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

Una Arquitectura basada en la Web Semántica para la Gestión de Versiones de Familias de Producto

Sonzini María Soledad[1, 2], Leone Horacio Pascual[1].

[1] INGAR, Instituto de Desarrollo y Diseño, Avellaneda 3657, Santa Fe, Arg.
[2] Universidad Nacional de La Rioja, Luis M. de la Fuente S/N, La Rioja, Arg.
[1]{ssonzini, hleone}@santafe-conicet.gob.ar

Resumen. El objetivo principal del trabajo es presentar el desarrollo de una arquitectura y su implementación en una aplicación prototipo que permite la utilización de una ontología de gestión de versiones para la integración y gestión unificada de variantes y versiones de Familias de Productos. Se propone un modelo de arquitectura compuesto de 3 módulos que se comunican entre sí cumpliendo funciones específicas, haciendo uso de diversas herramientas de la Web Semántica. La herramienta propuesta puede ser utilizada para lograr la gestión de la variabilidad en el tiempo y en el espacio de Familias de Producto en diversos tipos de industrias. En particular, en este artículo se introduce su aplicación, para la gestión de versiones y variantes, de una Familia de teléfonos celulares.

1 Introducción

El desarrollo de la tecnología de la web semántica ha introducido un cambio de paradigma en la industria, donde las empresas están adoptando las ontologías como una herramienta de integración semántica. La traducción de los modelos de datos empresariales en ontologías, permiten construir un vocabulario común para ser comprendido e intercambiado entre diversos actores en un entorno distribuido. Por lo tanto, numerosas herramientas software como: Protégé, Pellet, Jena Framework, Fuseki, entre otras, fueron diseñadas para el procesamiento de lenguajes (OWL, RDF, SPARQL, SWRL) que permiten la formalización de estos modelos de datos.

El intercambio de modelos de datos, es posible mediante la introducción de formatos de intercambio de datos. Existen dos formatos estándares: XML (eXtensible Markup Language) y JSON (JavaScript Object Notation). Estos formatos son independientes y sencillos de implementarse en herramientas de software. Establecen comunicación por medio de interfaces que reciben solicitudes y envían peticiones logrando la interoperabilidad entre sistemas heterogéneos [1].

Por su parte, los sistemas de información para la gestión de ciclo de vida de producto o PLM (Product Lifecycle Management), requieren soluciones para representar los modelos de datos de producto de forma consistente, con el fin de compartir y operar con otras organizaciones, procesos, etapas del ciclo de vida y stakeholders (personas o entidades involucradas en las actividades). Además, durante el ciclo de vida de

un producto, es común que se introduzcan cambios en el diseño y/o configuración debido a cuestiones comerciales, de mercado o avances tecnológicos.

La capacidad de cambio de un producto recibe el nombre de *variabilidad* y permite que un producto sea extendido, personalizado o configurado para su utilización en un cierto dominio [2].

Pohl y colab. en [3], sostienen que es fundamental hacer una distinción entre variabilidad en el tiempo y variabilidad en el espacio. La primera de ellas se define como "la existencia de diferentes versiones de un producto que es válido en diferentes tiempos", denotando su evolución. Por el contrario, la variabilidad en el espacio se define como "la existencia de un producto en diferentes formas en un mismo tiempo". Esta dimensión abarca de manera simultánea el uso de diferentes variantes de productos que coexisten en un mismo instante de tiempo. Es importante mencionar que los métodos utilizados actualmente en la gestión de variabilidad espacial, no pueden aplicarse del mismo modo para la gestión de variabilidad temporal. Por esto, surge la necesidad de definir un mecanismo apropiado para administrar los cambios en el tiempo y que pueda ser aplicado conjuntamente con los modelos de representación de variantes existentes.

Con el fin de contar con un modelo formal para la gestión integrada de variantes, así como un vocabulario común para la representación de los cambios que modifican la información de productos durante su ciclo de vida, en un trabajo previo [4] se desarrolló una ontología para la gestión de versiones. Esta ontología define conceptos genéricos, que pueden ser extendidos por modelos de gestión de variantes de Familias de Producto, como PRONTO (PRoduct ONTOlogy) [5] y el modelo de característica o FM (Feature Model) [6].

En este trabajo se propone una arquitectura que posibilite la implementación de una herramienta de la Web Semántica, denominada VERONTO, para la gestión de versiones de Familias de Producto. La arquitectura propuesta tiene a la ontología de versiones como uno de sus componentes e integra diversas herramientas y tecnología de la Web Semántica. La implementación de esta arquitectura permitirá el desarrollo de una aplicación que gestione los cambios de un producto de modo holístico, contemplando desde la identificación del cambio, el análisis de su impacto sobre los elementos afectados y el registro adecuado para prevenir las inconsistencias en la información de Familias de Producto.

A partir de esta breve introducción, el presente artículo se organiza de la siguiente forma: en la Sección 2 describe los modelos de gestión de variabilidad de Familias de Producto. El desarrollo del modelo de arquitectura se explica en detalle en la Sección 3. En la sección 4 se desarrolla un caso de estudio y su aplicación mediante las herramientas de la web semántica VERONTO. Finalmente en la Sección 5, se presentan las conclusiones y trabajos futuros.

2 Gestión de Variabilidad en dos dimensiones: Temporal y Espacial

La importancia de gestionar la información de los productos en todas las fases de su ciclo de vida, está en la ocurrencia de un evento de cambios en una etapa, el cual podría propagarse y afectar la consistencia e integridad de la información en otras

etapas. Además, existen cuestiones sin resolver, tal como la gestión de dependencias entre elementos afectado (puntos de variación) y variantes de un producto, lo que ha impulsado algunos estudios sobre las relaciones que existen (no siempre explícitas) entre estos.

A partir de esto, la definición de la ontología de versiones de [4], tiene como punto de partida la necesidad de gestionar de forma simultánea las dos dimensiones de variabilidad. Para ello, busca administrar los cambios ocurridos en la información de Familias de Producto, durante su ciclo de vida, por medio del registro de versiones temporales. Como se dijo anteriormente, su conceptualización incluye conceptos genéricos, para ser extendidos por diversos modelos de gestión de variantes. Para una mejor comprensión, en la Fig. 1 se representa el modo en que la ontología de versiones es extendida por los modelos de variantes de productos PRONTO y FM (Features Model), a través de los puntos de extensión, obteniendo así modelos de productos versionados, tal como: PRONTO versionado y FM Versionado.

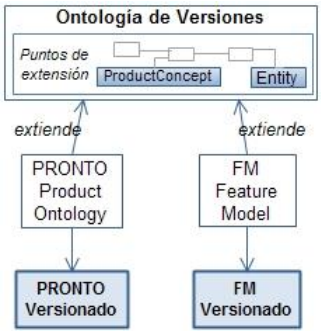

Fig. 1. Extensión de PRONTO y FM en la ontología de Versiones.

En la Fig. 2, se muestra los conceptos principales de la ontología de versiones, los cuales se describen brevemente en esta sección. El principal concepto es *ProductConcept,* el cual permite representar la información del producto cuyas versiones se van a gestionar. Cada *ProductConcept*, se vincula al concepto *History* mediante la relación *hasHistory*. Este segundo concepto, reúne el conjunto de versiones del producto analizado, representando su evolución en distintos instantes de tiempo. Cada versión temporal (*Version*) se construye a partir de la ocurrencia de un evento de cambio (*ChangeEvent*) y representa la configuración del producto válida durante un lapso comprendido desde el instante en que se crea la versión (indicado por *DateTime*), hasta el instante de creación de una nueva versión temporal.

En el modelo, cada versión temporal, excepto la inicial, es relacionada con su versión predecesora, a través de la asociación *previous*. Esta relación posibilita la reconstrucción de la versión actual de un producto a partir de su versión inicial. Además, por medio del concepto *Specification*, se registran datos adicionales acerca de la versión generada. Asimismo, un evento de cambio se relaciona con la actividad que ocasiona el cambio sobre una entidad (*Entity)*. Las actividades básicas consideradas por el modelo son: el agregado (*Add*) y la eliminación (*Delete*) de una entidad, así como la modificación (*Edit*) de un atributo (*Attribute*), identificando su nombre y el nuevo valor. Dependiendo el dominio de aplicación, no todas las actividades pueden ser

aplicadas a una entidad. El concepto *ActivityConstraint* permite restringir el tipo de actividad soportada por una entidad.

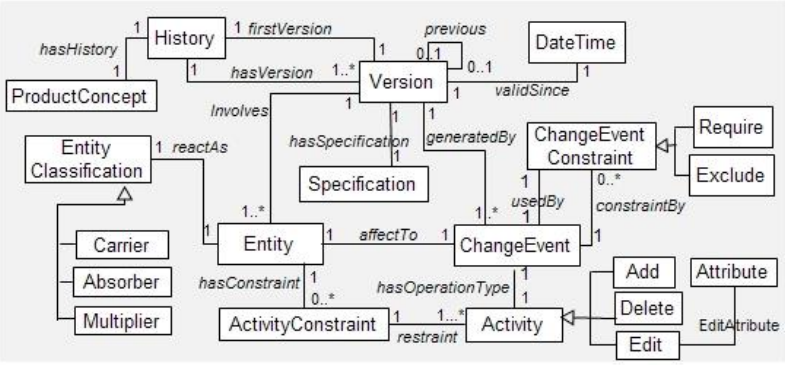

Fig. 2. Modelo Conceptual para la gestión de variabilidad temporal.

En el contexto de la información de producto, una entidad (*Entity)* es un concepto abstracto que debe ser extendido para representar los elementos del modelo de producto utilizado, que pueden ser afectados por los cambios, por ejemplo: un componente, una relación, una restricción o un atributo. Una entidad forma parte de una estructura y, un cambio en ella podría afectar a otras entidades de dicha estructura, produciéndose una serie de nuevos eventos de cambios. Para representar esta situación, también conocida como propagación de cambios, se introduce el concepto *ChangeEventConstraint* para especificar las restricciones asociadas a un evento de cambio.

El modelo propone una clasificación de tres tipos de reacciones, como respuesta de una entidad ante la ocurrencia de eventos de cambio que la afectan: Absorbentes (*Absorber*), para las entidades que absorben un número de cambios mayor de los que pueden transmitir; Portadoras (*Carrier*), para las entidades que pueden absorber el mismo número de cambios que el que transmiten a otras entidades; y Multiplicadoras (*Multiplier*), para entidades que generan un número mayor de eventos de cambios de los que pueden absorber. A partir de lo expuesto, la ontología de versiones permite gestionar la variabilidad temporal de Familias de Producto, identificando los componentes afectados, cómo éstos fueron modificados por un tipo de actividad, la causa de los eventos de cambios y el instante en que una versión comienza a ser válida. Además, es posible analizar la evolución de una familia de productos mediante el conjunto de versiones temporales que se generaron durante su ciclo de vida.

Como se mencionara anteriormente, la ontología de versiones propuesta puede ser extendida por diferentes modelos de productos que gestión variabilidad espacial. En particular, en el presente trabajo se muestra la extensión de VERONTO utilizando los conceptos de PRONTO, los cuales se introducen brevemente a continuación.

Por su parte, el modelo de productos PRONTO, representa la información de productos y sus variantes en el dominio de la industria de manufactura. PRONTO, representa datos de productos en diferentes niveles de abstracción y en diferentes dominios de la industria, por medio de la introducción de una jerarquía estructural (SH- Structu-

ral Hierarchy) para representar la información concerniente a los productos y componentes que participan en la manufactura de productos, y una jerarquía de abstracción (AH- Abstraction Hierarchy) que permite la representación de información no estructural de productos en diferentes niveles de abstracción y la representación de procesos de agregación y desagregación de información entre estos niveles. A continuación se describen brevemente los conceptos de PRONTO que se utilizan para extender la ontología de versiones (ver Fig.1). Para más detalles del modelo de PRONTO ver [5].

Por cada nivel de la AH, se identifican conceptos de PRONTO que pueden ser afectados por algún cambio. Estos conceptos, son susceptibles de ser modificados y especializan el concepto *Entity* de la ontología de versiones. Entre estos conceptos, se identifican relaciones de estructurales como *CRelation* y *DRelation,* atributos del valor (*Value*) de cada relación, cambios en la estructura (*Change*) y restricciones (*Restrictions*). La gestión de versiones se da por cada nivel de la AH, por lo tanto los conceptos de *Family*, *VariantSet* y *Product*, son especializaciones del concepto *ProductConcept.*

Desde otra perspectiva, existe una serie de trabajos, los cuales realizan una descripción de arquitecturas de herramientas para la gestión de versiones de información. De esta forma, en el contexto de las ontologías, Völkel y Groza [7] proponen la implementación de un sistema basado en ontologías, denominado SemVersion, para gestionar la evolución de una ontología desde una perspectiva estructural. La arquitectura de SemVersion consiste en varias capas dependientes entre sí, y son desarrolladas puramente con librerías Java. Por su parte, Redmon y colab. [8] proponen un sistema para gestionar los cambios de una ontología en un entorno donde se soportan múltiples editores. Este sistema tiene una arquitectura cliente-servidor, donde el servidor mantiene los estados actuales y el historial de cambios de todas las ontologías administradas, y las comunicaciones iniciadas por un cliente, pueden ser de 3 tipos: GET, Diff y PUT, para solicitar una revisión, un conjunto de cambios entre dos revisiones y crear una nueva revisión, respectivamente. Bass y colab. en [9] desarrollan diferentes modelos de arquitectura para el desarrollo de sistemas software, entre ellos, el modelo cliente-servidor y el modelo MVC (Modelo-Vista-Controlador) para entornos web. Las propuestas de los trabajos [7,8] se enfocan particularmente en la gestión de cambios y versiones de la ontología desde una perspectiva estructural, en dominios específicos. En contraste, la ontología propuesta incluye conceptos genéricos para ser extendidos independientemente de la ontología de variabilidad espacial implementada. Por otra parte, es importante mencionar que la gestión de variabilidad mediante el uso de ontologías, permite construir modelos de datos consistentes e íntegros, que pueden interoperar en entornos distribuidos con otras unidades como: organizaciones, procesos, e incluso, etapas del ciclo de vida.

3 Arquitectura de VERONTO como herramienta de la Web Semántica para la gestión de la variabilidad de productos.

En esta sección, se describe la arquitectura propuesta para la implementación de una herramienta de la Web Semántica para la gestión unificada de variantes y versiones de producto. Dicha herramienta se denomina VERONTO, y tiene como componentes las implementaciones en OWL (Ontology Web Language) de las ontologías de versiones

propuesta y la ontología de productos. Se propone la construcción de una arquitectura que distinga tres aspectos esenciales: la capa de usuario, la capa de procesamiento de datos y la capa de almacenamiento. Para ello, se adoptan el enfoque del patrón de arquitectura MVC (Modelo-Vista-Controlador) [9], el cual separa en tres componentes disjuntos, y en lo posible desacoplados, el modelo (datos y su almacenamiento), la vista (interfaz del usuario) y el controlador (algoritmos de procesamiento de datos). Este patrón busca desacoplar los tres componentes con el fin de modificar un componente de forma independiente sin afectar a los demás. Dentro del enfoque MVC, el componente Modelo contempla la información del dominio con la que interactúa la herramienta y su almacenamiento. El componente Vista es la representación gráfica e interactiva con los stakeholders, y el nexo entre estos dos componentes es el Controlador, encargado del control del flujo de los eventos generados en la Vista y la ejecución de las acciones pertinentes en el Modelo.

A fin de introducir la herramienta en un entorno distribuido, donde los stakeholders puedan interoperar y hacer uso de la misma, en la Fig. 3 se muestra los tres componentes contenidos en un servidor web Apache Tomcat. Este servidor es desarrollado bajo la tecnología Java por los miembros de la fundación de software Apache [10] y se comunica, por medio del protocolo HTTP, con los navegadores web o servicios web. Esta comunicación consiste en la recepción de solicitudes y la presentación de resultados, del mismo modo en el que se comunican en una estructura cliente-servidor. Por su parte, el flujo de comunicación entre los componentes MVC y las funciones que cumplen dentro del servidor se resume en las siguientes subsecciones.

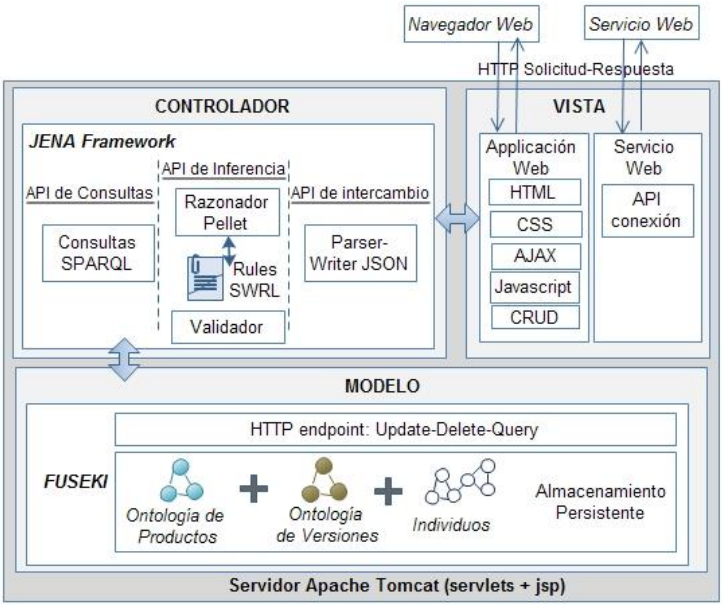

Fig. 3. Modelo de Arquitectura para la herramienta VERONTO

3.1 Vista

El componente Vista tiene comunicación directa con los stakeholders y recibe los eventos de petición, que no son más que las consultas que se traducen y se ejecutan al Modelo. Estas consultas se resumen bajo las siglas de CRUD (Create, Read, Update y Delete), y se presentan como formularios y tablas, insertos en una estructura dada por HTML y CSS, la cual permiten la interacción y la exhibición de los resultados.

AJAX (Asynchronous JavaScript And XML) es una tecnología que permite actualizarse de forma dinámica los resultados sin necesidad de recargar completamente la interfaz (página web) presentada, y de esta forma no se genera un procesamiento redundante en servidor [11]. Por medio de JavaScript se reciben los datos desde la página web, y se los envía al componente Controlador para ser procesados, el cual luego envía una respuesta que es interpretada de nuevo por JavaScript en la página web. Si bien el intercambio de datos ha sido realizado inicialmente en formato XML, y en base a la evaluación de formatos expuesta en [1], JSON se ha convertido en el nuevo formato estándar, con una notación más simple y ligera durante su procesamiento. Ambos formatos son metalenguajes basados en reglas simples y estrictas, que permite definir una gramática específica y, por tanto, permite el intercambio de información estructurada y legible de los datos contenidos. Además, el componente Vista puede ser presentado como la interfaz de comunicación entre servicios web, correspondiente a una arquitectura orientada a servicios (SOA, siglas del inglés Service Oriented Architecture), donde el intercambio de datos se realiza mediante una aplicación que recibe y envía los formatos XML y JSON a través del protocolo HTTP.

Dada las limitaciones de espacio, en el presente trabajo se describen las tecnologías para la construcción del componente Vista, sin entrar en detalle en las características de las interfaces de comunicación con los stakeholders.

3.2 Controlador

Los eventos de solicitud capturados por la Vista son transformados en cadena de texto con formato JSON. Estas cadenas son enviadas al *API de intercambio* donde son analizadas (Parse) para obtener un conjunto de valores y luego se los transforma (writers) en recursos clasificados como *Subject*, *Property* y *Object*. Este proceso de conversión e interpretación de formatos recibe el nombre de *serialización* y permite construir estructuras de tripletas, conformadas por recursos y sus identificadores URI (Uniform Resource Identifier), para ser manipuladas por Jena.

Jena [12] es un framework desarrollado en Java para construir aplicaciones de la web semántica mediante librerías que manipulan un conjunto de lenguajes recomendados por la W3C, tal como: RDF, RDFS, OWL y SPARQL. De esta forma, se desarrollan aplicaciones para ejecutar las consultas CRUD (*API de Consultas*) y así gestionar las tripletas RDF en el modelo.

Jena está diseñado para ser ensamblado con diversos motores de inferencia [13]. Por lo que se opta integrar en esta arquitectura, el razonador Pellet para ser invocado dentro de la *API de Inferencia*. El principal aporte de los motores de inferencia está en que mediante su ejecución, es posible obtener nuevas aserciones que se logran a partir del razonamiento de un conjunto de axiomas y reglas escritas en SWRL.

Los razonadores son diseñados para ser implementados de forma general aplicando un conjunto de reglas genéricas para el procesamiento de tripletas RDF. De acuerdo al dominio de aplicación, se pueden definir nuevas reglas que amplíen la capacidad del razonador para inferir nuevo conocimiento (tripletas RDF). Además, para el caso en donde los stakeholders desean gestionar sus propias reglas, es posible construir un archivo externo que las contenga (ver *Rules SWRL* del componente *API de inferencia* en la Fig.3). Seguido de la ejecución del razonador, se valida el modelo teniendo en cuenta las tripletas inferidas. Por lo general los lenguajes de ontologías expresan restricciones, y mediante la validación, se detecta cuando tales restricciones son violadas por algún conjunto de datos, generando inconsistencias en el modelo.

El flujo de comunicación entre los componentes Controlador y Modelo, ya sea para solicitar o enviar información generadas por el *API de Consultas*, son mediante el protocolo HTTP y pueden ser por medio de dos métodos: GET y POST; ambos métodos son válidos y ampliamente utilizados en entornos web.

3.3 Modelo

Como se mencionó anteriormente, el Modelo representa el conjunto de datos del dominio, el cual se construye sobre el concepto de tripletas, una estructura ordenada de tres elementos: sujeto, predicado y objeto. Para la arquitectura propuesta, se adopta la herramienta Apache Jena Fuseki, en su segunda versión, por ser compatible con el framework Jena y de simple implementación.

Fuseki es un servidor SPARQL para gestionar tripletas RDF [14], donde el almacenamiento puede ser de dos modos: persistente (se almacena en disco) o en memoria (no se almacena). Además, las tripletas almacenadas pueden ser accedidas por el protocolo HTTP. Esta herramienta puede operar de tres formas: como un servicio de un sistema operativo, como una aplicación web Java y como un servidor independiente, el cual se ejecuta localmente. Además, proporciona un conjunto de interfaces de monitoreo y administración en un entorno seguro basado en el framework Apache Shiro [15].

Las consultas SPARQL que recibe Fuseki desde el componente Controlador pueden clasificarse en 4 grupos: *Update* para modificar y crear nuevas instancias, *Create* para crear nuevos recursos en la ontología, *Delete* para eliminar recursos existentes y *Query* para consultar el modelo de tripletas almacenado. La presentación de los resultados a partir de estas consultas, puede ser configurada mediante tablas HTML, RDF/XML, Turtle, y JSON, para la interacción con otras aplicaciones o usuarios seres humanos que comprendan tal configuración.

Fuseki, a través de su entorno de administración, permite crear un modelo de datos e importar las ontologías para su gestión y almacenamiento. Para la propuesta del presente trabajo, se importa la ontología PRONTO versionado (PRONTO y Ontología de Versiones unificadas), previamente exportada en formato RDF desde la herramienta Protégé versión 5. El almacenamiento de la ontología importada es de tipo persistente, con el fin de que los datos permanezcan ante los posibles reinicios de Fuseki. Además, mediante el procesamiento de consultas, Fuseki gestiona los individuos de la ontología de productos versionada.

Para cada modelo de dato creado en Fuseki, se genera automáticamente un archivo de configuración que contiene información acerca de los servicios disponibles con los

permisos de los extremos (*endpoint*) de comunicación vía HTTP y el método de envío de solicitudes y repuestas soportado por el modelo generado. A partir de lo mencionado, la interfaz de consultas (*API de consultas)* dentro del Controlador, establece conexión con el Modelo y envía las consultas por medio del método GET y a través del protocolo HTTP con extremo *Update*, *Create*, *Delete* o *Query*, para indicar el tipo de consulta. Fuseki ejecuta la consulta en el modelo de datos y envía los resultados al framework Jena para inferir conocimiento y validar los modelos en busca de inconsistencias. Los resultados se traducen en el formato JSON y se transmiten al componente Vista para la presentación de los resultados de las consultas realizadas por los stakeholders.

4 Integración de las tecnologías a través de un Caso de Estudio

En esta sección se desarrolla un caso de estudio en el que se utiliza la aplicación prototipo que resulta de implementar los componentes Modelo y Controlador de la arquitectura propuesta en la sección 3. En particular, el caso de estudio se basa en la gestión de variantes y versiones de productos de la telefonía celular. En particular, se considera la manufactura de tres líneas de productos pertenecientes a la familia *Samsung*. Estas líneas se identifican como *Galaxy S*, *Galaxy A* y *Galaxy J*. Cada una de estas incluyen diversos modelos (ver Tabla 1), donde *Galaxy S* comprende los modelos *S6* y *S7*; *Galaxy A* comprende los modelos *A7* y *A8*, y *Galaxy J* comprende al modelo *J1*. Cada producto posee un conjunto de componentes con características propias de cada modelo. A fin de simplificar el caso para presentar con mayor claridad la estrategia adoptada, se consideran solo 4 componentes en la estructura de producto: *procesador*, *sistema operativo, cámara lateral* y *cámara frontal*. Cada componente posee diferentes particularidades y fabricantes, por lo que pueden ser clasificados también. Es decir, para el componente procesador se identifican 3 fabricantes: *Exynos*, *Cortex* y *Snapdragon Qualcomm*; y, en base a las características de procesamiento, puede sub clasificarse *Exynos* en *Exynos7x* y *Exynos8x*; *Cortex* en *A7*, y *Snapdragon Qualcomm* en *Series600*. Del mismo modo, en la Tabla 1, se clasifican los componentes restantes.

Table 1. Producto Teléfono Celular de la Familia Samsung.

Líneas	Modelo	Procesador	Sistema Operativo	Cámara Lateral	Cámara Frontal
Galaxy S	S6	Exynos7x	Lollipop5x	BSI16Mp	5Mp
	S7	Exynos8x	Marshmallow6x	BSI12Mp	5Mp
Galaxy A	A7	Series600	Lollipop5x	CMOS13Mp	5Mp
	A8	Exynos7x	Lollipop5x	CMOS16Mp	5Mp
Galaxy J	J1	A7	KitKat4x	CMOS5Mp	2Mp

Por medio de las dos jerarquías: SH y AH, introducidas por la ontología PRONTO, en la Fig. 4 se representa la información simplificada de la familia de productos *Samsung* en niveles de Familias, Conjunto de Variantes y Producto; y, por ser una estructura de composición, las relaciones *componentOf* y *memberOf*, para vincular las par-

tes componentes que intervienen y su pertenencia a niveles superiores en la AH. En la figura se representan las relaciones de composición únicamente para el producto *Samsung Galaxy S6* y conjunto de variante *Samsung Galaxy S,* a fin de no afectar la claridad de interpretación de la misma.

Para gestionar los cambios, que podrían afectar a la información de la familia de productos Samsung y sus versiones, se identifican las entidades de PRONTO susceptibles de ser modificadas, que extienden a VERONTO. Así, la información a nivel de producto del Samsung Galaxy S6, podría ser modificada ante un evento en el que se desea actualizar el sistema operativo Lollipop5x por Marshmallow6x.

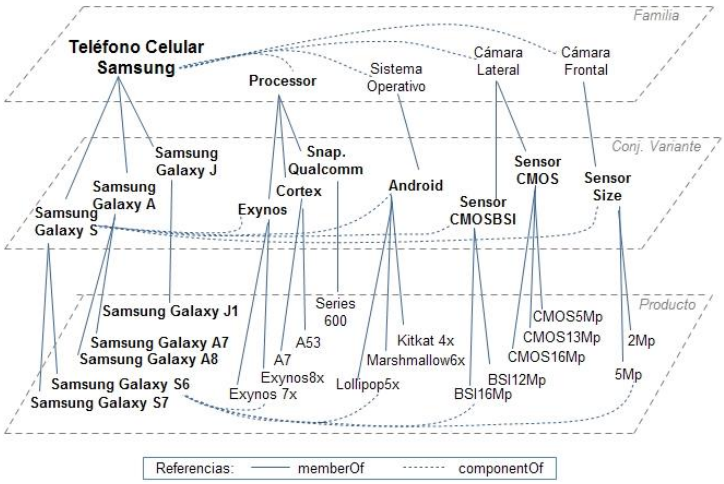

Fig. 4. Representación Simplificada del Caso de Estudio en PRONTO

La manipulación de tripletas para gestionar la variabilidad del producto, provienen de las consultas CRUD escritas en la Vista y ejecutadas en Jena (Controlador). Sin embargo, el presente trabajo no entra en detalle en las interfaces de los stakeholders, por lo que se escriben las consultas desde el Controlador.

En la parte superior de la Fig.5 se ilustra parte del código JAVA que permite escribir la consulta SPARQL, para generar la nueva versión de la línea *Samsung Galaxy S6*. Asimismo, en la parte inferior de dicha figura, se muestran los resultados obtenidos, los cuales permiten obtener información respecto al producto afectado y al evento de cambio acontecido.

En la parte inferior de la Fig. 5, se han etiquetado con letras (de "a" a "g") los diferentes datos obtenidos como respuesta a la consulta. La interpretación del resultado obtenido es explicado a continuación referenciando dichas etiquetas. De esta forma, en la Fig. 5 se observa la nueva versión *SGS6Version3* identificada por *a*, que comienza a ser válida desde una fecha *02feb2016_10:00am* (*b*) y es generada por el evento de cambio (*SOChangeSGS6*) (*f*). Este cambio afecta la relación de composición *chosenProductSO* (*c*), que vincula al producto *Samsung Galaxy S6* con el sistema operativo *Lollipop5x,* mediante su eliminación (*Delete)* (*d*). La nueva versión incluye una especificación (*e*) para incluir información respecto a la causa del evento. Este evento de cambio se asocia a una restricción (*g*) para requerir un nuevo evento de

cambio que incorpora una nueva relación de composición con el sistema operativo *Marshmallo6x*.

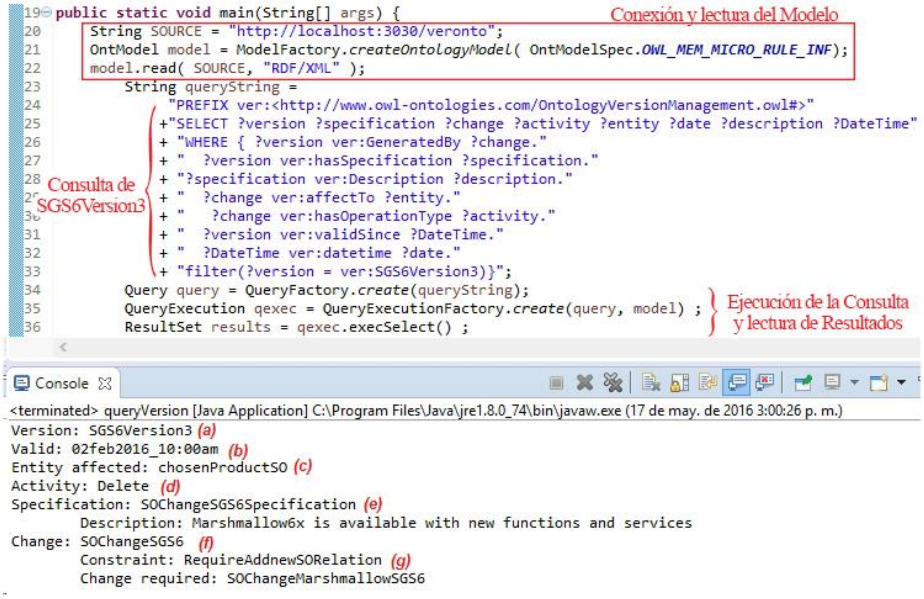

Fig. 5. Fragmento en Jena para crear una nueva versión por medio de consultas SPARQL.

Por otra parte, mediante los mecanismos de inferencia es posible conocer el historial del producto *Samsung Galaxy S6,* para obtener información de los cambios que la afectaron durante su ciclo de vida. Para inferir el orden de versiones que acontecieron a lo largo del ciclo de vida, se escribe una regla con un mecanismo de deducción hacia adelante y en un archivo externo, de modo que los stakeholders puedan agregar o modificar reglas sin afectar el código de inferencia del modelo (ver Fig.6). Esta regla aplica la ley de transitividad para evaluar el conjunto de versiones mediante la navegación por el predicado *previous*. Es decir, si la nueva versión *SGS6Version3* se vincula con su predecesora *SGS6Version2* a través de *previous, y SGS6Version2* se vincula con su predecesora *SGS6Version1,* por lo que *SGS6Version1* también es predecesora de *SGS6-Version3.* De este modo, en la Fig. 6, el razonador invoca el archivo contenedor (*A en Fig. 6*) de la regla *rules.txt* (representado en la Fig. 3 dentro de la interfaz de inferencia del componente Controlador) e infiere, sobre el modelo *model1,* el vínculo entre las versiones *SGS6Version1* y *SGS6Version3* a través de *previous (B en Fig. 6).* Además, mediante una regla es posible inferir la reacción de la entidad afectada basada en la cantidad de eventos que impactan en ella y la cantidad de eventos que se generan. Para el caso estudiado, la entidad *chosenProductSO* se clasifica como Portadora (*Carrier*) *(C en Fig. 6)* por transmitir la misma cantidad de eventos que los que impactan en ella. Seguido del razonamiento, se valida el modelo en busca de inconsistencias, dando un resultado satisfactorio (*D en Fig. 6*) para la situación analizada.

De esta forma, la gestión de versiones para la familia *Samsung Galaxy S6*, se basa en un conjunto de consultas CRUD y el motor de inferencias que permiten identificar

el cambio, el impacto de los elementos afectados, las restricciones asociadas para la propagación de cambios y la reconstrucción del historial de cambios. También, es posible conocer el instante en que comienza a ser válida la nueva versión, y la especificación, para registrar información concerniente al cambio. Este registro de versiones, contribuye a prevenir las inconsistencias en la información de la familia de producto y poder conocer y reconstruir estructuras de productos en un instante determinado a lo largo de su ciclo de vida.

Una vez obtenido los resultados de las consultas, estos son sometidos al proceso de serialización, dentro de la interfaz de Intercambio (*API de Intercambio* de la Fig.3), para convertir los resultados de las consultas en un formato de intercambio y ponerlo a disposición del componente Vista. Jena soporta diferentes scripts para el proceso de serialización de formatos, para este caso los resultados son serializados en formato JSON. Esta conversión consiste en un único objeto que contiene un miembro "head" y un miembro "results" o "boolean", dependiendo el tipo de consulta.

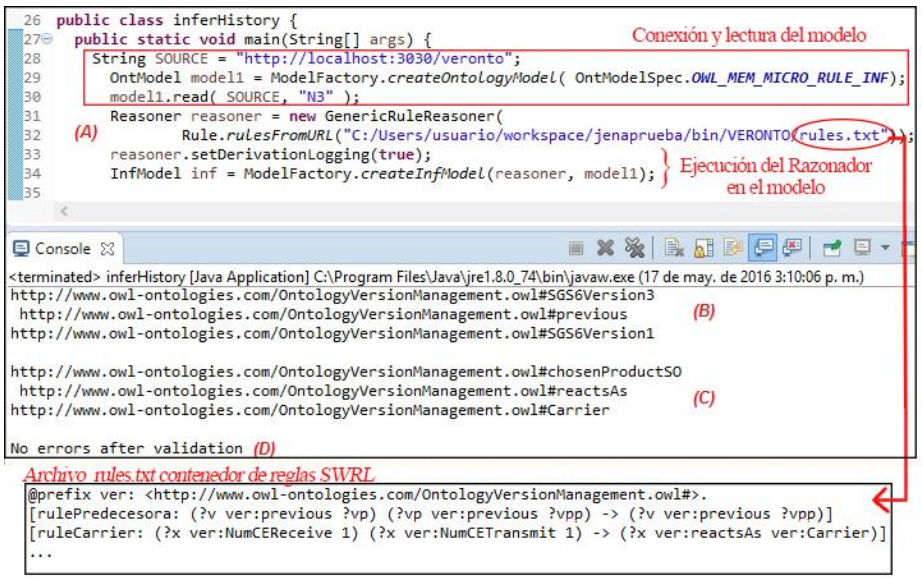

Fig. 6. Razonamiento y validación del Modelo.

La Fig. 7 muestra un fragmento producto del proceso de serialización para la consunta SPARQL que genera la versión *SGS6Version3*. El resultado conversión se compone del miembro "head" para identificar tres recursos: sujeto (S), propiedad (P) y objeto (O), que permiten constituir la estructura de la tripleta, y con el identificador "bindings" (parte del miembro "results"), se integran los resultados en un vector, para cada recurso con su identificador URI y valor.

```
{  "head": { "vars": [ "s" , "p" , "o" ]  } ,
  "results": { "bindings":
    [ { "s": { "type": "uri" , "value": "http://www.owl-
ontologies.com/OntologyVersionManagement.owl#SGS6History" } ,
        "p": { "type": "uri" , "value": "http://www.owl-
ontologies.com/OntologyVersionManagement.owl#hasVersion" } ,
        "o": { "type": "uri" , "value": "http://www.owl-
ontologies.com/OntologyVersionManagement.owl#SGS6Version3" }
      } ]
  } }
```


Fig. 7. Serialización en JSON de la consulta para la versión *SGS6Version3*

Del mismo modo, el proceso de serialización convierte los resultados obtenidos para cada consulta realizada y procesada por el *API de Consulta* y el *API de Inferencia* del componente Controlador

5 Conclusiones y Trabajos Futuros

Con el fin de gestionar la variabilidad en dos dimensiones de forma simultánea, se propone la herramienta VERONTO, construida con tecnologías de la Web Semántica, que permite la gestión de versiones y variantes de productos, a través de la integración de la ontología de versiones que puede ser extendida por parte de diversos modelos de producto que gestionen variantes. En particular, en este trabajo se presenta la especialización de la ontología de versiones mencionada con la ontología PRONTO.

Asimismo, se describe la propuesta de una arquitectura, que permita la implementación de VERONTO, inserta en un entorno Cliente-Servidor e integra tres componentes, basado en el patrón de arquitectura MVC: Modelo (almacenamiento de datos de Familia de Producto), Vista (interacción con los stakeholders) y Controlador (procesamiento de datos). Los componentes interactúan entre sí a través de diversas herramientas: Jena y Fuseki, y lenguajes de representación, consultas e intercambio de datos como RDF, SPARQL y JSON.

Se presenta, también, la implementación de dos de los componentes principales de la arquitectura: el procesamiento de tripletas (Controlador) y la ejecución de consultas y almacenamiento (Modelo). Esta aplicación prototipo ha sido utilizada para la gestión de variantes y versiones de un caso de estudio basado en la telefonía celular. La gestión de versiones del caso de estudio fue posible mediante la construcción de consultas de tipo CRUD, donde se registra adecuadamente una versión de una familia de producto para prevenir inconsistencias en la información de producto. Esta versión reúne información relevante acerca del evento de cambio, los elementos afectados, restricciones, propagación de los cambios y reconstrucción del historial de cambios.

Queda pendiente y como trabajo futuro, el desarrollo de interfaces para la comunicación con los stakeholders, considerando la interacción con otros servicios web. Se pretende que la herramienta brinde un entorno de administración que permita al usuario escribir sus propias consultas y reglas de inferencia, para lograr una gestión eficiente de la información con la que opera.

Otra línea de trabajo a considerar, es el agregado a la herramienta de un módulo que permita la extensión de la ontología de versiones directamente en VERONTO, sin tener que extenderla en protégé y luego importarla en la herramienta.

Por otro lado, se proyecta implementar la herramienta en diferentes casos de estudios más robustos para su validación y para identificar los comportamientos repetitivos que se producen antes los cambios. Estos comportamientos, permitirán definir patrones de predicción de los efectos de tales cambios y sus dependencias, para aplicar procedimientos eficientes a la gestión de cambios.

Agradecimientos

Se agradece el apoyo brindado por estas instituciones: CONICET, Universidad Tecnológica Nacional (PID 3810) y Universidad Nacional de La Rioja.

Referencias

1. Nurseitov, N., Paulson, M., Reynolds, R., Izurieta, C. Comparison of JSON and XML data interchange formats: a case study. In: The International Conference on Computer Applications in Industry and Engineering. ISCA, Cary, NC, USA, pp. 157–162. (2009)
2. Asikainen T., Mannisto T., Soininen T. Kumbang: A domain ontology for modelling variability in software product families. Advances engineering informatics. (2006)
3. Pohl K., Bockle G., Van Der Linden F.: Software Product Line Engineering. Foundations, principles, and Techniques. ISBN-10 3-540-24372-0 Springer Berlin Heidelberg New York. (2006)
4. Sonzini M. Soledad, Vegetti Marcela, Leone Horacio. Towards an ontology for product version management. International Journal of Product Lifecycle Management (IJPLM), Vol. 8, No. 1, (2015).
5. Vegetti M., Leone H, Henning, G.. PRONTO: An ontology for comprehensive and consistent representation of product information. Engineering Applications of Artificial Intelligence 24 (8), pp. 1305-1327. (2011)
6. Kang K.C. Lee K., Lee J.: Feature Oriented Product line Software Engineering: Principles and guidelines. Domain-Oriented Systems Development: Practices and Perspectives. Cap. 2. ISBN 0203711874. (2003)
7. Völkel M., Groza T. SemVersion: and RDF-based Ontology Versioning System. In: Proc. 5th IADIS International Conference on WWW/Internet. IADIS Press (2006)
8. Redmond T., Smith M., Drummond N., and Tudorache T.. Managing change: An ontology version control system. In OWL: Experiences and Directions, 5th Intl. Workshop, OWLED 2008, Karlsruhe, Germany (2008)
9. Bass L, Clements P., Kazman R., Software Architecture in Practice 3rd Edition, SEI Series in Software Engineering, Addison Wesley (2012)
10. The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/. Acceso: 26/04/2016.
11. Ying M., Miller J. Refactoring legacy AJAX applications to improve the efficiency of the data exchange component. Journal of Systems and Software, no 86(1), pp. 72-88, (2013)
12. The Apache Software Foundation. What's Jena? https://jena.apache.org/about_jena/about.html. Accedido 26-04-2016
13. The Apache Software Foundation. Reasoners and rule engines: Jena inference support. https://jena.apache.org/documentation/inference/#reasonerAPI. Accedido 26-04-2016.
14. The Apache Software Foundation. Apache Jena Fuseki. https://jena.apache.org/ documentation/fuseki2/fuseki-run.html. Accedido 26-04-2016.