia en el tiempo de los eventos E_4 y E_{10} debe ser mayor a ***2000 ms*** (retardo en el almacén y corresponde a la definición de la función noFunctional del patrón de crónica, $T_{10} - T_4 > 2000$).
- *Salida:*
  - emite un evento enlazador llamado EB_{Delay} al diagnosticador en el almacén).

- ***Subchronicle Almacén Retardo:*** Es generada usando el patrón de la Figura 6.19 (S_k SLA OR QoS):
  - *Entrada:*
    - EB_{Delay} es un evento mantenido por ***15500 ms*** y EB_{Delay1} es un stream, y son generados por la sub-crónica S_j SLA OR QoS en el Proveedor (la primera y segunda son la misma, pero reflejan momentos distintos de la misma falla), ambos tienen los atributos ***id***, time y lp.
  - *Restricciones:*
    - La cantidad de eventos recibidos EB_{Delay} debe ser mayor que ***2***. En esta sub-crónica no se ha considerado una función noFunctional para el diagnosticar en el Almacén, debido a que no es necesario

agregar ninguna restricción sobre las propiedades no funcionales.

- *Salida:*

  - Emite el evento al Reparador en el Almacén con la información de la falla. Para ello se ha añadido información adicional en los atributos del evento, para indicarle al Reparador el nombre y tipo de falla (name = Quality Of Service type = Delay) y la parte del flujo afectada (event_init = E_8 y event_end = E_8, el flujo afectado en este caso es un solo servicio E_8). La información suministrada en event-init y event_flow facilita la búsqueda de mecanismos de reparación en la metadata para esta falla.

Además, en base a la estructura de Regiones definida en el Capítulo 3, para definir los mecanismos de reparación para solventar las fallas en las aplicaciones SOA, se deben caracterizar en la metadata los mismos. En este caso de estudio, se han agregado 5 mecanismos de reparación en la metadata en la Tabla 7.1:

Métodos de Reparación Disponibles						
Id	**Weight**	**RepairMethod**	**Transition**	**Event_init**	**Event_end**	**Operations**
1	1	substituteflow	{E6, E7, E8}	E5	E9	remFlow({E5, E6, E7, E8, E9}) addFlow({E5,E6, E7,E8,E9})
2	1	parametersUpdate	{Ø}	E6	E6	setting(E6)
3	1	CompleteMissingParameter	{Ø}	E4	E5	remFlow({E4, E5}) addFlow({E4,Compensate,E5)
4	1	substituteflow	E8	E8	E8	remFlow({ E7, E8, E9}) addFlow({E7,E9})
5	1	substituteflow	E7	E7	E7	remFlow({E6, E7, E8}) addFlow({E6,E8})

Tabla 7.1: Métodos disponibles para reparar la aplicación de comercio electrónico

Según la Tabla 7.1, para el ejemplo los reparadores de ARMISCOM tienen disponibles 5 mecanismos de reparación para las fallas de la aplicación de comercio electrónico. Pasemos a describir los mecanismo de reparación:

- substituteflow (Id = 1) afecta el flujo de la composición del evento E_5 hasta el evento E_9 (repara las operaciones E_5, E_6, E_7, E_8 y E_9, sustituyendo todas las operaciones del Almacén por unas nuevas).
- parametersUpdate repara la operación E_6 actualizando su parámetros de operación (modifica "buscar los productos" en el evento E_6 del almacén, usando la operación "*ExternalSearch*", para que pueda buscar productos en otros almacenes).
- CompleteMissingParameter (repara la operación con una compensación de la entrada en la operación E_5, utilizando para esto una especie de filtro que transforma la salida del evento E_4 a una que pueda ser suministrada a E_5).

- substituteflow (repara la operación E_8 sustituyendo la operación "empaqueta los productos y envía" (packs and ships) por otra equivalente).
- substituteflow (repara la operación de actualizar el inventario E_7 por otra equivalente en el Almacén).

7.1.1. Prueba de la Aplicación de Comercio Electrónico Utilizando el Componente de Conocimiento

Para verificar el funcionamiento de los componentes de Conocimiento de ARMISCOM, se despliega la aplicación de Comercio Electrónico en OpenESB, y se conectan los servicios Diagnosticadores y Reparadores distribuidos entre todos los servicios de la composición (instancias Tienda, Proveedor y Almacén). Adicionalmente, se hace uso de la operación *setTuneDelay p*ara agregar retardo en el tiempo de respuesta del servicio. Así, 3 invocaciones de la aplicación son realizadas (id = {1, 2, 3}), donde se ha colocado el valor de TuneDelay en 3000 ms (inducir múltiples retardos). Posteriormente, se invoca la aplicación (id = {4}) con un valor de TuneDelay de 6000ms, que causaría un fuera de tiempo en la aplicación. Los resultados son mostrados en la Figura 7.2. Para facilitar la lectura de la información generada por el diagnóstico de las fallas, la consulta de la ontología Fault-Recovery, y la respuesta de la metadata, el contenido de los archivos mostrado en la Figura 7.2 se describe a continuación:

- **Diagnóstico de fallas por la Crónica Distribuida**:
  - ***id:*** corresponde al identificador de la invocación de la aplicación de comercio electrónico (ej: id = 4).
  - ***fault:*** Nombre de la falla que es diagnosticada (ej: ServiceLevelAgreement) y que es usada como fuente para el servicio gestor de la ontología.

- ***faulttype:*** corresponde al tipo de falla, es usado generalmente para agregar información en el diagnóstico (ej: SearchProducts).
- ***time:*** corresponde al tiempo en milisegundos cuando es generado el diagnóstico.
- ***lp:*** La lista de productos (lp) que tiene el evento enlazador que generó el diagnóstico.
- ***event_init:*** El evento inicial del flujo que es afectado por la falla, y es usado como insumo para la búsqueda en la metadata (Ej: event_init = E_5).
- ***event_end:*** El evento final del flujo que es afectado por la falla, y es usado como insumo para la búsqueda en la metadata (Ej: event_init = E_9).
- ***Timestamp:*** corresponde al tiempo en formato de tiempo entendido por los humanos.

- **Respuesta de la Ontología:**
  - ***getRepairMethod:*** Retorna los distintos mecanismos de reparación que pueden aplicarse cuando una falla sucede en la aplicación.
- **Respuesta de la metadata**:
  - ***methodrepair:*** Indica el método de reparación que se puede aplicar para corregir la aplicación (Ej: substituteFlow).
  - ***id:*** corresponde al identificador de la invocación de la aplicación de comercio electrónico (ej: id = 4).
  - ***event_init:*** El evento inicial del flujo que es afectado por la falla, y es usado como insumo para la búsqueda en la metadata (Ej: event_init = E_5).

- ***event_end:*** El evento final del flujo que es afectado por la falla, y es usado como insumo para la búsqueda en la metadata (Ej: event_init = E_9).

Falla	Respuesta del diagnóstico de la Crónica Distribuida	Respuesta del servicio con la Ontología "Fault-Recovery"	Respuesta del servicio con la selección de la reparación (metadata)
Calidad de Servicio (Delay)	<?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput4_MsgObj xmlns:msgns="warehouseChronicle _iep"> <id>3</id> <fault>QualityOfService</fault> <faulttype>Delay</faulttype> <time>1408573440066</time> <lp>10</lp> <event_init>E8</event_init> <event_end>E8</event_end> <Timestamp>2014-08-20T17:54:05.927-04:30</Timestamp> </msgns:StreamOutput4_MsgObj>	<?xml version="1.0" encoding="utf-8"?> <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope"> <S:body> <ns2:getRepairMethodResponse xmlns:ns2="http://ws/"> <return>reassign;substitute;substituteFlow</return> </S:body> </S:Envelope>	<?xml version="1.0" encoding="utf-8"?> <msgrepair:warehouse> <methodrepair>substituteflow</methodrepair> <id>4</id> <event_init>E5</event_init> <event_end>E9</event_end> </msgrepair:warehouse>
Fuera de Tiempo (Timeout)	<?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput2_MsgObj xmlns:msgns="warehouseChronicle_iep"> <id>4</id> <fault>timeout</fault> <faulttype>N/A</faulttype> <time>1408573625710</time> <lp>2</lp> <event_init>E5</event_init> <event_end>E9</event_end> <Timestamp>2014-08-20T17:57:06.025-04:30</Timestamp> </msgns:StreamOutput2_MsgObj>	<?xml version="1.0" encoding="utf-8"?> <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope"> <S:body> <ns2:getRepairMethodResponse xmlns:ns2="http://ws/"> <return>reassign;retrysubstitute;substituteflow;skipService;skipFlow</return> </S:body> </S:Envelope>	<?xml version="1.0" encoding="utf-8"?> <msgrepair:warehouse> <methodrepair>substituteflow</methodrepair> <id>1</id> <event_init>E5</event_init> <event_end>E9</event_end> </msgrepair:warehouse>

Figura 7.2: Fuentes de conocimiento generadas en las fallas Quality Of Service (Delay) y fuera de tiempo (Crónicas, Ontología y Metadata)

Como se muestra en la Figura 7.2, ARMISCOM fue capaz de diagnosticar y emitir el mecanismo de reparación correcto para las fallas de Calidad de Servicio (Delay) y Timeout. En el caso de Calidad de Servicio, el diagnosticador en el Almacén reconoce la crónica mostrada en la Figura 7.1, y emite el evento a su reparador con la información de la falla (fault: QualityOfService, fault type: Delay,

event_init = 5 y event_end = 9, ver la primera columna en la primera fila: Respuesta del diagnóstico de la Crónica Distribuida, en la Figura 7.2). Con esta información, el reparador en el Almacén realiza la consulta del servicio que contiene la implementación de la ontología Fault-Recovery para la falla de calidad de Servicio, para esto utiliza la información suministrada en el diagnóstico de la falla del diagnosticador (campo fault: QualityOfService del diagnóstico). Así, consulta la ontología internamente dentro del servicio, la cual retorna (infiere) los posibles métodos de solución que pueden ser implementados para corregir esta falla (reassign, substitute y substituteFlow; ver segunda columna en la primera fila de la Figura 7.2: Respuesta del servicio con la Ontología "Fault-Recovery"). Entonces, con la información de los métodos posibles para solventar la falla y la suministrada por el diagnosticador del nivel en que a ocurrido la falla (event_init = E_5 y event_end = E_9), el Reparador realiza la búsqueda en la metadata: el servicio primero busca los métodos reassign y substitute, y debido a que ellos no se encuentran implementados (ver Tabla 7.1), determina que el método a aplicar es substituteFlow con event_init = 5, event_end = 9, y transition = {E_6, E_7, E_8}, aplicando el Algoritmo 3.1. Así, el Algoritmo 3.1 encuentra una Región Equivalente en la metadata (corresponde a id = 1 en la Tabla 7.1). El diagnóstico y corrección de la falla Fuera de Tiempo es similar. Primero, el diagnosticador en el Almacén reconoce la crónica de la Figura 7.1 y emite la información de la falla a su reparador (fault: timeout, fault type: N/A, event_init = E_5 y event_end = E_9, ver la primera columna en la segunda fila de la Figura 7.2: Respuesta del diagnóstico de la Crónica Distribuida). El reparador nuevamente con el nombre "timeout" de la falla invoca el servicio que sirve de interfaz a la ontología Fault-Recovery, la cual retorna (infiere) los posibles métodos a implementar (reassign, retrysubstitute, substituteflow, skipService y skipFlow, vea la segunda columna en la segunda fila de la Figura 7.2: Respuesta del servicio con la Ontología "Fault-Recovery"). Por último, realiza una consulta para determinar el método de reparación a usar en la

metadata, retornando el método de reparación “substituteflow” con id = 1 (ver la Tabla 7.1, y la tercera columna y segunda fila de la Figura 7.2: Respuesta del servicio con la selección de la reparación (metadata)), que corresponde a una Región equivalente para el flujo: event_init = E_5, event_end = E_9 y transition = {E_6, E_7, E_8}, el cual es el único que se encuentra disponible.

7.1.2. Análisis de Resultados

En este caso de estudio se ha utilizado la aplicación SOA de comercio electrónico para comprobar el componente Fuente de Conocimiento de ARMISCOM, utilizando un conocimiento híbrido para gestionar los diferentes aspectos necesarios para garantizar la tolerancia a fallas de una aplicación SOA, primero realizando el diagnóstico de las fallas con la *base de crónicas distribuidas* en lenguaje CQL, generando el diagnóstico de la falla y la parte del flujo de aplicación que se encuentra afectada (event_init y event_end). Posteriormente, con la información de la falla se consulta la *ontología Fault-Recovery*, para correlacionar el nombre de la falla con los posibles métodos de reparación. Por último, con la parte del flujo afectada (event_init y event_end), y los mecanismos almacenados en la *metadata*, es posible obtener el mejor método disponible para solventar la falla en tiempo de ejecución. En la próxima sección se extiende esta prueba poniendo a funcionar a todo ARMISCOM, en varios escenarios, sobre una aplicación SOA real.

Los diferentes patrones de las crónicas distribuidos se utilizan para diagnosticar las fallas (En este caso, se ha mostrado el funcionamiento de los patrones de crónicas Calidad de Servicio QoS (Retardo) y Fuera de tiempo). Cuando se reconoce una crónica distribuida, el diagnóstico genera un archivo con el diagnóstico, que es leído por el componente Reparador. En este experimento se ha comprobado que los patrones de crónicas para estas 2 fallas propuestos en el

Capítulo 4 funcionan, generando los resultados esperados con la información del diagnóstico de la falla y la parte del flujo afectada en la aplicación. Las configuraciones realizadas para obtener los patrones de crónicas para esta falla se muestran en la Tabla 7.2.

Patrón de Crónica	Propiedad no Funcional	Nivel de Aplicación	Configuraciones	Restricciones Atemporales
Fuera de Tiempo (Timeout)	N/A	Flujo	**Proveedor:** Subchronicle Proveedor Fuera-Tiempo generada de Subchronicle S_j Fuera de Tiempo de la Figura 4.29: • ***Equivalencia de eventos:*** ◦ $E_{INVE2} \approx E_4$ (Proveedor). ◦ $E_4 \approx E_{10}$ (Proveedor, $\Delta T_{timeout}$ = 5000ms y ΔT_{espera} = 4000 ms). • ***Acción:*** Emitir el evento enlazador ($BE_{Timeout}$) a Diagnosticador Almacén. **Almacén:**Subchronicle Almacén Fuera-Tiempo generada de Subchronicle S_k Fuera de Tiempo de la Figura 4.29: • ***Equivalencia de eventos:*** ◦ $E_2 = E_5$ (Almacén). ◦ $BE_{Timeout} = BE_{Timeout}$ (generado en Proveedor). • ***Acción:*** Invocar reparador Almacén.	N/A
Retardo (QoS)	Retardo de tiempo	Flujo	**Proveedor:** Subchronicle Proveedor Retardo generada de Subchronicle S_j SLA OR QoS de la Figura 6.19: ◦ ***Equivalencia de eventos:*** ▪ $E_{k1} \approx E_4$ (Proveedor). ▪ $E_{k2} \approx E_{10}$ (Proveedor). ◦ ***Acción:*** Emitir el evento enlazador ($BE_{SLAORQoS} = EB_{Delay}$) a Diagnosticador Proveedor. ◦ **Almacén:**Subchronicle Almacén Retardo generada de Subchronicle S_k SLA OR QoS de la Figura 6.19: • ***Equivalencia de eventos:*** ◦ $BE_{SLAORQoS} = BE_{Delay}$ (generado en Proveedor) con ocurrencia de 3 ($n_1 = 3$, $n_2 > 3$). • ***Acción:*** Invocar reparador Proveedor.	Función noFunctional en el Proveedor (flujo= {E_4, E_{10}}): • $T_{10} - T_4 \geq 2000$ • $T_{10} - T_4 < 5000$ Función noFunctional en el Almacén (flujo= {E_5, E_6, E_7, E_8}): N/A

Tabla 7.2: Configuración de los patrones de crónicas para las fallas Fuera de Tiempo y retardo

El módulo de reparación permite a ARMISCOM inferir las estrategias de reparación de fallas en la composición de los servicios, teniendo en cuenta la

información del contexto de la falla y el problema de la composición del flujo, en tiempo de ejecución. En trabajos anteriores [6, 19, 21] se han correlacionado las acciones de recuperación con los tipo de fallas, en este trabajo se utiliza una ontología fault-recovery para correlacionar las fallas con las acciones de recuperación. Esta ontología puede aumentar (ej. utilizando enfoques de aprendizaje de ontológicas) para incluir nuevas fallas, mecanismos de reparación, entre otras. Además, al convertir el flujo de aplicación en Regiones que son almacenadas en la metadata, proporciona un mecanismo a ARMISCOM para la inferencia precisa de los posibles métodos de reparación a usar, realizando la búsqueda de regiones equivalentes y super-regiones (ver Capítulo 3). Actualmente, el ingreso de la información de la metadata es agregada a ARMISCOM por un ente humano, pero podría agregarse un segundo nivel meta al middleware reflexivo que contenga los componentes necesarios para realizar el proceso de manera automática, requiriendo posiblemente mecanismos de gestión de contexto para encontrar servicios y/o regiones semejantes.

7.2. Auto-reparación (Self-healing) de Fallas en la Composición de Servicios

En experimentos anteriores se han probado los distintos componentes de ARMISCOM, en esta sección se pondrá a prueba la capacidad de ARMISCOM para realizar auto-reparación en la composición de servicios. Para esto, es necesario conectar los distintos componentes del MAPE-K de la arquitectura autonómica en su implementación en OpenESB (presentada en el Capítulo 5, en la Figura 5.2) y se despliega la aplicación de la Industria del Mueble y comercio electrónico en OpenESB. Para realizar los experimentos se tomaron los dos casos de estudio descritos en la sección 6.1.

7.2.1. Caso Industria del Mueble

Para probar el mecanismo de auto-reparación de ARMISCOM, se toma la crónica de violación de SLA que se ha utilizado previamente en el experimento de la sección 6.4.1 (Figura 6.14). La implementación de la crónica en el componente IEP de OpenESB se muestra en la Figura 7.3.

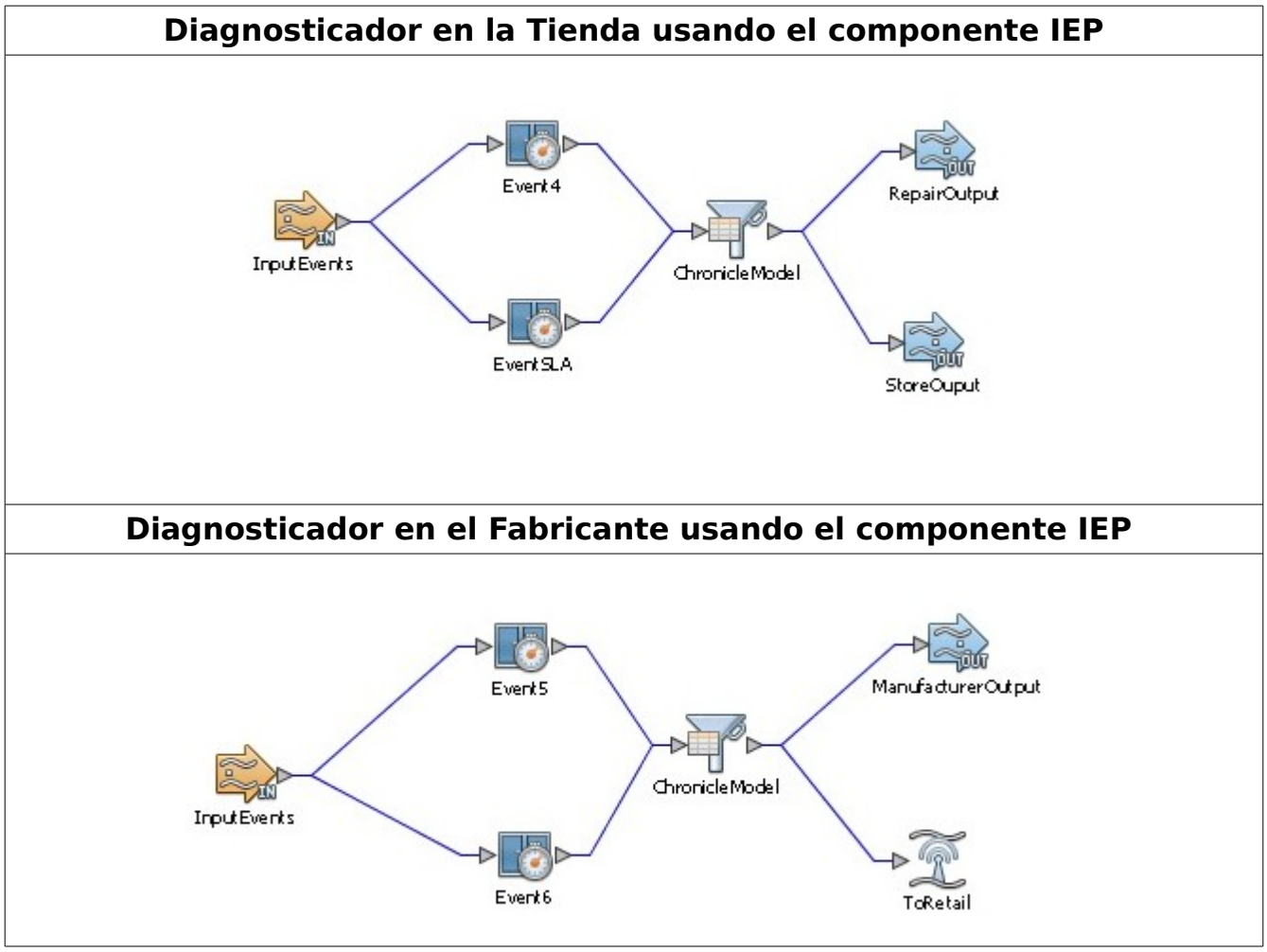


Figura 7.3: Diagnosticador de la Tienda y el Fabricante usando el componente iep de OpenESB

Como se muestra en la figura anterior, el Diagnosticador de la Tienda maneja dos relaciones (relations en CQL) de eventos E_4 y E_{SLA} (evento enlazador generado

en la sub-crónica de el Fabricante), y genera 2 eventos al momento del reconocimiento de la crónica en forma de archivos: RepairOutput (utilizado por el reparador para solventar la falla) y StoreOutput (usado como log y permite mostrar los resultados del reconocimiento). Por otro lado, el Diagnosticador del Fabricante reconoce los eventos E_5 y E_6, y genera al momento de su reconocimiento el evento enlazador BE_{SLA} (provee una llamada SOAP al Diagnosticador en la Tienda), y el mensaje ManufacturerOutput (usado como un registro de incidencias o log). Por otro lado, el único método de reparación de los servicios disponibles en cada sitio en este caso de estudio, y su metadata, se muestra en la Tabla 7.3:

Métodos de Reparación Disponibles						
Id	**Weight**	**RepairMethod**	**Transition**	**Event_init**	**Event_end**	**Operations**
1	1	substituteflow	{Ø}	E4	E5	turnprovider

Tabla 7.3: Métodos disponibles para reparar la aplicación de la Industria del Mueble

El método de reparación “substituteflow” en la metadata consiste en considerar al servicio del Fabricante 2 en la composición (sustituir al servicio por defecto Fabricante 1) cuando sucede una falla en Fabricante 1. Para esto se utiliza la operación disponible en la aplicación de la Industria del Mueble en OpenESB turnprovider (ver sección 6.1.1.1). La implementación del componente Reparador de la instancia de la Tienda en OpenESB se muestra en la Figura 7.4; su funcionamiento se ejecuta en tiempo real: lee el diagnóstico de la falla en el puerto newWSDL_inboundPort, luego realiza la consulta al servicio ontología con el contenido de la falla usando el puerto OntologyPort. Seguidamente, realiza la consulta de la metadata internamente y genera la corrección invocando la operación turnprovider (puerto RetailPort), almacenando la reparación en un archivo usando el puerto outputfile_OutboundPort.

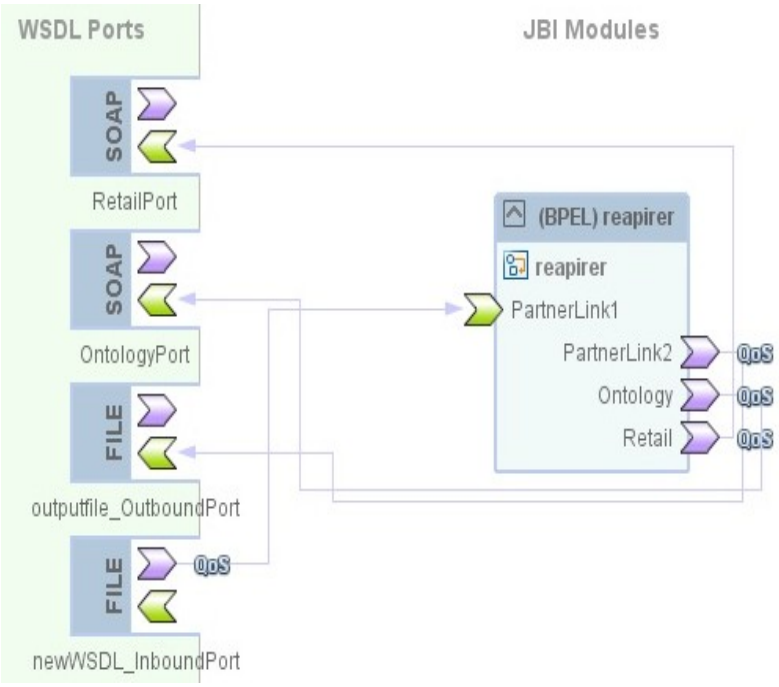


Figura 7.4: Implementación del Reparador en la instancia Tienda en OpenESB

Las posibles correcciones que realice sobre el flujo de la composición, no serán percibidas en la ejecución donde se generó la falla, sino en futuras ejecuciones de la aplicación (invocaciones posteriores a la falla). Esto es debido a que ARMISCOM no tiene un mecanismo que permita realizar la operación de Rehacer (volver a ejecutar la aplicación desde el comienzo), para lo cual requeriría de mecanismos de puntos de chequeo, retorno, etc. Si se desea colocar un mecanismo de rehacer en ARMISCOM, al ser una capa intermedia entre los servicios de la composición, puede detener la ejecución de la aplicación, realizar las reparaciones necesarias, y volver a comenzar la ejecución de la aplicación. Estos aspectos podrán ser desarrollados en futuros trabajos.

7.2.1.1 Aplicación de la Prueba de Auto-sanación en OpenESB

Para verificar el funcionamiento del mecanismo de auto-reparación de ARMISCOM, se utilizó la operación llamada **tuneQualityBehavior** dentro de cada servicio Fabricante 1 y 2 (Manufacturer 1 y 2), para inducir la falla de violación de SLA de Calidad de los Productos. Se ejecuta la aplicación la Industria del Mueble 7 veces (con diferentes patrones de pruebas, en distintos escenarios de funcionamiento: normal y fallas de SLA), añadiendo un atributo id para diferenciar cada invocación. En la Tabla 7.4 se muestran las diferentes invocaciones, con los diferentes valores utilizados para id, Quality, y tuneQualityBehavior.

ID	tuneQualityBehavior		Resultado de la Operación (Calidad)								
			E1	E2	E3	E4 (PSvar)	E5 (PSvar)	E5' (PSvar)	E6	E7	E8
	Manuf1	Manuf2					*Manuf1*	*Manuf2*			
1	FALSE	FALSE	7	14	14	*14*	*14*	-	*14*	*14*	*14*
2	FALSE	FALSE	1	7	7	7	7	-	7	7	7
3	**TRUE**	FALSE	**5**	**5**	**5**	***5***	***4***	**-**	***4***	***4***	***4***
4	FALSE	FALSE	16	16	16	16	-	16	16	16	16
5	**FALSE**	**TRUE**	**14**	**14**	**14**	**14**	**-**	**13**	**13**	**13**	**13**
6	FALSE	FALSE	2	2	2	2	2	-	2	2	2
7	FALSE	FALSE	10	10	10	10	10	-	10	10	10

Tabla 7.4: Invocaciones de la aplicación Industria del Mueble

Para cada sub-crónica que es reconocida, los eventos enlazadores y el diagnóstico de las fallas generados se almacenan en los archivos (ManufacturerOutput.xml y RepairOutput.xml). Adicionalmente, el componente de reparación genera un archivo llamado RetailRepairResult.xml, donde almacena el resultado de aplicar el mecanismo de auto-reparación. Así, el contenido de los archivos generados por el reconocimiento de fallas por cada sub-crónica, y el método de reparación implementado para corregir las fallas, para la aplicación Industria del Mueble, se muestra en la Figura 7.5. Para facilitar la lectura de la

información generada por los eventos enlazadores, el diagnóstico de las fallas y la reparación de las fallas, el contenido de los archivos se describe a continuación:

- **Evento enlazador** (primera o tercera fila, y primera columna, de la Figura 7.5):
  - ***id:*** corresponde al identificador de la invocación de la aplicación de la industria del mueble (ej: id = 4).
  - ***event:*** indica el evento enlazador que es generado cuando la sub-crónica es reconocida.
  - ***time:*** corresponde al tiempo en milisegundos cuando es generado el evento enlazador.
  - ***Timestamp:*** corresponde al tiempo cuando el evento enlazador es generado, expresado en formato de tiempo entendido por los humanos.
  - ***lp:*** La lista de productos (lp) cuando el evento enlazador es generado.
- **Diagnóstico de fallas** (primera o tercera fila, y segunda columna, de la Figura 7.5):
  - ***id:*** corresponde al identificador de la invocación de la aplicación de la industria del mueble (ej: id = 3).
  - ***fault:*** nombre de la falla que es diagnosticada (ej: ServiceLevelAgreement), y que es usada como fuente para el servicio gestor de la ontología.
  - ***faulttype:*** corresponde al tipo de falla, es usado generalmente para agregar información en el diagnóstico (ej: Quality).
  - ***time:*** corresponde al tiempo en milisegundos cuando es generado el diagnostico.

- ***lppre:*** la lista de productos (lp) en un evento previo al diagnóstico, y añade información al diagnóstico (ej: el lp en el evento E_5).
- ***lp:*** la lista de productos (lp) que tiene el evento enlazador que genero el diagnóstico.
- ***event_init:*** el evento inicial del flujo que es afectado por la falla, y es usado como insumo para la búsqueda en la metadata (Ej: event_init = E_4).
- ***event_end:*** el evento final del flujo que es afectado por la falla, y es usado como insumo para la búsqueda en la metadata (Ej: event_end = E_5).

- **Reparación de la Falla** (segunda o cuarta fila, y segunda columna, de la Figura 7.5):
  - ***id:*** corresponde al identificador de la invocación de la aplicación de la industria del mueble (ej: id = 4).
  - ***fault:*** nombre de la falla que es reparada (ej: SLA), que es usada como fuente para el servicio gestor de la ontología.
  - ***measure:*** agrega información acerca de la métrica de la falla que es reparada (Ej: Quality).
  - ***methodspossibles:*** los posibles métodos de reparación que pueden ser aplicados por el reparador, separados por punto y coma (;), y corresponde a la consulta al servicio que gestiona la ontología (Ej: parametersUpdate;substitute;substituteFlow) .
  - ***methodrepair:*** indica el método de reparación que se ha aplicado para corregir la aplicación (Ej: substituteFlow).
  - ***repair_state:*** indica si la falla fue reparada: **TRUE** si se corrigió y **FALSE**

si no fue posible reparar la aplicación. En algunos casos no es posible reparar la aplicación si los métodos de reparación que se puedan aplicar (methodspossibles) no se encuentran en la metadata.

- ***time:*** corresponde al tiempo en milisegundos cuando es generado la reparación.
- ***previuosstate:*** agrega información del estado en que se encontraba la aplicación antes de la reparación.
- ***currentstate:*** agrega información del estado en que se encuentra la aplicación durante la reparación.

Id	Fabricante (Output)	Tienda (Output)
3	**Diagnosticador** <?xml version="1.0" encoding="utf-8"?> <msgns:ManufacturerOutput_MsgObj xmlns:msgns="ManufacturerChronicle_iep"> <id>3</id> <event>BESLA</event> <time>1410352813133</time> <Timestamp>2014-09-10 T08:10:13.239-04:30</Timestamp> <lp>4</lp> </msgns:ManufacturerOutput_MsgObj> **ManufacturerOutput.xml**	**Diagnosticador** <?xml version="1.0" encoding="utf-8"?><msgns:StoreOuput_MsgObj xmlns:msgns="RetailChronicle_iep"> <id>3</id> <fault>ServiceLevelAgreement(SLA)</fault> <faulttype>Quality</faulttype> <lpretail>4</lpretail> <lp>5</lp> <event_init>E4</event_init > <event_end>E5</event_end> <time>1410352813133</time> </msgns:StoreOuput_MsgObj> **RepairOutput.xml**
	-	**Reparador** <?xml version="1.0" encoding="utf-8"?> <msgrepair> <id>3</id> <fault>ServiceLevelAgreement(SLA)</fault> <measure>Quality</measure> <methodspossibles>parametersUpdate;substitute;substituteFlow</methodspossibles> <methodrepair>substitute</method> <repair_state>true</repair_state> <time>1410352815444</time> <previuosstate>Furniture Manufacturer 1</previuosstate> <currentstate>Furniture Manufacturer 2</currentstate> </msgrepair> **RetailRepairResult.xml**
5	**Diagnosticador** <?xml version="1.0" encoding="utf-8"?> <msgns:ManufacturerOutput_MsgObj xmlns:msgns="ManufacturerChronicle_iep"> <id>5</id> <event>BESLA</event> <time>1410353325526</time> <lp>13</lp> <Timestamp>2014-09-10 T08:18:45.632-04:30</Timestamp> </msgns:ManufacturerOutput_MsgObj> **ManufacturerOutput.xml**	**Diagnosticador** <?xml version="1.0" encoding="utf-8"?><msgns:StoreOuput_MsgObj xmlns:msgns="RetailChronicle_iep"> <id>3</id> <fault>ServiceLevelAgreement(SLA)</fault> <faulttype>Quality</faulttype> <lpretail>13</lpretail> <lp>14</lp> <event_init>E4</event_init > <event_end>E5</event_end> <time>11410353325526</time> </msgns:StoreOuput_MsgObj> **RepairOutput.xml**
	Reparador	**Reparador** <?xml version="1.0" encoding="utf-8"?> <msgrepair> <id>5</id> <fault>ServiceLevelAgreement(SLA)</fault> <measure>Quality</measure> <methodspossibles>parametersUpdate;substitute;substituteFlow</methodspossibles > <methodrepair>substitute</method> <repair_state>true</repair_state> <time>1410353328495</time> <previuosstate>Furniture Manufacturer 2</previuosstate> <currentstate>Furniture Manufacturer 1</currentstate> </msgrepair> **RetailRepairResult.xml**

Figura 7.5: Resultado del reconocimiento de las crónicas y los mecanismos de reparación de la aplicación Industria del Mueble

Como se muestra en la Figura 7.5, cuando el funcionamiento de la aplicación de la Industria del Mueble es normal (id = {1, 2, 4, 6, 7}, ver Tabla 7.4) ninguna

sub-crónica es activada. De lo contrario, en las invocaciones id = {3, 5} (ver Tabla 7.4) se genera el reconocimiento de la falla de violación de SLA para la calidad de los productos. En ambos casos, los eventos generados son almacenados en el archivo ManufacturerOutput.xml (emite el evento enlazador BE_{SLA} hacia la Tienda (ver Figura 7.5, Diagnosticador Fabricante, id = 3, primera fila y segunda columna), que es leído por el diagnosticador en la Tienda en su sub-crónica, y emite evento con el reconocimiento de la falla (archivo RepairOutput.xml, ver Figura 7.5, segunda columna y fila, diagnosticador Tienda, id = 3).

Posteriormente, con el reconocimiento de la falla el componente Reparador en la Tienda obtiene el diagnóstico del archivo generado por el diagnosticador (RepairOutput.xml), y genera las operaciones necesarias para solventar la falla en la composición (operación **turnprovider** de la Tienda, ver sección 6.1.1.1), guardando el resultado de la reparación en el archivo RetailRepairResult.xml (ver Figura 7.5, segunda columna y fila, reparador Tienda, id = 3).

Para hacer más comprensible las reparaciones realizadas por el reparador de la Tienda mostradas en el archivo RepairOutput.xml en ambas invocaciones (id = 3 y 5), la Figura 7.6 muestra gráficamente los cambios realizados en la composición. La aplicación comienza con el servicio Fabricante 1, en el instante en que la falla de violación de SLA para la calidad sucede y es reconocida en la ejecución con id = {3} (ver Tabla 7.4 y Figura 7.5), el servicio Fabricante 1 es sustituido por el Fabricante 2. Posteriormente, en la invocación con id = {5} el servicio Fabricante 2 es sustituido por el servicio Fabricante 1. La operación invocada para implementar el mecanismo de reparación turnprovider ha sido explicada en la sección 6.1.1.1 del caso de estudio.

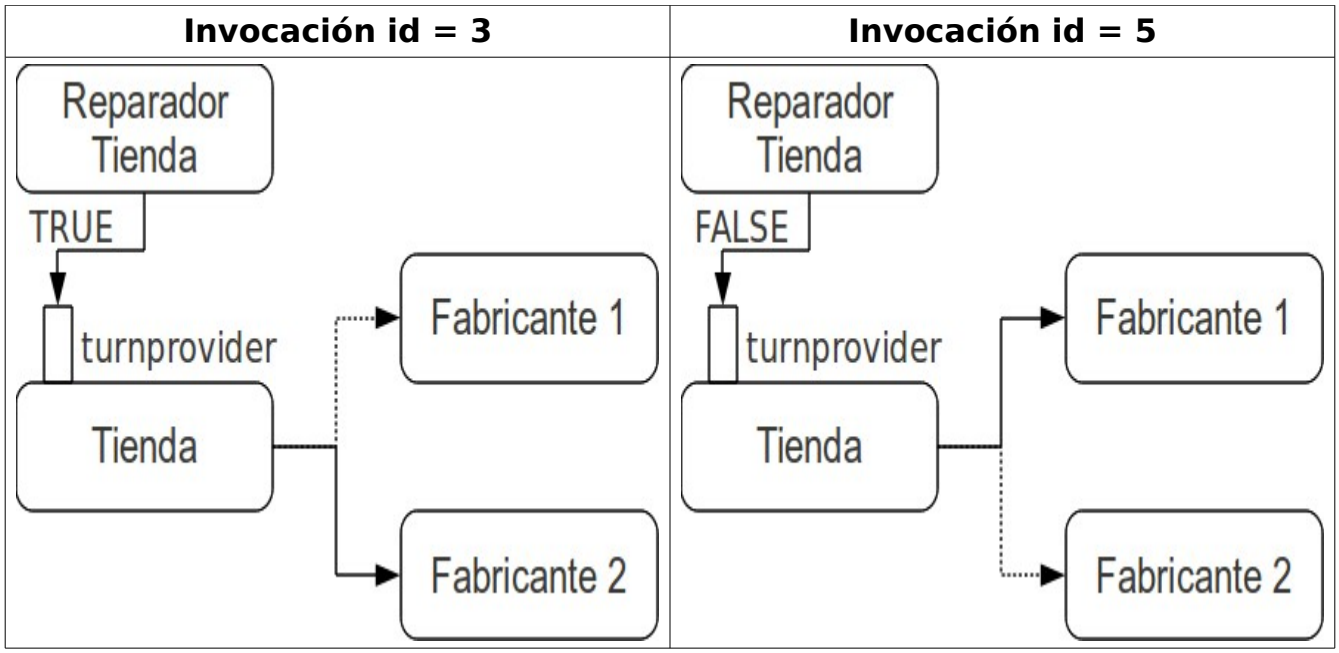


Figura 7.6: Reparaciones realizadas por el componente Reparador de la Tienda

7.2.2. Caso Comercio Electrónico

Para comprobar el funcionamiento de ARMISCOM en este escenario, se toman las siguientes fallas:

- Violación de SLA para el Almacén (caso Búsqueda de Productos).
- Violación de SLA para el Almacén (caso Empaqueta y envía).
- Calidad de Servicio - QoS (Delay).
- Fuera de Tiempo (Timeout).
- La Interfaz pudo Haber Cambiado (Interface Might Have Changed).

El diseño de las crónicas y su implementación en el lenguaje CQL para las cuatro primeras fallas, han sido mostradas en los experimentos anterior (Figura

6.16, Figura 6.17 y Figura 7.1). En el caso de la la crónica de retardo del almacén, para la frecuencia de ocurrencia se ha seleccionado el valor de 2 ($n_1 = 2$, $n_2 > 2$). En el caso de la falla de la Interfaz pudo Haber Cambiado, se toma el patrón de crónicas distribuido para este tipo de falla de la Figura 4.9. Su implementación en el lenguaje CQL se muestra en la Figura 7.7:

Crónica distribuida: Interfaz pudo Haber Cambiado

```
Subchronicle Proveedor Interfaz pudo Haber Cambiado {
 SELECT
      ISTREAM(
       id => E4.id,
                                fault     => 'InterfaceMightHaveChanged',
       faulttype => 'N/A',
       time => EVPARAMICOM.time,
       lpsupplier => EVPARAMICOM.lp,
       lp => E4.lp,
       event_init => 4,
       event_end => 5,
       to => 'Reparador Proveedor',
     )
 FROM
     E4[86401],
     ENOEVENT[86401],
     E4'[15000],
     EVPARAMICOM[now]
 WHERE
     ENOEVENT.time >= E4.time + 5000 AND
     EVPARAMICOM.id = E4.id AND
     E4'.id <> E4.id
 }
```

```
Subchronicle Almacén Interfaz pudo Haber Cambiado {
 SELECT
     ISTREAM(
       id => E5.id,
       event => 'EBParamIcomp',
       time => E5.time,
       lp => E5.lp,
       to => 'Diagnosticador Proveedor',
     )
 FROM
     E5PRIMA[85400]
     E5[now]
 WHERE
     E5.pl <= 0
 }
```

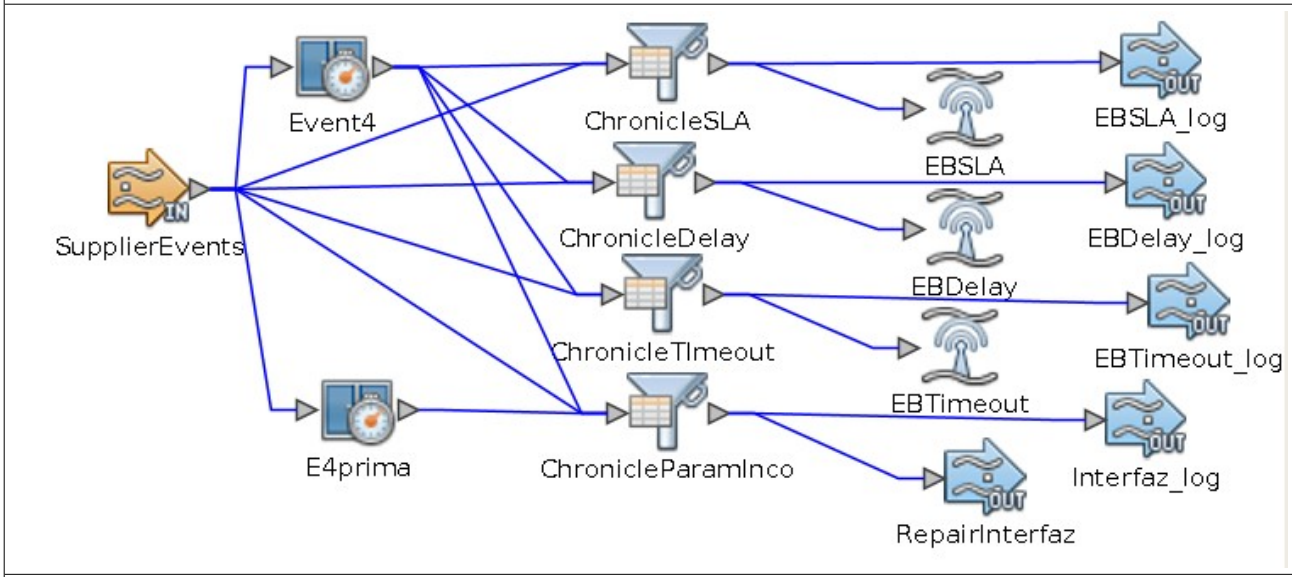
Figura 7.7: Patrón de Crónica distribuida para la falla Interfaz pudo Haber Cambiado en CQL

La falla Interfaz pudo Haber Cambiado consiste en recibir un evento con incompatibilidad de parámetros en su invocación, y en invocaciones previas la aplicación ha funcionado sin ningún problema. Así, para mostrar la configuración de la crónica de la Figura 7.7, en el diagnosticador Almacén, se considera que ha recibido previamente el evento E_{5PRIMA} (se coloca como una relación de E_5), se

invoca previamente a la operación de recibir el requerimiento del Proveedor (se almacena en la relación por un día - 86400 seg) sin problemas, y posteriormente (ahora, now) se recibe el evento E_5 con incompatibilidad de parámetros. Al reconocer la sub-crónica, el Almacén genera el evento enlazador $EB_{ParamIcomp}$ hacia el diagnosticador del Proveedor. Con el evento enlazador recibido, las invocaciones que se han realizado previamente de la aplicación (E_4 y E_4' en forma de relaciones) sin problemas, y al no haber recibido previamente ningún evento enlazador haciendo referencia al problema de la incompatibilidad de parámetros ($E_{NOEVENT}$, el problema es la primera vez que sucede en la aplicación), la crónica en el Proveedor reconoce la falla y envía su contenido a su reparador.

La implementación de las crónicas en el componente IEP de OpenESB se muestra en la Figura 7.8.

Crónicas en CQL usadas por el dignosticador Proveedor



Crónicas en CQL usadas por el dignosticador Almacén

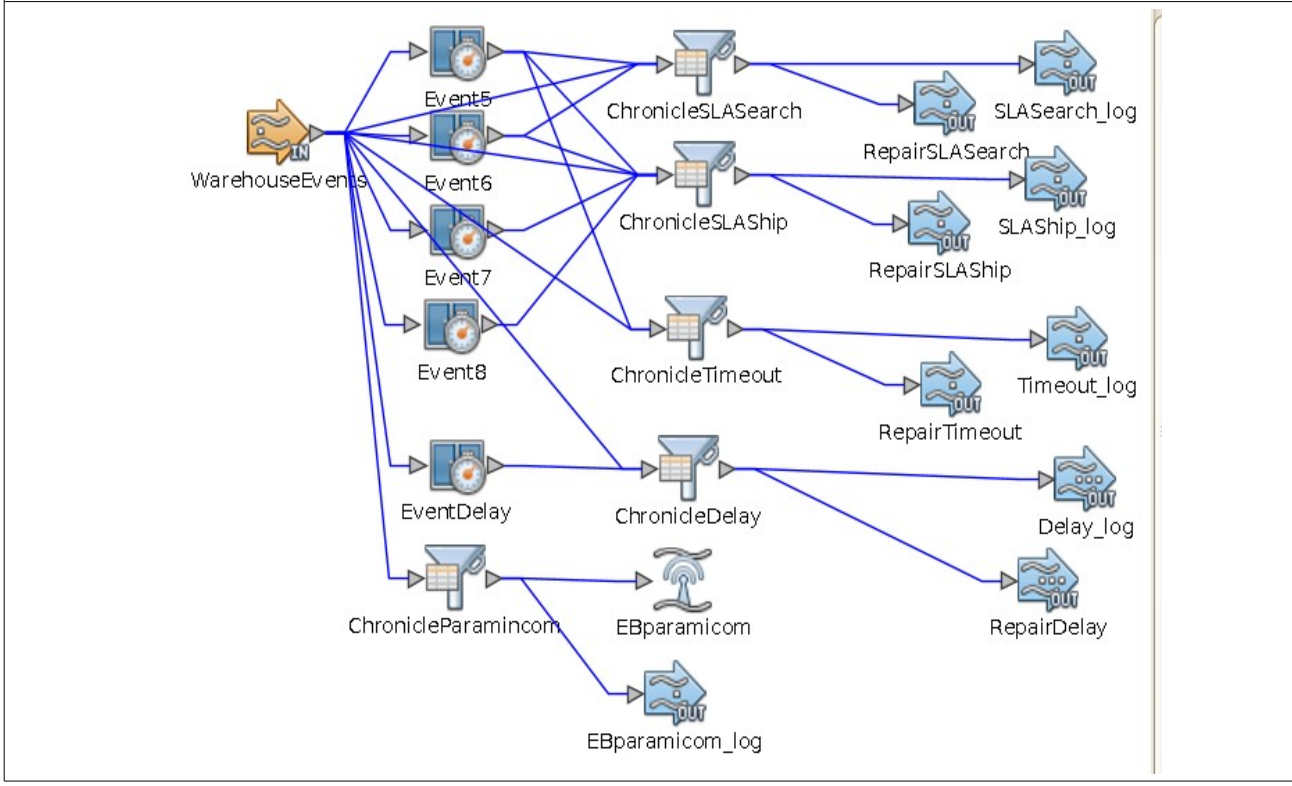


Figura 7.8: Crónicas en CQL de los diagnosticadores de la aplicación de Comercio electrónico usando el componente IEP de OpenESB

El diagnosticador en el Proveedor gestiona dos relaciones de eventos E_4 y E_{4prima} y un stream. El diagnosticador genera 3 eventos enlazadores: el evento enlazador EB_{SLA} para la sub-crónica en el Almacén cuando la sub-crónica de violación de SLA es activada en los casos de "Búsqueda de Productos" y "Empaqueta y envía" (ver la sección 6.4.2, sub-crónica Proveedor SLA), y su contenido es almacenado en el archivo EBSLA_log.xml; el evento enlazador EBt_{imeout}, que es enviado a la sub-crónica Almacén Fuera-Tiempo en el Almacén, cuando la sub-crónica Proveedor Fuera-Tiempo es reconocida en el Proveedor, y su contenido es almacenado en el archivo EBTimeout_log; y el evento enlazador EB_{Delay} para el Almacén (reparador en el Almacén, ver la sección 7.1), cuando la sub-crónica Proveedor Retardo es reconocida, y su contenido es almacenado en el archivo EBDelay_log.xml. Adicionalmente, el diagnosticador en el Proveedor genera el diagnóstico de la falla Interfaz pudo Haber Cambiado en el archivo RepairInterfaz.xml, que es leído por el reparador para corregir la falla. En una acción similar, El reparador Proveedor almacena el resultado de la reparación en el archivo interfazR.xml (Incompatibilidad de Parámetros), usados como un registro de incidencias o log.

Por otro lado, el diagnosticador en el Almacén funciona de manera similar a su par Proveedor, genera el evento enlazador $EB_{paramincom}$ cuando es activada la sub-crónica Almacén Interfaz pudo Haber Cambiado en el Almacén (la crónica de esta falla se muestra en la Figura 7.7), y su contenido es almacenado en el archivo EBparamicom_log.xml, Adicionalmente, el diagnosticador del Almacén es el encargado de diagnosticar las fallas de violación de SLA en los casos "Búsqueda de Productos" y "Empaqueta y envía" (Figura 6.16, Figura 6.17) y retardo de tiempo (Figura 7.1), para esto recolecta los eventos en forma de relaciones Event5, Event6, Event7 y Event8 (E_5, E_6, E_7, E_8, respectivamente), EventDelay (EB_{Delay}) y un Stream (EB_{SLA}, corresponde al evento enlazador generado en el Proveedor cuando se viola alguna restricción SLA, y EBDelay cuando se informa

del retardo de tiempo por parte del Proveedor) que sirven de insumo para reconocer las sub-crónicas en el Almacén SLA Búsqueda, Almacén SLA Empaqueta y Almacén Retardo, respectivamente (se genera el evento enlazador en los tres casos, la crónicas en el Almacén se activan dependiendo del contexto de los demás eventos en diagnóstico). El resultado del diagnóstico de las 3 sub-crónicas es almacenado en los archivos RepairSLASearch.xml, RepairSLAShip.xml (Figura 7.8: Diagnosticador Almacén) y RepairDelay.xml. Adicionalmente, el diagnosticador en el Almacén recolecta la información de la relation Event5 y el stream (EBtiemout) para reconocer la sub-crónica Almacén Fuera-Tiempo y reconocer la falla de Fuera de Tiempo (el diagnóstico se almacena en el archivo RepairTimeout.xml). Por último, el resultado de la operaciones de reparación del Almacén es almacenado en los archivos SLASearchR.xml (Búsqueda de Productos), SLAShipR.xml (Empaqueta y envía), timeoutR.xml (fuera de tiempo) y delayR.xml, usados como un registro de incidencias o log.

La implementación de los componentes reparadores de las instancias del Proveedor y Almacén en OPenESB son similares a las mostradas en el caso de la aplicación de la Industria del Mueble (Figura 7.4). Las posibles correcciones que se realizan sobre el flujo de la composición, no son percibidas en la actual ejecución, sino en las ejecuciones posteriores de la aplicación.

En cuanto al contenido de la metadata utilizada por los reparadores en esta aplicación, son los mismos que se han mostrado y descritos en la Tabla 7.3 (sección 7.2.1). Para facilitar la implementación de los mecanismos de reparación disponibles en la metadata de la Tabla 7.3, se han agregado las siguientes operaciones al reparador del Proveedor y del Almacén, y se han usado algunas operaciones propias del Almacén definidas en la sección 6.1.2.1 (ver Figura 7.9):

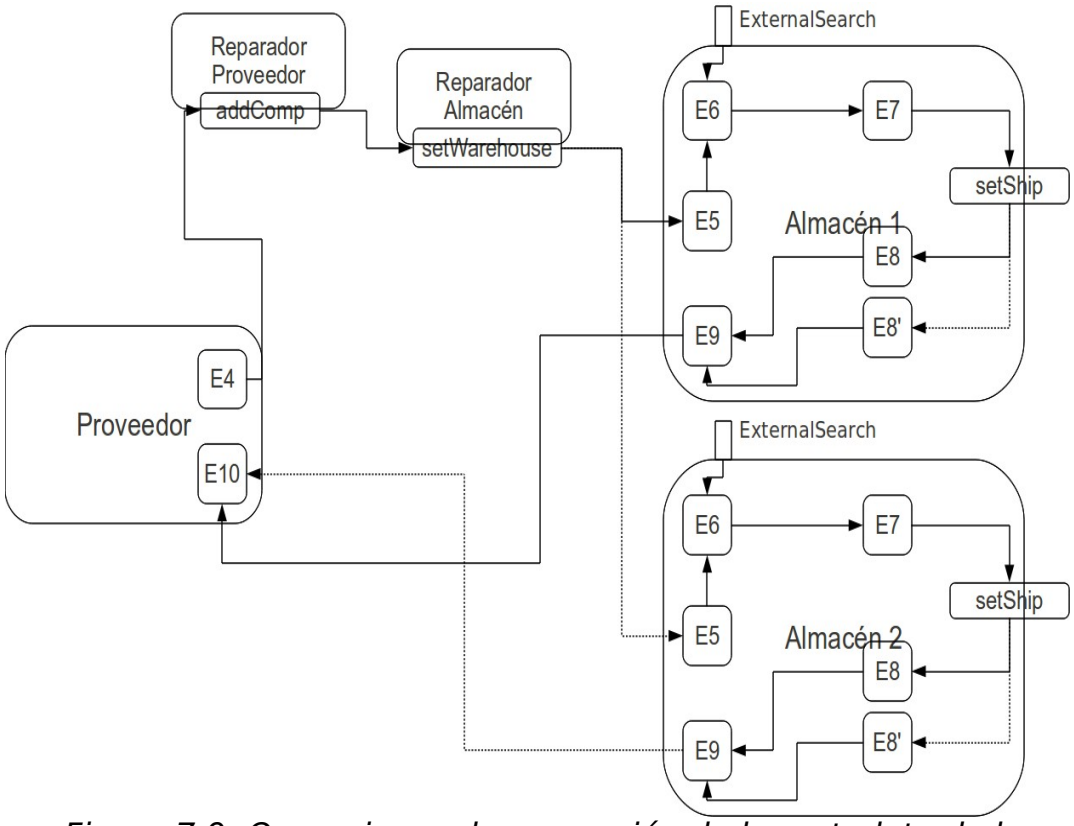


Figura 7.9: Operaciones de reparación de la metadata de los reparadores de la aplicación de comercio electrónico

Donde las operaciones de reparación son:

- **Reparador Almacén:**
  - ***setWarehouse:*** Permite cambiar el servicio Almacén (todas las operaciones E_5, E_6, E_7, E_8 y E_9) por uno equivalente Almacén 2. Si el valor suministrado a la operación es warehouse = TRUE, el servicio que se encuentra en funcionamiento es el Almacén 1. Por el contrario, si warehouse = FALSE, el servicio en funcionamiento es Almacén 2. El valor por defecto de esta operación es warehouse = TRUE. Es usada por el reparador del Almacén por el mecanismo de reparación substituteflow

con id = 1 de la metadata de la Tabla 7.3.

- ***SetShip:*** Permite modificar la empresa para realizar los envíos de los productos (definida en la sección 6.1.2.1). Es usada por el reparador del Almacén por el mecanismo de reparación substituteflow con id = 4 de la metadata de la Tabla 7.3.

- **Servicio Almacén:**

  - ***ExternalSearch:*** Permite ajustar las búsqueda hacia otros Almacenes (definida en la sección 6.1.2.1). Es usada por el reparador del Almacén, por el mecanismo de reparación parametersUpdate con id = 2 de la metadata de la Tabla Tabla 7.3.

- Reparador Proveedor:

  - ***addComp:*** Permite agregar un filtro compensador entre los eventos E_4 y E_5, cuya función es transformar el mensaje provisto por el Proveedor (lp) y usado por el reparador Proveedor. La operación realiza una transformación básica de valor absoluto al valor de lp: si el valor lp provisto por el Proveedor es negativo (ej. -1), el compensador transforma ese valor a un entero positivo como entrada al Almacén (lp = 1). Esta operación es utilizada por el mecanismo de reparación CompleteMissingParameter con id = 3 de la metadata de la Tabla 7.3.

7.2.2.1 Aplicación de la Prueba de Auto-sanación en OpenESB

Para realizar las pruebas, se ejecuta la aplicación de Comercio Electrónico 12 veces (se han definido un conjunto de pruebas, para comprobar la detección de fallas en distintos escenarios), añadiendo un atributo id para diferenciar cada

invocación. Para inducir las fallas en la aplicación, se han usado nuevamente las operaciones definidas en la sección 6.1.2.1: tuneSearchBehavior (violación de SLA de Búsqueda de Productos, ver sección 6.4.2.1), tuneShip (violación de SLA de Empaqueta y envía, ver sección 6.4.2.1) y tuneDelay (retardos del Almacén, ver sección 7.1.1). En la Tabla 7.5 se muestran las diferentes invocaciones de la aplicación comercio electrónico, con los diferentes valores utilizados para el atributo id, Products list (lp, Resultado de las Operaciones), el servicio Almacén utilizado (Almacén 1 y Almacén 2), tuneSearchBehavior, tuneShip y tuneDelay.

id	tuneSearchBehavior	tuneShip	tuneDelay	Resultado de las Operaciones (lp)														
				E1	E2	E3	E4	E5	E6	E7	E8	E8'	E9	E10	E11	E12	E13	E14
Almacén 1																		
Almacén 2																		
1	FALSE	FALSE	0	11	11	11	11	11	11	11	11	-	11	11	11	11	11	11
	FALSE	FALSE	0					-	-	-	-	-	-					
2	FALSE	FALSE	0	8	8	8	8	8	8	8	8	-	8	8	8	8	8	8
	FALSE	FALSE	0					-	-	-	-	-	-					
3	FALSE	FALSE	0	5	5	5	5	5	5	5	5	-	5	5	5	5	5	5
	FALSE	FALSE	0					-	-	-	-	-	-					
4	**TRUE**	**FALSE**	**0**	**4**	**4**	**4**	**4**	**4**	**0**	**0**	**0**	**-**	**0**	**0**	**0**	**0**	**0**	**0**
	FALSE	**FALSE**	**0**					**-**	**-**	**-**	**-**	**-**	**-**					
5	FALSE	FALSE	0	6	6	6	6	6	6	6	6	-	6	6	6	6	6	6
	FALSE	FALSE	0					-	-	-	-	-	-					
6	FALSE	FALSE	0	3	3	3	3	3	3	3	3	-	3	3	3	3	3	3
	FALSE	FALSE	0					-	-	-	-	-	-					
7	FALSE	FALSE	0	2	2	2	2	2	2	2	2	-	2	2	2	2	2	2
	FALSE	FALSE	0					-	-	-	-	-	-					
8	**FALSE**	**TRUE**	**0**	**0**	**0**	**0**	**0**	**10**	**10**	**10**	**9**	**-**	**9**	**9**	**9**	**9**	**9**	**9**
	FALSE	**FALSE**	**0**					**-**	**-**	**-**	**-**	**-**	**-**					
9	**FALSE**	**FALSE**	**6000**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**-**	**7**	**7**	**7**	**7**	**7**	**7**	**7**
	FALSE	**FALSE**	**0**					**-**	**-**	**-**	**-**	**-**	**-**					
10	**FALSE**	**FALSE**	**0**	**9**	**9**	**9**	**9**	**-**	**-**	**-**	**-**	**-**	**-**	**9**	**9**	**9**	**9**	**9**
	FALSE	**FALSE**	**3000**					**9**	**9**	**9**	**-**	**9**	**9**					
11	**FALSE**	**FALSE**	**0**	**14**	**14**	**14**	**14**	**-**	**-**	**-**	**-**	**-**	**-**	**14**	**14**	**14**	**14**	**14**
	FALSE	**FALSE**	**3000**					**14**	**14**	**14**	**-**	**14**	**14**					
12	**FALSE**	**FALSE**	**0**	**6**	**6**	**6**	**-1**	**-1**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
	FALSE	**FALSE**	**0**					**-**	**-**	**-**	**-**	**-**	**-**					

Tabla 7.5: Invocaciones de la aplicación de Comercio Electrónico

Para facilitar la lectura de la información generada por los eventos enlazadores, diagnóstico de fallas y la reparación de las fallas, el contenido de los archivos se describe a continuación:

- **Evento enlazador** (los archivos llevan por nombre **BEnombredelaFalla_log.xml**, Ej: para la falla fuera de tiempo - timeout el nombre del archivo es BETimeout_log.xml, ver primera columna y fila de la

Figura 7.10):

- ***id:*** corresponde al identificador de la invocación de la aplicación comercio electrónico (ej: id = 4).
- ***event:*** indica el evento enlazador que es generado cuando la sub-crónica es reconocida.
- ***time:*** corresponde al tiempo en milisegundos cuando es generado el evento enlazador.
- ***Timestamp:*** corresponde al tiempo en formato de tiempo entendido por los humanos.
- ***lppre:*** la lista de productos (lp) en un evento previo al diagnóstico y añade información al diagnóstico (ej: el lp en el evento E_5).
- ***lp:*** la lista de productos (lp) cuando el evento enlazador es generado.

- **Diagnóstico de fallas** (los archivos llevan por nombre **RepairnombredelaFalla.xml**, Ej: para la falla fuera de tiempo - timeout el nombre del archivo es RepairTimeout.xml, ver primera fila y segunda columna de la Figura 7.10) :
  - ***id:*** corresponde al identificador de la invocación de la aplicación comercio electrónico (ej: id = 4).
  - ***fault:*** nombre de la falla que es diagnosticada (ej: ServiceLevelAgreement), y que es usada como fuente para el servicio gestor de la ontología.

- ***faulttype:*** corresponde al tipo de falla, es usado generalmente para agregar información en el diagnóstico (ej: SearchProducts).
- ***time:*** corresponde al tiempo en milisegundos cuando es generado el diagnóstico.
- ***lp:*** la lista de productos (lp) que tiene el evento enlazador que genero el diagnóstico
- ***event_init:*** el evento inicial del flujo que es afectado por la falla, y es usado como insumo para la búsqueda en la metadata (Ej: event_init = E_5).
- ***event_end:*** el evento final del flujo que es afectado por la falla, y es usado como insumo para la búsqueda en la metadata (Ej: event_init = E_9).

- **Reparación de la Falla** (los archivos llevan por nombre **nombredelaFallaR.xml**, Ej: para la falla fuera de tiempo - timeout el nombre del archivo es TimeoutR.xml, ver segunda columna, y segunda o cuarta fila, de la Figura 7.10):
  - ***id:*** corresponde al identificador de la invocación de la aplicación comercio electrónico (ej: id = 4).
  - ***fault:*** nombre de la falla que es reparada (ej: ServiceLevelAgreement), que es usada como fuente para el servicio gestor de la ontología.
  - ***measure:*** agrega información acerca de la métrica de la falla que es reparada (Ej: SearchProducts).

- ***methodspossibles:*** los posibles métodos de reparación que pueden ser aplicados por el reparador separador por punto y coma (;), y corresponde a la consulta del servicio que gestiona la ontología (Ej: parametersUpdate;substitute;substituteFlow) .

- ***methodrepair:*** indica el método de reparación que se ha aplicado para corregir la aplicación (Ej: substituteFlow).

- ***repair_state:*** indica si la falla fue reparada: **TRUE** si se corrigió y **FALSE** si no fue posible reparar la aplicación. En algunos casos no es posible reparar la aplicación, si los métodos de reparación que se puedan aplicar (methodspossibles) no se encuentran en la metadata.

- ***time:*** corresponde al tiempo en milisegundos cuando es generado la reparación.

- ***previuosstate:*** agrega información del estado en que se encontraba la aplicación antes de la reparación.

- ***currentstate:*** agrega información del estado actual en que se encuentra la aplicación.

La aplicación comienza con la composición de comercio Electrónico con el servicio Almacén 1 (comienza por defecto con este servicio). Como se muestra en la Tabla 7.5, cuando el funcionamiento de la aplicación es normal (id = {1, 2, 3, 5, 6, 7}) ninguna sub-crónica es activada. En la invocaciones id = {4, 8} se generan las fallas de violación de SLA "Búsqueda de Productos" y "Empaqueta y envía" (ver Tabla 7.5, las columnas para los eventos E_6, E_7, E_8 y E_9 de las invocaciones id = {4, 8}). En este caso, se generan dos eventos enlazadores EB_{SLA} en el diagnosticador del Proveedor (sub-crónica Proveedor SLA, Figuras 6.16 y

6.17), y son enviados a las sub-crónicas Almacén SLA Búsqueda (Figura 6.16) y Almacén SLA Empaqueta (Figura 6.17) en el Almacén (Figura 7.10, Diagnosticador Proveedor, id = {4, 8}, archivo EBSLA_log.xml). En la invocación id={4} la sub-crónica “Búsqueda de Productos” es reconocida en el Almacén, y la falla de violación de SLA es diagnosticada en la sub-crónica Almacén SLA Búsqueda (archivo RepairSLASearch.xml en Figura 7.10, Reparador del Almacén. Id = 4), y enviada al reparador del Almacén. Posteriormente, con el diagnóstico de la falla de SLA = fault: ServiceLevelAgreement, fault type: SearchProducts, event_init = E_6 y event_end = E_6, el reparador en el Almacén realiza la búsqueda en la ontología (Figura 7.10, campo methodspossibles del documento XML generado por el reparador) y la metadata (Figura 7.10, campo methodrepair del documento XML generado por el reparador), seleccionando el mecanismos de reparación de la metadata con id = 2 (parametersUpdate), reconfigurando la operación de Búsqueda de Productos (E_6) para permitir al servicio Almacén realizar búsqueda de productos en otros almacenes, usando para esto la operación **ExternalSearch** del Almacén (ver SLASearchR.xml en el reparador del Almacén de la Figura 7.10, y gráficamente en la Figura 7.11). Note que en el resto de las invocaciones de la aplicación (id > 4), esta falla no sucede nuevamente.

ID	PROVEEDOR	ALMACÉN
4	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput0_MsgObj xmlns:msgns="supplierChronicle_iep"> <id>4</id> <event>EBSLA</event> <time>1412630170762</time> <lppre>4</lppre > <lp>0</lp> <Timestamp>2014-10-06 T16:46:10.375-04:30</Timestamp> </msgns:StreamOutput0_MsgObj> **Archivo: EBSLA_log.xml (Generar el Evento Enlazador EB_{SLA} en el Diagnosticador Proveedor hacia el Almacén)**	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:RepairOutput0_MsgObj xmlns:msgns="warehouseChronicle_iep"> <id>4</id> <fault>ServiceLevelAgreement</fault> <faulttype>SearchProducts</faulttype> <event_init>E6</event_init> <event_end>E6</event_end> <time>1412630170762</time> <lp>0</lp> </msgns:RepairOutput0_MsgObj> **Archivo: RepairSLASearch.xml (Diagnostico de la Falla SLA - Búsqueda de Productos en Almacén)**
	REPARADOR	**REPARADOR** <?xml version="1.0" encoding="utf-8"?> <msgrepair:warehouse> <id>8</id> <fault>ServiceLevelAgreement</fault> <measure>SearchProducts</measure> <methodspossibles>parametersUpdate;substitute;substituteFlow</methodspossibles> <methodrepair>parametersUpdate</method> <repair_state>true</repair_state> <time>1412630172587</time> <previuosstate>Not Allow External Search</previuosstate> <currentstate>Allow External Search</currentstate> </msgrepair:warehouse> **Archivo: SLASearchR.xml (Reparación de la Falla SLA - Búsqueda de Productos en el Almacén)**
8	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput0_MsgObj xmlns:msgns="supplierChronicle_iep"> <id>8</id> <event>EBSLA</event> <time>1412630507762</time> <lppre>10</lp> <lp>9</lp> <Timestamp>2014-10-06 T16:51:47.418-04:30</Timestamp> </msgns:StreamOutput0_MsgObj> **Archivo: BSLA_log.xml (Generar el Evento Enlazador EB_{SLA} en el Diagnosticador Proveedor hacia el Almacén)**	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:RepairOutput1_MsgObj xmlns:msgns="warehouseChronicle_iep"> <id>8</id> <fault>ServiceLevelAgreement</fault> <faulttype>Ship and packed</faulttype> <event_init>E8</event_init> <event_end>E8</event_end> <time>1412630507762</time> <lp>9</lp> </msgns:RepairOutput1_MsgObj> **Archivo: RepairSLAShip.xml (Diagnostico de la Falla SLA - Empaqueta y envía en el Almacén)**
	REPARADOR	**REPARADOR** <?xml version="1.0" encoding="utf-8"?> <msgrepair:warehouse> <id>8</id> <fault>ServiceLevelAgreement</fault> <measure>Ship and packed</measure> <methodspossibles>parametersUpdate;substitute;substituteFlow</methodspossibles> <methodrepair>substitute</method> <repair_state>true</repair_state> <time>1412630509928</time> <previuosstate>Ship and packed 1</previuosstate> <currentstate>Ship and packed 2</currentstate> **</msgrepair:warehouse>** **Archivo: SLAShipR.xml (Reparación de la Falla SLA - Empaqueta y envía en el Almacén)**

Figura 7.10: Resultado del reconocimiento de las crónicas y los mecanismos de reparación de la aplicación de comercio electrónico para la falla de violación de SLA

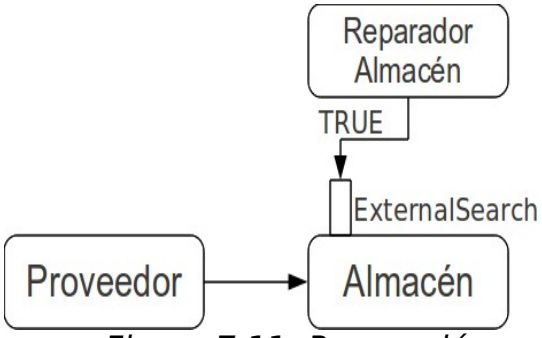


Figura 7.11: Reparación realizada por el Reparador Almacén cuando se produce la violación de SLA de Búsqueda de Productos en el id = 4

En el otro caso de violación de SLA en la invocación (id = {8}), la sub-crónica Almacén SLA Empaqueta (Figura 6.17) es reconocida (archivo RepairSLAShip.xml, Figura 7.10, Diagnosticador Almacén, id = 8) y enviada al reparador en el Almacén. Con el diagnóstico de la falla = fault: ServiceLevelAgreement, fault type: Ship and packed, event_init = E_8 y event_end = E_8, el reparador consulta en la ontología (Figura 7.10, campo methodspossibles del documento XML SLAShipR.xml generado por el reparador) y en la metadata (Figura 7.10, campo methodrepair del documento XML SLAShipR.xml generado por el reparador), seleccionando el mecanismo de reparación con id = 4 de la metadata, el cual implementa modificando el flujo de la composición al sustituir el servicio E_8 por E_8' (ver SLAShipR.xml, Figura 7.10, Reparador en el Almacén, id = 8, y gráficamente en la Figura 7.12). Note que en el resto de las invocaciones de la aplicación (id > 8) esta falla no sucede nuevamente (la operación E_8 es sustituida por E_8' en el resto de las invocaciones de la aplicación, ver Tabla 7.5, Operación del Almacén SetShip).

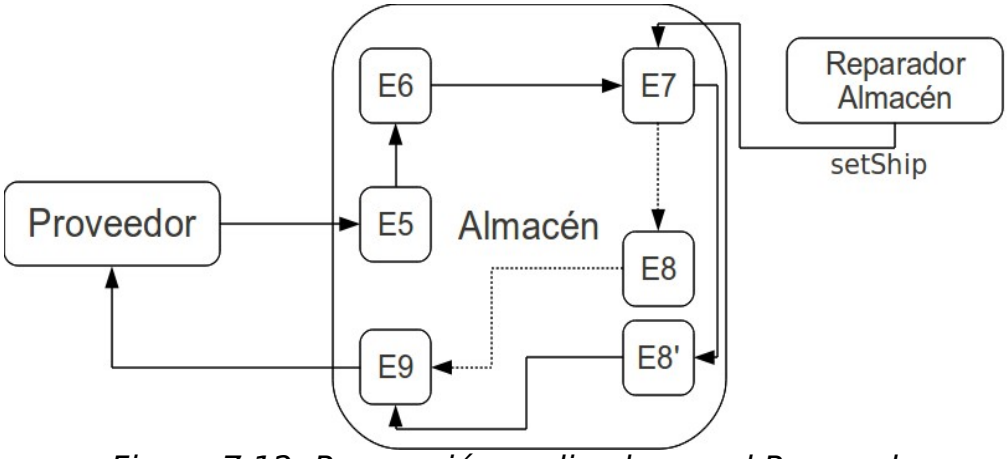


Figura 7.12: Reparación realizada por el Reparador Almacén cuando se produce la violación de SLA de Empaqueta y envía en el id = 8

Por otro lado, en la invocación id={9} sucede la falla de Fuera de Tiempo (Timeout) (Tabla 7.5, id = 9). Para esto, la sub-crónica en el Proveedor es activada debido a que no se ha recibido el evento E_{10} en el tiempo máximo de espera de 5000 ms (recuerde que se ha ajustado en esta invocación el parámetro **tuneDelay** = 6000ms para provocar esta falla, se sucede el evento $E_{NOEVENT}$ al no haber recibido E_{10}, para los valores de $\Delta T_{timeout}$ = 5000 ms, ΔT_{espera} = 4000 ms). El reconocimiento de la sub-crónica "Proveedor Fuera-Tiempo" en el Proveedor genera el evento enlazador $E_{BTimeout}$ hacia el diagnosticador en el Almacén (Archivo EBTimeout_log.xml de la Figura 7.13, Diagnosticador Proveedor). Con el evento enlazador recibido y con el evento E_5 que ha ocurrido previamente, el diagnosticador en el Almacén reconoce la sub-crónica "Almacén Fuera-Tiempo" y genera el diagnóstico de la falla a su reparador (Archivo RepairTimeout.xml de la Figura 7.13, Diagnosticador Almacén), el cual realiza una búsqueda en la ontología de Fault-Recovery y la metadata usando la información de la falla = fault: timeout, fault type: N/A, event_init = E_5 y event_end = E_9 (para ver el proceso de inferencia de la ontología y la metadata, consulte para esta misma falla la sección 7.1. El resultado de la consulta se muestra en la Figura 7.13 para

la ontología, en particular el campo methodspossibles, y para la metadata el campo methodrepair del documento XML del reparador), obteniendo como resultado que el método de reparación que más se ajusta es el almacenado en la metadata con el id = 1, el cual consiste en modificar el flujo de la composición sustituyendo las operaciones E_5, E_6, E_7, E_8 y E_9 del servicio Almacén 1 por las producidas por el servicio Almacén 2 (ver TimeoutR.xml en el Almacén de la Figura 7.13 y gráficamente en la Figura 7.14). Note que en la Tabla 7.5, luego de la corrección de esta falla en el id = 9, la aplicación de comercio electrónico comienza a funcionar con los eventos E_5, E_6, E_7, E_8 y E_9 del Almacén 2 (fila inferior en estas operaciones).

ID	PROVEEDOR	ALMACÉN
9	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput1_MsgObj xmlns:msgns="supplierChronicle_iep"> <id>9</id> <event>EBTimeout</event> <time>1412630604710</time> <lp>7/lp> </msgns:StreamOutput1_MsgObj> **Archivo: EBTimeout_log.xml (Generar el Evento Enlazador $EB_{tiemout}$ en el Diagnosticador Proveedor hacia el Almacén)**	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput2_MsgObj xmlns:msgns="warehouseChronicle_iep"> <id>9</id> <fault>timeout</fault> <faulttype>N/A</faulttype> <time>1412630604710</time> <lp>7</lp> <event_init>E5</event_init> <event_end>E9</event_end> <Timestamp>2014-10-06 T16:53:24.025-04:30</Timestamp> </msgns:StreamOutput2_MsgObj> **Archivo: RepairTimeout.xml (Diagnostico de la Falla fuera de tiempo en el Almacén)**
	REPARADOR	**REPARADOR** <?xml version="1.0" encoding="utf-8"?> <msgrepair:warehouse> <id>9</id> <fault>timeout</fault> <measure>N/A</measure> <methodspossibles>reasign;retry;substitute;substituteFlow</methodspossibles> <methodrepair>substituteflow</method> <repair_state>true</repair_state> <time>1412630608426</time> <previuosstate>Warehouse 1</previuosstate> <currentstate>Warehouse 2</currentstate> </msgrepair:warehouse> **Archivo: TimeoutR.xml (Reparación de la Falla Fuera de Tiempo en el Almacén)**

Figura 7.13: Resultado del reconocimiento de las crónicas y los mecanismos de reparación de la aplicación de comercio electrónico para la falla fuera de tiempo (Timeout)

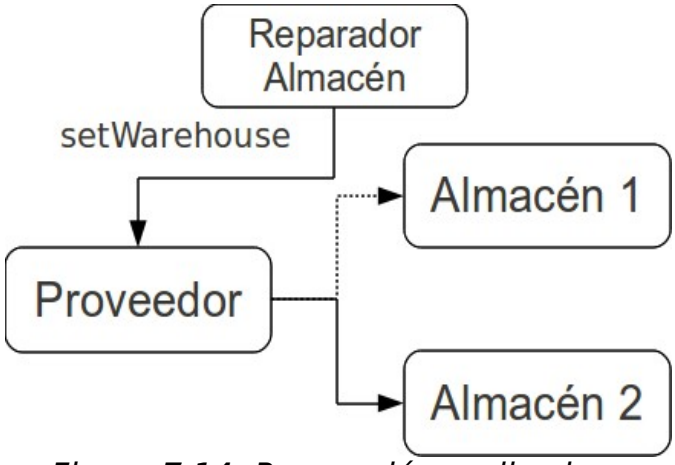


Figura 7.14: Reparación realizada por el Reparador Almacén cuando se produce la falla de Fuera de Tiempo en el id = 9

De igual manera, en las ejecuciones de la aplicación de comercio Electrónico con id= {10, 11} se producen retardos de tiempo al recibir la respuesta del Almacén (Tabla 7.5), activando la sub-crónica "Proveedor Retardo" (ver crónica en la Figura 7.1) en el Proveedor y generando un evento enlazador EB_{Delay} en las dos ocasiones hacia el diagnosticador del Almacén (ver el contenido del archivo EBDelay_log.xml del Proveedor para los dos id = {10, 11} de la Figura 7.16). Los cuales son recibidos por la sub-crónica "Almacén Retardo" en el Almacén, permitiendo el reconocimiento de la falla de calidad de servicio (archivo RepairDelay.xml de la Figura 7.15, id = 11, Diagnosticador Almacén). El diagnóstico de la falla es transmitido al reparador en el Almacén (ver el contenido del archivo DelayR.xml, Figura 7.16), el cual se encarga de solventar la situación sustituyendo completamente el flujo de las operaciones donde interviene el servicio Almacén 2 en la composición, por sus equivalentes provistas por el servicio Almacén 1, usando nuevamente el mecanismo de reparación almacenado en la metadata con el id = 1, operación setWarehouse (FALSE) en el reparador del Almacén, invocando la operación setWarehouse(TRUE) (ver DelayR.xml de la

Figura 7.16, Reparador del Proveedor, y gráficamente en la Figura 7.15).

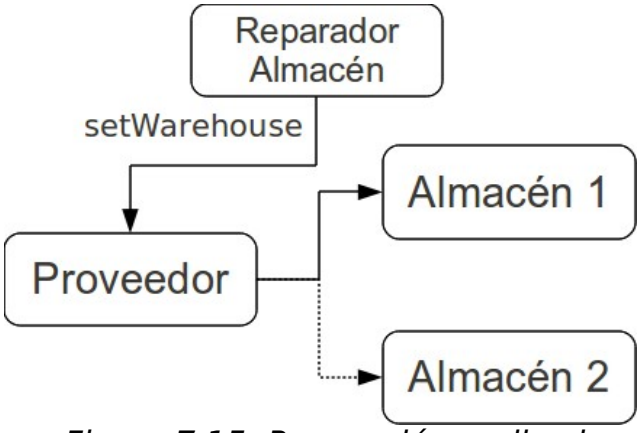


Figura 7.15: Reparación realizada por el Reparador Almacén cuando se produce la falla de Calidad de Servicio (Retardo) en el id = 11

ID	PROVEEDOR	ALMACÉN
10	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput1_MsgObj xmlns:msgns="supplierChronicle_iep"> <id>10</id> <event>EBDelay</event> <time>1412630707292</time> <lp>9</lp> <Timestamp>2014-10-06 T16:55:07.761-04:30</Timestamp> </msgns:StreamOutput1_MsgObj> **Archivo: EBDelay_log.xml (Generar el Evento Enlazador EB_{DELAY} en Proveedor hacia el Proveedor)**	**DIAGNOSTICADOR**
	REPARADOR	**REPARADOR**
11	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput1_MsgObj xmlns:msgns="supplierChronicle_iep"> <id>11</id> <event>EBDelay</event> <time>1412630719054</time> <lp>14</lp> <Timestamp>2014-10-06 T16:55:19.761-04:30</Timestamp> </msgns:StreamOutput1_MsgObj> **Archivo: EBDelay_log.xml (Generar el Evento Enlazador EB_{DELAY} en Proveedor hacia el Proveedor)**	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:RepairOutput2_MsgObj xmlns:msgns="warehouseChronicle_iep"> <id>11</id> <fault>QualityOfService</fault> <faulttype>Delay</faulttype> <time>1412630723294</time> <lp>14</lp> <event_init>E5</event_init> <event_end>E9</event_end> </msgns:RepairOutput2_MsgObj> **Archivo; RepairDelay.xml (Diagnostico de la Falla de Calidad de Servicio en el diagnosticador Almacén)**
	REPARADOR	**REPARADOR** <?xml version="1.0" encoding="utf-8"?> <msgrepair:warehouse> <id>8</id> <fault>QualityOfService</fault> <measure>Delay</measure> <methodspossibles>reasign;substitute;substituteFlow</methodspossibles> <methodrepair>substituteflow</method> <repair_state>true</repair_state> <time>1412630723294</time> <previuosstate>Warehouse 2</previuosstate> <currentstate>Warehouse 1</currentstate> <msgrepair:warehouse> **Archivo: DelayR.xml (Reparación de la Falla Calidad de Servicio QoS en el Almacén)**

Figura 7.16: Resultado del reconocimiento de las crónicas y los mecanismos de reparación de la aplicación de comercio electrónico para la falla de Calidad de Servico - QoS (Retardo)

Por último, en la invocación id= {12} se ha producido una falla de incompatibilidad de parámetros que es diagnosticada en la sub-crónica "Almacén Interfaz pudo Haber Cambiado" en el diagnosticador del Almacén, produciendo el

evento enlazador $EB_{paramincom}$ hacia el Proveedor (ver el contenido del archivo EBParamicom.xml, Figura 7.17, Diagnosticador Almacén). Con el evento enlazador, el diagnosticador en el Proveedor le es posible reconocer la sub-crónica "Proveedor Interfaz pudo Haber Cambiado" y de esta forma detectar la falla de Interfaz pudo Haber Cambiado (se ha sucedido previamente la llamada al evento E_5 en 11 oportunidades previas sin este problema, id = {1, ..11}, ver Tabla 7.5). El resultado del diagnóstico de esta falla puede consultarse en el archivo RepairInterfaz.xml generado por el diagnosticador Proveedor de la Figura 7.17. Una vez detectada la falla, el reparador en el Proveedor obtiene el diagnóstico y gestiona el mecanismo apropiado para corregir la falla (ver archivo InterfazR.xml de la Figura 7.17, del Reparador del Proveedor). Para corregir la aplicación, el reparador utiliza los datos obtenido del diagnóstico (RepairInterfaz.xml) fault: InterfaceMightHaveChanged, fault type: N/A, event_init = E_4 y event_end = E_5, para luego consultar la ontología (resultado de la consulta Figura 7.17, campo methodspossibles del InterfazR.xml del Reparador), y la metadata (Figura 7.17, campo methodrepair del documento InterfazR.xml generado por el reparador), resultando finalmente seleccionado el mecanismo de reparación de la metadata con id = 3 (ver el archivo InterfazR.xml de la Figura 7.17, Reparador Proveedor), el cual consiste en completar los parámetros perdidos en la invocación del evento E_5, haciendo uso de un compensador (ver InterfazR.xml de la Figura 7.17, Reparador Proveedor, gráficamente en la Figura 7.18).

ID	PROVEEDOR	ALMACÉN
12	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:RepairOutput1_MsgObj xmlns:msgns="supplierChronicle_iep"> <id>12</id> <fault>parameterIncompatibility</fault> <faulttype>N/A</faulttype> <time>1412630848652</time> <lppre>6</lp> <lp>-1</lp> <event_init>E4</event_init> <event_end>E5</event_end> </msgns:RepairOutput1_MsgObj> **Archivo: RepairInterfaz.xml (Diagnostico de la Falla Interfaz pudo Haber Cambiado en el Proveedor)**	**DIAGNOSTICADOR** <?xml version="1.0" encoding="utf-8"?> <msgns:StreamOutput2_MsgObj xmlns:msgns="warehouseChronicle_iep"> <id>12</id> <event>EBparamincom</event> <time>1412630848652</time> <Timestamp>2014-10-06 T16:57:28.649-04:30</Timestamp> <lp>-1</lp> </msgns:StreamOutput2_MsgObj> **Archivo: EBparamicom_log.xml (Generar el Evento Enlazador $EB_{paramincom}$ en el Almacén hacia el Proveedor)**
	REPARADOR <?xml version="1.0" encoding="utf-8"?> <msgrepair:supplier> <id>12</id> <fault>InterfaceMightHaveChanged</fault> <measure>N/A</measure> <methodspossibles>CompleteMissingParameters;substituteFlow</methodspossibles> <methodrepair>CompleteMissingParameters</method> <repair_state>true</repair_state> <time>1412630854590</time> <previuosstate>Not Compensate Input (Filter)</previuosstate> <currentstate>Compensate Input (Filter)</currentstate> <msgrepair:supplier> **InterfazR.xml (Reparación de la Falla Interfaz pudo Haber Cambiado en el Proveedor)**	**REPARADOR**

Figura 7.17: Resultado del reconocimiento de las crónicas y los mecanismos de reparación de la aplicación de comercio electrónico para la falla Interfaz pudo Haber Cambiado

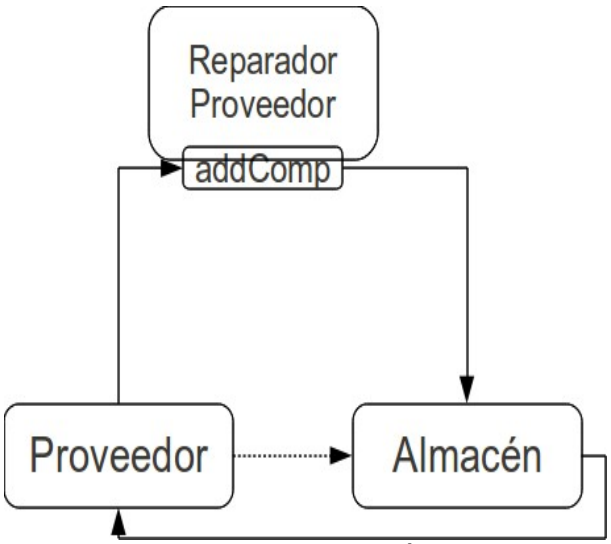


Figura 7.18: Reparación realizada por el Reparador Almacén cuando se produce la falla de Interfaz pudo Haber Cambiado en el id = 12

7.2.3. Análisis de Resultados

En este experimento se ha mostrado el funcionamiento de todos los componentes de ARMISCOM para realizar la gestión automática de fallas en la composición de servicios. Además, se ha mostrado como se realiza la interacción entre los componentes de ARMISCOM, vía los archivos Xml de salida que generan los mismos. Por ejemplo, en el caso de la *Industria del Mueble* se ha podido diagnosticar y corregir las fallas de violación de SLA para el caso calidad (Quality) en dos oportunidades, haciendo uso de los patrones de crónicas diseñadas para tal fin. Al momento que el componente diagnosticador en la Tienda reconoce la falla, genera un archivo con el contenido del diagnóstico, que es usada por el componente reparador en la Tienda para realizar la corrección, que en este caso se ejecuta usando el método de substitución del flujo de la composición.

Los experimentos para el caso de la aplicación SOA en *comercio electrónico*

fueron más numerosos, reconociéndose 5 fallas diferentes: SLA (caso de "Búsqueda de Productos" y "Empaqueta y envía"), Calidad de Servicio QoS (Retardo), Fuera de Tiempo (Timeout) y Interfaz pudo Haber Cambiado. Las mismas fueron reconocidas para un abanico de situaciones, que implico usar un número de métodos de reparación mucho mayor, en distintas instancias de los diagnosticadores del Proveedor y del Almacén, para varios flujos de la composición (tanto a nivel de un solo servicio como modificando parte del flujo).

En la implementación de los patrones de crónicas usadas en esta sección, se ha añadido información acerca de la parte exacta de la aplicación que ha sido afectada con la falla, agregando al diagnóstico generado información acerca de donde la falla comienza **event_init** y termina **event_end**. Adicionalmente, con el resto de información obtenida en el diagnóstico (fault y fault type: falla y tipo de falla) es posible consultar el servicio que contiene la interfaz con la ontología Fault-Recovery para obtener (inferir) los distintos métodos a aplicar para corregir la falla. Posteriormente, es posible realizar la consulta a la metadata para encontrar regiones equivalentes, usando el resultado de la consulta de la ontología (métodos a aplicar) y la información del flujo afectado (event_init y event_end).

En ese orden de ideas, en este experimento se logro implementar y acoplar todos los componentes del Gestor Autonómico de las distintas instancias de ARMISCOM propuestas en el Capítulo 5 como servicios (SOA), usando OpenESB; verificando su total funcionamiento, al poder realizar correctamente las tareas de diagnóstico y reparación que les fueron encomendadas al middleware (ver sección 7.2), verificando de esta manera el comportamiento autonómico del middleware, al poder realizar tareas de auto-sanación para 6 fallas en 2 escenarios distintos.

7.3. Conclusiones

En este Capítulo se ha probado, mediante diferentes experimentos, el funcionamiento de los distintos componentes de ARMISCOM, y su funcionamiento como un todo, para realizar la auto-reparación de fallas en la composición de Servicios. En un primer momento se mostró el funcionamiento del componente de Fuente de Conocimiento: Las crónicas distribuidas, la ontología Fault-Recovery y la metadata; y su interacción con el resto de componentes del MAPE, para realizar el diagnóstico de las fallas en las aplicaciones SOA, y obtener el mecanismo de reparación correcto para cada falla.

Seguidamente, se ha comprobado nuevamente la capacidad del lenguaje CQL para la definición y el reconocimiento de crónicas, permitiendo dar mayor expresividad a las restricciones no temporales usadas en las crónicas, al permitir añadir expresiones matemáticas, tal como se ha mostrado en los casos de estudio cuando se producen las fallas de violación de SLA y de Interfaz pudo Haber Cambiado. Adicionalmente, la posibilidad de implementar el reconocedor de crónicas en una arquitectura SOA permite la implementación de las distintas instancias del Gestor Autonómico de ARMISCOM como componentes distribuidos débilmente acoplado, al heredar todas las ventajas de esta arquitectura, lo que le permite al MAPE una gran flexibilidad.

En cuanto a los patrones distribuidos de crónicas para el diagnóstico de fallas en la composición de servicios, en este Capítulo se ha comprobado el funcionamiento de las crónicas distribuidas para el diagnóstico de fallas de violación de SLA, servicios fuera de tiempo (timeout), Calidad de Servicio QoS (retardo) e Interfaz pudo Haber Cambiado. Es importante destacar que al final se configuraron los distintos diagnosticadores de ARMISCOM para detectar un gran abanico de fallas (violación de SLA en distintos niveles, servicio fuera de tiempo,

calidad de servicio, cambios de interfaces) en distintos escenarios (industria del Mueble y comercio electrónico), pudiendo discriminar y encontrar correctamente la falla en la composición, y la parte afectada del flujo (event_init y event_end).

Asimismo, se ha diseñado un módulo de reparación que permite a ARMISCOM inferir las estrategias de reparación de fallas en la composición de los servicios, teniendo en cuenta la información del contexto (falla) y la composición del flujo, la cual se realiza en tiempo de ejecución. En trabajos anteriores [6, 19, 21] se han correlacionado/emparejado las acciones de recuperación con los tipo de fallas, en este trabajo el componente reparador utiliza una ontología (“fault-recovery”) diseñada en el Capítulo 3 para correlacionar las fallas con las acciones de recuperación, la cual fue implementada como un servicio web usando el componente BC Sun Java EE SE, para encapsular el lenguaje JAVA como un servicio, y el motor de inferencia FACT++. Las distintas consultas al servicio de la ontología para cada falla mostraron las respuesta correspondientes (métodos de reparación a usar) en los reparadores de las fallas: por ejemplo, en la figura 7.2 (columna Respuesta del servicio con la Ontología "Fault-Recovery" para las fallas fuera de tiempo y calidad de servicio), y en las Figura 7.5, Figura 7.10, Figura 7.13, Figura 7.16 y Figura 7.17 (campo methodspossibles del documento XML generado por el reparador). Esta ontología puede aumentarse (ej. utilizando enfoques de aprendizaje de ontológicas) para incluir nuevas fallas, mecanismos de reparación, entre otras.

Por otro lado, la metadata proporciona a ARMISCOM múltiples planes de recuperación para solucionar fallas en la composición de servicios web. En los distintos experimentos realizados en esta Capítulo se mostró la manera de almacenar los distintos mecanismos de reparación basado en Regiones (propuesto en el Capítulo 3), y realizar la consulta a la metadata para encontrar el

mecanismos que más se adapta a la parte del flujo afectado. La inferencia de los mecanismos de solución se realiza utilizando la información derivada del diagnóstico de la falla: métodos posibles de reparación (suministrado por el servicio de ontología), y la parte del flujo afectado (campos event_init y event_end del diagnóstico de la crónica). Con eso, el Algoritmo 3.1 encuentra las Regiones equivalentes y/o super-regiones. Los resultados por el uso de la metadata pueden verificarse a lo largo de los experimentos realizados en este Capítulo, consultando las respuesta de los reparadores de las fallas: por ejemplo, ver Figura 7.2 (columna Respuesta del servicio con la selección de la reparación para las fallas fuera de tiempo y calidad de servicio) y Figura 7.5, Figura 7.10, Figura 7.13, Figura 7.16 y Figura 7.17 (campo methodrepair del documento XML generado por el reparador). Asimismo, una de las capacidades a evaluar en este trabajo es la capacidad de auto-reparación de ARMISCOM para corregir fallas en la composición de servicios; para esto, en las Figura 7.5, Figura 7.10, Figura 7.13, Figura 7.16 y Figura 7.17 se muestra la información generada por cada componente del MAPE, y utilizada para comunicarse entre si en el proceso de auto-reparación: comunicación entre los distintos diagnosticadores usando los eventos enlazadores, comunicación entre los diagnosticadores y reparadores (diagnóstico de falla usando crónicas), y comunicación entre los sub-componentes del reparador (planificador y ejecutor), logrando finalmente la auto-reparación de un gran abanico de fallas: distintas violaciones de SLA y de calidad (QoS), fuera de tiempo, problemas de interfaz, etc.

Por último, en este Capítulo se ha comprobado el objetivo primordial de ARMISCOM, que es garantizar la gestión automática de fallas en la composición de servicios. Para esto, se ha comprobado en el apartado 7.2 el funcionamiento y acoplamiento de las distintas instancias distribuidas del middleware, y su capacidad de gestionar y reparar fallas en la composición de servicios, tanto en

problemas a nivel de un solo servicio (caso de violación de SLA en los casos de estudio de la Industria del Mueble y Comercio Electrónico) como de flujos que involucra varios servicios (caso de fuera de tiempo, calidad de servicio QoS e Interfaz pudo Haber Cambiado), comprobando de esta manera la capacidad autonómica del middleware para realizar tareas de auto-sanación.

El conjunto de experimentos realizados, ha permitido comprobar la capacidad autonómica del middleware para realizar tareas de auto-sanación. Asimismo, se han verificado las capacidades reflexivas de ARMISCOM, a saber:

- **La Introspección:** Esta capacidad se comprueba, porque el middleware fue capaz de diagnosticar las distintas fallas que suceden en la aplicación SOA. Para ello, uso su base de conocimiento de crónicas distribuidas, los mensajes intercambiados por los servicios que componen la aplicación SOA, y la información almacenada en los diferentes componentes de las plataformas de despliegue del paradigma SOA (WSDL, etc.). Estas dos últimas cosas se encuentran en el nivel base.

- **La Intersección:** Esta capacidad se ha comprobado porque el middleware realiza cambios en la aplicación SOA (el cual se ejecuta en el nivel base), en el momento en que se produce una falla, para corregir el funcionamiento de la aplicación. Según el método de reparación usado, el nivel Meta puede modificar el flujo de la composición de la aplicación SOA, los servicios usados en la composición, entre otras cosas.

-

Capítulo 8
Conclusiones y Perspectivas

8.1. Conclusiones

En este trabajo se ha desarrollado un middleware reflexivo distribuido llamado ARMISCOM, para el diagnóstico de fallas en la composición de servicios, donde sus componentes interactúan para obtener una visión global del sistema. El middleware esta compuesto por 2 niveles de la torre reflexiva: Nivel base (Sistema y Aplicación SOA) y nivel meta (encargado de realizar la introspección e intersección). Adicionalmente, el nivel meta ha sido diseñado siguiendo la Arquitectura de Computación Autonómica, agregando un gestor autonómico (MAPE) a cada servicio, obteniendo de esta manera un conjunto de componentes MAPE distribuidos, que realizan las tareas propias de introspección e intersección de un middleware reflexivo. De esta manera, esta arquitectura reduce el problema de escalabilidad, en particular, el volumen de las comunicaciones para la gestión de las aplicaciones distribuidas. La arquitectura distribuida del middleware contrasta con otras arquitecturas propuestas en otros trabajos [12, 21, 22, 24, 64, 71] (ver Tabla 3.3). Así, para el desarrollo de ARMISCOM fue necesario realizar el diseño e implementación de nuevos mecanismos, modelos, estructuras de datos, entre otras cosas, que permitieran la implementación de ARMISCOM como un sistema para la corrección de fallas distribuida en la composición de servicios distribuida. Los aspectos nuevos desarrollados fueron:

8.1.1. Crónicas Distribuidas para el Diagnóstico de Fallas en la Composición de Servicios

En otros trabajos se han estudiado las fallas presentes en la composición de servicios, realizando el análisis de los mensajes intercambiados entre los diferentes participantes de la aplicación SOA [18, 19, 20]. En el Capítulo 2 de este trabajo se ha ampliado el estudio de las fallas en las aplicaciones SOA, enfocándose en describir estas como un conjunto de eventos con restricciones temporales entre si, que permiten caracterizar y diferenciar los episodios de fallas en una aplicación SOA. Debido a que las fallas se encuentran definidas en términos de eventos, en este trabajo ha sido posible el desarrollo de diagnosticadores distribuidos usando crónicas.

8.1.1.1 Extensión de las Crónicas

Las crónicas han sido usadas exitosamente para el diagnóstico de fallas y la detección de situaciones no deseadas en distintos sistemas [10, 11, 12, 13, 14, 15, 16, 71]. Una de las debilidades que presenta el uso de las crónicas es su complejidad al intentar su implementación en sistemas con un gran número de componentes, al estar intrínseco el problema de escalabilidad descrito al comienzo de este Capítulo. En este trabajo se han seleccionado las crónicas como mecanismo para realizar el diagnóstico de las fallas en la composición de servicios en ARMISCOM, pero las arquitecturas tradicionales de implementación de las crónicas basadas en enfoques centralizados [11, 13, 14, 15, 16] o descentralizados [12, 30, 71], rompían con el esquema planteado en el middleware reflexivo distribuido.

Así, en el Capítulo 4 se ha realizado una extensión al mecanismo de crónicas para permitir el reconocimiento distribuido de las crónicas, donde cada

diagnosticador local cuenta con su propio reconocedor de crónicas (CRS). Para esto, ha sido necesario la introducción de lo que se ha llamado "eventos enlazadores" (binding events), que permiten la propagación de los diferentes diagnósticos locales, haciendo posible el reconocimiento de la crónica global sin necesitar de un coordinador para gestionar sus interacciones. Esto último reduce notablemente el problema de escalabilidad (ver Tabla 4.1), resultando en un avance notable en esta área, al poder disminuir la cantidad de mensajes intercambiados por reconocedor, y distribuir el procesamiento de eventos entre los distintos diagnosticadores. La extensión de crónicas distribuidas desarrollada en este trabajo no resulta de uso exclusivo en aplicaciones SOA, por el contrario, puede ser usado en distintos ambientes en que la distribución sea natural, como simulaciones distribuidas, comunidad de agentes de software, entre otros.

8.1.1.2 Patrones Genéricos de Crónicas Distribuidas

Para el diagnóstico de fallas en la composición de servicios usando crónicas, en trabajos previos [13, 71] normalmente han sido caracterizadas las mismas (sus eventos y sus restricciones temporales) por expertos que analizaba la aplicación, resultando extremadamente difícil su implementación ante el gran abanico de fallas que pueden suceder en diferentes escenarios.

En el Capítulo 2 se han analizado las fallas comunes en la composición de servicios en base al conjunto de evento generados, y en el Capítulo 4 se han definido un conjunto de patrones de crónicas distribuidas genéricas que permiten relacionar el conjunto de eventos observables en las fallas en la composición de servicios. Cada patrón de crónica es construido distribuyendo los distintos eventos que se generan en la aplicación en sub-crónicas, las cuales se asignan a cada diagnosticador local, conectados entre ellos por eventos enlazadores para propagar sus diagnósticos. Cada sub-crónica local es construida tomando en

cuenta los eventos que caracterizan las causas de la falla y su propagación en el resto de la composición (sus consecuencias). Por otro lado, como se muestra en los casos de estudio (Capítulos 6 y 7), los patrones genéricos desarrollados en este trabajo para diferentes fallas son fáciles de parametrizar/configurar, para usarlos en diferentes escenarios y aplicaciones.

Adicionalmente, en caso de ser necesario la implementación de crónicas con un solo diagnosticador, aun cuando representa una desmejora al mecanismo distribuido planteado en este trabajo, los patrones para la detección de crónicas desarrollados pueden ser ajustados para funcionar en arquitecturas centralizadas. Para ello, solo es necesario realizar la unión de los distintos eventos distribuidos en las distintas sub-crónicas que componen un patrón de diagnóstico, y eliminar los eventos enlazadores (es un proceso inverso al descrito en la demostración de la definición 4.1), obteniendo de esta manera un patrón único para cada falla.

8.1.1.3 Implementación de un Sistema Reconocedor de Crónicas Distribuido, Usando el Lenguaje CQL

El diagnóstico de crónicas se realiza usualmente con la ayuda de una herramienta desarrollada por Dousson, llamada CRS, y expandida en otros trabajos para permitir un diagnóstico descentralizado. Como se muestra en la Tabla 5.5, el uso de CRS presenta una serie de desventajas al intentar su implementación en una arquitectura totalmente distribuida y en software libre como ARMISCOM. Así, en este trabajo se procedió a la búsqueda de una herramienta alternativa que se ajustara a los requerimientos de ARMISCOM.

El lenguaje CQL representa un lenguaje poderoso para realizar el análisis de flujos de eventos, permitiendo una gran expresividad en la inferencia realizada sobre los atributos de los eventos. Así, en el Capítulo 4 se mostró que los

predicados temporales reificados clásicos utilizados comúnmente en las crónicas (holds, events, not events y occurs), pueden ser expresados usando el lenguaje CQL y la expresividad que ofrece el lenguaje para realizar inferencias sobre las variables atemporales (atributos presentes en los eventos). En particular, esto último aumenta la información que pueden gestionar las crónicas, al no necesitarse traductores para reconocer circunstancias particulares sobre los atributos (por ejemplo: E_i.rate + E_j.rate < 50). Además, al parecerse la sintaxis de CQL al lenguaje SQL usado en los gestores de base de datos, su documentación es extensa y de fácil aplicación.

El componente diagnosticador de ARMISCOM, consiste en un conjunto de diagnosticadores locales usando crónicas, las cuales son enlazadas usando los eventos enlazadores. Así, en el Capítulo 4 se diseñaron los diagnosticadores usando el componente *IEP* de OpenESB, el cual usa el lenguaje CQL para expresar las restricciones sobre los eventos. Al ser el componente IEP una herramienta para ser aplicada en ambientes SOA, la construcción de las distintas interfaces para comunicar el diagnosticador con los demás componentes de MAPE resultó de forma natural. Como se mostró en los Capítulos 6 y 7, los patrones de fallas seleccionados para realizar las distintas pruebas de ARMISCOM, pudieron expresarse fácilmente usando el lenguaje CQL. Por otro lado, estar el diagnosticador inmerso en una arquitectura SOA, el envío de los distintos eventos enlazadores entre las sub-crónicas, y el reconocimiento de las fallas, resultaron fáciles de implementar en la composición SOA del gestor autonómico.

8.1.2. Fuentes de Conocimiento para la Reparación de Fallas

Para la gestión de las fallas en la composición de servicios, en el Capítulo 3 se

ha diseñado una ontología llamada Fault-Recovery, que permite correlacionar las fallas presentes con los mecanismos de reparación. La ontología describe una taxonomía sobre las fallas y los mecanismos de reparación de fallas para una aplicación SOA. La ontología está basada en el trabajo [18], donde definen una taxonomía de fallas y las causas que las generan. Posteriormente, en el Capítulo 5 se ha implementado la ontología utilizando la herramienta protégé, y se ha encapsulado como un servicio web, con la librería FACT++ como motor de inferencia, y el BC Sun Java EE SE para implementar el código Java como un servicio web. Los resultados mostrados en el Capítulo 7 para correlacionar las fallas con los mecanismos de reparación son motivadores, porque se pudo inferir la relación entre las fallas y los mecanismos de reparación. La ontología Fault-Recovery puede ser enriquecida en el futuro para permitir inferencias acerca de situaciones más complejas.

También, en el Capítulo 3 se han definido un conjunto de conceptos que permiten descomponer las aplicaciones en Regiones de eventos, para facilitar el reemplazo de sub-flujos de aplicación SOA que se encuentran en falla. El trabajo de reparación de una aplicación SOA consiste en encontrar y almacenar flujos alternativos que puede seguir la aplicación en el momento que sucede una falla. La idea de flujos equivalentes consiste en encontrar flujos tomando algún criterio de similitud basado en las propiedades funcionales y no funcionales (posiblemente usando ontologías que permitan definir a los servicios semánticamente) de los servicios que forman parte de la aplicación [58, 59, 60, 61, 62]. Algunos trabajos almacenan los sub-flujos modelándolos como un conjunto de servicios que están interconectados entre si, usando para ello redes de Petri o grafos de conexiones. En este trabajo se han modelado los flujos como eventos con restricciones temporales, para estar en consonancia con el diagnosticador basado en crónicas. Así, se han definido los conceptos de Regiones

Equivalentes y Super-Región, como un mecanismo para contar con un almacén de flujos substitutos/alternativos. Adicionalmente, se ha diseñado una metadata para almacenar la información que caracteriza a las distintas regiones: Evento inicial del flujo (Event_init), secuencia de eventos que lo componen (Transition), evento final del Flujo (Event_end), la estrategia de reparación posible a implementar (RepairMethod) y el orden de importancia de esa región equivalente (Weight), permitiendo discriminar entre regiones equivalentes semejantes.. Esto ha, resultado ser un mecanismo eficaz para realizar la inferencia sobre los flujos afectados y por quien deben ser sustituidos, como se ha mostrado en el Capítulo 7.

Ahora bien, el proceso de automatizar la búsqueda de regiones equivalentes aun no se ha realizado en ARMISCOM, hasta ahora se usa un almacén que es configurado por un actor humano. Sin embargo, en la sección prospectiva de este Capítulo se propone un posible sistema a implementar para desarrollar esta tarea. Si bien este componente le conferiría aún más autonomía al middleware, no es el elemento determinante de dicha propiedad. La capacidad de auto-sanación (auto-gestión) es alcanzada por los componentes propuestos en este trabajo, tal como se ha comprobado en los experimentos del apartado 7.2.

8.1.3. Componente MAPE-K como una Aplicación Orientada a Servicios

Al ser ARMISCOM un middleware para la gestión de fallas en aplicaciones SOA, el desarrollo de todas las instancias del gestor autonómico están diseñadas para ser desplegadas como un servicio. En el Capítulo 5, todos los componentes que forman parte de las diferentes instancias locales del gestor autonómico fueron desarrollados e implementados como un servicio web, con interfaces bien definida

que puede ser invocadas con el protocolo SOAP, los cuales fueron conectados usando una composición SOA. Adicionalmente, los componentes encargados para realizar la interfaz en la consultas a las fuentes de conocimiento (ontología Fault-Recovery y metadata) también siguen la arquitectura de servicios.

El desarrollo de MAPE-K siguiendo el paradigma SOA, a cada instancia del Gestor Autonómico de ARMISCOM, hace que este último funcione con componentes distribuidos débilmente acoplados, lo que le da a ARMISCOM una gran flexibilidad. Así, cada gestor autonómico fue construido para funcionar como una composición de un conjunto de servicios, en específico: Diagnosticador, Reparador y Gestores de Conocimiento, cada uno con propiedades para funcionar independientemente de los demás, permitiendo una fácil adaptación de la arquitectura MAPE-K si se requiere, por ejemplo, cambiar el Diagnosticador si no se está satisfecho con el Diagnosticador basado en crónicas. Así, puede implementarse otro servicio para el diagnóstico de fallas con otro paradigma, y solo seria necesario reemplazar el servicio Diagnosticador por el nuevo componente en la aplicación SOA que define la composición del MAPE-K.

Por otro lado, cada componente puede funcionar en una composición distinta al gestor autonómico. Por ejemplo, el servicio diagnosticador puede ser usado independientemente de los demás. En el Capítulo 6 se realizaron las pruebas sobre el componente Diagnosticador, como un componente independiente del middleware reflexivo. Se ejecutó una aplicación SOA, y se interconectó el flujo de eventos en su composición, con el sistema de reconocimiento de crónicas del componente Diagnosticador que permite realizar el diagnóstico de fallas. Adicionalmente, en el Capítulo 7 se realizaron pruebas sobre la composición de las instancias MAPE-K, realizando para esto pruebas sobre la composición de todos los servicios que componen el gestor autonómico (funcionamiento y acoplamiento

de las distintas instancias).

8.1.4. Auto-sanación Distribuida con ARMISCOM

El objetivo principal del trabajo es el diseño e implementación de una arquitectura de un middleware reflexivo para realizar el diagnóstico y reparación distribuida de fallas en aplicaciones SOA. Para esto, se han desarrollado un conjunto de mecanismos para realizar el diagnóstico y la reparación distribuida de fallas en la composición de servicios. En el Capítulo 7 se ha comprobado el funcionamiento completo de ARMISCOM, logrando realizar la gestión automática de fallas en la composición de servicios. Particularmente, se ha comprobado en el apartado 7.2 el funcionamiento y acoplamiento de las distintas instancias distribuidas del middleware, y su capacidad de gestionar y reparar fallas en la composición de servicios, tanto en problemas a nivel de un solo servicio (caso de violación de SLA en los casos de estudio de la Industria del Mueble y comercio electrónico), como de flujos que involucran varios servicios (casos de fuera de tiempo, calidad de servicio QoS, e Interfaz pudo Haber Cambiado).

Lograr la auto-reparación en la composición de servicios con una arquitectura distribuida mejora notablemente el problema de escalabilidad, en particular la cantidad de cálculos y volumen de información intercambiada, como se evidencia a lo largo del desarrollo de este trabajo, y especialmente en Tabla 3.3 y Tabla 4.1.

De esta manera, los objetivos específicos fueron alcanzados de la siguiente manera:

- Diseñar una arquitectura de un middleware reflexivo para realizar el diagnóstico y reparación distribuida de fallas en aplicaciones SOA.

  Este objetivo fue alcanzado con la propuesta de ARMISCOM (ver Capítulo 3),

el cual es un middleware reflexivo para la gestión de aplicaciones orientadas a servicios. El middleware está diseñado para ser distribuido a través de todos los servicios de la aplicación SOA [74]. La arquitectura que se utilizó para el desarrollo del nivel meta esta basada en el modelo de computación autonómica, que le permite una fácil adaptación para procesos de auto-reparación [17].

- Especificar un conjunto de patrones que permitan el diagnóstico de fallas en Aplicaciones Orientadas a Servicios Web.

  Básicamente, lo señalado en la sub-sección 8.1.1.2 resume el aporte en este ámbito. En el Capítulo 4 se han definido un conjunto de patrones de crónicas distribuidas genéricas que permiten detectar fallas en la composición de servicios, relacionando el conjunto de eventos observables en la aplicación SOA con las fallas. Los eventos son distribuidos en sub-crónicas que están a cargo de cada diagnosticador local (diagnosticador por servicio), los cuales logran un diagnóstico global usando eventos enlazadores. Los patrones de crónicas diseñados en el Capítulo 4 han sido probados a lo largo de los Capítulos 6 y 7.

- Diseñar e implementar mecanismos para definir y reconocer crónicas en un ambiente distribuido, que representan fallas en las Aplicaciones Orientadas a Servicios Web .

  En el Capítulo 4 se ha cumplido este objetivo al desarrollar una extensión al mecanismo de crónicas para permitir el reconocimiento distribuido de las crónicas. Para esto, ha sido necesario la introducción de lo que se ha llamado "eventos enlazadores", que permiten la propagación de los diferentes diagnósticos locales, haciendo posible el reconocimiento de la

crónica global sin necesitar de un coordinador para gestionar sus interacciones. El lograr distribuir el reconocimiento de las crónicas reduce notablemente el problema de escalabilidad (ver Tabla 4.1), resultando en un avance notable en esta área. Adicionalmente, en el Capítulo 5 se ha desarrollado un reconocedor distribuido basado en el lenguaje CQL [76] que constituye una alternativa a las herramientas usadas comúnmente en el reconocimiento de crónicas propuestas por Dousson [10, 32], y sus extensiones CarDeCRS [13] y Matrac [71].

- Diseñar y desarrollar herramientas de programación bajo software libre, que implementen un middleware reflexivo para la gestión de fallas en Aplicaciones Orientadas a Servicios Web, basado en los mecanismos previamente desarrollados.

  Se ha mostrado la implementación de ARMISCOM en openESB y Protégé, las cuales tienen licencias de sofware libre. Los distintos componentes del middleware fueron construidos como servicios independientes y autónomos, lo que facilita su reuso y le dan una gran flexibilidad para poder ser construidos, al ser definidos como una composición de servicios, permitiendo que en un futuro puedan ser modificados y extendidas su estructura, principalmente los componentes del gestor autonómico. También, al ser reusable los distintos servicios/componentes, pueden ser utilizados separadamente, como por ejemplo el servicio Diagnosticador que podría usarse como una herramienta de reconocimiento de crónicas. Finalmente, las herramientas CRS, CarDeCRS y Matrac no están liberadas en software libre y el acceso a las fuentes de la aplicación, como su uso, esta restringido a la concesión de un permiso. En este trabajo se ha construido un reconocedor de crónicas en software libre [76], basado en el lenguaje

CQL y el procesador de eventos complejos de GlashFish (IEP SE).

- Definir e implementar casos de prueba utilizando el middleware reflexivo.

  El Capítulo 7 es dedicado solo a este objetivo, en el mismo se ha puesto a prueba mediante diferentes experimentos, el funcionamiento de los distintos componentes de ARMISCOM, y su funcionamiento como un todo, para realizar la auto-reparación de fallas en la composición de Servicios. En un primer momento (apartado 7.1) se muestra el funcionamiento del componente de Fuente de Conocimiento: Las crónicas distribuidas, la ontología Fault-Recovery y la metadata; y su interacción con el resto de componentes del MAPE, para realizar el diagnóstico de las fallas en las aplicaciones SOA, y obtener el mecanismo de reparación correcto para cada falla. Adicionalmente, en el apartado 7.2 se prueba el funcionamiento y acoplamiento de las distintas instancias distribuidas del middleware, y sus capacidades para gestionar y reparar fallas en la composición de servicios, tanto en problemas a nivel de un solo servicio (caso de violación de SLA en los casos de estudio de la Industria del Mueble y Comercio Electrónico), como de flujos que involucran varios servicios (casos de fuera de tiempo, calidad de servicio QoS e Interfaz pudo Haber Cambiado).

8.2. Perspectivas

En base a los resultados obtenidos en el desarrollo de ARMISCOM, se proponen a continuación una serie de trabajos futuros que se piensa puedan mejorara la implementación del middleware. En la sub-secciones 8.2.1 y 8.2.2 se plantea el desarrollo de 2 nuevos componentes para la configuración de crónicas y el almacenamiento de flujos equivalentes, y en la sub-sección 8.2.3 se propone una extensión a ARMISCOM. Aun cuando los dos nuevos componentes formarían parte

de la nueva extensión, se consideran de forma separada, ya que pudieran funcionar en otras arquitecturas distintas al Middleware.

8.2.1. Componente Configurador de Patrones de Crónicas

En el Capítulo 4 se ha diseñado un conjunto de patrones para el diagnóstico de fallas en la composición de servicios usando crónicas distribuidas. Posteriormente, en los Capítulo 6 y 7 se han realizado pruebas sobre los patrones, consiguiendo resultados favorables en sus fiabilidades como diagnosticadores de estas fallas. Sin embargo, aun cuando cada configuración realizada para ajustar cada crónica distribuida resulto sencilla, fue realizada por un actor humano experto.

Así, para lograr un mayor nivel de autonomía de ARMISCOM, se recomienda realizar un nuevo componente que realice la tarea de configurar los patrones de crónicas para las fallas de manera automática. Esto es un tipo de aprendizaje, llamado en la literatura “aprendizaje de parámetros”.

Para poder configurar los patrones de crónicas de falla es necesario conocer la interacción entre los servicios en la composición (los eventos generados propios de la aplicación SOA), esto se puede obtener realizando una inferencia en documentos que están en el lenguaje XML:

- Documento WS-CDL, en el apartado interacciones (interaction) se definen los participantes y su secuencia de llamadas (quien envía (send) recibe (receive))

- Documento BPEL, en el apartado secuencia (secuence) se describe el orden en que se realizan las invocaciones de los servicios en la composición (invoque).

Para esto, se pueden tomar las siguientes consideraciones para cada patrón de falla:

8.2.1.1 Las fallas Físicas (servicio no disponible) y Flujo de Trabajo Inconsistente

Las configuraciones para estas fallas se muestra en la Tablas 8.1:

Falla debido a servicio no disponible (Figura 4.5)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
Invoca	S_i Servicio no-disponible 1	Invocar servicio PING (genera el resultado de la operación PING (evento $\mathbf{E_{PING}}$).	• **E_1:** La salida de la operación que se encuentra antes de la invocación del servicio a invocar. • **E_2:** Evento generado por el protocolo SOAP al no poder invocar al servicio	Conocer la ubicación del servicio PING y la dirección ip de la ubicación del servicio que recibe la solicitud
		Generar el evento enlazador $\mathbf{BE_{invPING}}$		Conocer la ubicación del diagnosticador que recibe el evento enlazador
	S_i Servicio no-disponible 2	Generar el reconocimiento de la falla.	La fuente de eventos de esta sub-crónica son la consecuencia del reconocimiento de la sub-crónica S_i Servicio no-disponible 1: • **$BE_{invPING}$:** Evento generado cuando se reconoce la sub-crónica S_i Servicio no-disponible 1 que indica que no esta disponible (status = unavailable). • **EPING:** Es el resultado de invocar	Conocer la ubicación del reparador que recibe el diagnóstico de la falla. ΔT debe seleccionarse de acuerdo a un estudio previo del tiempo promedio que tarda en llegar el evento E_{PING}.

Falla debido a servicio no disponible (Figura 4.5)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
			la operación PING. Se estima su llegada en un tiempo $T_{invPING}$ + ΔT, donde es la llegada del evento enlazador $BE_{invPING}$.	
Recibe	N/A	N/A	N/A	N/A
Falla debido a Flujo de trabajo inconsistente (Figura 4.11)				
Invoca	S_i Servicio no-disponible 1	Invocar servicio PING (genera el resultado de la operación PING (evento **E_{PING}**).	• **E_1:** La salida de la operación que se encuentra antes de la invocación del servicio a invocar. • **E_2:** Evento generado por el protocolo SOAP al no poder invocar al servicio	Conocer la ubicación del servicio PING y la dirección ip de la ubicación del servicio que recibe la solicitud
		Generar el evento enlazador **$BE_{invPING}$**		Conocer la ubicación del diagnosticador que recibe el evento enlazador
	S_i Servicio no-disponible 2	Generar el reconocimiento de la falla.	La fuente de eventos de esta sub-crónica son la consecuencia del reconocimiento de la sub-crónica S_i Servicio no-disponible 1: • **$BE_{invPING}$:** Evento generado cuando se reconoce la sub-crónica S_i Servicio no-disponible 1 que indica que el servicio esta disponible (status = available). • **EPING:** Es el resultado de invocar la operación PING. Se estima su llegada en un tiempo $T_{invPING}$ + ΔT, donde es la	Conocer la ubicación del reparador que recibe el diagnóstico de la falla. ΔT debe seleccionarse de acuerdo a un estudio previo del tiempo promedio que tarda en llegar el evento E_{PING}.

Falla debido a servicio no disponible (Figura 4.5)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
			llegada del evento enlazador $BE_{invPING}$.	
Recibe	N/A	N/A	N/A	N/A

Tablas 8.1: Configuraciones para las fallas de servicio no disponible y Flujo de trabajo inconsistente

Como se muestra en la Tablas 8.1, los patrones de crónicas son muy similares, solo difieren en la respuesta a la invocación del servicio PING. Las configuraciones prácticamente están basadas en extraer los eventos de los documentos WS-CDL o BPEL, y seleccionar el valor de ΔT en base a la experiencia que se tenga sobre el tiempo que tardara en llegar el resultado de la operación PING al diagnosticador D_i. Estos patrones de crónicas deben agregarse en cada sub-flujo de la aplicación en la que un servicio S_j dependa de la salida de un servicio S_i (debe colocarse en el diagnosticador de D_i).

8.2.1.2 Fallas de Incompatibilidad de Parámetros, Interfaz Podría Haber Cambiado, Debido a Respuesta de Error, Debido a Incorrecto Orden

Las configuraciones para estas fallas se muestra en la Tabla 8.2.

Falla Incompatibilidad de Parámetros (Figura 4.7)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
Invoca (Servicio S_i)	sub-cronica S_i Incompatibilidad de Parámetros	Generar el reconocimiento de la falla.	• **E_1:** La salida de la operación que se encuentra antes de la invocación del servicio a invocar.	El tiempo de retardo inherente en las comunicaciones ΔT debe

Falla Incompatibilidad de Parámetros (Figura 4.7)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
			• **$BE_{ParamIncom}$:** Evento generado cuando se reconoce la sub-crónica S_j "Incompatibilidad de Parámetros".	seleccionarse de acuerdo a un estudio previo del tiempo promedio que tarda un evento en llegar a D_j desde D_i. Adicionalmente, se debe conocer la ubicación del reparador que recibe el diagnóstico de la falla.
Recibe (Servicio S_j)	sub-cronica S_j Incompatibilidad de Parámetros	Generar el evento enlazador **$BE_{ParamIncom}$**	• **$\neg E_2$:** No haberse invocado el servicio S_j. • **E_2:** La invocación de la operación del servicio S_j (viene de S_i) resultado del protocolo SOAP al intentar invocar con parámetros incompatibles a S_j.	Conocer la ubicación del diagnosticador que recibe el evento enlazador (D_i)
Falla Interfaz podría haber Cambiado (Figura 4.9)				
Invoca (Servicio S_i)	sub-cronica S_i Interfaz podría haber Cambiado	Generar el reconocimiento de la falla.	• **E_1:** Invocaciones previas al servicio S_j. • **$\neg BE_{ParamIncom}$:** NO haber recibido previamente el evento enlazador (la sub-cronica S_j "Interfaz podría haber Cambiado" no se ha activado nunca). • **E_1':** La salida de la operación en el momento actual que se encuentra	El tiempo de retardo inherente en las comunicaciones ΔT debe seleccionarse de acuerdo a un estudio previo del tiempo promedio que tarda un evento en llegar a D_j desde D_i. Adicionalmente, se debe conocer la ubicación del

Falla Incompatibilidad de Parámetros (Figura 4.7)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
			antes de la invocación del servicio S_j. • **$BE_{ParamIncom}$:** Evento generado cuando se reconoce la sub-crónica S_j "Interfaz podría haber Cambiado".	reparador que recibe el diagnóstico de la falla.
Recibe (Servicio S_j)	sub-cronica S_j Interfaz podría haber Cambiado	Generar el evento enlazador **$BE_{ParamIncom}$**	• **E_2:** La invocación de la operación del servicio S_j se ha realizado sin problemas en un tiempo previo. • **E_2':** La invocación actual de la operación del servicio S_j (viene de S_i) resultado del protocolo SOAP al intentar invocar con parámetros incompatibles a S_j.	Conocer la ubicación del diagnosticador que recibe el evento enlazador (D_i)
Falla debido a Respuesta de Error (Figura 4.20)				
Invoca (Servicio S_i)	N/A	N/A	N/A	N/A
Recibe (Servicio S_j)	sub-cronica S_j Respuesta de Error	Generar el reconocimiento de la falla.	• **E_2:** La invocación actual de la operación del servicio S_j sin problemas. • **E_3:** Resultado de la operación del servicio S_j con mensaje de error interno generado por el protocolo SOAP.	Conocer la ubicación del reparador que recibe el diagnóstico de la falla.
Falla debido a Incorrecto Orden (Figura 4.27)				
Invoca	sub-cronica S_i	Generar el	• **E1:** Corresponde a	El tiempo de

Falla Incompatibilidad de Parámetros (Figura 4.7)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
(Servicio S_i)	Incorrecto Orden	reconocimiento de la falla.	la entrada del servicio S_i. • **BE_{IncOrd}:** Evento generado cuando se reconoce la sub-crónica S_j "Incorrecto Orden".	retardo inherente en las comunicaciones ΔT debe seleccionarse de acuerdo a un estudio previo del tiempo promedio que tarda un evento en llegar a D_j desde D_i. Adicionalmente, se debe conocer la ubicación del reparador que recibe el diagnóstico de la falla.
Recibe (Servicio S_j)	sub-cronica S_j Incorrecto Orden	Generar el evento enlazador **BE_{IncOrd}**	• **$E_{j,Si}$:** Corresponde a la salida del servicio S_i. • **E_j:** Corresponde a la salida del servicio S_j	La restricción temporal Tjsi > Tj se infiere facilmente del documento WS-CDL o BPEL al depender el servicio j del servicio i. Adicionalmente, se debe conocer la ubicación del diagnosticador que recibe el evento enlazador (D_i).

Tabla 8.2: Configuraciones para las fallas de Incompatibilidad de Parámetros, Interfaz podría haber Cambiado, debido a Respuesta de Error, debido a Incorrecto Orden

Al igual que los patrones de crónicas de la Tablas 8.1, los eventos de las

crónicas para estas fallas son generados por la aplicación SOA: propios de la aplicación (interacciones entre los servicios) y fallas generados por el protocolo SOAP. El valor de ΔT se determina en cada crónica en base a la experiencia que se tenga sobre el tiempo que tardan en llegar los eventos enlazadores a cada diagnosticador D_i. Estos patrones de crónicas deben agregarse en cada sub-flujo de la aplicación en la que un servicio S_j dependa de la salida de un servicio S_i.

8.2.1.3 Fallas Debido a Acción no-Determinada, debido a Servicio Incorrecto y Debido a Mal Comportamiento del Flujo Ejecución

Las configuraciones para estas fallas se muestra en la Tabla 8.3.

Falla debido a Acción no-Determinada (Figura 4.13)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
Invoca (Servicio S_i)	sub-cronica S_i Acción no-Determinada	Generar el reconocimiento de la falla.	• **E_1:** Invocaciones previas al servicio Sj que han resultado correctas. • **$\neg BE_{NdetAct}$:** NO haber recibido previamente el evento enlazador (la sub-cronica S_j "Acción no-Determinada" no se ha activado nunca). • **E_1':** La salida de la operación en el momento actual que se encuentra antes del servicio S_j. • **$BE_{NdetAct}$:** Evento generado cuando se reconoce la sub-crónica S_j "Acción	El tiempo de retardo inherente en las comunicaciones ΔT debe seleccionarse de acuerdo a un estudio previo del tiempo promedio que tarda un evento en llegar a D_j desde D_i. Adicionalmente, se debe conocer la ubicación del reparador que recibe el diagnóstico de la falla.

Falla debido a Acción no-Determinada (Figura 4.13)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
			no-Determinada".	
Recibe (Servicio S_j)	sub-cronica S_j Acción no-Determinada	Generar el evento enlazador **$BE_{NdetAct}$**	• **E_2:** La invocación de la operación del servicio S_j se ha realizado sin problemas en un tiempo previo. • **E_2':** La invocación actual de la operación del servicio S_j (viene de S_i) genera una mala respuesta (BadResponse).	Conocer la ubicación del diagnosticador que recibe el evento enlazador (D_i). El proceso para detectar que el evento E_2' es una mala respuesta es el resultado
Falla debido a Servicio Incorrecto				
Invoca (Servicio S_i)	sub-cronica S_i Comportamiento incomprendido	Generar el reconocimiento de la falla.	• **E_1:** La salida de la operación en el momento actual que se encuentra antes del servicio S_j. • **$BE_{IncServ}$:** Evento generado cuando se reconoce la sub-crónica S_j "Comportamiento incomprendido".	El tiempo de retardo inherente en las comunicaciones ΔT debe seleccionarse de acuerdo a un estudio previo del tiempo promedio que tarda un evento en llegar a D_j desde D_i. Adicionalmente, se debe conocer la ubicación del reparador que recibe el diagnóstico de la falla.
Recibe (Servicio S_j)	sub-cronica S_j Comportamiento incomprendido	Generar el evento enlazador **$BE_{IncServ}$**	• **$\neg E_2$:** La invocación de la operación del servicio S_j no se ha realizado previamente. • **E_2:** La invocación actual de la	Conocer la ubicación del diagnosticador que recibe el evento enlazador (D_i).

Falla debido a Acción no-Determinada (Figura 4.13)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
			operación del servicio S_j (viene de S_i) genera una mala respuesta (BadResponse).	
Falla debido a mal comportamiento del flujo Ejecución (Figura 4.17)				
Invoca (Servicio S_{i-k})	sub-cronica S_{ji-k} Mal comportamiento del flujo Ejecución	Generar el evento enlazador **BE_{TRUE}**	• **$E_{i-k,\ i-k-1}$:** La invocación actual de la operación del servicio S_{i-k} que no se sabe si fue correcta (status ≠TrueResponse). • **BE_{Misbe_Flow}:** El evento enlazador generado en la sub-cronica S_i Mal comportamiento del flujo Ejecución. • **BE_{TRUE}:** El evento enlazador generado en la sub-cronica S_{i-k+1} Mal comportamiento del flujo Ejecución	Conocer la ubicación del diagnosticador previo a la invocación del servicio S_{i-k} (servicio S_{i-k-1}).
	sub-cronica S_{i-k} Mal comportamiento del flujo Ejecución 1	Generar el reconocimiento de la falla.	• **$E_{i-k,\ i-k-1}$:** La invocación actual de la operación del servicio S_{i-k} que no se sabe que fue correcta (status = TrueResponse) o es el diagnosticador al principio de la composición (i-k = 1). • **BE_{Misbe_Flow}:** El evento enlazador generado en la sub-cronica S_i Mal comportamiento del flujo Ejecución.	Conocer la ubicación del reparador que recibe el diagnóstico de la falla.

Falla debido a Acción no-Determinada (Figura 4.13)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
			• **BE_{TRUE}:** El evento enlazador generado en la sub-cronica S_{i-k+1} Mal comportamiento del flujo Ejecución	
Recibe (Servicio S_i)	sub-cronica S_i Mal comportamiento del flujo Ejecución	Generar el evento enlazador **BE_{Misbe_Flow}.**	• **$E_{i, i-1}$:** La invocación actual de la operación del servicio S_i con una mala rerspuesta (BadResponse).	Conocer la ubicación del resto de diagnosticadores para emitir el evento enlazador.
		Generar el evento enlazador **BE_{TRUE}**		Conocer la ubicación del diagnosticador previo a la invocación del servicio S_i (servicio S_{i-1}).

Tabla 8.3: Configuraciones para las fallas de Acción no-Determinada, Servicio Incorrecto y mal comportamiento del flujo Ejecución

Al igual que los patrones de crónicas de la Tablas 8.1, los eventos de las crónicas para estas fallas son generados por la aplicación SOA: propios de la aplicación (interacciones entre los servicios) y eventos con estado sobre la calidad de la información contenida en los eventos (BadResponse y TrueResponse). El valor de ΔT se determina en cada crónica, en base a la experiencia que se tenga sobre el tiempo que tardan en llegar los eventos enlazadores a cada diagnosticador D_i. Los patrones de crónicas Acción no-Determinada y Servicio Incorrecto deben agregarse en cada sub-flujo de la aplicación en la que un servicio S_j dependa de la salida de un servicio S_i. En el caso de mal comportamiento del flujo Ejecución, debe agregarse a cada diagnosticador que

forma parte de la aplicación.

8.2.1.4 Falla Debido a la Violación del Acuerdo de Nivel de Servicio (SLA), Calidad de Servicio (QoS) y Fuera de Tiempo

Las configuraciones para estas fallas se muestra en la Tabla 8.4.

Falla debido a Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (Figura 6.19)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
Invoca	S_j SLA OR QoS	Generar el reconocimiento de la falla.	• **$E(_{j.1}, T_{j.1})$, .., $E(_{j.m1}, T_{j.m1})$:** Los m1 eventos internos del servicio S_j, con restricciones no funcionales expresados en la función noFunctional(Flow({$E(_{j.1, Tj.1})$, .., $E(_{j.m1, Tj.m1})$})) • **$BE_{SLAORQoS}$:** Corresponde al evento enlazador cuando se reconoce la sub-crónica S_k SLA OR QoS, el cual su frecuencia de ocurrencia se encuentra entre n_1 y n_2.	Para construir las restricciones de SLA o QoS en la función noFunctional es necesario contar con una ontología para definirlas que permitan extraer las restricciones realizando inferencia sobre la ontología y poder expresarlas posteriormente en la sub-crónica. Igualmente paso con los parámetros de la frecuencia de ocurrencia n1 y n2, lo cuales deben apoyarse sobre la misma ontología de SLA o QoS para poder definirlos. Conocer la ubicación del reparador que recibe el diagnóstico de la falla.
Recibe	S_k SLA OR QoS	Generar el evento enlazador **$BE_{SLAORQoS}$**	• **$E(_{k.1}, T_{k.1})$, .., $E(_{k.m2}, T_{k.m2})$:** Los m2 eventos internos del servicio S_k, con restricciones no funcionales	Para construir las restricciones de SLA o QoS en la función noFunctional es necesario contar con una ontología para definirlas que permitan

Falla debido a Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (Figura 6.19)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
			expresados en la función noFunctional(Flow({$E_{(k.1,\ Tk.1)}$, .., $E_{(k.m1,\ Tk.m1)}$}))	extraer las restricciones realizando inferencia sobre la ontología y poder expresarlas posteriormente en la sub-crónica. Conocer la ubicación del diagnosticador que recibe el evento enlazador (D_j).
Falla debido a Fuera de Tiempo (Figura 4.29)				
Invoca	S_j Fuera de Tiempo	Generar el reconocimiento de la falla.	• **E_2:** La invocación actual de la operación del servicio S_j (viene de S_{j-1}). • **$BE_{Timeout}$:** El evento enlazador generado cuando se reconoce la sub-crónica S_k Fuera de Tiempo, el cual puede ser recibido en un tiempo de $T_2+\Delta T$, donde T_2 es la ocurrencia del evento E_2.	El tiempo de retardo inherente en las comunicaciones ΔT debe seleccionarse de acuerdo a un estudio previo del tiempo promedio que tarda un evento en llegar a D_j desde D_k. Adicionalmente, se debe conocer la ubicación del reparador que recibe el diagnóstico de la falla.
Recibe	S_{jk} Fuera de Tiempo	Generar el evento enlazador **$BE_{Timeout}$**	• **$INVE_2$:** Generado por el middleware cuando no se ha generado una respuesta del servicio Sj (no se ha generado el evento E4, entonces se informa a Sk de	ΔT_{espera} y $\Delta T_{Timeout}$ son configurados de acuerdo a un estudio previo del servicio S_j o extraido de una ontología (posiblemente QoS o SLA) que permita inferirlos (son ofrecidos por el proveedor de Sj y puesto a disposición).

Falla debido a Violación del Acuerdo de Nivel de Servicio (SLA) y Calidad de Servicio (Figura 6.19)				
Tipo de Operación	**Diagnosticador**		**Configuraciones**	
	Patrón Sub-crónica	**Resultado del Reconocimiento**	**Eventos**	**Otras**
			la invocación de E_2) en un tiempo de ΔT_{espera}. • **$\neg E_4$:** La invocación final del servicio Sk no se ha realizado en un tiempo $\Delta T_{Timeout}$.	Conocer la ubicación del diagnosticador D_j

Tabla 8.4: Configuraciones para las fallas de SLA, QoS y fuera de tiempo

La configuración de estos patrones de crónicas difieren de los mostrados previamente, ya que es necesario contar con información sobre los atributos no funcionales y de desempeño de los servicios de la aplicación. Así, para la generación de las restricciones no funcionales de los eventos del patrón de crónica de SLA y QoS, expresada en la función noFunctional en cada subcrónica, es necesario apoyarse en algunas estructuras y herramientas que permitan a un ente no humano inferir dicha información y poder colocarla en las sub-crónicas. En esta tarea, el generador de patrones se puede apoyar en ontologías (posiblemente expresada en OWL), que permitan describir las restricciones no temporales (restricciones no temporales como los mostrados en las pruebas de los Capítulos 6 y 7), que son suministradas por el proveedor de cada servicio. Adicionalmente, para las restricciones temporales de fuera de tiempo y algunos casos de SLA y QoS (como retardo), pueden utilizarse un conjunto de invocaciones que permitan extraer la información de cuanto tarda el servicio en dar su respuesta. En cuanto a la implementación de los patrones de crónicas, esta puede realizarse en distintos sub-flujos de la aplicación (empezando desde la iteración de dos servicios a sub-flujos más complejas).

Es importante mencionar que la dificultad de implementar estas crónicas esta en la construcción del analizador que permita detectar cuando la información contenida en un evento no es correcta.

8.2.2. Buscador de Flujos Equivalentes

Al momento de realizar el diagnóstico y corrección de fallas en una aplicación SOA, un componente puede estar trabajando pro-activamente para encontrar flujos alternativos de la aplicación. Los flujos alternativos de una aplicación están compuestos por un conjunto de servicios que al realizar su composición dan como resultado la funcionalidad que provee la aplicación. Así, el trabajo de encontrar flujos equivalentes consiste en modificar el flujo actual de la aplicación por otra que cumpla la misma función. Una manera de realizarlo es reemplazando uno o más servicios de la composición por otros equivalentes (por ejemplo, configurándolo con nuevos valores, tomando una instancia del servicio en otro servidor, u otro servicio equivalente, entre otros), siempre que se generen los mismos resultados que se obtenían con la aplicación original.

La exactitud que confiere encontrar servicios equivalentes está basada en la calidad de información que se tiene sobre la aplicación SOA (a nivel de los servicios y sus interacciones), la cual consiste en conocer la información de un servicio en tres aspectos:

- La operación del servicio: Se debe poseer una descripción de la operación que realiza el servicio.
- Variables de entrada y salida: Se debe conocer las variables de entrada y salida al servicio.
- Propiedades no funcionales: Se debe conocer las propiedades no

funcionales de SLA y QoS de un servicio.

La principal fuente de información para describir un servicio son los documentos que contienen la descripción de los servicios (WSDL), las distintas interacciones de los servicios dentro de la aplicación (WS-CDL), y el directorio de servicios UDDI. Ahora bien, esto no es suficiente para garantizar que el servicio que se encuentra en la composición pueda ser reemplazado por otro servicio de otro proveedor. Así, usualmente en la búsqueda de servicios equivalentes se unen las herramientas provistas por la arquitectura SOA (WSDL, WS-CDL, UDDI, entre otros) con ontologías que permitan describir semánticamente los servicios (conocer el contexto en que se desenvuelve la aplicación), para de esta manera poder realizar inferencias que permitan encontrar flujos equivalentes.

Las ontologías usadas comúnmente para describir los servicios están basadas en OWL-S [77] o WSMO [78]:

- OWL-S esta compuesta por el perfil (service profile) que describe que hace el servicio, el modelo de proceso (process model) que describe como un cliente puede interactuar con un servicio, y la conexión (service grounding) para conocer los detalles para que se pueda dar su invocación.

- WSMO: permite la descripción de un servicio tanto a nivel de capacidades (funcionales) como de uso (interfaz).

Por otro lado, este nuevo componente no necesariamente debería formar parte de la arquitectura distribuida de ARMISCOM, donde se coloca una instancia por cada servicio; por el contrario, debería seguir una arquitectura distribuida distinta basada posiblemente en agentes de software, que pueden realizar búsquedas independientes en Internet, haciendo uso de Ontologías (OWL-S o WSMO) las

herramientas SOA (el directorio UDDI de servicios, el documento de descripción del servicio WSDL, y la descripción de la composición (WS_CDL) y posiblemente alguna estrategia de búsqueda inspirada en las técnicas comúnmente usadas en inteligencia artificial para realizar tareas similares, como las colonias de hormigas, entre otras.

8.2.3. Extensión de la Arquitectura de ARMISCOM

En el Capítulo 7 se ha comprobado el funcionamiento completo de ARMISCOM, logrando realizar la gestión automática de fallas en la composición de servicios (auto-sanación), lo que le confiere al middleware capacidades de auto-gestión (autonómico), al lograr descubrir, diagnosticar y reparar fallas en aplicaciones SOA, tal como se ha comprobado en los experimentos del apartado 7.2.

Como se ha puesto en evidencia en las sub-secciones anteriores (8.2.1 y 8.2.2), es necesario la construcción de dos nuevos componente en ARMISCOM para realizar el ajuste/adaptación/adecuación de las crónicas, y la actualización de la base de datos con los distintos flujos alternativos para solventar las fallas.

La arquitectura de ARMISCOM corresponde a un middleware reflexivo de 2 niveles, base y meta, causalmente conectados (intercambio de mensajes, Sistema SOA, etc.), el cual puede ser extendido con más niveles. Por ejemplo, un nuevo nivel podría definirse para realizar las tareas de aprendizaje señaladas en las sub-secciones anteriores, conectándose con el nivel meta para realizar su introspección e intersección, necesario en los procesos de aprendizaje por definir, tal que los niveles adyacentes (base y meta) no deban conocer de su existencia. Así, en base a los requerimientos de configuración de crónicas y flujos alternativos para automatizar ARMICOM, un posible nuevo nivel meta 2 se muestra en la Figura 8.1.

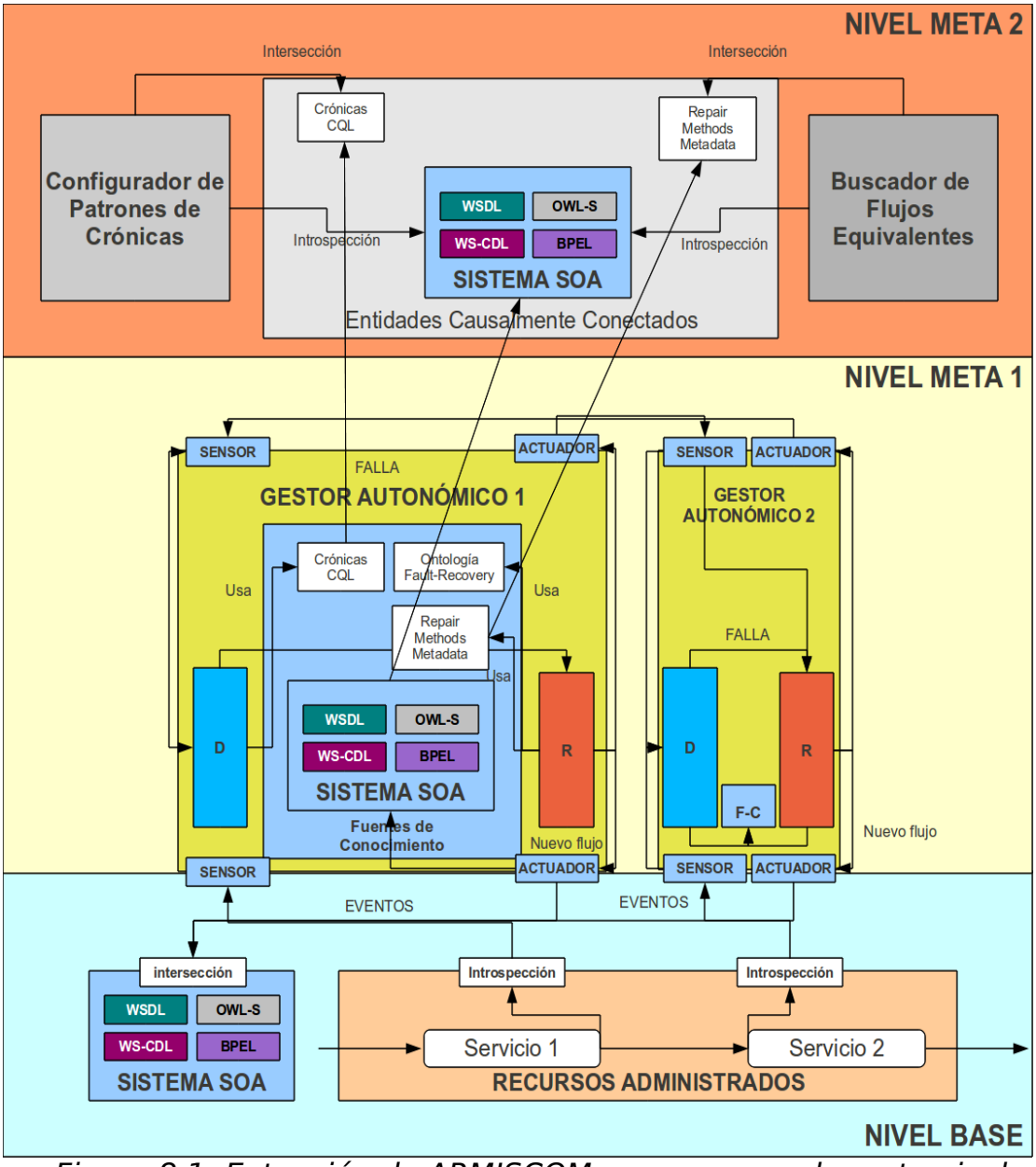


Figura 8.1: Extensión de ARMISCOM con un segundo meta nivel (meta nivel 2)

Como se muestra en la Figura 8.1, la extensión del middleware propuesta en esta sección mantiene los dos primeros niveles desarrollados en este trabajo (nivel base y meta 1), pero añade un segundo nivel meta para gestionar lo

relacionado a la configuración automática de los patrones distribuidos de crónicas, y el ingreso de nuevas regiones que funcionen como flujos alternativos cuando sea necesario reparar la aplicación SOA. Es decir, ese nivel meta gestionaría todos los procesos de aprendizaje a incorporarle a ARMISCOM, para conferirle más autonomía.

El nivel meta 2 cuenta con dos componentes: Configurador de Patrones de Crónicas y Buscador de Flujos Equivalentes (descritos en las secciones 8.2.1 y 8.2.2, respectivamente), los cuales se encuentran causalmente conectados con el nivel meta 1, en particular, con el sistema SOA, las crónicas en CQL y la metadata. Ambos componentes realizan la introspección e intersección del nivel meta 1, actualizando los patrones de crónicas y la metadata. Adicionalmente, si se busca rapidez al momento de configurar los patrones de crónicas, el componente Configurador de Patrones de Crónicas podría conectarse causalmente con la metadata y realizar la configuración de los patrones de crónicas para todas las regiones equivalentes que se encuentran almacenadas, facilitando la tarea de ajustar las crónicas cuando se repara la aplicación (se realiza previamente).

Referencias Bibliográficas

[1] [1] M. Josuttis, "SOA in Practice The Art of Distributed System Design, " O'Reilly, ISBN-13: 978-0-596-52955-0.

[2] R. Dongning, J. Zhihua y J. Yunfei, "Fault Tolerant Web Services Composition as Planning, " en International Conference on Intelligent Systems and Knowledge Engineering (ISKE 2007), 2007.

[3] R. Lapadula, R. Pugliese y F. Tiezzi, "A Calculus for Orchestration of Web Services, " en Proc. of 16th European Symposium on Programming (ESOP'07), volumen 4421 , paginas 33-47, 2007.

[4] J.Camara, C. Canal y J. Cubo, J, "Issues in the formalization of Web Service Orchestrations, " en Second International Workshop on Coordination and Adaptation Techniques Entities (WCAT05), 2005.

[5] O. Kopp y F. Leymann, Frank (2008). "Choreography Design Using WS-BPEL, " en IEEE Data Engineering 31(2), paginas 31-34, 2008.

[6] K. S. Chan, J. Bishop y L. Baresi, J. Steyny, "A Fault Taxonomy for Web Service Composition", en Service-Oriented Computing - ICSOC 2007 Workshops: ICSOC 2007, International Workshops, 2007.

[7] K. Czajkowski, S. Fitzgerald, I. Foster y C. Kesselman, C, "Grid Information Services for Distributed Resource Sharing, " en Proceeding of the 10th IEEE International Symposium on High Performance Distributed Computing, paginas 181-184, 2001.

[8] X. Pucel, "Unified Point of View on Diaagnosability, " Tesis Doctoral, Universidad de Toulouse, Toulouse - Francia, 2008.

[9] WS-Diamond, "WS-Diamond, IST-516933, Deliverable D5.1, Characterization of diagnosability and repairability for self-healing Web Services, " 2005. Disponible en: http://freepdfs.net/ws-diamond-ist-516933-deliverable-d51/2340639935f8d5e5c0708b0bbdf6d2d6/

[10] C. Dousson, P. Gaborit y M. Ghallab, "Situation recognition:representation and algorithms, " en: Proc. of the Int. Joint Conf. on Artificial Intelligence (IJCAI'93), pàginas: 166–172, 1993.

[11] C. Dousson, "Suivi d'évolutions et reconnaissance de chroniques, " Tesis doctoral, Universidad de Toulouse, Toulouse - Francia, 1994.

[12] X. Le Guillou, M. O. Cordier, S. Robin y L. Rozé, "Chronicles for On-line Diagnosis of Distributed Systems, " en Proceeding of the 2008 conference on ECAI 2008: 18th European Conference on Artificial Intelligence, paginas: 194-198, 2008.

[13] M. O. Cordier y C. Dousson, "Alarm driven monitoring based on chronicles, " en: Proc. of Safeprocess'2000, pàginas 286–291, 2000.

[14] M. O. Cordier, J. Krivine, P. Laborie, P y S. Thi´ baux, "Alarm processing and reconfiguration in power distribution systems", en: Proc. of IEA-AIE'98, paginas 230–240, 1998.

[15] R. Quiniou, M. O. Cordier, G. Carrault y F. Wang, "Application of ilp to cardiac arrhythmia characterization for chronicle recognition, " en: ILP'2001, pàginas: 220–227, 2001.

[16] WS-Diamond, "WS-Diamond, IST-516933, Deliverable D4.3, Specification of diagnosis algorithms for Web Services – phase 2, " Version 0.5, 2008. Disponible en: http://wsdiamond.di.unito.it/Reports/d4_3.pdf

[17] J. Vizcarrondo, J. Aguilar, E. Exposito, E y A. Subias, "ARMISCOM: Autonomic Reflective COMposition, " en: Infrastructure and MIddleware Proceedings Networking of for the management 4th Symposium Global (GIIS Service Information 2012), IEEE Communication Society, Choroni, Venezuela, 2012.

[18] J. Vizcarrondo, J. Aguilar, E. Exposito, E y A. Subias, "Crónicas Distribuidas para el Reconocimiento de Fallas, " en Ciencia e Ingeniería, AVol 36, No 3, paginas 73-84, 2015.

[19] M.G. Fugini y E. Mussi, E, "Recovery of Faulty Web Applications through Service Discovery, " en: Proceedings of the 1st SMR-VLDB Workshop, Matchmaking and Approximate Semantic-based Retrieval: Issues and Perspectives, 32nd International Conference on Very Large Databases, paginas 67-80, September 2006.

[20] D. Ardagna, C. Cappiello, M. Fugini, E. Mussi, B. Pernici y P. Plebani, "Faults and recovery actions for self-healing web services", en 15th Int. World Wide Web Conf., 2006.

[21] G. Huang, X. Liu y H. Mei, "SOAR: Towards Dependable Service-Oriented Architecture via Reflective Middleware, " en International Journal of Simulation and Process Modelling, Vol. 3, Issue 1/2, paginas 55-65, 2007.

[22] R. Halima, E. Fki, K. Drira and M. Jmaiel, "Experiments results and large scale measurement data for web services performance assessment, " en Proceeding of the IEEE Symposium on Computers and Communications, paginas. 83-88, 2009.

[23] WS-Diamond project, "WS-Diamond, IST-516933, Deliverable D4.3, Specification of diagnosis algorithms for Web Services – phase 2, " disponible en: http://wsdiamond.di.unito.it/.

[24] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi y M. Segnan, "Enhancing Web Services with Diagnostic Capabilities, " en Proceedings of the Third European Conference on Web Services, paginas 182-191, 2005.

[25] A. Sherif, A. Z. Gurguis, "Towards autonomic web services: achieving self-healing using web services, " en DEAS '05 Proceedings of the 2005 workshop on Design and evolution of autonomic application software, paginas 1-5, 2005.

[26] C. Diop, "An Autonomic Service Bus for Service-based Distributed Systems, " Tesis doctoral, Universidad de Toulouse, Toulouse - Francia, Abril 2015.

[27] M., C. Ghedira y Z. Maamar, "Towards a Self-Healing Approach to Sustain Web Services Reliability, " en 2011 {IEEE} Workshops of International Conference on Advanced Information Networking and Applications (March 2011), paginas 267-272, doi:10.1109/WAINA.2011.101 Key: citeulike:9844763.

[28] H. Psaier, L. Juszczyk, F. Skopik, D. Schall y S. Dustdar, "Runtime

Behavior Monitoring and Self-Adaptation, " en Service-Oriented Systems. En Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference, paginas 164-173, ISBN:978-0-7695-4232-4.

[29] A. Boufaied, A. Subias y M. Combaceau, "Distributed fault detection with delays consideration, " en Proc. of the 15th Int. Workshop on Principles of Diagnosis (DX'04), 2004.

[30] A. Mhalla, N. Jerbi, S. Collart, E. Craye y M. Benrejeb, "Distributed Monitoring Based on Chronicles Recognition for Milk Manufacturing Unit, " en Journal. of Aut. & Syst. Eng. (JASE), Vol. 4(1), 2010.

[31] A. Boufaied, A. Subias y M. Combacau, "Chronicle modeling by Petri nets for distributed detection of process failures, " en Second IEEE International Conference on Systems, Man and Cybernetics (SMC 2002), IEEE Computer Society Press, 2002.

[32] C. Dousson, "Extending and unifying chronicle representation with event counters, " en ECAI, paginas 257–261, 2002.

[33] A. Artikis, G. Paliouras, F. Portet, A. Skarlatidis, "Logic-based representation, reasoning and machine learning for event recognition, " en Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, 2010.

[34] T. Erl, "Service Oriented Architecture (SOA) Concepts, Technology and Design, " Prentice Hall, 2005.

[35] Microsoft Corporation, "La Arquitectura Orientada a Servicios (SOA)

de Microsoft aplicada al mundo real, " Microsoft Corporation, 2006.

[36] Microsoft Corporation, "Service Oriented Architecture (SOA) in the Real World, " Microsoft Corporation, Free Book, 2007.

[37] Y. Vasiliev, "SOA and WS-BPEL Composing Service-Oriented Solutions :with PHP and ActiveBPEL, " Pack Publishing, Birmingham-Mumbai, ISBN 978-1-847192-70-7, 2007.

[38] IBM, "Implementing an SOA Using an Enterprise Service Bus, " IBM Redbooks publication, 2004.

[39] B. Matjaz, R. Juric, S. Loganathan, F. Jennings, "SOA Approach to Integration XML, Web services, ESB, and BPEL in real-world SOA projects, " Pack Publishing, Birmingham-Mumbai, ISBN 978-1-904811-17-6, 2007.

[40] W3C, "Web Service Choreography Interface (WSCI) 1.0. Revisado Diciembre 2014, ", Disponible en: http://www.w3.org/TR/wsci/.

[41] A. Barros, M. Dumas y P. Oaks, "A Critical Overview of the Web Services Choreography Description Language (WS-CDL), " BPTrendsNewsletter, 2005.

[42] K. Václav, "Automatic generation the fault signature table due to classification", en Opotřebení, spolehlivost, diagnostika. Brno: Univerzita obrany, 2006. ISBN 80-7231-165-4.

[43] B. Dubuisson, M. Staroswiecki, J. P. Cassar y T. Denoeux, "Surveillance des systèmes continus, " en Ecole d'Été d'Automatigue

de Grenoble, 1996.

[44] E. Curry, "Adaptive and Reflective Middleware, " en Chapter in Middleware for Communications, John Wiley and Sons, Chichester, England, paginas 29-52, 2004.

[45] W. Emmerich, "Software engineering and middleware: A roadmap, " en Communications of the ACM, paginas 117–129, 2000.

[46] F. Kon, F. Costa, G. Blair, H. Campbell, "The case for reflective middleware, " en Commun. ACM, Vol. 45, No. 6, 2002.

[47] C. Quintero, "Arquitectura de Software Dinamica basada en Reflexión, " Tesis doctoral, Universidad de Valladolid, Valladolid - España, 2002.

[48] F. Rideau, "Métaprogrammation et libre disponibilité des sources, " En Autour du Libre 1999, 1999.

[49] Y. Sizhong, L. Jinde y L. Zhigang, "RECOM: a reflective architecture of middleware, " en Info-tech and Info-net, 2001.

[50] G. Coulson, G, "What is Reflective Middleware?, " IEEE Distributed Systems, 2002. Disponible en: http://dsonline.computer.org/middleware/RMarticle1.htm

[51] H. Gang, L. Xuanzhe y M. Hong, M, "SOAR: Towards Dependable Service Oriented Architecture via Reflective Middleware, " en International Journal of Simulation and Process Modelling (IJSPM), Volume 3, Issue 1/2, paginas 55-65, 2007.

[52] G. S. Blair, G. Coulson, A. Andersen, M. Clarke, F. Costas, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, K. Saikoski, "The Design and Implementation of Open ORB V2, " en IEEE Distributed Systems Online, Vol. 2, No. 6, 2001.

[53] IBM Corporation, "An architectural blueprint for autonomic computing, " Fourth Edition, 2006. Disponible en: http://www.ginkgo-networks.com/IMG/pdf/AC_Blueprint_White_Paper_V7.pdf

[54] S. Cartes, "The New Language of Business SOA & Web 2.0, " IBM Press, 2007. ISBN-13: 978-0-13-195654-4

[55] C. Dousson, P. Gaborit, M. Ghallab, M, "Situation recognition: representation and algorithms", en Proc. of the Int. Joint Conf. on Artificial Intelligence (IJCAI'93), paginas 166-172, 1993.

[56] P. Maes, "Concepts and Experiments in Computational Reflection", en Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, paginas 147-155, 1987.

[57] D. Chiribuca, D. Hunyadi y E. Popa, "The Educational Semantic Web", en 8th WSEAS International Conference on Applied Informatics and Communications, paginas 314-319, 2008.

[58] X. Feng, H. Wang, Q. Wu y B. Zhou, "An adaptive algorithm for failure recovery during dynamic service composition," en Pattern Recognition and Machine Intelligence, ser. Lecture Notes in Computer Science, A. Ghosh, R. De, and S. Pal, Eds. Springer Berlin / Heidelberg, vol. 4815,

paginas 41-48, 2007.

[59] X. Feng, Q. Wu, H. Wang, Y. Ren y C. Guo, "ZebraX: A model for service composition with multiple QoS constraints", en Advances in Grid and Pervasive Computing, ser. Lecture Notes in Computer Science, C. Cerin and K.-C. Li, Eds. Springer Berlin / Heidelberg, vol. 4459, paginas 614-626, 2007.

[60] G. Canfora, M. Di Penta, R. Esposito y M. L. Villani, "A framework for QoS-aware binding and re-binding of composite web services", en Journal of Systems and Software, vol. 81, paginas 1754-1769,October 2008.

[61] J. Lin, J. Zhang, Y. Zhai y B. Xu, "The design and implementation of service process reconfiguration with end-to-end QoS constraints in SOA," en Service Oriented Computing and Applications (SOCA), vol. 4, no. 3, paginas 157–168, 2010.

[62] H. Saboohi, A. Amini, y H. Abolhassani, "Failure recovery of composite semantic web services using subgraph replacement," en International Conference on Computer and Communication Engineering (ICCCE), Kuala Lumpur, Malaysia, paginas 489-493, 2008.

[63] S. Poonguzhali, R. Sunitha, y G. Aghila, “Self-Healing in Dynamic Web Service Composition", en International Journal on Computer Science and Engineering, Vol. 3, No. 5. paginas 2054-2060, 2011.

[64] OpenESB community web site, "Welcome to OpenESB community web site", OpenESB community web site, 2015.Disponible en:

http://www.open-esb.net/

[65] Oracle White Paper, "Enterprise Service Buses Where are they going?", 2006. Disponible en: https://java.net/downloads/oraclesoasuite11g/OSB/osb_examples_tutorials_111140_and_later.pdf

[66] Protégé, "A free, open-source ontology editor and framework for building intelligent systems", 2015. Disponible en:http://protege.stanford.edu/

[67] R. Kajic, "Evaluation of the Stream Query Language CQL", Institutionen för informationsteknologi, Uppsala universitet, 2010.

[68] Sun Microsystems, "Intelligent Event Processor (IEP): User's Guide", Parte No: 821–1070–11, 2010.

[69] A. Arasu, S. Babu, J. Widom, "The CQL continuous query language: semantic foundations and query execution", en Journal The VLDB Journal - The International Journal on Very Large Data Bases, Volume 15 Issue 2, 2006.

[70] B. Morin, H. Debar, "Correlation on intrusion: an application of chronicles", en 6th International Conference on recent Advances in Intrusion Detection RAID, Pittsburgh, USA, 2003

[71] X. Le Guillou, M. O. Cordier, S. y L. Roz, "Monitoring WS-CDL-based choreographies of Web Services", en DX'09 (20th International Worshop On Principles of Diagnosis) , Stockholmo, Suecia, 2009.

[72] Imagine project, "Innovative end-to-end Management of Dynamic

Manufacturing Networks, ". Disponible en: http://www.imagine-futurefactory.eu/index.dlg, revisado en Diciembre 2014.

[73] Imagine project, "Furniture Manufacturing Description". Disponible en: http://furnituremanufacturing.imagine-futurefactory.eu/articles/furniture-manufacturing-description/5000000000139303, revisado en Diciembre 2014.

[74] J. Vizcarrondo, J. Aguilar y E. Exposito, "Reflective Middleware for Automatic Management of service-oriented applications using the theory of Signatures of Failure", en Proceeding of the 14th WSEAS International Conference on Mathematical Methods, Computational Techniques and Intelligent Systems (MAMECTIS '12), paginas 183-188, Julio 2012.

[75] J. Vizcarrondo, J. Aguilar, E. Exposito y A. Subias, "Distributed Chronicles for Recognition of Failures in Web Services Composition", en Proceedings of the XXXIX Conferencia Latinoamericana en Informática (CLEI 2013), IEEE Xplore, Vol. 2, paginas. 92-101, Naiguatá, Venezuela, Octubre 2013.

[76] J. Vizcarrondo, J. Aguilar, E. Exposito y A. Subias, "Building Distributed Chronicles for Fault Diagnostic in Distributed Systems using Continuous Query Language (CQL)," En International Journal of Engineering Development and Research (IJEDR), ISSN:2321-9939, Vol.3, Issue 1, paginas 131-144, 2015.

[77] W3C, "OWL-S: Semantic Markup for Web Services," 2004. Disponible en: http://www.w3.org/Submission/OWL-S/

[78] D. Roman, U. Keller, H. Lausen, J. Bruijn, R. Lara R, M. Stollberg, A. Polleres, C. Feie, C. Bussler y D. Fensel D, "Web Service Modeling Ontology," en Journal

Applied Ontology archiveVolume 1 Issue 1, paginas 77-106, 2005.

[79] Mule ESB, Mule ESB | Enterprise Service Bus | Open Source ESB | MuleSoft, 2015. Disponible en https://www.mulesoft.com.

[80] Apache ServiceMix, Welcome to Apache ServiceMix!, 2015. Disponible en https://servicemix.apache.org/

Apéndices

Apendice A.1

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package ws;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import java.util.Iterator;
import java.util.Set;
import java.io.File;
import java.io.*;
import org.semanticweb.owlapi.apibinding.OWLManager;
import org.semanticweb.owlapi.model.IRI;
import org.semanticweb.owlapi.model.OWLClass;
import org.semanticweb.owlapi.model.OWLDataFactory;
import org.semanticweb.owlapi.model.OWLNamedIndividual;
import org.semanticweb.owlapi.model.OWLObjectProperty;
import org.semanticweb.owlapi.model.OWLOntology;
```

```
import org.semanticweb.owlapi.model.OWLOntologyCreationException;
import org.semanticweb.owlapi.model.OWLOntologyManager;
import org.semanticweb.owlapi.reasoner.*;
 //import uk.ac.manchester.cs.factplusplus.owlapiv3.*;
import uk.ac.manchester.cs.factplusplus.owlapiv3.FaCTPlusPlusReasoner;
/**
 *
 * @author juan Vizcarrondo
 */
@WebService(serviceName = "Ontology")
public class Ontology {
    /**
     * Web service operation
     * getRepairMethod: Get repair methods based on failure
     * input:
     *   Fault: a name fault to search
     *output:
     *   repair_methods: Repairs methods to apply
     */
    @WebMethod(operationName = "getRepairMethod")
        public String getRepairMethod(@WebParam(name = "fault_load") String
fault_load) {
        //TODO write your implementation code here:
        String repair_methods = "";
```

```
        try{
            String aux = "";
            //file src fault repair
            File file = new File("/home/juan/faultRepairsmethods.owl");
            //load ontology
            OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
            OWLOntology ont = manager.loadOntologyFromOntologyDocument(file);
            // Create a console progress monitor
                                ConsoleProgressMonitor progressMonitor = new
ConsoleProgressMonitor();
            // Specify the progress monitor setting
                                    OWLReasonerConfiguration config = new
SimpleConfiguration(progressMonitor);
            //load reasoner
                                  //OWLReasonerFactory reasonerFactory = new
Reasoner.ReasonerFactory();
                                      FaCTPlusPlusReasoner reasoner = new
FaCTPlusPlusReasoner(ont,config,BufferingMode.BUFFERING);
            reasoner.precomputeInferences();
            boolean consistent = reasoner.isConsistent();
            OWLDataFactory fac = manager.getOWLDataFactory();
                                           OWLNamedIndividual fault_ind =
fac.getOWLNamedIndividual(IRI.create("http://www.co-ode.org/ontologies/ont.owl#" +
fault_load));
                                        OWLObjectProperty hasrepairmethod =
fac.getOWLObjectProperty(IRI.create("http://www.co-
ode.org/ontologies/ont.owl#hasrepairmethod"));
            if (fault_ind == null){
```

```
                    return "";

                }

                                    NodeSet<OWLNamedIndividual>  repairValuesNodeSet  =
reasoner.getObjectPropertyValues(fault_ind, hasrepairmethod);

                Set<OWLNamedIndividual> values = repairValuesNodeSet.getFlattened();

                for(OWLNamedIndividual ind : values) {

                    String name = ind.toString();

                    name = name.substring(name.indexOf("#")+1,name.length()-1);

                    repair_methods += aux + name;

                    aux = ";";



                }

            }

            catch(UnsupportedOperationException exception) {

                repair_methods = "Unsupported reasoner operation.";

            }

            catch (OWLOntologyCreationException e) {

                    repair_methods = "Could not load the fault-repair ontology: " +
e.getMessage();

            }

            return repair_methods;

        }

    }
```