Localización de Fallas dirigida por Taxonomía en Aplicaciones Orientadas a Servicios

Ezequiel Scott
Director: Dr. Álvaro Soria

Instituto de Investigación ISISTAN, Facultad de Cs. Exactas, UNCPBA
Campus Universitario, Paraje Arroyo Seco, CP B7001BBO, Tandil, Bs. As., Argentina

ezequiel.scott@isistan.unicen.edu.ar, alvaro.soria@exa.unicen.edu.ar

Resumen El paradigma de computación orientada a servicios resulta de particular interés tanto para la industria cómo para la investigación debido a sus numerosos beneficios. En particular sus propiedades de interoperabilidad y capacidad de generar componentes altamente desacoplados, permite generar software mediante la composición de servicios. Pese a ser altamente relevante detectar fallas en estos sistemas tanto cómo para incrementar la producción, contribuir a la verificación y facilitar el reemplazo de servicios en la composición, el área de localización de fallas ha sido poco explorada. En este contexto se plantea un enfoque de localización de fallas dirigido por una taxonomía específica, que se vale del modelo arquitectónico para aumentar la expresividad de los resultados. El objetivo es brindar una guía que asista al usuario permitiendo reducir los tiempos de localización, aumentar la precisión e incrementar el conocimiento acerca de los diferentes tipos de fallas que pueden suceder en estos sistemas particulares, como lo corroboran los resultados experimentales llevados a cabo.

1. Introducción

La posibilidad de construir sistemas complejos y distribuidos cuya funcionalidad es descompuesta en términos de servicios, donde algunos de ellos son desarrollados completamente y otros ya existentes son reutilizados, hacen que las arquitecturas orientadas a servicios (SOA) sean una atractiva propuesta. De esta manera, las aplicaciones basadas en SOA son desarrolladas a través del descubrimiento y la composición de servicios, lo que hace a este estilo arquitectónico diferente de los convencionales, centrándose en las capacidades de coordinar eficientemente la interacción entre componentes (servicios)

dentro de ambientes inestables y de continua evolución [11]. Dichas capacidades hacen que los sistemas sean capaces de descubrir y componer servicios por sí mismos en tiempo de ejecución, haciendo necesario localizar servicios alternativos y reemplazar los existentes en caso de que fallen para satisfacer los requerimientos funcionales previstos.

Nuevas dificultades se presentan en este contexto distribuido pudiendo provocar comportamientos anormales, desvíandose el sistema desde sus estados correctos. Esta desviación es conocida como *error* (el comportamiento anormal observado), y la causa o raíz que se le adjudica a ese error se conoce como *falla* [3]. Entre las propiedades de un sistema basado en SOA, la localización de fallas es de vital importancia para garantizar su correcto funcionamiento, ya que resulta el punto de partida para el reemplazo de servicios y las características de auto-recuperación que contribuyen a la tolerancia a fallas como atributo de calidad [20,4]. Sin embargo, satisfacer éste atributo no es sencillo, debido a los diversos factores en juego. Por ejemplo, la reutilización de servicios cuya implementación no siempre se encuentra accesible, producto de la terciarización, prima en el paradigma. Además, pueden involucrarse múltiples invocaciones, requiriendo interacciones complejas entre clientes y proveedores, debido a la naturaleza heterogénea de los componentes.

Pese a ser una característica estratégica, la localización de fallas en SOA es un área poco explorada y los enfoques existentes tienen como característica principal su estrecha relación con la implementación del sistema, forzando al desarrollador a focalizar el análisis en las porciones de código fuente [5,19]. De esta manera, se incrementa el riesgo de perder la visión global del sistema y del problema a resolver. Estos aspectos se contradicen con el desarrollo orientado a servicios, ya que se centra en el descubrimiento y la composición de componentes en vez de la codificación de funcionalidad. Desde esta perspectiva, la mayoría de las propuestas de localización de fallas dejan de ser efectivas. Otro inconveniente se desprende del desconocimiento acerca de las fallas, convirtiendo las mismas ventajas de SOA en desventajas por las numerosas fallas que existen de diferentes naturalezas.

Es un hecho que existe una relación entre la complejidad de un sistema y las fallas que pueden ocurrir [8]. Como antes se mencionó, la terciarización de servicios provoca que a menudo la implementación

no se encuentre disponible, por ello resulta de especial importancia prestar atención a la información sobre la ejecución del sistema. Una clave importante en el éxito del tratamiento de fallas en tiempo de ejecución es conocer, en principio, cuáles son los errores a buscar [13,20]. Es decir, si el sistema posee la habilidad de distinguir entre los diferentes tipos de errores, se puede conocer las fallas en la que puede derivar, adicionar información relevante sobre las consecuencias, e incluso posibilidades de recuperación. En este sentido, el amplio espectro de errores que pueden surgir puede ser abordado desde una perspectiva taxonómica [6,16]. La información que brinda una clasificación sistemática de las posibles fallas que pueden ocurrir en SOA es valiosa ya que permite identificar aquellas que pueden ser excluidas cuando un error se produce. De esta manera, dado un error que se manifiesta en tiempo de ejecución, se puede utilizar la taxonomía para excluir aquellas partes del sistema irrelevantes para el error y circunscribir el análisis en las porciones del sistema que potencialmente contengan la falla que lo originó.

En este contexto, se presenta un enfoque que asiste al desarrollador en la localización de fallas en aplicaciones orientadas a servicios, con foco principal en el uso de la información provista por una taxonomía de fallas específica de SOA y las trazas de ejecución del sistema. En este tipo de sistemas la comunicación de las posibles causas de un error requiere de un mayor nivel de abstracción que el código fuente. Para esto, el enfoque propuesto utiliza un modelo de la arquitectura orientada a servicios y su instanciación en el sistema bajo análisis para proveer un contexto adecuado en tiempo de ejecución que contribuye a un mejor entendimiento de las razones y posibles consecuencias de las fallas existentes en dicho sistema.

El presente artículo se organiza como sigue: en la Sección 2 se presenta el enfoque propuesto, explicando la heurística de localización de fallas mediante un ejemplo. Luego, se explica cómo el enfoque fue evaluado con una serie de experimentos. La Sección 4 muestra los trabajos relacionados con la localización de fallas en SOA. Finalmente, se presenta una Conclusión acerca del enfoque.

2. Localización de fallas dirigida por taxonomía

El enfoque propuesto está centrado en el uso de la taxonomía de fallas SOA, que captura los posibles errores que podrían manifestarse y clasifica las fallas. La importancia del uso de una taxonomía yace en la organización consistente acerca de las fallas, formalizando y unificando este conocimiento mediante una organización sistemática. Si este conocimiento es combinado con la información de la comunicación entre servicios, los datos y el contexto ocurrido en tiempo de ejecución, es posible detectar las fallas ocurridas dentro de dicho contexto. Así, las trazas de ejecución se explotan para detectar las fallas que han ocurrido en función de la información aportada por la taxonomía.

La Figura 1 muestra un esquema conceptual del enfoque. El corazón de éste, la taxonomía de fallas SOA, es incluida por defecto como un artefacto del asistente para evitar que el usuario deba construirla. Se aprecian los dos roles de importancia para el asistente, el arquitecto y el desarrollador. Del primero se requiere de su conocimiento para elaborar el modelo arquitectónico. Éste consta de la especificación de n escenarios arquitectónicamente relevantes, siendo n establecido bajo el criterio y responsabilidad del arquitecto. Sin embargo, la cantidad de escenarios debieran ser los necesarios para describir el comportamiento y funcionalidad del sistema. Inmersos en un proceso de desarrollo centrado en arquitectura, el modelado de escenarios mediantela notación de Use Case Maps [7] proporciona una visión integrada y abstracta del sistema, señalando las tareas que se realizan dentro de su propio contexto. Por lo tanto, la simplicidad de la notación y su alto nivel de expresividad, proporcionan un nivel de entendimiento mayor del sistema [1]. A continuación, quien desempeña el rol del desarrollador localiza las fallas en la aplicación de forma asistida. En función de los dos roles protagonistas se puede dividir el proceso de localización de fallas asistida en dos etapas denominadas *fase de configuración* y *fase de ejecución.* En la primera se establecen los artefactos que serán utilizados por el algoritmo, mientras que en la segunda fase, el desarrollador ejercita los escenarios generando trazas de ejecución, e interactúa con el asistente hasta localizar la falla. Todos los escenarios documentados pueden ser analizados, o bien sólo aquellos que el desarrollador seleccione en

función de su experiencia, circunscribiendo el espacio de búsqueda. Los errores arrojados durante el ejercicio de cada escenario son resaltados dentro de la traza y en función del contexto arquitectónico provisto se señala la falla. Dicho contexto resulta estratégico para el enfoque, ya que provee una perspectiva más abstracta, y supone una mejora en la comprensión acerca de las fallas que ocurrieron en el sistema y su impacto sobre la totalidad del sistema. El objetivo de la propuesta es acortar la brecha existente entre la información sobre la falla (causas, consecuencias, y errores), el modelo mental que se posee de la totalidad del sistema y la correspondiente implementación, ya que se permite una navegación bidireccional entre cada uno de estos aspectos.

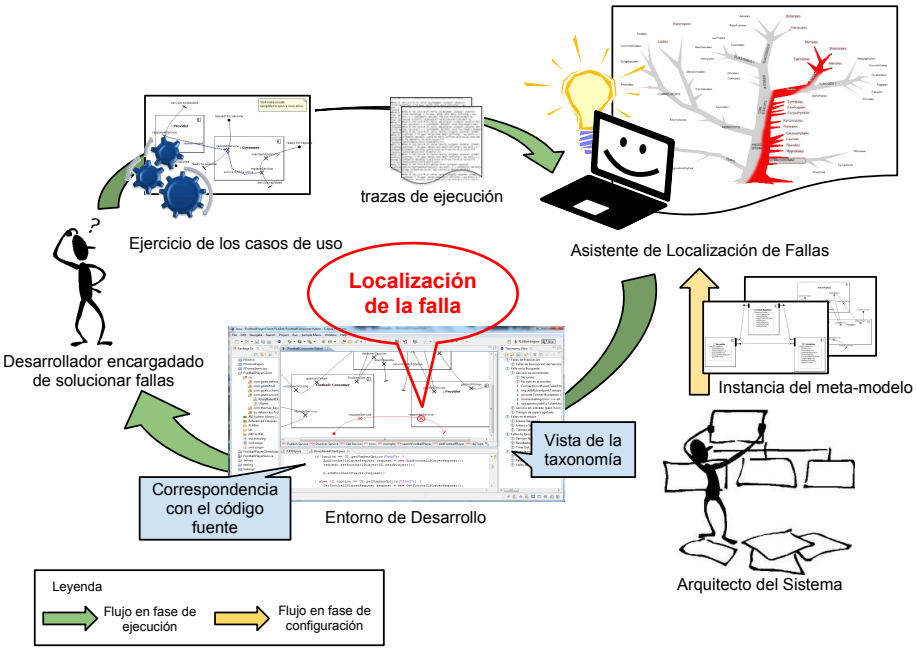


Figura 1: Modelo conceptual del enfoque.

2.1. Asistente de Localización de fallas

La taxonomía de fallas SOA considera fallas físicas, de desarrollo e interacción que pueden causar una gran variedad de errores obser-

vables en la ejecución normal de un sistema orientado a servicios. Recorriendo de las hojas hacia el tronco de la taxonomía se acumula a la información pertinente a la falla que ha ocurrido, junto con los efectos y posibles causas. Esta visión inversa de hojas hacia el tronco, surge debido a que la taxonomía puede traspolarse a un diagrama causa-efecto de Ishikawa [15] como se muestra en la Figura 2. De esta manera, un error o excepción en la hoja se interpreta como causa de los nodos intermedios en la taxonomía. Así mismo, cada nodo intermedio puede llegar a ser causa del siguiente nodo intermedio, y así sucesivamente hasta llegar a la raíz de la taxonomía. Luego, cada uno de los pares causa-efecto pueden ser traducidos en términos de reglas. Al ocurrir una determinada excepción en el sistema, se la considera como un hecho o causante de la falla. Estos hechos se pueden diferenciar dentro de la taxonomía, y mediante la aplicación sucesiva de reglas, se puede arribar a nuevas conclusiones y localizar la falla. El siguiente paso en el proceso de localización es el de expresar los resultados al usuario, de manera clara y precisa. En este punto el enfoque utiliza los beneficios de la documentación arquitectónica.

El modelo conceptual de SOA esta basado en un estilo arquitectónico que define un modelo de interacción entre tres partes principales: el *Proveedor* del servicio, el *Consumidor* y el *Registro*. El meta-modelo que describe estas relaciones [2] puede traducirse de la misma manera con la notación propuesta, es decir, estableciendo los elementos arquitectónicos de SOA como los componentes involucrados en un conjunto de escenarios típicos descritos mediante UCM, presentes en la mayoría de las arquitecturas orientadas a servicios. Dichos escenarios serían los de búsqueda de servicios, publicación de servicios e invocación (ver Apéndice A). Por lo tanto, el arquitecto debe construir la especificación arquitectónica de la aplicación a analizar, utilizando este meta-modelo en forma de plantilla. De esta manera, extendiendo los componentes y modificando tanto los escenarios como el mapeo a la implementación deseada según crea necesario construye una instancia del meta-modelo SOA sobre la cual serán reportadas las fallas.

La asistencia al desarrollador se brinda cuando éste otorga la información de ejecución obtenida en forma de trazas luego de ejercitar los escenarios correspondientes. Los errores manifestados se evalúan mediante la taxonomía SOA que permite conocer más información

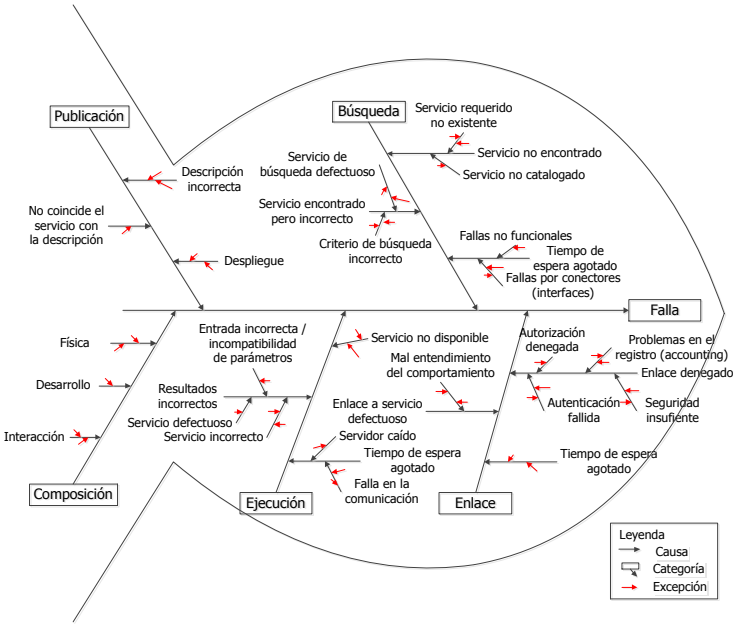


Figura 2: Diagrama causa-efecto de la taxonomía de fallas SOA.

acerca de la falla y discriminar la porción de código fuente que contiene la falla, junto con la correspondencia de este código hacia el elemento arquitectónico responsable. De esta manera se provee de los resultados al desarrollador mediante la visualización a un nivel de abstracción superior, como lo es el arquitectónico, que colabora con el entendimiento del contexto y de la falla del sistema.

2.2. Algoritmo de Localización de fallas

El enfoque se centra principalmente en la heurística de localización de fallas cuyo pseudocódigo se muestra en el Algoritmo 1. Éste toma como entrada los siguientes artefactos: un conjunto de trazas de ejecución L de la aplicación orientada a servicios sobre las que se extraen los errores, una taxonomía de fallas específica para SOA t donde se ubican dichos errores, y una instanciación del meta-modelo arquitectónico propuesto UCM que es utilizado como medio de presentar los reportes de la localización, junto al código fuente de la aplicación (*code*).

Algoritmo 1 Algoritmo de localización de fallas

```
1: procedure LocateFault(L, t, UCM, code)
2:     E ← collectExceptions(L)
3:     for all e ∈ E do
4:         taxo_ubication ← locate(e, t)
5:         adjustProbs(e,t)
6:         additionalActions(taxo_ubication)
7:         info ← info(taxo_ubication)
8:         R ← ∅
9:         for all ucm ∈ UCM do
10:            S ← locateResponsability(e, ucm)
11:            R ← R ∪ S
12:        end for
13:        for all r ∈ R do
14:            line ← getMapping(r, code)
15:            informFault(line, ucm, info)
16:        end for
17:    end for
18: end procedure
```

El algoritmo logra relacionar diferentes niveles de abstracción en la explicación de la falla. Por un lado, a un nivel mayor de abstracción permite identificar la responsabilidad fallida dentro de la arquitectura SOA dada. Además, dado que existe una relación de correspondencia entre la responsabilidad y su materialización en el código fuente, permite al desarrollador localizar en la aplicación la clase y/o método que provoca la falla. Adicionalmente, gracias a la información proporcionada por la taxonomía de fallas, se puede dar al desarrollador un nuevo conocimiento acerca de la falla que está tratando, y de sus posibles causas y consecuencias, asistiendo a su conocimiento.

Con el objeto de clarificar el algoritmo, se expone un ejemplo pequeño. Se supone que existe una aplicación de gestión de jugadores de fútbol basada en SOA. El sistema posee una falla crítica, no permitiendo recuperar la información respectiva a un determinado jugador de fútbol.

Se tiene la especificación arquitectónica (ver Apéndice B), notando que puede estar documentada previamente a la detección del error. Si no existiese dicha especificación arquitectónica, es condición necesaria construir una para asistir en la localización de fallas mediante el presente enfoque. Aquí el rol del arquitecto se hace presente, ya que

es él quien posee el conocimiento suficiente para realizar esta tarea. Sin embargo, se provee el meta-modelo antes descrito que puede ser utilizado como plantilla o punto de partida para facilitar y reducir el esfuerzo de esta tarea.Una vez que se tiene una instanciación del modelo arquitectónico, su correspondiente mapeo a la implementación, y también la taxonomía de fallas SOA (dada por defecto), la denominada *fase de configuración* se encuentra completa.

Ahora el rol que cumple relevancia es el del desarrollador, cuya tarea es localizar la falla y en medida de lo posible, subsanarla. Para ello inicia la *fase de ejecución* del enfoque. Esta fase consiste en primer medida en ejercitar los correspondientes escenarios donde el error se manifiesta. Para el ejemplo, el escenario sería *"Búsqueda de un jugador de fútbol"*, que debido a la información provista por el arquitecto, posee su correspondiente UCM. Luego de cada ejecución, mediante la instrumentación del código en tiempo de ejecución, se obtiene una traza de ejecución que representa los flujos ejecutados.

A continuación, el enfoque se encarga de ejecutar los algoritmos correspondientes para asistir al desarrollador en la localización de fallas. El asistente primero recolectará las excepciones de las trazas, en el ejemplo la excepción es *"java.net.ConnectException".* Esta excepción es ubicada dentro de la taxonomía de fallas, y entonces ahí se obtiene ya una categoría e información acerca de la posible causa del error que se manifestó. Esto, junto a la información arquitectónica provista, permite localizar en qué sección del sistema de gestión de jugadores de fútbol sucedió la falla. Notar que dicha información es provista en un alto nivel de abstracción. Para el ejemplo, el desarrollador luego de ejecutar el asistente, tendrá la información acerca de la localización de la falla a nivel arquitectónico e información sobre la naturaleza del error, que en este caso sería: *"falla de búsqueda del servicio: conexión rechazada - asegúrese que el servidor se encuentra activo*". En otras palabras, la taxonomía permite descubrir que hubo un error en la búsqueda del servidor, y la arquitectura nos informa del componente específico, el flujo y la responsabilidad, que son responsables de llevar a cabo esta tarea.

Una vez que el desarrollador posee esta información, logra descubrir cuál es la causa real de la manifestación de ese error en la aplicación: el servidor no estaba corriendo en ese momento. Es así como en el enfoque asiste al usuario, eliminando una serie de opciones y

posibles fallas gracias a la taxonomía, y focalizando en una categoría. Esta categoría en conjunción a la información arquitectónica, permite guiar al desarrollador en una reducción del espacio de búsqueda necesario para localizar la falla.

En el ejemplo, si el desarrollador no contara con la guía del enfoque, debería analizar todas las posibles causas por la cual ha sido rechazada la conexión con el servidor, (como por ejemplo podría ser que la conexión no funcione, haya problemas en los protocolos, el cliente se comporte erróneamente, etc.) e inspeccionar cada una de ellas en el código fuente, lo que sin duda implicaría mayor esfuerzo.

3. Evaluación

Para verificar si las suposiciones hechas a nivel conceptual pueden ser realizadas en la práctica, el asistente fue evaluado a través de la inyección de cinco fallas en un sistema orientado a servicios. Dichas fallas provocaron un comportamiento anormal de la aplicación, y consecuentemente los desarrolladores debían localizarlas. Las fallas fueron seleccionadas de manera de contar con una falla representativa por cada categoría principal de la taxonomía.

Para los experimentos se seleccionaron seis desarrolladores de un grupo de estudiantes avanzados de la Facultad de Ciencias Exactas perteneciente a la UNICEN. Ellos fueron divididos en dos grupos: *sin soporte* y *con soporte*. A cada grupo se les asignaron 5 fallas. Al primer grupo de desarrolladores se le solicitó que localizara las fallas utilizando el ambiente de desarrollo Eclipse[1] junto con un conjunto de documentos que describen el diseño de alto nivel del sistema, mientras que al segundo grupo se instó además a utilizar el asistente. Este segundo grupo también tuvo acceso a la documentación de diseño del sistema.

Para monitorear el comportamiento de los participantes durante la localización de fallas, se utilizó el plug-in Mylyn[2], ya que éste puede realizar el seguimiento de los métodos, clases, y atributos a medida que son accedidos por medio de Eclipse. Al finalizar el experimento, dicha información fue utilizada para el cómputo de las

[1] http://www.eclipse.org/
[2] http://www.eclipse.org/mylyn/

siguientes métricas: LOC (*Lines Of Code*) y el tiempo consumido para localizar la falla. LOC mide la cantidad de líneas de código que fueron inspeccionadas, sin considerar blancos ni comentarios, mientras que el tiempo en minutos se utiliza cómo métrica de desempeño. En particular, la métrica LOC se obtiene mediante el plugin Metrics[3] evaluado sobre los métodos, clases y atributos detectados por Mylyn.

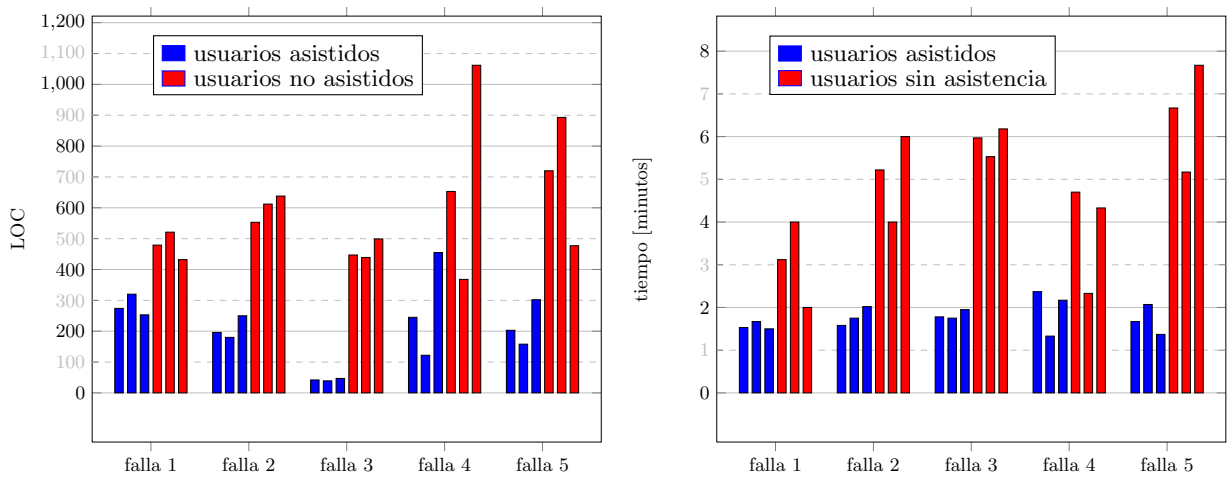
3.1. Resultados obtenidos

Luego de someter a la localización de fallas sobre los escenarios bajo la estructura de evaluación antes mencionada, se obtuvieron los resultados que muestran en la Figura 3. Para cada falla, se resalta el comportamiento de los desarrolladores que fueron asistidos en la localización respecto a aquellos sin asistencia.

Según la Figura (a), el asistente logra que se exploren aproximadamente el 30 % de las líneas de código necesarias para localizar la falla, si se compara con un enfoque de localización tradicional. Por consiguiente, el dominio de búsqueda sobre el cual se intenta localizar la falla se ve reducido, provocando consecuentemente una reducción del esfuerzo necesario.

En cuanto al tiempo necesario para la localización de fallas, la evaluación evidencia que se requiere, en promedio, menos del 50 % si el usuario se encuentra asistido por el enfoque (ver Figura (b)). A pesar de esta mejora en rendimiento, hay que destacar que las mediciones expuestas anteriormente se basan sobre un supuesto relevante: los artefactos requeridos en la *fase de configuración* ha sido construidos previamente. Sin embargo, no hay que obviar el tiempo y esfuerzo requerido para la construcción de dichos artefactos. En particular, la taxonomía puede omitirse y utilizar la otorgada por defecto, pero en cambio, el modelado de los escenarios mediante UCM no se puede obviar. Pese a esto, no se consideró en los experimentos realizados el tiempo que insume la *fase de configuración* y el mantenimiento que puede adicionar, ya que se asume que la localización de fallas asistida por este enfoque se encuentra inmersa en un proceso dirigido por la arquitectura.

[3] http://metrics.sourceforge.net/



(a) Líneas de código inspeccionadas (b) Tiempo de localización

Fig. 3: Métricas evaluadas durante los experimentos

3.2. Lecciones Aprendidas y Limitaciones

Fruto de la evaluación del enfoque llevada a cabo, los siguientes puntos resaltan como dignos de mencionar. En primer lugar cabe destacar la generalización de los resultados. Algunos aspectos que pueden afectar dicha generalización es le hecho de que los participantes tenían una determinada experiencia y conocimiento relacionados con el lenguaje de programación y los diagramas UCM, lo cual puede impactar en la eficiencia que posee enfoque a la hora de asistir.

Por otro lado, se encuentra la flexibilidad que posee la taxonomía. En particular, se ofrece una taxonomía de fallas SOA predefinida que resulta transparente al desarrollador. No obstante, se ofrece la posibilidad de modificar dicha taxonomía agregando fallas propias del dominio de la aplicación. Proveer esta característica conlleva a un esfuerzo adicional que impacta en el desempeño de la asistencia.

De la misma manera que se requiere obligatoriamente de una taxonomía de fallas SOA, también se necesita de los escenarios UCM para servir a la visualización de la falla de manera amigable al usuario. Pero la eficiencia y utilidad del enfoque varía en función del tiempo que implica construir estos escenarios. Aún así, cuanto mayor es el numero de fallas localizadas, la reutilización de los escenarios

y el tiempo de modelado provocan que el tiempo invertido en la instanciación del meta-modelo SOA sea amortizado.

4. Trabajos Relacionados

Desde una perspectiva histórica, se consideran las primeras técnicas de naturaleza puramente intuitiva. Un ejemplo es localizar los *bugs* de un sistema analizando su respectivo volcado de memoria. Otra es insertar sentencias de tipo *print* alrededor del código sospechoso, de manera que el desarrollador pueda observar el valor de las variables y el estado actual dentro del flujo de ejecución en el que se encuentra el programa. Ambas técnicas sufren de ineficiencia e intratabilidad debido a la enorme cantidad de datos que deben ser examinados, viéndose sujeta al tamaño de la aplicación.

Otras técnicas se basan en los estados de ejecución [9,21,12], es decir un conjunto de variables y sus valores, en un punto particular durante la ejecución del programa. Un enfoque general utilizado este concepto, consiste en modificar los valores de algunas variables para determinar cual de ellas es la causa de la ejecución fallida. El problema es el coste relativamente elevado, ya que pueden existir miles de estados en la ejecución de un programa.

Los trabajos centrados en el análisis de los contextos, proponen un enfoque basado en la clasificación y minería de datos para predecir los comportamientos del software, y así obtienen una generalización de las fallas que permite la detección de fallas similares [17]. Otros proponen la minería de reglas temporales con el fin de asistir al monitoreo, la verificación, y la comprensión del sistema [18], y algunos se centran en proveer un conjunto de métricas que permitan medir las propiedades de un determinado contexto proporcionando las bases para su comparación y asegurar de esta manera su calidad [14].

En este contexto, se puede decir que existe la necesidad de herramientas que permitan la localización de las fallas desde un punto mayor de abstracción, cuya representación se acerque más al modelo mental que el desarrollador posee del sistema, conservando una distancia de la implementación adhiriendo al carácter agnóstico del paradigma.

5. Conclusión

Las contribuciones principales realizadas por este trabajo residen principalmente en el desarrollo de un enfoque de localización de fallas dirigida por una taxonomía específica para SOA, ya que estos sistemas poseen muchas diferencias respecto a los sistemas clásicos, y las diferentes fallas que pueden ocurrir logran ser cubiertas taxonómicamente. La taxonomía de fallas SOA, en conjunción con la información sobre la ejecución del sistema, permite asistir en la localización de las fallas. Además, se provee al desarrollador y/o arquitecto de información suficiente para realizar un análisis en un nivel mayor de abstracción, que facilite la comunicación de la falla y sus causas hacia todos los interesados en el sistema, permitiendo no sólo la localización de fallas puntuales en el código sino que además fallas en el diseño del sistema SOA, elevando el nivel de abstracción que adhiera al carácter agnóstico de la implementación del paradigma orientado a servicios.

Mediante el análisis de diferentes escenarios para un caso de estudio, sujetos a pruebas experimentales, se tiene como conclusión que el prototipo contribuye a la reducción de los esfuerzos asociados a las actividades de localización de fallas en aplicaciones orientadas a servicios reales, aumentando la eficacia y precisión de la localización. De esta manera, se asiste al desarrollador en la localización de fallas en aplicaciones orientadas a servicios, centrandose en la información de ejecución y en la información provista por la taxonomía de fallas SOA.

Referencias

1. Daniel Amyot. Use Case Maps and UML for Complex Software-Driven Systems. pages 1–16.
2. Ali Arsanjani. Service-Oriented Modeling and Architecture: How to Identify, Specify, and Realize your Services. Technical report, IBM developerWorks, 2004.
3. Algirdas Avizienis, JC Laprie, and Brian Randell. Dependability and its threats: a taxonomy. *Building the Information Society*, (July 1834), 2004.
4. Luciano Baresi and Sam Guinea. An Introduction to Self-Healing Web Services. In *International Conference on Engineering of Complex Computer Systems*, 2005.
5. Lionel C Briand, Yvan Labiche, and Xuetao Liu. Using Machine Learning to Support Debugging with Tarantula. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, pages 137–146, Washington, DC, USA, 2007. IEEE Computer Society.

6. Stefan Bruning, Stephan Weissleder, Miroslaw Malek, Stefan Br, and Stephan Weiß leder. A Fault Taxonomy for Service-Oriented Architecture. *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, pages 367–368, 2007.
7. R.j.a. Buhr. Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering*, 24:1131–1155, 1998.
8. Usha Chhillar and Sucheta Bhasin. Establishing Relationship between Complexity and Faults for Object-Oriented Software Systems. 8(5):437–442, 2011.
9. Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.
10. John Erickson and Keng Siau. Web Services, Service-Oriented Computing, and Service-Oriented Architecture: Separating Hype from Reality. *Journal of Database Management*, 19(3):42–54, 2008.
11. Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.
12. Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 263–272, New York, NY, USA, 2005. ACM.
13. Sherif A Gurguis and Amir Zeid. Towards autonomic web services: achieving self-healing using web services. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.
14. Abdelwahab Hamou-Lhadj. *Techniques to simplify the analysis of execution traces for program comprehension*. PhD thesis, Ottawa, Ont., Canada, Canada, 2006.
15. K Ishikawa. *Introduction to quality control*. 3A Corporation, 1990.
16. Mariani Leonardo. A Fault Taxonomy for Component-Based Software. In *International Workshop on Test and Analysis of Component Based Systems satellite workshop at the European Joint Conferences on Theory and Practice of Software, 13 Aprile, Warsaw (Poland)*. Elsevier, 2003.
17. David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 557–566, New York, NY, USA, 2009. ACM.
18. David Lo, Siau-Cheng Khoo, and Chao Liu. Mining past-time temporal rules from execution traces. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08, pages 50–56, New York, NY, USA, 2008. ACM.
19. Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P Reiss. Automated fault localization using potential invariants. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, pages 273–276, Ghent, Belgium, 2003.
20. Guoquan Wu, Jun Wei, and Tao Huang. Towards self-healing web services composition. In *Proceedings of the First Asia-Pacific Symposium on Internetware*, Internetware '09, pages 15:1—-15:5, New York, NY, USA, 2009. ACM.
21. Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

A. Meta-modelo SOA

El estilo arquitectónico que define SOA describe un conjunto de patrones y guías para crear soluciones que se encuentren débilmente acopladas, servicios alineados al negocio que debido a la separación que concierne a la descripción, la implementación y el ligue entre servicios, proveen una flexibilidad sin precedentes.

El modelo conceptual de una Arquitectura Orientada a Servicios esta basado en un estilo arquitectónico que define un modelo de interacción entre tres partes principales:

El Proveedor del servicio. Es la entidad que publica una descripción del servicio y provee su implementación.

El Consumidor del servicio. La entidad que tanto puede usar un identificador uniforme de recurso (URI) para conectar con la descripción del servicio directamente, o bien puede buscar la descripción del servicio en un registro de servicios (por ej. UDDI) y luego conectarse e invocar el servicio encontrado.

Un Registro de servicios. Siendo el responsable de proveer y mantener el registro de servicios, pese a que hoy en día el uso de registros públicos de servicios no es muy popular.

El meta-modelo que describe a SOA [2,10] puede traducirse con la notación propuesta, es decir, se pueden especificar los componentes de la arquitectura SOA desde una vista tanto estructural como comportamental, mediante Diagramas de Componentes y UCM en las Secciones A.1 y A.2 respectivamente.

A.1. Componentes del meta-modelo

Según lo descrito anteriormente existen tres componentes principales que conforman el esqueleto SOA, los cuales pueden ser especificados en el diagrama de componentes de la Figura 4, formalizando las interfaces requeridas como expuestas de cada uno de ellos, así como la funcionalidad de las cuales son responsables.

Proveedor. El componente que representa el Proveedor del servicio es quien ofrece e implementa el servicio, eventualmente publicándolo en un Registro. Por ello posee una interfaz requerida por un Registro en caso de que se quiera publicar el servicio y

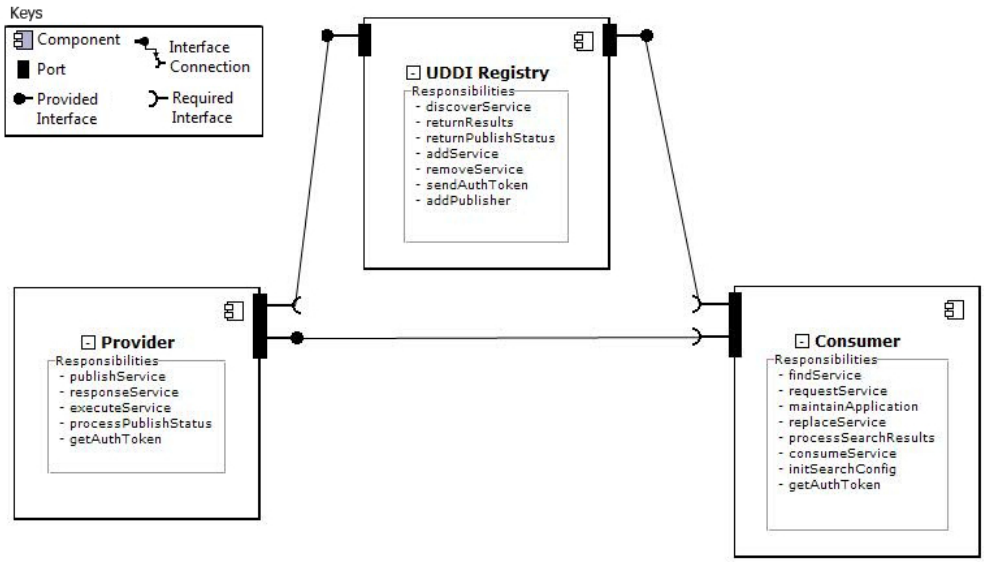


Figura 4: Componentes del Meta-modelo SOA

una interfaz expuesta que tiene como objeto ofrecer el servicio que implementa. Cuenta con las responsabilidades de publicar un servicio (*publishService*), solicitar autorización (*getAuthToken*), y ejecutar el servicio (*excecuteService*) entre otras.

Consumidor. Este componente es el encargado de hacer uso de los servicios, consumiendo de uno o más proveedores a los cuales accede mediante un vínculo directo si conoce a su proveedor y al URI del servicio, o bien desconocidos a los cuales accede previa búsqueda en un Registro. El consumidor posee dos interfaces requeridas, una de ellas permite obtener los resultados de búsqueda de servicios desde un Registro, y la otra consume el servicio ofrecido por el proveedor una vez que se lo conoce, ya sea mediante una conexión directa o previa búsqueda. Las responsabilidades que involucra son: buscar (*findService*), solicitar la ejecución (*requestService*) y reemplazar servicios (*replaceService*), entre otros.

Registro. Representa la entidad encargada de mantener, indexar y permitir recuperar diferentes servicios que previamente han sido publicados. Por ello provee dos interfaces, una de ellas al consumidor con el objetivo de proveer el servicio de búsqueda

de servicios, y la otra al proveedor de servicios, permitiendo la posibilidad de darsede alta en él. Las responsabilidades que contiene son las de buscar un servicio (*discoverService*), retornar resultados de búsqueda (*returnResults*), y agregar nuevos servicios (*addService).*

A.2. Escenarios UCM del meta-modelo

Utilizando los componentes previamente descritos en la Subsección A.1, y basándonos en los comportamientos básicos que poseen las aplicaciones orientadas a servicios, podemos especificar los escenarios funcionales a nivel arquitectónico mediante la notación de UCM, expresando los flujos de ejecución característicos de SOA. Cada uno de estos comportamientos se desglosa en tres escenarios relevantes: la publicación de un Servicio en el Registro, la búsqueda de un Servicio en el Registro y el enlace directo entre el Consumidor y el Proveedor donde el segundo ejecuta el servicio y el primero procesa el resultado obtenido.

Escenario de publicación de Servicio. Este escenario describe la situación que sucede cuando un servicio se encuentra implementado y se decide publicarlo en un Registro para que otras aplicaciones o Consumidores puedan conocer de su existencia, y posteriormente hacer uso de él. Como se observa en la Figura 5, se cuentan con dos instancias de componentes: el Proveedor del Servicio y el Registro. A modo de ejemplo, se han instanciado con *"aServiceProvider"* y *"jUDDIv3"* haciendo referencia a que el Registro es una implementación de tecnología jUDDIv3 en particular.

El flujo comienza desde el Proveedor quien solicita un token de autorización mediante la responsabilidad *getAuthToken*, a la cual responde el Registro enviando la cadena de caracteres mediante *sendAuthToken*, proveyendo seguridad durante la sesión. Además el Registro da de alta la entidad que realiza la publicación mediante *addPublisher*. Seguido de esto el flujo retorna al Proveedor quien prepara la información requerida para realizar la publicación, como el nombre del servicio, su descripción, y el nombre del socio de negocio. Esta información es enviada al Registro, quien la utiliza para agregar el servicio a su base del datos y retorna el estado de publicación,

que sería un informe manifestando si el servicio publicado ya existía, si fue publicado satisfactoriamente o cualquier otro tipo de error. Finalmente el flujo retorna al Proveedor, quien procesa este resultado y realiza las acciones adicionales correspondientes.

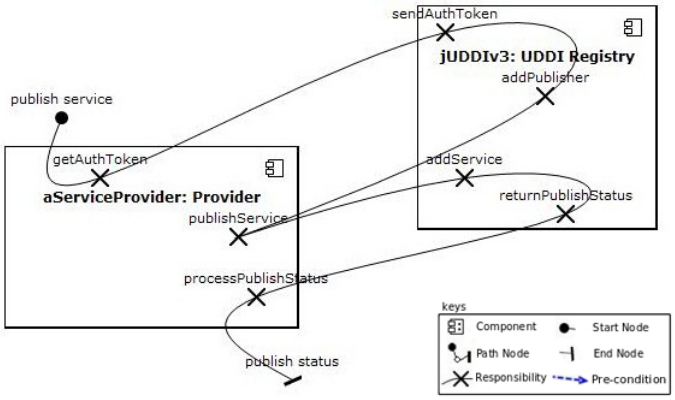


Figura 5: Escenario de Publicación de Servicio en el Meta-modelo.

Escenario de búsqueda de Servicio. Otro escenario relevante dentro del meta-modelo de SOA es la búsqueda de un determinado servicio en el Registro y se muestra en el diagrama UCM de la Figura 6. El flujo comienza desde el Consumidor quien inicializa la búsqueda con los parámetros necesarios, como por ejemplo la dirección URI donde se encuentra el Registro. De la misma manera que durante la publicación, ahora el Consumidor es quien solicita un token de autorización que mantenga segura la sesión. El registro envía dicho token al consumidor y este luego elabora la consulta correspondiente que será enviada al Registro. Este procesa la consulta y recupera aquellos servicios que coincidan con la información solicitada por medio de la consulta. Finalmente el Registro retorna los resultados encontrados para ser posteriormente utilizados por el Consumidor.

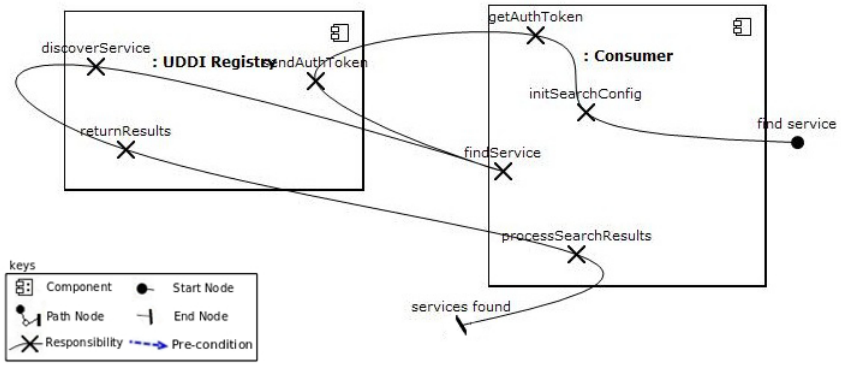


Figura 6: Escenario de Búsqueda de Servicio en el Meta-modelo.

Escenario de solicitud y ejecución del servicio. En el caso que el Consumidor conozca quién es el Proveedor del servicio que necesita utilizar, pasa a realizar un ligue que le permita enviar la información requerida para ejecutar el servicio dentro del Proveedor. El Consumidor puede conocer de ante mano donde se encuentra el Proveedor (posee el URI del servicio), o bien puede buscar en un Registro mediante el nombre o descripción del servicio con las características deseadas. El flujo es simple, ya que consta de una única vinculación, donde el consumidor realiza un pedido de Servicio, incluyendo los argumentos necesarios, y el Proveedor responde con los resultados de ejecución del servicio o bien informando del error si es que lo hay.

A.3. Mapeo del meta-modelo a la implementación

Una vez que el meta-modelo SOA se encuentra especificado tanto en cuanto a los componentes involucrados como a los escenarios por los que atraviesa una Arquitectura Orientada a Servicios tipo, se realiza una correspondencia entre cada responsabilidad involucrada y su implementación. En particular, para este trabajo el mapeo se realiza hacia una implementación Orientada a Objetos, específicamente métodos y clases Java. En este caso se utilizan tecnologías específicas

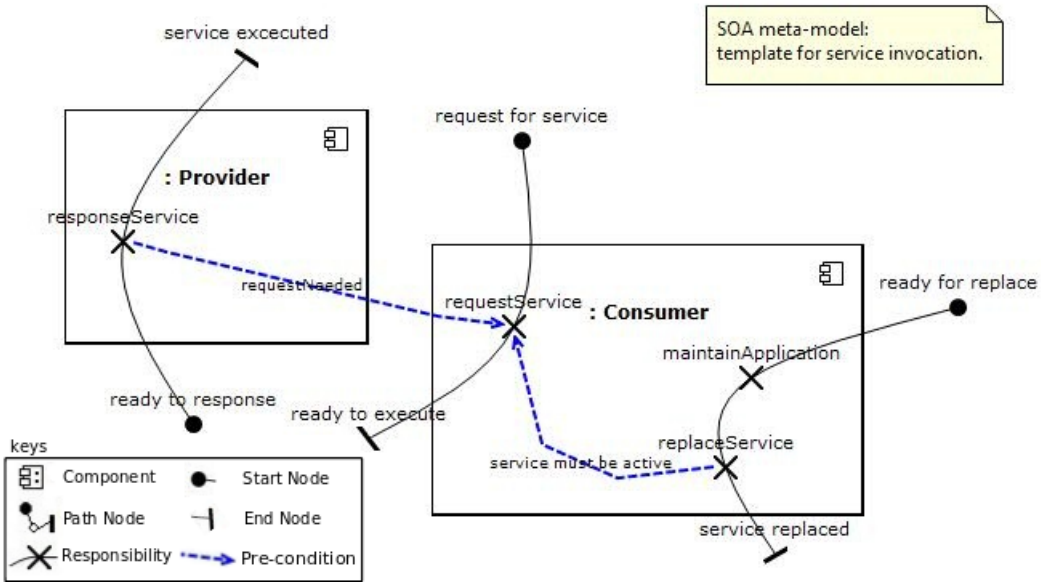


Figura 7: Escenario de Solicitud y Ejecución del Servicio en el Metamodelo.

que dan soporte a esta implementación como lo son jUDDIv3[4] y Apache Axis[5].

Apache Axis 2 es una plataforma Open Source, que posee implementaciones tanto en Java como en C de un servidor SOAP, y varias utilidades y APIs para proveer soporte en la creación, despliegue y ejecución de Aplicaciones que consuman y/o provean Servicios Web. La utilización de esta plataforma disminuye en gran medida el esfuerzo requerido por los desarrolladores, permitiendo crear aplicaciones basadas en SOA interoperables y distribuidas fácilmente. Axis esta desarrollado bajo el auspicio de Apache Software Fundation.

Por otro lado, jUDDI es también una plataforma Open Source que implementa la especificación del Registro para Web Services UDDI (Universal Description, Discovery, and Integration - UDDIv3) y también esta desarrollado bajo el auspicio de Apache Software Fundation.

[4] http://juddi.apache.org/
[5] http://axis.apache.org/axis2/java/core/

Con estas dos tecnologías se realiza el mapeo del modelo arquitectónico de SOA. Consiste en asociar a cada una de las responsabilidades en el meta-modelo a las clases y métodos que lo materializan utilizando las bibliotecas específicas. En caso de utilizar otras, el arquitecto debe realizar éste mapeo nuevamente a la nueva implementación.

Por cuestiones de legibilidad, se muestra a continuación un ejemplo. Cuando un Proveedor de servicio desea publicarlo en el Registro, primero se autentica, y a partir de ese momento ambos conocen el token de autorización. Este token es materializado por un objeto provisto en la API de jUDDIv3 que instancia la clase org.uddi.appi_v3.AuthToken. Gráficamente podemos visualizar esta situación en la Figura 8.

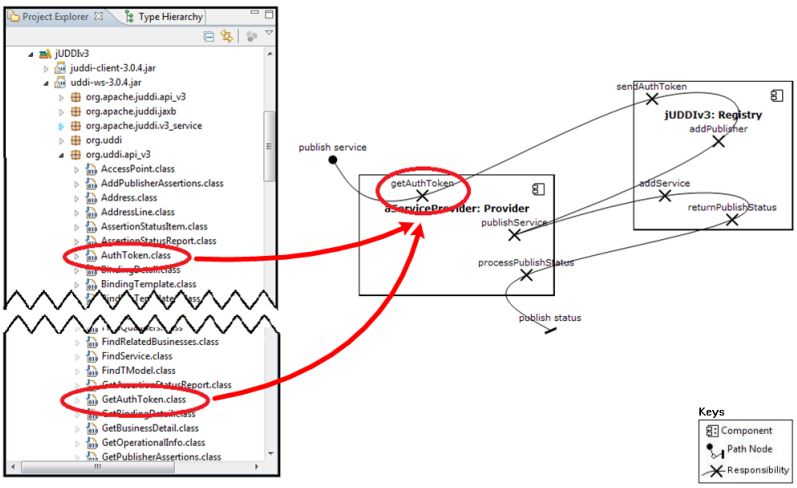


Figura 8: Correspondencia entre una responsabilidad del meta-modelo y las clases que la materializan

B. Ejemplo instanciado

Para el ejemplo del sistema de gestión de jugadores de fútbol, el arquitecto deberá crear los componentes *FootballPlayerProvider* y *FootballPlayerConsumer* que se correspondan al proveedor del servicio y al consumidor respectivamente. El diagrama de componentes donde se detallan ambos como extensiones del meta-modelo SOA se muestra en la Figura 9. En éste se resaltan con color azul los nuevos componentes específicos del ejemplo que extienden de los existentes en la plantilla (meta-modelo).

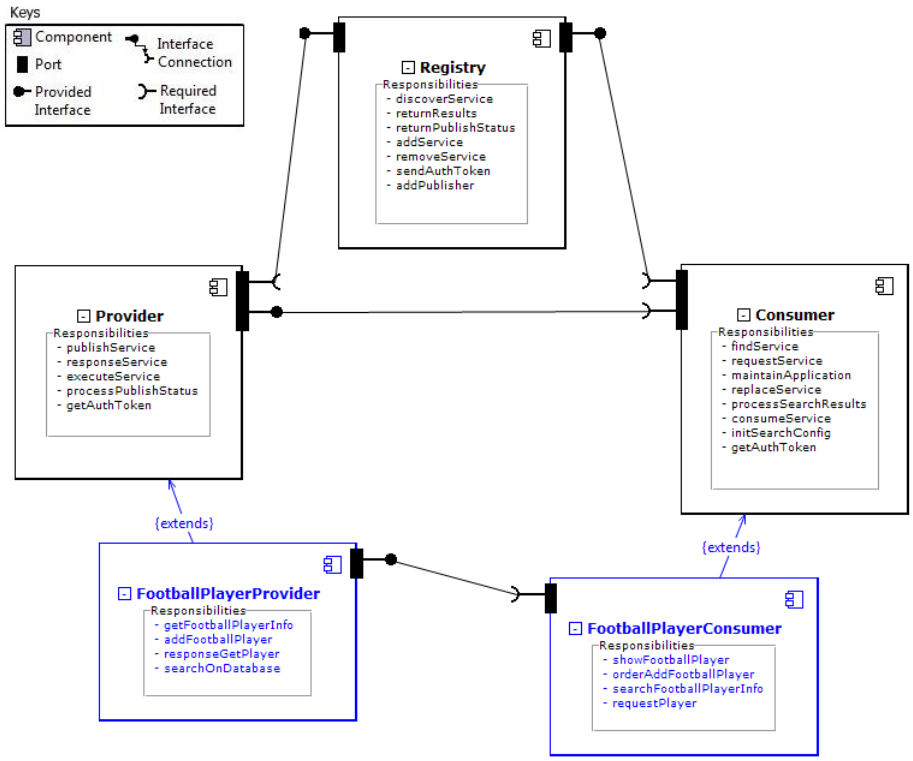


Figura 9: Componentes del ejemplo sobre jugadores de fútbol.

Adicionalmente deberá realizar un diagrama Use Case Map por cada escenario funcional. Para continuar con el ejemplo, se muestra en la Figura 10 el escenario de búsqueda de un jugador de fútbol determinado.

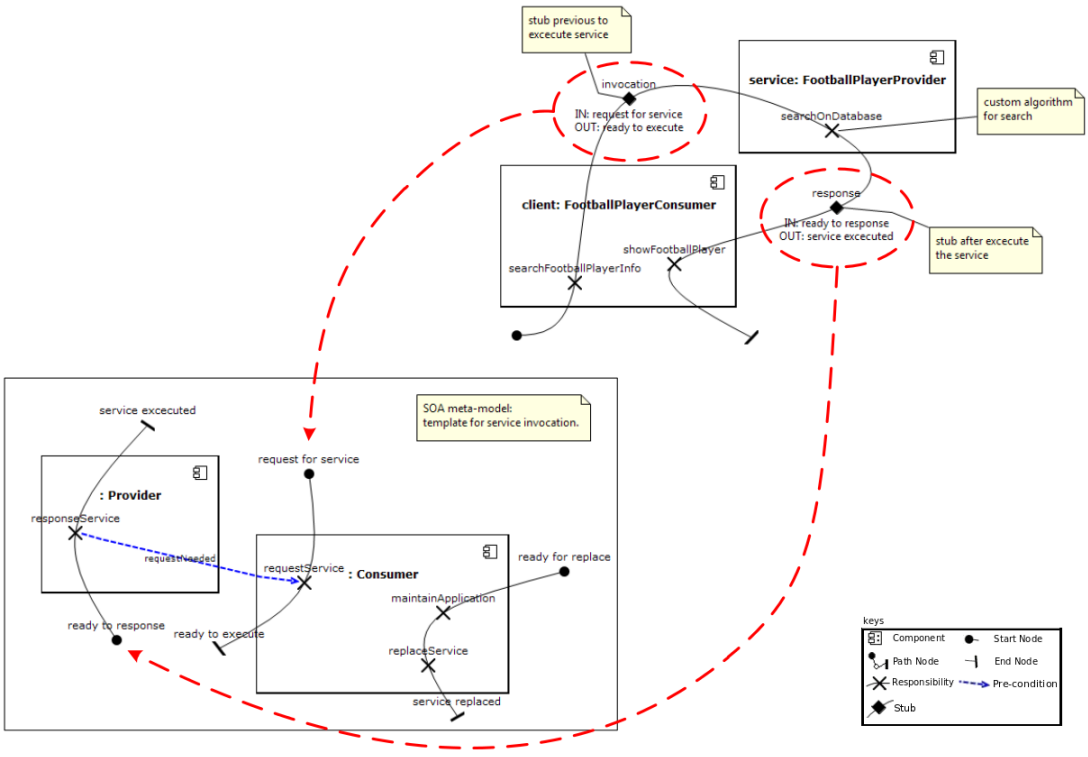


Figura 10: Use Case Map para el escenario de búsqueda de un jugador de fútbol.