View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Servicio de Difusión de la Creación Intelectual

Uso de Ontologías para mapear una Arquitectura de Software con su Implementación

H. C. Vázquez[1,2], J. A. Díaz Pace[1,2], and C. Marcos[1,3]

[1] ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (249) 4439682.
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).
[3] CIC, Committee for Scientific Research B1900AYB, La Plata, Argentina.

Resumen La arquitectura de software de un sistema es un activo importante para una organización que desarrolla software. Para maximizar los beneficios que provee una arquitectura, ésta debe estar en correspondencia con la implementación del sistema. En muchos proyectos existe cierta documentación de la arquitectura, pero sin embargo, la *información de mapeos* entre los elementos de dicha arquitectura y su implementación en código es escasa o inexistente. Este problema trae aparejadas dificultades de entendimiento de los elementos de código en relación a la arquitectura originalmente diseñada, lo que repercute negativamente sobre el aseguramiento de la calidad y los esfuerzos de mantenimiento del sistema. Si bien la provisión manual de estos mapeos es factible, es una tarea compleja y proclive a errores, particularmente a medida que la implementación del sistema evoluciona en el tiempo. En este contexto, las técnicas de alineación de ontologías se presentan como una alternativa para producir mapeos en forma automática. Por esta razón, el presente trabajo propone un enfoque automatizado y basado en ontologías para la generación de mapeos entre la arquitectura de un sistema y su implementación.

1. Introducción

La arquitectura de un sistema de software [3] se refiere a la organización de alto nivel del sistema, que comprende elementos de software, las propiedades externamente visibles de esos elementos, y las relaciones entre ellos. Una arquitectura de software [3] proporciona un modelo que hace explícitas las principales decisiones de diseño respecto a la satisfacción de atributos de calidad (por ej., performance, disponibilidad, modificabilidad, entre otros). A medida que un sistema evoluciona, comienzan a existir discrepancias entre la arquitectura "según se documentó" y su implementación "en código" [10]. La acumulación de cambios (típicamente, en la implementación) tiende a "erosionar" la arquitectura inicial. Este problema impacta negativamente sobre los esfuerzos de mantenimiento y el aseguramiento de la calidad del sistema.

Una práctica que permite mantener la arquitectura alineada (y consistente) con su implementación es la conformidad arquitectural [16]. Esta práctica se refiere a verificar periódicamente las relaciones entre los elementos arquitectónicos y sus contrapartes en la implementación, previniendo posibles violaciones de las reglas de arquitectura y reparando aquellas que puedan existir. En la actualidad, existen varias herramientas que ayudan al arquitecto a desarrollar con la conformidad arquitectural (por ej., Archium [9], ArchJava [1]). Para esto, el prerrequisito es contar con mapeos entre los elementos

de la arquitectura (por ej., componentes, responsabilidades, escenarios) y los elementos de la implementación (por ej., paquetes, clases, métodos). Es habitual encontrar descripciones de la arquitectura prevista del sistema, pero generalmente los mapeos entre dicha arquitectura y el código son escasos o inexistentes. Esto no permite capitalizar en los beneficios que provee la arquitectura. Si bien es factible que un arquitecto/desarrollador defina en forma manual los mapeos de la arquitectura a una versión dada del sistema, esta tarea suele ser compleja y requiere un análisis detallado de la implementación. Este problema se acentúa a medida que la implementación del sistema evoluciona en el tiempo.

En este contexto, un enfoque alternativo para producir mapeos en forma automática son las *técnicas de alineación de ontologías* [6]. Una ontología es una formulación explícita de un esquema conceptual, que permite la abstracción de entidades y sus relaciones dentro de una estructura determinada [8]. Si se considera la arquitectura de software como un representación conceptual del sistema, tanto la semántica de la arquitectura como el código correspondiente pueden ser utilizados para crear ontologías que representen sus conceptos y relaciones más importantes. En este trabajo se propone la utilización de dos ontologías, una para representar la arquitectura y otra para su implementación. Estas ontologías pueden ser luego alineadas de forma automática para establecer correspondencias entre sus entidades. Dichas correspondencias proveen los mapeos entre elementos de la arquitectura y su implementación.

El resto del trabajo se estructura de la siguiente manera. En la Sección 2 se presentan los aspectos más importantes del enfoque propuesto para la generación de mapeos. La Sección 3 discute algunos resultados de la evaluación del enfoque con casos de estudio. En la Sección 4, se analizan algunos trabajos relacionados. Finalmente, la Sección 5 presenta las conclusiones del trabajo y menciona trabajos futuros.

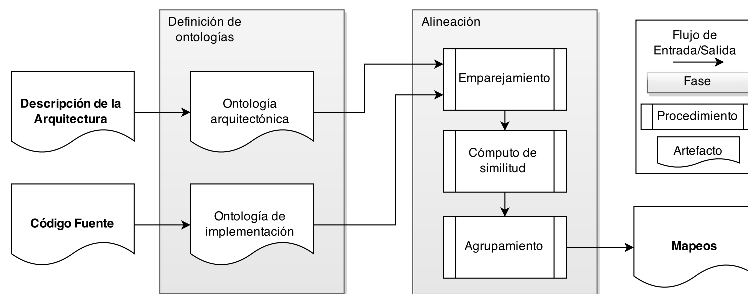

Figura 1: Un enfoque para establecer mapeos entre elementos de la arquitectura y su implementación.

2. Enfoque propuesto

La ausencia de mapeos entre la arquitectura de un sistema y su implementación dificulta: el entendimiento del código en relación a su diseño arquitectónico original, el mantenimiento evolutivo del sistema y la práctica de chequeo de conformidad arquitec-

tural. Por esta razón, en este trabajo se propone un enfoque para la generación automática de mapeos que permita identificar correspondencias entre elementos de la arquitectura y su implementación (Figura 1). A partir de una descripción de la arquitectura y del código fuente, se definen dos ontologías, denominadas "ontología arquitectónica" y "ontología de la implementación", respectivamente. Luego, estas dos ontologías son tomadas como entradas por el proceso de alineación, el cual tiene como objetivo encontrar las correspondencias entre ambas ontologías. Dicho proceso está compuesto por tres actividades: emparejamiento, cómputo de similitud y agrupamiento. En el emparejamiento, se generan todas las posibles parejas entre las entidades de las dos ontologías. A partir de este emparejamiento, se realiza el cómputo de similitud de cada pareja de entidades basándose en sus características. Luego del cómputo de similitud se realiza el agrupamiento de las parejas de entidades de acuerdo al grado de similitud presentado entre estas. Finalmente, se selecciona como resultado de la alineación el grupo que posee parejas de mayor similitud, las cuales se presentan como salida al usuario.

2.1. Definición de las ontologías

Una ontología es una especificación explícita de una conceptualización [8]. Algunas de las características más representativas de las ontologías que se adecuan a nuestro problema son: la capacidad de adaptarse a distintos niveles de abstracción, y la posibilidad de realizar correspondencias de ontologías estableciendo relaciones entre los elementos de una o más ontologías, para establecer conexiones, especializaciones, y generalizaciones, entre otras. En este trabajo, las ontologías son definidas a través del lenguaje OWL (*Ontology Web Lenguage* [14]). OWL tiene como objetivo proporcionar un lenguaje que puede ser utilizado para describir las clases y las relaciones entre ellas, que son inherentes a documentos y aplicaciones Web. Sin embargo, es un lenguaje que también puede ser utilizado para representar otros tipos de descripciones como, por ejemplo, la arquitectura de software y su implementación.

A continuación, se presentan las definiciones de la ontología arquitectónica y la ontología de la implementación. Notar que la generación de ambas ontologías se realiza en forma automática.

Ontología arquitectónica Existen diversas formas de representar la arquitectura de un sistema y la relación entre sus componentes, este trabajo se basa en una vista parcial de la arquitectura denominada *blueprint* [11]. El *blueprint* es una vista de los componentes de alto nivel de un sistema y de sus interfaces hacia otros componentes. Un componente en esta vista arquitectónica representa una porción de código que tiene asignadas responsabilidades (o *concerns*) del sistema. Estos componentes son vistos como "cajas negras" y su interacción con el resto del sistema se realiza solo mediante sus interfaces. Las interfaces pueden ser provistas o requeridas, determinando así una forma de dependencia entre los componentes.

La representación *blueprint* se traduce a una ontología utilizando el lenguaje OWL (Figura 2). Los componentes se representan como clases *OWL:Class*, la dependencia entre estos dada por las interfaces se traduce como propiedades *OWL:ObjectProperty*, y los *concerns* simplemente se traducen como *strings* utilizando *OWL:DatatypeProperty*. Por ejemplo, en la figura 2, los componentes 1 y 2 son traducidos a dos clases OWL homónimas, la "interface1to2" a una relación *ObjectProperty*, y los *concerns* a dos *strings.*

Ontología de la implementación La implementación de un sistema se lleva a cabo mediante la producción de código. Para poder realizar los mapeos, este trabajo se basa en código escrito en el lenguaje Java. El código Java se traduce a una ontología OWL (Figura 2). En esta traducción, las clases Java se representan como *OWL:Class*, las dependencias entre clases vía invocaciones a métodos se representan como propiedades de objetos *OWL:ObjectProperty*. Adicionalmente, también pueden informarse *concerns* de la misma forma que en la ontología anterior utilizando *OWL:DatatypeProperty*. Esto permite que un *concern* pueda ser asociado a una clase cuando exista información de trazabilidad. Por ejemplo, en la figura 2, las clases Java 1 y 2 son traducidas a dos clases OWL homónimas, la invocación a "method2" como una relación *ObjectProperty*, y en el caso de la clase 1, se informa el concern1 como *string*.

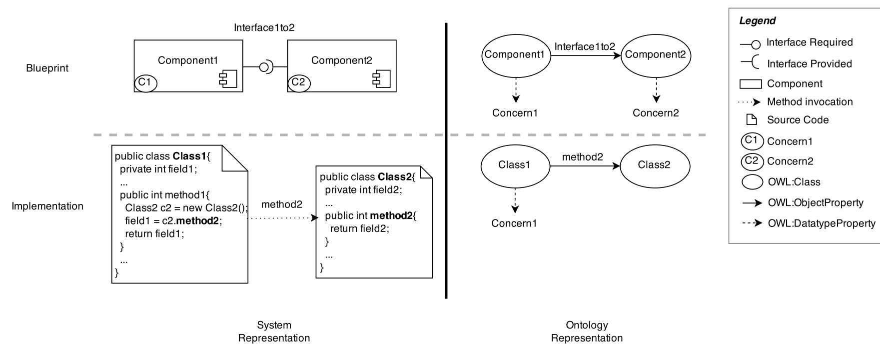

Figura 2: Definición de las ontologías a partir de la representación del sistema.

2.2. Alineación de las ontologías

A partir de las ontologías anteriores, es posible trazar una correspondencia entre los conceptos y relaciones de las mismas, lo que se conoce como *alineación de ontologías* [6]. La Figura3 muestra un pequeño ejemplo de arquitectura y su implementación. En la parte inferior se muestra la descripción (parcial) de un sistema dada por su *blueprint* y su implementación (vista como clases e invocaciones a métodos); y en la parte superior se muestra la representación ontológica de ambas.

Emparejamiento El emparejamiento de entidades de las ontologías se refiere al producto cartesiano entre los componentes (de la arquitectura) y las clases (de la implementación). Para el ejemplo de la Figura 3, el emparejamiento de entidades se observa en el Cuadro 1, donde los componentes GUI_Elements, Business_Rules y Data_Manager, son cruzados con las clases GUI_Adapter, BusinessManager, BusinessRule, BusinessDataManager y DataObject. Para problemas complejos (en términos de tamaño de las ontologías), por cuestiones de performance, podría pensarse en una forma de limitar los emparejamientos con información adicional, no obstante, ese problema no será abordado en este trabajo.

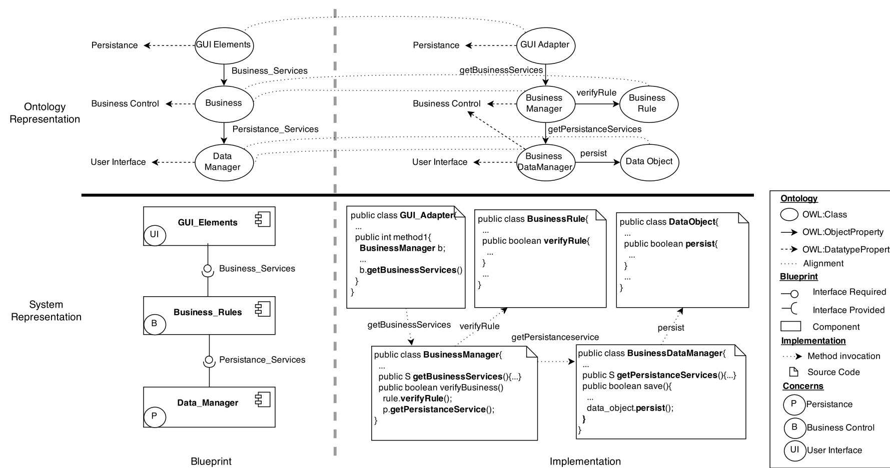

Figura 3: Ejemplo de generación de ontologías a partir de la representación del sistema.

Cómputo de similitud A cada par de entidades resultante de la fase anterior se le aplican funciones de similitud que lo evalúan de acuerdo a distintas propiedades. Este enfoque utiliza tres funciones de similitud (lingüística, de *concerns*, y estructural) para cubrir todos los aspectos en lo que dos entidades puedan ser comparadas. Cada función de similitud produce, para cada pareja de entidades, un valor entre 0 (baja similitud) y 1 (alta similitud) generando así una matriz de similitud.. El Cuadro 1 resume las tres matrices de similitud en una única matriz para los emparejamientos del ejemplo de la figura 3.

	GUI_Adapter			BusinessManager			BusinessRule			BusinessDataManager			DataObject		
	L	C	E	L	C	E	L	C	E	L	C	E	L	C	E
GUI_Elements	0.5	1,00	0.84	0.25	0,00	0.52	0.28	0,00	0,00	0.22	0,00	0,00	0.06	0,00	0,00
Business_Rules	0.0	0,00	0,00	0.63	1,00	1,00	1,00	1,00	0.38	0.6	0.66	0.49	0.23	0,00	0,00
Data_Manager	0.0	0,00	0,00	0.71	0,00	0,00	0.28	0,00	0.37	0.9	0.66	0.37	0.5	1,00	0.42

Cuadro 1: Similitud lingüística (L), de *concerns* (C) y estructural (E) para los elementos de la Figura 3.

Similitud Lingüística Se basa en la comparación de los nombres de las entidades de las ontologías. Los nombres son divididos en palabras de acuerdo a espacios, guiones, y divisiones de mayúsculas, entre otros criterios. Una vez divididas las palabras, se obtiene una oración que es utilizada como entrada de la función de similitud. En particular, para el cálculo de similitud lingüística se evaluó el uso de tres técnicas: Greedy [19], Optima [19] y el método Corley Mihelcea [4], empleando para ello el API de similitud semántica SEMILAR [20]. Por ejemplo, en el Cuadro 1, puede observarse en la columna L el valor de similitud lingüística entre las entidades usando el método Corley Mihelcea.

Similitud de Concerns Un *concern* de software es cualquier aspecto que impacte sobre la implementación de un sistema de software [22], generalmente impactando sobre varios artefactos de software. Ejemplos típicos de *concerns* pueden ser aspectos de persistencia, seguridad, o performance. En nuestra representación, los *concerns* que afectan ciertos elementos de la arquitectura, también pueden a afectar a los elementos de código que materializan dichos componentes. La ecuación 1 muestra el cálculo de similitud de *concerns* entre dos entidades E1 y E2, calculada como la suma de los *concerns* de E1 y E2, sobre la suma de los *concern* de E1 que se encuentran en E2 más la suma de los *concerns* de E2 que se encuentran en E1. Esta información es representada en la columna C del Cuadro 1.

$$Similitud(E1,E2) = \frac{\sum concerns_{E1} + \sum concerns_{E2}}{\sum concerns_{E1enE2} + \sum concerns_{E2enE1}} \quad (1)$$

Similitud Estructural Este tipo de similitud proviene de considerar las ontologías como grafos dirigidos para poder analizar su estructura y determinar la correspondencia entre sus nodos. En base a esta característica, se utiliza el algoritmo de matching de grafos Similarity Flooding (SFA) [15]. El SFA toma dos grafos como entrada y produce como salida un mapeo entre los nodos de ambos grafos. Cabe destacar que el SFA utiliza como entrada un mapeo inicial (o semilla), a fin de reducir su espacio de búsqueda. En este enfoque, para evitar la intervención del usuario, este mapeo inicial es resuelto utilizando las funciones de similitud lingüística y de *concerns*. Es decir, primeramente se buscan los mapeos para los cuales las funciones de similitud anteriores hayan producido un alto valor de similitud (muy cercano a 1), y se utilizan estos como semilla del SFA. Este proceso es descripto por el pseudocódigo 1. En la Cuadro 1, se puede observar en la columna E, el valor de similitud estructural usando el algoritmo SFA.

Algoritmo 1 Pseudocódigo del proceso de similitud estructural.

1. G1 = Graph(Ontology1);
2. G2 = Graph(Ontology2);
3. initialMap = maxSimilarityMappings(G1, G2);
4. similarityMatrix = SFA(G1, G2, initialMap);

Agrupamiento A partir de las matrices, se obtienen instancias de los pares de entidades y sus valores de similitud. Finalmente, estas instancias son agrupadas mediante la utilización de una técnica de *clustering* [18] con el fin de reunir los pares de entidades de acuerdo a la similitud de sus características. Particularmente, se utilizó la técnica de *clustering* K-Means [2]. K-Means tiene como objetivo la partición de un conjunto de n observaciones en k grupos, en el que cada observación pertenece al grupo más cercano a la media. Para este trabajo, el resultado de la alineación es el clúster con la media más cercana a 1, que es presentado en última instancia al usuario, como mapeos entre componentes y clases. En la Figura 3, la línea punteada entre las entidades expone la alineación final, por ejemplo entre el componente Data_Manager y las clases BusinessDataManager y DataObject

3. Evaluación

En el área de alineación de ontologías, se utilizan generalmente los criterios de *precision* y *recall* evaluar la performance del sistema de alineamiento [5]. Estas medidas son definidas por las ecuaciones siguientes:

$$Precision = \frac{(givenAlignment \cap correctAlignment)}{givenAlignment} \tag{2}$$

$$Recall = \frac{(givenAlignment \cap correctAlignment)}{correctAlignment} \tag{3}$$

Se denomina *precision* a la fracción de instancias recuperadas que son relevantes, mientras que *recall* es la fracción de instancias relevantes que han sido recuperadas. Además de estas dos medidas, se utilizó *F-Measure* como una medida armónica de *precision* y *recall*. Esta medida está definida por la siguiente ecuación:

$$F-Measure = \frac{2 * Precision * Recall}{Precision + Recall} \tag{4}$$

F-measure es utilizada para evaluar qué tan lejos está la solución de la utilidad teórica, y también para observar qué tan balanceadas están las medidas de *precision* y *recall*.

Descripción	HWS	MM
Líneas de código (LOC)	8697	3015
Componentes	7	25
Interfaces	14	70
Clases	135	51
Métodos	1090	251

Cuadro 2: Descripción de los proyectos HWS y MM

Se utilizaron dos casos de estudio para validar el enfoque propuesto: i) el sistema Mobile Media (MM) [24] que es una SPL (Software Product Line) para aplicaciones móviles, y ii) el sistema Health Watcher (HWS) [13] que es un sistema para recoger y gestionar quejas y notificaciones relacionadas con la salud pública. Los mapeos de referencia para MM y HWS fueron provistos por expertos en forma manual, a fin de realizar comparaciones contra los mapeos obtenidos automáticamente mediante nuestro enfoque. En la Cuadro 2 se provee una descripción breve de ambos proyectos, tanto a nivel de arquitectura como de su implementación.

Se experimentó con el enfoque en los dos casos de estudio, y se realizaron pruebas variando el valor K (cantidad de clusters) y la técnica de similitud lingüística, obteniendo los resultados mostrados en la Cuadro 3. Estos resultados evidencian alrededor de K=20 un valor de *f-measure* superior al conseguido con otros K, lo que significa un mayor balance entre *precision* y *recall*. La técnica de similitud lingüística con la que el proceso de alineación arrojó resultados más precisos fue *Greedy*, con cerca del 95 % para K=20, mientras que el *recall* se mantuvo cerca del 50 %. Como era de esperar, para valores más altos de K=20 la medida *precision* aumenta, pero *recall* disminuye como puede observarse en la Figura 4. Esto se debe a que el aumento en el valor de K disminuye la cantidad promedio de instancias por cada clúster, lo que genera un *trade-*

	MM									HWS								
	CorleyMihalcea			Greedy			Optimum			CorleyMihalcea			Greedy			Optimum		
K	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
2	0,07	0,68	0,13	0,08	0,68	0,15	0,09	0,68	0,16	0,22	0,98	0,37	0,22	0,98	0,37	0,22	0,98	0,37
4	0,22	0,37	0,27	0,25	0,48	0,33	0,34	0,59	0,43	0,85	0,58	0,69	0,77	0,55	0,64	0,85	0,53	0,66
6	0,23	0,21	0,22	0,51	0,41	0,45	0,53	0,39	0,45	0,85	0,58	0,69	0,89	0,40	0,55	0,85	0,53	0,66
8	0,38	0,35	0,36	0,62	0,39	0,48	0,60	0,39	0,47	0,85	0,58	0,69	0,96	0,39	0,56	0,89	0,41	0,56
10	0,79	0,31	0,44	0,62	0,39	0,48	0,66	0,38	0,48	0,89	0,39	0,54	0,96	0,40	0,56	0,89	0,40	0,55
12	0,79	0,31	0,44	0,71	0,35	0,47	0,69	0,34	0,45	0,94	0,40	0,56	0,96	0,40	0,56	0,96	0,40	0,56
14	0,41	0,34	0,37	0,69	0,35	0,47	0,74	0,32	0,45	0,94	0,39	0,55	0,96	0,40	0,56	0,96	0,40	0,56
16	0,88	0,30	0,44	0,71	0,35	0,47	0,74	0,32	0,45	0,94	0,39	0,55	0,96	0,40	0,56	0,96	0,40	0,56
18	0,88	0,30	0,44	0,73	0,34	0,46	0,73	0,34	0,46	0,94	0,40	0,56	0,96	0,40	0,56	0,96	0,40	0,56
20	**0,88**	**0,30**	**0,44**	**0,95**	**0,30**	**0,45**	**0,88**	**0,31**	**0,46**	**0,94**	**0,40**	**0,56**	**0,96**	**0,40**	**0,56**	**0,96**	**0,40**	**0,56**
22	0,87	0,28	0,43	0,88	0,31	0,46	0,88	0,31	0,46	0,94	0,40	0,56	0,96	0,40	0,56	0,96	0,40	0,56
24	0,91	0,28	0,43	0,95	0,28	0,43	0,88	0,30	0,44	0,94	0,40	0,56	0,96	0,40	0,56	1,00	0,38	0,55
26	0,86	0,27	0,41	0,95	0,28	0,43	0,88	0,31	0,46	1,00	0,17	0,29	0,96	0,39	0,56	1,00	0,38	0,55
28	0,92	0,17	0,29	0,95	0,28	0,43	0,88	0,31	0,46	1,00	0,17	0,29	0,96	0,39	0,56	1,00	0,38	0,55
30	0,92	0,17	0,29	0,95	0,28	0,43	0,88	0,30	0,44	1,00	0,17	0,29	0,96	0,39	0,56	1,00	0,38	0,55
32	0,90	0,27	0,41	0,95	0,28	0,43	0,88	0,30	0,44	1,00	0,17	0,29	0,96	0,39	0,56	1,00	0,38	0,55
34	0,92	0,17	0,29	0,93	0,18	0,31	0,88	0,31	0,46	1,00	0,17	0,29	1,00	0,38	0,55	1,00	0,38	0,55
36	0,92	0,17	0,29	0,93	0,18	0,31	0,88	0,31	0,46	1,00	0,17	0,29	1,00	0,38	0,55	1,00	0,38	0,55
38	0,92	0,17	0,29	0,95	0,25	0,40	0,95	0,27	0,42	1,00	0,17	0,29	1,00	0,38	0,55	1,00	0,38	0,55
40	0,92	0,17	0,29	0,95	0,27	0,42	0,95	0,25	0,40	1,00	0,17	0,29	1,00	0,38	0,55	1,00	0,38	0,55

Cuadro 3: Resultados de precision (P), recall (R) y f-measure (F) de la herramienta de alineación.

off entre *precision* y *recall*. Entre estas dos medidas se prioriza *precision*, dado que es más importante que los mapeos que se recuperen sean correctos, a que se recuperen muchos mapeos a costa de introducir incorrectos. Mapeos erróneos pueden dar lugar a razonamientos incorrectos sobre los análisis que puedan hacerse entre la arquitectura y su implementación. En la Figura 4 puede verse que los valores de *precision*, *recall y f-measure* tienden a estabilizarse más rápido en el proyecto MM que el HWS. Esto se debe a que la alineación depende exclusivamente de la correspondencia semántica entre las ontologías, y esta puede ser más evidente en algunos proyectos y más difusa otros.

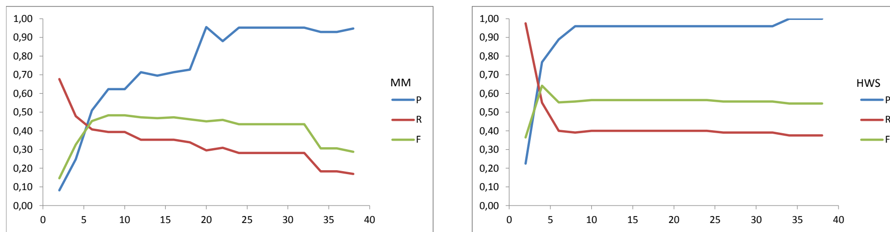

Figura 4: Gráfico de de *precision* (P), *recall* (R) y *f-measure* (F) para distintos valores de K (eje x) utilizando Greedy.

4. Trabajos relacionados

Obtener un nivel de comprensión adecuado de la arquitectura de un sistema de software es importante por muchas razones [10]. En los últimos años, varias investigaciones han explorado el uso de técnicas de Inteligencia Artificial para recuperar la arquitectura

de un sistema a partir de su código fuente. En [17], se analizaron técnicas de Machine Learning para condensar diagramas de clases en pos de obtener una representación de más alto nivel del diseño del sistema. Otros enfoques como [12] intentan recuperar la arquitectura del código fuente utilizando *clustering*. Un enfoque similar al anterior fue utilizado en [7], donde el interés está primero en la recuperación de *concerns* del sistema implementado, y luego junto a información estructural, se intenta identificar automáticamente componentes y conectores. Sin embargo, estos enfoques apuntan a recuperar la arquitectura desde la implementación en casos en los que no existe información acerca de la arquitectura original del sistema. Este trabajo se centra en la recuperación de mapeos entre una arquitectura dada y la implementación del sistema, considerando que dicha arquitectura se conoce de antemano (ya sea porque existe algo de documentación, o porque se ha consultado a un experto). Por otro lado, Jing Sung et al. [21] plantean una descripción ontológica de una arquitectura representada por componentes y conectores, y exponen algunas de las ventajas de las ontologías como representación alternativa a los lenguajes de descripción de arquitecturas (ADLs). A partir de eso, proponen un enfoque de diseño y verificación de modelos de arquitecturas de software usando tecnología de web semántica. Si bien en nuestro enfoque es posible aprovechar técnicas de razonamiento ontológico, el aporte del presente trabajo consiste en considerar la implementación del sistema como una ontología que se adecue a la ontología de la arquitectura, para luego obtener mapeos mediante la alineación de dichas ontologías.

5. Conclusiones

En este trabajo se presentó un enfoque para la generación automática de mapeos entre elementos de una descripción arquitectónica y elementos de la implementación, aprovechando técnicas existentes de alineación de ontologías. Una constribución novedosa del enfoque es la combinación de tres medidas de similitud (lingüística, de *concerns* y estructural). El enfoque fue evaluado en dos casos de estudio, obteniendo buenos resultados en cuanto a precisión aunque con un *recall* moderado. Por otra parte, se encontraron puntos a mejorar en el proceso de alineación. Uno de estos refiere a la función de similitud estructural, cuando existen diferencias de tamaño entre los grafos correspondientes al *blueprint* y a la implementación Java, ya que el algoritmo SFA no está preparado para alinear grafos de tamaños muy dispares. Por lo tanto es necesario, como un paso previo a la alineación, llevar ambas representaciones a un nivel de abstracción similar, o bien, experimentar con otros algoritmos. Otra posible mejora consiste en aportar más información estructural al proceso de alineamiento mediante la inclusión de una función de similitud basada en métricas de SNA (*Social Network Analisys*) [23].

Referencias

1. Aldrich, J., Chambers, C., Notkin, D.: Archjava: connecting software architecture to implementation. In: Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on. pp. 187–197. IEEE (2002)
2. Arthur, D., Vassilvitskii, S.: k-means++: The advantages of careful seeding. In: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 1027–1035. Society for Industrial and Applied Mathematics (2007)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Professional, 3rd edn. (2012)

4. Corley, C., Mihalcea, R.: Measuring the semantic similarity of texts. In: Proceedings of the ACL workshop on empirical modeling of semantic equivalence and entailment. pp. 13–18. Association for Computational Linguistics (2005)
5. Do, H.H., Rahm, E.: Matching large schemas: Approaches and evaluation. Information Systems 32(6), 857–885 (2007)
6. Euzenat, J., Shvaiko, P., et al.: Ontology matching, vol. 18. Springer (2007)
7. Garcia, J., Popescu, D., Mattmann, C., Medvidovic, N., Cai, Y.: Enhancing architectural recovery using concerns. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. pp. 552–555. IEEE Computer Society (2011)
8. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge acquisition 5(2), 199–220 (1993)
9. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on. pp. 109–120. IEEE (2005)
10. Kazman, R., Woods, S.G., Jeromy Carriere, S.: Requirements for integrating software architecture and reengineering models: Corum ii. In: Reverse Engineering, 1998. Proceedings. Fifth Working Conference on. pp. 154–163. IEEE (1998)
11. Kruchten, P.B.: The 4+ 1 view model of architecture. Software, IEEE 12(6), 42–50 (1995)
12. Maqbool, O., Babri, H.A.: Hierarchical clustering for software architecture recovery. Software Engineering, IEEE Transactions on 33(11), 759–780 (2007)
13. Massoni, T., Soares, S., Borba, P.: Requirements health-watcher version 2.0. Early Aspects, ICSE 7 (2007)
14. McGuinness, D.L., Van Harmelen, F., et al.: Owl web ontology language overview. W3C recommendation 10(10), 2004 (2004)
15. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In: Data Engineering, 2002. Proceedings. 18th International Conference on. pp. 117–128. IEEE (2002)
16. Moriconi, M., Qian, X., Riemenschneider, R.A.: Correct architecture refinement. Software Engineering, IEEE Transactions on 21(4), 356–372 (1995)
17. Osman, M.H., Chaudron, M.R., Van Der Putten, P.: An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on. pp. 140–149. IEEE (2013)
18. Rohlf, F.J.: NTSYS-pc: numerical taxonomy and multivariate analysis system. Applied Biostatistics (1992)
19. Rus, V., Lintean, M.: A comparison of greedy and optimal assessment of natural language student input using word-to-word similarity metrics. In: Proceedings of the Seventh Workshop on Building Educational Applications Using NLP. pp. 157–162. Association for Computational Linguistics (2012)
20. Rus, V., Lintean, M.C., Banjade, R., Niraula, N.B., Stefanescu, D.: Semilar: The semantic similarity toolkit. In: ACL (Conference System Demonstrations). pp. 163–168. Citeseer (2013)
21. Sun, J., Wang, H.H., Hu, T.: Design software architecture models using ontology. In: SEKE. pp. 191–196 (2011)
22. Sutton Jr, S.M., Rouvellou, I.: Modeling of software concerns in cosmos. In: Proceedings of the 1st international conference on Aspect-oriented software development. pp. 127–133. ACM (2002)
23. Wasserman, S.: Social network analysis: Methods and applications, vol. 8. Cambridge university press (1994)
24. Young, T.J.: Using aspectj to build a software product line for mobile devices. Ph.D. thesis, The University of British Columbia (2005)