UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

CLAVES XML: UNA IMPLEMENTACIÓN DE ALGORITMOS DE IMPLICACIÓN Y VALIDACIÓN

EMIR FERNANDO MUÑOZ JIMÉNEZ

2010

UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

CLAVES XML: UNA IMPLEMENTACIÓN DE ALGORITMOS DE IMPLICACIÓN Y VALIDACIÓN

EMIR FERNANDO MUÑOZ JIMÉNEZ

Tesis de Grado presentada en conformidad a los requisitos para obtener el Grado de Magíster en Ingeniería Informática

Comisión integrada por los profesores: **Dr. Mauricio Marín**
Dr. Flavio Ferrarotti
Dr. Mauricio Solar
Dr. Jorge Pérez
Dr. Héctor Antillanca

Santiago de Chile
2010

AGRADECIMIENTOS

Esta tesis es fruto del esfuerzo, apoyo, y preocupación de muchas personas a las cuales me gustaría agradecer. Siempre seré un agradecido del apoyo y cariño fundamental e incondicional entregado por mis sobrinos Thaira, Axel y Diego, mis hermanas Yessenia y Daniela, y de mi incondicional madre Rosa, a quienes espero retribuir todo lo que me han dado. También, agradezco el apoyo de mis amigos desde el colegio: Davinia Cubillo, Eduardo Pérez, Joaquín Pavez, Diego Rojas, y Rodrigo Saud, a quienes les agradezco por su apoyo, y sus intentos por sacarme de la rutina y disfrutar de la vida. Y de aquellos que conocí en la Universidad: Pablo Torres, Miguel Navarro, Eduardo Miranda, Ivan Silva, y todo el grupo, con quienes compartí buenos y malos momentos, pero que finalmente nos han llevado a superar este gran desafío.

Agradezco a la Fundación San Marcelino Champagnat, por su apoyo durante todos mis estudios universitarios. A la Universidad de Santiago, y al Departamento de Ingeniería Informática, por haber sido mi segundo hogar durante todos estos años, y haberme permitido cursar los programas de Ingeniería Civil y Magíster en Informática. Particularmente, a la Sra. Soledad Miranda, la Srta. María Paz González, y la Srta. Pamela Campos por su buena disposición a responder todas mis consultas, y ayudarme en los asuntos anexos.

Especialmente tengo que agradecer a mis profesores guías, el Dr. Mauricio Marín y el Dr. Flavio Ferrarotti, por haber aceptado el desafío de guiarme al poco tiempo de conocerme, por todos los conocimientos y la ayuda entregada, y por las tantas y largas conversaciones mantenidas sin importar la distancia o el cambio horario. En ellos encontré un ejemplo de empeño y dedicación por la investigación y desarrollo del área a todo nivel, que tomaré como parte fundamental de mi desarrollo personal y profesional. También, agradezco al Dr. Diego Arroyuelo, por sus revisiones y apoyo prestado para entender varios de los conceptos de este trabajo.

A la Fundación para la Transferencia Tecnológica, especialmente al proyecto Yahoo Research Latin America, por el apoyo entregado durante el desarrollo de esta tesis. Al equipo humano del Laboratorio Yahoo en Chile, del cual he aprendido que las ganas por hacer bien las cosas implican superación. Y en quienes encontré grandes personas y amigos: Carolina Bonacic, Carlos Gómez, Alonso Inostrosa, Oscar Rojas y Alejandro Figueroa, gracias por compartir conmigo su experiencia, apoyarme, enseñarme, hacerme reír, y soportarme durante este último año, y por seguir haciéndolo durante un buen tiempo más.

Con todo mi corazón a mis queridos sobrinos,

hermanas, y a mi incondicional madre.

RESUMEN

La flexibilidad sintáctica, y el complejo anidamiento de los datos en una estructura tipo árbol dificulta expresar propiedades deseables de los datos XML, ofreciendo una capacidad limitada para expresar semántica. En esta tesis se presenta un estudio de las claves como restricciones de integridad sobre documentos XML, implementando algoritmos para los problemas de implicación y validación, con el fin de mostrar la factibilidad de usar las capacidades semánticas que éstas entregan, y que XML como modelo requiere.

Este trabajo presenta una perspectiva teórico-práctica del concepto de claves XML. Se implementa el algoritmo más eficiente conocido para su implicación, comprobándose la utilidad en la práctica de éste. Se diseña e implementa un algoritmo para la validación de documentos XML contra claves que complementa a los ya existentes. Además, se introduce la noción de *covers* no redundantes para claves XML, la cual es luego utilizada para optimizar el algoritmo de validación.

A partir de las implementaciones desarrolladas, se experimenta con diferentes conjuntos de datos, lo que permite comprobar las principales hipótesis. Primero, se observa un buen comportamiento en la práctica del algoritmo para el problema de implicación de claves XML con conjunto de caminos clave simples no vacíos. Y segundo, debido a que la validación de documentos XML contra claves depende en parte del tamaño del conjunto de claves, siempre es preferible calcular primero el *cover* no redundante del conjunto de claves XML, antes de realizar la validación. El tiempo que toma calcular el conjunto *cover* no redundante es despreciable en comparación al tiempo que toma la validación del documento contra una clave que sea implicada por las demás.

Las soluciones desarrolladas tienen un amplio rango de aplicación, permitiendo avanzar a futuro en trabajos con XML como validación de consistencia, diseño de esquemas, integración de datos, intercambio y limpieza de datos, optimización y reescritura de consultas, indexación, y respuesta consistente a consultas.

Palabras Claves: XML; Claves XML; Implicación de claves; Validación de documentos XML; Cover no redundante

ABSTRACT

The syntactic flexibility and complex tree-like nested data make it challenging to express desirable properties of XML data, offering a limited capability to express semantic. In this thesis, we present a study of keys as integrity constraints on XML documents, implementing algorithms for implication and validation problems, with the aim of showing the feasibility of using the semantic capabilities that keys gives and XML as a model requires.

This work presents a theory-practice perspective about XML keys concept. We implement the most efficient algorithm known for key implication proving its practical utility. We design and implement a XML document validation algorithm against a XML key set that complements the existing ones. Moreover, we introduce the non-redundant cover sets notion for XML keys, which is then used as an optimization for the validation algorithm.

From the implementations made we experimented with different datasets, which allow us to prove the main hypothesis. First, we observed a good behavior in practice of the algorithm for the implication problem of XML keys with nonempty sets of simple key paths. And in second place, due to XML document validation against keys partly depends on key set size, always is recommended calculate first the non-redundant cover set for XML key set before making the validation. The time for calculate the non-redundant cover set is inconsiderable compared with the time that takes to make the document validation against a key implied for the rest.

The developed solutions have a wide range of applications enabling future progress in working with XML as consistency validation, XML schema design, data integration, data exchange and cleaning, XML query optimization and rewriting, indexing, and consistent query answering.

Keywords: XML; XML keys; Key implication; XML document validation; Non-redundant cover

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ALGORITMOS

CAPÍTULO 1. INTRODUCCIÓN

1.1 ANTECEDENTES Y MOTIVACIÓN

El rápido desarrollo de la Web ha generado nuevos problemas y áreas de investigación en las Ciencias de la Computación e Informática. Junto a eso ha iniciado el desarrollo de (innumerables) nuevas tecnologías, y la evolución de otras. La comunicación e interoperabilidad de estas tecnologías es un tema crucial para mantener el desarrollo de la Web. En particular, XML (*eXtensible Markup Language* o Lenguaje extensible de marcado) ha surgido como un modelo de datos estándar para almacenar e intercambiar datos en la Web. Su rol en el intercambio de datos ha pasado de simplemente transmitir la estructura de los datos, a uno que también transmite su semántica (Benedikt et al., 2003; Davidson et al., 2007).

XML (Bray et al., 2006) es la propuesta del *World Wide Web Consortium* (W3C) como lenguaje de intercambio e interoperabilidad en la Web. Este lenguaje proporciona un alto grado de flexibilidad sintáctica, pero ofrece una capacidad limitada para expresar la semántica de los datos. Esta flexibilidad sintáctica, y el complejo anidamiento de los datos en una estructura tipo árbol, dificulta expresar propiedades deseables de los datos XML. En consecuencia, el estudio de restricciones de integridad ha sido reconocido como una de las áreas de investigación en XML más difíciles (Fan, 2005; Suciu, 2001; Vianu, 2003; Widom, 1999).

El estudio de restricciones de integridad ha sido reconocido como una de las áreas más difíciles de investigación en XML (Vianu, 2003). En el modelo relacional, las restricciones han sido estudiadas extensamente (Fagin & Vardi, 1984; Thalheim, 1991), y son esenciales para el diseño de esquemas, la optimización de consultas, y métodos eficientes de acceso y almacenamiento (Abiteboul et al., 1995). Varias clases de restricciones de integridad han sido definidas para XML, incluyendo claves (Buneman et al., 2002), restricciones de camino (Buneman et al., 2001, 2000), restricciones de inclusión (Fan & Libkin, 2002; Fan & Siméon, 2003), y dependencias funcionales

(Arenas & Libkin, 2004; Hartmann & Trinh, 2006; Vincent et al., 2004). Sin embargo, la mayoría de las clases de restricción, dada la compleja estructura de datos XML, resultan en problemas de decisión que son intratables, y es difícil encontrar clases de restricciones XML que sean naturales y útiles, y que puedan ser razonadas de manera eficiente (Fan, 2005; Fan & Siméon, 2003; Fan & Libkin, 2002; Suciu, 2001; Vianu, 2003; Arenas et al., 2002). Las principales candidatas de esas clases son las claves absolutas y relativas (Buneman et al., 2003, 2002) que son definidas en base a un modelo de árbol para XML como el propuesto por DOM (Apparao et al., 1998) y XPath (Clark & DeRose, 1999), de manera independiente a alguna especificación del tipo[1] de un documento XML como DTD[2] o XML *Schema* (Thompson et al., 2004).

Las claves son fundamentales para todo modelo de datos (Abiteboul et al., 1995; Ramakrishnan & Gehrke, 2002). Son usadas para describir la consistencia de los datos (restricciones), para realizar referencias a los datos (claves foráneas), y para actualizar datos de manera inequívoca. Permiten identificar datos de manera única sin ambigüedad y, por lo tanto, introducir métodos basados en valor para localizar elementos en un documento XML. Si se desea mejorar la semántica de los datos XML, la noción de claves debe ser estudiada en profundidad.

La expresividad entregada por las claves XML está relacionada con la elección de un lenguaje navegacional de caminos, y por la elección de una noción adecuada de igualdad en valor. En Buneman et al. (2003, 2002) las claves XML son definidas usando: (i) expresiones de camino formadas a partir de las etiquetas de los nodos, mediante recursivas aplicaciones de los operadores *child* y *descendant-or-self*, y (ii) subárboles isomorfos (identidad en cadenas de caracteres) como noción de igualdad en valor.

Debido a la compleja estructura de XML, la definición de claves no debiese estar restringida a cadenas de caracteres (valores de atributos), y debiese permitir la identificación de elementos dentro del ámbito de subdocumentos. Considerando esto, en Buneman et al. (2003, 2002) se tienen dos posibles contextos para una clave: si la clave se aplica a partir de la raíz del documento, su contexto es el documento completo, y se denomina *clave absoluta*; si se aplica a partir de algún otro elemento distinto de la raíz, su contexto es un subdocumento, y se denomina *clave relativa*.

Muchos de los aspectos de la propuesta de Buneman et al. (2003, 2002) han sido

[1] Un *tipo* en XML es considerado como una gramática extendida libre de contexto, asociada a restricciones en la estructura de los elementos de un documento.

[2] *Document Type Definition* o Definición de tipo de un documento XML.

adoptados dentro de XML *Schema* (Sperberg-McQueen & Thompson, 2000), reconociéndose la utilidad en la práctica de esta clase de claves XML. Es necesario considerar, que no existe la exigencia al momento de crear un documento XML, que éste cumpla con restricciones impuestas por algún XML *Schema* o DTD asociado, y gran parte de los documentos existentes no los tiene. Por lo tanto, la definición de claves debe ser independiente de algún esquema XML (Buneman et al., 2003, 2002; Hartmann & Link, 2009a).

Usando la definición de claves XML de Buneman et al. (2003, 2002), el reciente trabajo de Hartmann & Link (2009a) provee una axiomatización para una importante y expresiva clase de este tipo de claves XML. Caracterizando el problema de implicación de claves XML como un problema de alcanzabilidad de nodos en un grafo. Hartmann & Link (2009a) logran obtener un algoritmo que permite decidir la implicación de claves XML en tiempo cuadrático.

En la misma línea, el problema de validación de restricciones como las claves, ha sido muy estudiado en el modelo relacional, pero se ha visto que la validación de restricciones sobre documentos XML es más compleja de acuerdo a lo expuesto en los trabajos realizados en el área (Abrão et al., 2004; Alon et al., 2003, 2001; Bouchou et al., 2003; Chen et al., 2002; Liu et al., 2005, 2004; Suciu, 2002a).

A pesar, que la definición de claves ha comenzado a ser considerada tanto por XML *Schema* como por DTD, aún no existe una técnica estándar para validar que un documento XML satisfaga un conjunto de claves XML (Chen et al., 2002; Liu et al., 2005, 2004).

Abrão et al. (2004); Bouchou et al. (2003); Chen et al. (2002); Liu et al. (2005, 2004), han sido trabajos que considerando la definición de claves XML dada en Buneman et al. (2003, 2002), han presentado propuestas para validación de documentos XML. Chen et al. (2002) presenta un validador de documentos XML contra claves basado en un analizador de acceso secuencial para XML, utilizando *Simple API for XML* (SAX) (Megginson, 2004), basándose en una técnica de indexación de claves, y almacenando el documentos XML en una base de datos relacional. En Bouchou et al. (2003) y Abrão et al. (2004) utilizan un autómata árbol para validar un documento XML contra una clave XML. Liu et al. (2005, 2004) proponen otro enfoque, realizando la validación de un documento XML contra una claves mediante la traducción al problema de verificación estructural (tal como la revisión de un documento XML contra un DTD). En todos estos trabajos junto con utilizar la sintaxis para claves dada en Buneman et al. (2003, 2002), se asegura utilizar el concepto de igualdad en valor

propuesto en Buneman et al. (2003, 2002). Pero, hasta donde se sabe, los algoritmos presentados se restringen sólo a la igualdad de nodos texto (sólo permiten caminos clave asociados a un valor) (Davidson et al., 2008, 2007).

1.2 DESCRIPCIÓN DEL PROBLEMA

La definición de claves XML es más compleja que en el modelo relacional, debido a la compleja estructura de árbol que poseen los documentos XML.

En este trabajo se plantea, en primer lugar, determinar la utilidad de trabajar en la práctica con claves XML utilizando un algoritmo para el problema de implicación. Definir ésta utilidad práctica permitirá avanzar en la aceptación de las claves como restricciones sobre XML por parte de los profesionales, considerando el poder expresivo que estas entregan a XML. En segundo lugar, existe la necesidad de un método que permita determinar la validez de un documento XML contra un conjunto predefinido de claves XML. A partir de los trabajos realizados en validación de documentos XML contra claves (Abrão et al., 2004; Bouchou et al., 2003; Chen et al., 2002; Liu et al., 2005, 2004), se plantea diseñar un algoritmo que permita validar documentos XML contra claves XML como las definidas en Buneman et al. (2003, 2002), las cuales consideran la igualdad en valor entre nodos elemento: si los subárboles que tienen por raíz a estos nodos, son isomorfos por algún isomorfismo que para cadenas de texto se corresponde con la función identidad.

Finalmente, considerando que la complejidad del algoritmo de validación depende en parte del tamaño del conjunto de claves, se investiga un método para obtener una optimización del proceso de validación de documentos XML contra claves, utilizando el algoritmo de implicación de claves XML presentado por Hartmann & Link (2009a).

1.3 SOLUCIÓN PROPUESTA

Para dar solución al primer problema planteado, este trabajo presenta una implementación del algoritmo de implicación de claves XML con conjunto de caminos clave simples no vacío, y el desarrollo de experimentos con ella, considerando claves con diferentes características.

Determinando qué características son las más influyentes en los tiempos de implicación.

El segundo problema, se aborda, siguiendo parte de la estrategia utilizada en Liu et al. (2005, 2004) expresando las claves como consultas navegacionales de tipo XPath[3] sobre un árbol XML según lo planteado por DOM (Apparao et al., 1998); y respetando la igualdad en valor definida en Buneman et al. (2003, 2002), la cual no se restringe a la igualdad en texto o atributos.

Para el tercer problema, aprovechando la implementación del algoritmo de implicación de claves XML, se aplica la noción de *covers* no redundantes de claves XML, la cual se basa en la noción de *covers* no redundantes del modelo relacional (Maier, 1983, 1980). Esto en el modelo relacional permite obtener conjuntos reducidos de dependencias con igual clausura semántica, y en XML permitirá obtener conjuntos reducidos de claves XML. Esto motiva a analizar la utilidad en términos de optimización del cálculo de conjuntos *covers* no redundantes antes de validar un documento contra un conjunto de claves XML. Se procede de la siguiente manera: (1) se realiza la validación de un documento contra un conjunto de claves XML, obteniendo un determinado tiempo; luego, (2) se calcula el conjunto *cover* de claves (el cual contiene igual o menor número de claves) y (3) se realiza nuevamente la validación contra este conjunto. Obteniendo una disminución del tiempo de validación, al comparar ambos tiempos.

Para finalizar, se experimenta con las implementaciones desarrolladas para cada método, para lograr caracterizaciones de los peores y mejores casos, y así poder resaltar cuales son las variables que influyen en cada uno. Específicamente, en este trabajo se desarrollan implementaciones para los algoritmos de implicación de claves XML, validación de documentos XML contra claves, y obtención de *covers* no redundantes de claves XML.

1.4 OBJETIVOS Y ALCANCE DEL PROYECTO

1.4.1 Objetivo general

Demostrar empíricamente la utilidad de las capacidades semánticas que entregan las claves XML, mediante el desarrollo de *software* para los problemas de implicación y validación, para las claves XML definidas en Buneman et al. (2003, 2002), y estudiadas en Hartmann & Link (2009a).

[3]XPath, junto con XQuery son los lenguajes de consulta más populares para XML.

1.4.2 Objetivos específicos

Para la consecución del objetivo general, se plantean las siguientes metas intermedias:

1. Estudiar la noción de claves XML y las axiomatizaciones existentes.
2. Desarrollar la implementación de un algoritmo para el problema de implicación de claves XML.
3. Estudiar las propuestas para validación de documentos XML contra conjuntos de claves XML.
4. Diseñar un algoritmo de validación de documentos XML contra conjuntos de claves XML, que considere la igualdad en valor no restringida a texto.
5. Analizar la utilidad de la implementación del algoritmo de implicación de claves XML como un método de optimización del proceso de validación de documentos XML contra claves.
6. Desarrollar un validador de documentos XML contra claves, respetando la noción de igualdad en valor, en la cual dos nodos elemento son iguales en valor, si los subárboles a partir de ellos son isomorfos por un isomorfismo que para cadenas de texto se corresponde con la función de identidad.
7. Identificar y definir, colecciones de documentos XML y de claves XML, que sirvan como instancias de prueba para experimentar con las implementaciones desarrolladas.
8. Caracterizar las instancias de prueba a través de experimentos, determinando patrones y variables relevantes, que mejoren o empeoren complejidad temporal de los algoritmos trabajados.

1.4.3 Alcances

El presente trabajo considera principalmente la implementación de dos métodos, uno que decide la implicación de claves XML, y otro que determina la validez de un documento XML contra un conjunto de claves. La primera implementación, basada en el algoritmo propuesto en Hartmann & Link (2009a) decide la implicación para el fragmento de claves con conjunto de caminos

clave simples no vacío. Este fragmento tiene la particularidad de representar un conjunto muy relevante de la clase de claves XML definidas en Buneman et al. (2003, 2002) y al mismo tiempo poseer un algoritmo cuadrático para el problema de implicación (Hartmann & Link, 2009a). La implementación del método de validación, considera documentos XML de tamaño no superior a 25 MB, debido a las restricciones impuestas por el *framework* y las APIs que se utilizan, y las operaciones realizadas sobre estos documentos. De todas formas este tamaño de documentos es suficiente para los objetivos de experimentación propuestos. Se aplica la noción de *covers* no redundantes de conjuntos de claves XML, para optimizar el tiempo de ejecución del algoritmo de validación de documentos XML contra claves. Dado que el algoritmo implementado para el problema de implicación de claves XML se restringe a claves en el fragmento de claves XML con conjuntos de caminos simples no vacíos, esta optimización mediante el cálculo de *covers* no redundantes, sólo es aplicable a esta clase de claves XML. De todas formas, sin utilizar esta optimización, el algoritmo de validación puede validar un documento XML contra cualquier conjunto de claves en la clase definida en Buneman et al. (2003, 2002).

No se considera en este trabajo temas como claves foráneas en XML o la validación incremental de documentos XML.

Ambas implementaciones funcionan con entradas de texto con un formato específico, que se definen dentro de este trabajo. La validación de cada una de las implementaciones se realiza contra los ejemplos manejados en la bibliografía. Las implementaciones funcionan mediante línea de comandos, facilitando su posible inclusión en otros proyectos como bibliotecas externas.

La experimentación se realiza sobre un computador personal, y no se considera necesaria la utilización de máquinas de mayor poder de cómputo.

1.5 METODOLOGÍA Y HERRAMIENTAS UTILIZADAS

1.5.1 Metodología

El presente trabajo se relaciona con diferentes áreas de las Ciencias de la Computación e Informática tales como, teoría de bases de datos, XML, teoría de complejidad computacional, teoría

de grafos, y estructuras de datos. Se plantea un trabajo teórico-práctico que permita incrementar el conocimiento y uso de claves como restricciones en XML. Para lograr los objetivos propuestos, se sigue una metodología definida especialmente para este trabajo, que se pasa a describir a continuación.

La metodología de trabajo a seguida a lo largo de este trabajo, se compone de los siguientes pasos:

1. Recolección y selección del material bibliográfico sobre restricciones de integridad en XML que ha sido publicado.

2. Focalización por tema (implicación y validación de claves XML), repasando la bibliografía seleccionada y, redactando resúmenes y borradores.

3. Diseño e implementación de algoritmos de implicación de claves XML y validación de documentos XML.

4. Definición de las instancias de prueba, a partir de *benchmarks* existentes, y de ejemplos utilizados en la bibliografía.

5. Diseño de los experimentos para la comprobación de las hipótesis planteadas.

6. Evaluación experimental con las implementaciones de implicación y validación.

7. Caracterización de los resultados obtenidos en los experimentos.

La metodología considera de manera paralela, reuniones presenciales y no presenciales con los profesores guía del tema. Además, en cada paso se obtendrá un hito o entregable, que marcará un avance en el trabajo, dándose conjuntamente cumplimiento a los objetivos propuestos para el trabajo.

Para la realización de las implementaciones, se utilizará un modelo de desarrollo espiral (Boehm, 1986), que permita la inclusión de tareas o actividades no definidas *a priori*, enfocándose en mitigar el riesgo de aquellas tareas de mayor complejidad en trabajos de tipo teórico-práctico. Con este modelo, se reducen los riesgos del proyecto, y se asegura un cumplimiento exitoso de los objetivos. Las reuniones consideradas, ayudan a suplir la desventaja de requerir experiencia en la evaluación de los riesgos que se pueden presentar. Este modelo determina cuatro regiones, la determinación

de objetivos, el análisis de riesgos, el desarrollo y las pruebas, y la planificación para la siguiente iteración. Las cuales son aplicables a cada uno de los seis pasos descritos.

1.5.2 Herramientas de desarrollo

Para lograr los objetivos propuestos en este trabajo se requiere el soporte de algunas herramientas de *hardware* y *software*. En *hardware*, se requiere de un computador personal para el desarrollo y pruebas de las implementaciones de los algoritmos. Se cuenta con un equipo Intel(R) Core(TM) 2 Duo T7250 de 2.0 GHz, con 3 GB de memoria RAM y, un disco duro de 160 GB con velocidad de 5400 RPM. En cuanto a *software* se requiere de herramientas para el desarrollo de las implementaciones, y la generación de la documentación. Se cuenta con una distribución Linux x86, con kernel 2.6.32 como sistema operativo; para el desarrollo, se utiliza el compilador *g++* de GCC[4] para implementar la implicación de claves, y el *framework* Java versión 6 para implementar la validación de documentos XML, utilizando el IDE Netbeans 6.8; para la generación de documentación e informes se cuenta con LaTeXversión LaTeX 2_ε, Xfig 3.2.5, Gnuplot 4.2.6, Dia 0.97.1, y Doxygen 1.6.3.

1.6 RESULTADOS OBTENIDOS

Con el presente trabajo, se ha logrado la satisfacción de los principales objetivos planteados desde un comienzo. En primer lugar, se destaca el desarrollo de una implementación eficiente que permite decidir el problema de implicación de claves XML con conjunto de caminos clave simples no vacío. De esta manera se comprueba la efectividad práctica de la implicación de claves en el modelo de datos XML. Esta implementación permitirá, dado un conjunto de claves, determinar si una clave externa es implicada por este conjunto. Pudiendo ser utilizada a futuro en otras áreas de investigación en XML como por ejemplo, validación de consistencia, diseño de esquemas, integración de datos, intercambio y limpieza de datos, optimización y reescritura de consultas, indexación, y respuesta consistente a consultas (Chomicki, 2007; Davidson et al., 2007; Fan, 2005).

[4]GNU Compiler Collection, http://gcc.gnu.org/

El segundo resultado, está dado por el diseño y desarrollo de un algoritmo, que permite la validación de un documento XML contra un conjunto de claves. Esta implementación, permite dado un documento, y un conjunto de claves XML determinar si este documento es válido ante las restricciones impuestas por cada una de las claves del conjunto. Además, haciendo uso de la primera implementación (para implicación), se logra diseñar e implementar un módulo de optimización de este proceso, a través, del cálculo de conjuntos *covers* no redundantes para claves XML con un conjunto de caminos clave simples no vacío.

Todas las implementaciones desarrolladas, tienen un diseño modular independiente, que les permite ser agregadas a futuras aplicaciones como bibliotecas externas.

La experimentación llevada a cabo con estas implementaciones, ha permitido comprobar la viabilidad y utilidad de razonar sobre claves XML, permitiendo aportar con sus capacidades semánticas al desarrollo de la tecnología y modelo de datos XML.

1.7 ORGANIZACIÓN DEL DOCUMENTO

El presente trabajo está dividido en ocho capítulos considerando éste como el primero. En el Capítulo 2 se formalizan los fundamentos de documento XML, modelo de árbol XML, y lenguaje de definición de expresiones de camino para definir claves XML. La noción principal de clave XML es presentada en el Capítulo 3, junto con la axiomatización (válida y completa) existente para claves XML y el problema de implicación de claves en la clase definida en Hartmann & Link (2009a). Este último algoritmo de la literatura, caracteriza el problema de implicación de claves XML como un problema de alcanzabilidad de nodos en un grafo. A partir de lo anterior, la contribución de este trabajo comienza con la implementación del algoritmo de implicación para claves XML (Hartmann & Link, 2009a), presentada en detalle en el Capítulo 4. Luego, en el Capítulo 5 se presenta un algoritmo de validación de documentos XML contra claves con conjunto de caminos clave simples no vacío, optimizado mediante la introducción de la noción de *covers* no redundantes para conjuntos de claves XML (que hasta ahora sólo se manejaba en el modelo relacional). A partir de estos últimos algoritmos, siguiendo con el objetivo de realizar un análisis empírico, se implementa un validador de documentos XML, y los detalles de su implementación se presentan en el Capítulo 6. Con las

implementaciones desarrolladas se realizan experimentos detallados en el Capítulo 7, y también se describen las principales instancias de claves XML y documentos XML usadas en estos experimentos. Finalmente, en el Capítulo 8 se presentan las conclusiones a las cuales ha arribado en este trabajo, y se delinean los trabajos futuros. Las instancias de prueba restantes utilizadas son presentadas en los Anexos A, B, y C.

CAPÍTULO 2. MARCO TEÓRICO

En este capítulo, se presenta una introducción a los conceptos de documentos XML, modelo de árbol XML, y lenguajes para definir expresiones de camino, que permitan la definición de claves XML. Las claves XML estudiadas aquí son definidas sobre un modelo de árbol XML como el propuesto por DOM (Apparao et al., 1998) y XPath (Clark & DeRose, 1999).

2.1 DOCUMENTOS XML

XML es el acrónimo de *eXtensible Markup Language* o Lenguaje extensible de marcado. Es un formato de texto simple y flexible, que al igual que HTML (*HyperText Markup Language* o Lenguaje de Marcado de Hipertexto), deriva a partir de SGML (*Standard Generalized Markup Language* o Lenguaje de marcado generalizado estándar para el formato de documentos). XML es utilizado principalmente como formato de datos para la interoperabilidad de aplicaciones, con el fin de facilitar la integración de información desde múltiples y diferentes fuentes, especialmente para el intercambio de datos en la Web.

En la Figura 2.1 se muestra un documento XML y su representación de árbol. Este documento presenta de manera simple la estructura de un libro, con el título principal, y los capítulos y sus correspondientes títulos. Todo documento XML se compone de etiquetas (*tags*), los cuales pueden ser *etiquetas de apertura* o *etiquetas de cierre*, tales como `<libro>, <capitulo>`, y `</libro>, </capitulo>`, respectivamente. Estas etiquetas deben estar balanceadas, es decir, siempre que exista una etiqueta de apertura `<x>`, existirá otra de cierre `</x>`. De esta forma, son utilizadas para delimitar elementos. Las etiquetas pueden anidar a otras etiquetas, por tanto, todo elemento puede contener texto, otros elementos, o una mezcla de ellos. Por ejemplo, el elemento delimitado por `<libro>` contiene a otros tres elementos delimitados por `<titulo>`, `<capitulo>`, y

```
<?xml version="1.0" encoding="UTF-8"?
<libro>
  <titulo>El Juego</titulo>
  <capitulo id="cap1">
    <titulo>Parte 1</titulo>
  </capitulo>
  <capitulo id="cap2">
    <titulo>Parte 2</titulo>
  </capitulo>
</libro>
```

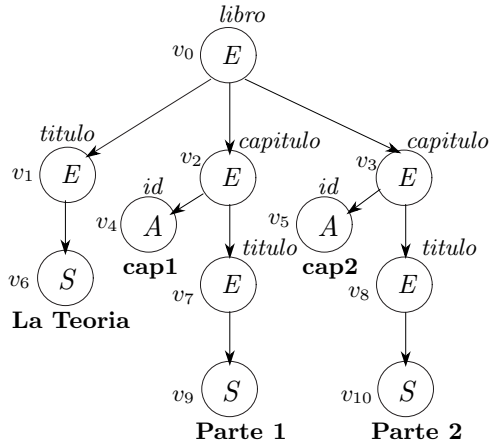


FIGURA 2.1: *Modelo de árbol para un documento XML.*

`<capitulo>`. Se dice que estos son subelementos de `<libro>`. También, los elementos pueden contener *atributos*, tal como `id` en `<capitulo id="cap1">`.

Un documento XML posee una estructura anidada, lo cual le da flexibilidad al momento de almacenar información. Para especificar la estructura de documentos XML, se puede (no es exigencia) especificar, al igual que en el modelo relacional, un *esquema*. No existe un estándar para la definición de esquemas. Las propuestas que predominan son DTD (Goldfarb, 1990) y XML *Schema* (Sperberg-McQueen & Thompson, 2000).

2.2 EL MODELO DE ÁRBOL XML

Dada la estructura jerárquica, todo documento XML puede ser visto como un árbol.

Se asume la existencia de un conjunto infinito contable **E** que denota etiquetas de elementos, un conjunto infinito contable **A** que denota nombres de atributos, y un conjunto unitario $\{S\}$ que denota texto (PCDATA). Se asume además que estos conjuntos son disjuntos tomados de a pares. Se define el conjunto $\mathcal{L} = \mathbf{E} \cup \mathbf{A} \cup \{S\}$, donde cada elemento de $\mathcal{L}$ es llamado *etiqueta*.

Un *árbol XML* es una 6-tupla $T = (V, lab, ele, att, val, r)$, donde V es el conjunto de nodos

de T, y lab es una función $V \to \mathcal{L}$, que asigna una etiqueta a cada nodo en V. Un nodo $v \in V$ es llamado *nodo elemento* si $lab(v) \in \mathbf{E}$, *nodo atributo* si $lab(v) \in \mathbf{A}$, o *nodo texto* si $lab(v) = S$. ele y att son funciones parciales que definen la relación de aristas de T: para todo nodo $v \in V$, si v es un nodo elemento, entonces $ele(v)$ es una lista de nodos elemento y texto en V, y $att(v)$ es un conjunto de nodos atributo en V. Si v es un nodo atributo o texto, entonces los valores de $ele(v)$ y $att(v)$ están indefinidos. La función parcial val asigna una cadena de texto (*string*) a cada nodo atributo o texto: para todo $v \in V$, $val(v)$ es una cadena de texto si v es un nodo atributo o texto, mientras $val(v)$ está indefinido en cualquier otro caso. Finalmente, r distingue el (único) nodo raíz de T. Se dice que T es *finito* si V es finito, y se dice que T es *vacío* si $V = \{r\}$.

Ejemplo 2.1 La Figura 2.1 muestra un documento XML en su parte superior, y su respectiva representación de árbol en la parte inferior. El árbol contiene un conjunto de nodos $V = \{v_i \mid i = 0, \ldots, 10\}$, donde el nodo $v_0 = r$ corresponde al nodo raíz; donde las etiquetas de, los nodos elemento son:

$$\begin{array}{lll} lab(v_0) = libro & lab(v_1) = titulo & lab(v_2) = capitulo \\ lab(v_3) = capitulo & lab(v_7) = titulo & lab(v_8) = titulo \end{array}$$

los nodos atributo: $lab(v_4) = id$, $lab(v_5) = id$, y nodos texto: $lab(v_6) = S$, $lab(v_9) = S$, $lab(v_{10}) = S$. También se tienen los valores asociados a cada nodo, donde $val(v_i)$ está indefinido para $i = 0, 1, 2, 3, 7, 8$, y

$$\begin{array}{lll} val(v_4) = \text{cap1} & val(v_5) = \text{cap2} & val(v_6) = \text{ La Teoria} \\ val(v_9) = \text{Parte 1} & val(v_{10}) = \text{Parte 2} & \end{array}$$

La estructura del árbol está dada por las funciones ele y att:

$$\begin{array}{ll} ele(v_0) = [v_1, v_2, v_3] & att(v_0) = \emptyset \\ ele(v_1) = [v_6] & att(v_1) = \emptyset \\ ele(v_2) = [v_7] & att(v_2) = \{v_4\} \\ ele(v_3) = [v_8] & att(v_3) = \{v_5\} \\ ele(v_4) = \text{indefinido} & att(v_4) = \text{indefinido} \\ ele(v_5) = \text{indefinido} & att(v_5) = \text{indefinido} \\ ele(v_6) = \text{indefinido} & att(v_6) = \text{indefinido} \\ ele(v_7) = [v_9] & att(v_7) = \emptyset \\ ele(v_8) = [v_{10}] & att(v_8) = \emptyset \\ ele(v_9) = \text{indefinido} & att(v_9) = \text{indefinido} \\ ele(v_{10}) = \text{indefinido} & att(v_{10}) = \text{indefinido} \end{array}$$

Un *camino* p en un árbol XML T es una secuencia finita de nodos $v_0, \ldots, v_m$ pertenecientes al conjunto V tal que (v_{i-1}, v_i) es una arista de T para todo $i = 1, \ldots, m$. Se llamará p a un camino desde v_0 hasta v_m, y se dice que v_m es *alcanzable* desde v_0 siguiendo el camino p. El camino p determina una palabra $lab(v_1) \ldots lab(v_m)$ sobre el alfabeto $\mathcal{L}$, denotado por $lab(p)$. Para un nodo $v \in V$, cada nodo w alcanzable desde v es llamado un *descendiente* de v. Para todo nodo $v \in V$, hay un único camino desde el nodo raíz r hasta v, debido a la estructura de árbol. Por ejemplo, en la Figura 2.1 el nodo v_4 con $lab(v_4) = id$ es descendiente de v_2 con $lab(v_2) = capitulo$, que a su vez es descendiente de $v_0 = r$ con $lab(v_0) = libro$; por tanto, v_4 es también descendiente de v_0.

2.3 IGUALDAD EN VALOR DE NODOS EN UN ÁRBOL XML

Para la definición de claves sobre árboles XML, es necesario definir un concepto equivalente al concepto de igualdad de valores atómicos en bases de datos relacionales. Es claro que este concepto de igualdad de valores no debe restringirse sólo a igualdad de nodos texto. Por ejemplo, si se desea utilizar como clave un nodo elemento con etiqueta "capitulo" que tiene como hijos un nodo con etiqueta "id" y un nodo con etiqueta "titulo", se necesita alguna regla para determinar cuando dos nodos con etiqueta "capitulo" son iguales en valor. En este trabajo, se considera que, dos nodos u y v de un árbol XML T son iguales en valor, si tienen la misma etiqueta, y además, si son nodos texto o atributo, tienen la misma cadena de texto o, en el caso de que los nodos sean nodos elemento, sus hijos son iguales en valor tomados de a pares. Más formalmente, dos nodos $u, v \in V$ son *iguales en valor* (denotado por $u =_v v$) si y sólo si, los subárboles con raíz u y v son isomorfos por un isomorfismo que para cadenas de texto se corresponde con la función de identidad. En otras palabras, dos nodos $u, v \in V$ son *iguales en valor* ($u =_v v$) cuando se satisfacen las siguientes condiciones:

(a) $lab(u) = lab(v)$,

(b) si u, v son nodos atributo o texto, entonces $val(u) = val(v)$,

(c) si u, v son nodos elemento, entonces:

  (i) si $att(u) = \{a_1, \ldots, a_m\}$, entonces $att(v) = \{a'_1, \ldots, a'_m\}$ y existe una permutación π sobre $\{1, \ldots, m\}$ tal que $a_i =_v a'_{\pi(i)}$ para $i = 1, \ldots, m$, y

  (ii) si $ele(u) = [u_1, \ldots, u_k]$, entonces $ele(v) = [v_1, \ldots, v_k]$ y $u_i =_v v_i$ para $i = 1, \ldots, k$.

Como se puede ver la noción de igualdad en valor toma en cuenta el orden del documento XML. Por ejemplo, en la Figura 2.2 el primer y tercer nodo *persona* (**P1** y **P3** de acuerdo al orden del documento) son iguales en valor. Dado que la igualdad en valor entre dos nodos cualesquiera de un árbol XML se corresponde con la existencia de un isomorfismo entre los subárboles con raíz en estos nodos, es claro que $=_v$ es una relación de equivalencia en el conjunto V de nodos de un árbol XML.

2.4 EXPRESIONES DE CAMINO PARA SELECCIÓN DE NODOS EN ÁRBOLES XML

Para definir claves sobre árboles XML, es necesario especificar un mecanismo para la selección de nodos. En este sentido, las expresiones de camino (también conocidas como consultas de camino) son una herramienta muy útil que ha sido muy utilizada tanto en la teoría como en la práctica (Clark & DeRose, 1999; Suciu, 2001). Las expresiones de camino se construyen usando las etiquetas de los nodos como alfabeto y describen una secuencia de aristas padre-hijo en un árbol XML. La respuesta a una consulta de camino evaluada en un nodo v sobre un árbol XML T, esta formada por el conjunto de nodos alcanzados desde v, siguiendo un camino en T que concuerda con el de la consulta. Se definirán a continuación los lenguajes de camino PL y PL_s. Estos lenguajes de camino tienen la particularidad de ser lo suficientemente expresivos como para definir claves XML de utilidad en la práctica, y lo suficientemente simples como para que el problema de implicación de claves XML definidas mediante estos lenguajes sea tratable (Buneman et al., 2002, 2003; Hartmann & Link, 2009a,b).

La siguiente gramática define la sintaxis del lenguaje de caminos PL,

$$P \to \ell \mid \varepsilon \mid P.P \mid \_^* \tag{2.1}$$

donde ℓ es una etiqueta en $\mathcal{L}$, ε denota el camino vacío (o palabra vacía), "." es el operador binario usado para concatenar dos expresiones de camino, y "$\_^*$" el comodín de longitud variable que representa cualquier secuencia (posiblemente vacía) de etiquetas.

Sean P, Q dos palabras de PL, P es un *refinamiento* de Q, denotado por $P \lesssim Q$, si P puede obtenerse a partir de Q reemplazando comodines en Q por palabras en PL. Por ejemplo,

```
<?xml version="1.0" encoding="UTF-8"?>
<db>
  <proyecto>
    <pnombre>Phobia</pnombre>
    <equipo>
      <persona>
        <idpersona>16.269.030-2</idpersona>
        <nombre>Juan</nombre>
        <apellido>Perez</apellido>
      </persona>
      ...
    </equipo>
    <jefe>
        <idpersona>11.435.940-1</idpersona>
        <nombre>Eduardo</nombre>
        <apellido>Jofre</apellido>
    </jefe>
  </proyecto>
  <proyecto>
    <pnombre>MediaLow</pnombre>
    <equipo> ... </equipo>
    <jefe> ... </jefe>
  </proyecto>
</db>
```

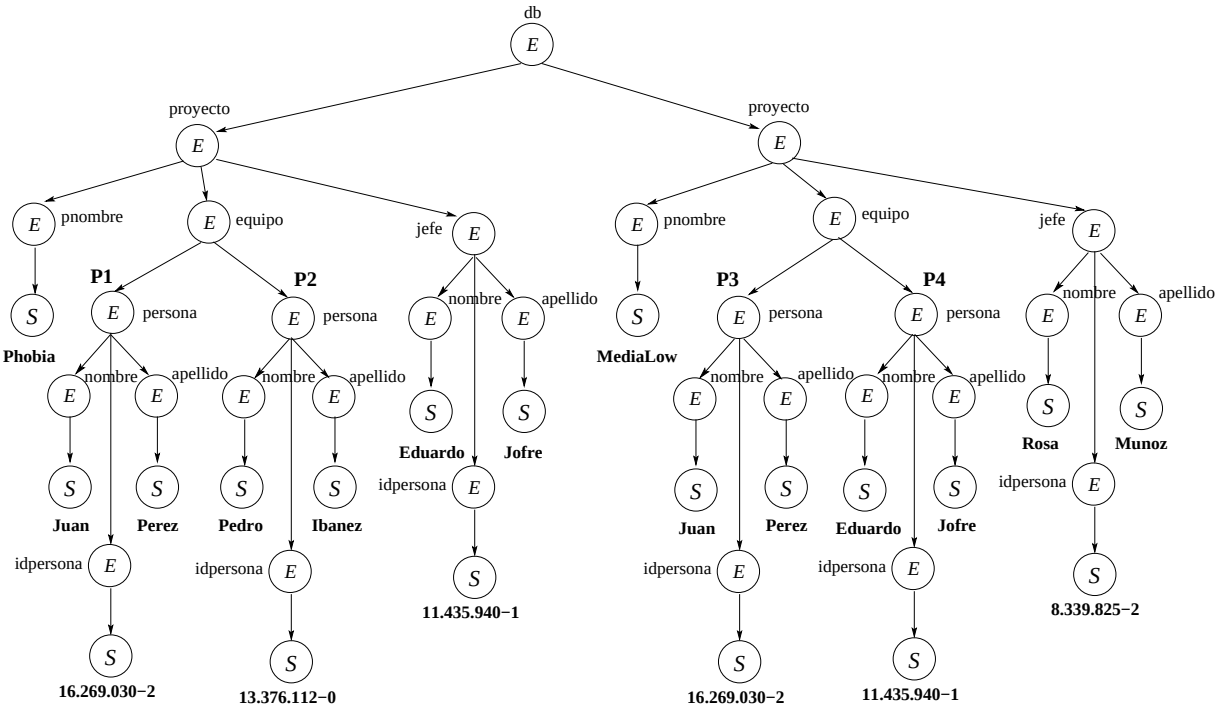


FIGURA 2.2: *Documento XML de proyectos y su representación de árbol.*

libro.capitulo.titulo es un refinamiento de *libro._*.titulo*. Nótese que la relación $\lesssim$ es un preorden en PL. Si $\sim$ denota la congruencia inducida en PL por la identidad $\_^*.\_^* = \_^*$, entonces $P \sim Q$ si y sólo si, $P \lesssim Q$ y $Q \lesssim P$.

A continuación se define la semántica de PL. Sea Q una expresión en PL, un camino p en el árbol XML T es llamado *camino-Q* si $lab(p)$ es un refinamiento de Q. Se usará $T \models Q(v, w)$, donde $v, w \in V$, para denotar que w es alcanzable desde v siguiendo un camino-Q en T. Por ejemplo, en el árbol de la Figura 2.2 todos los nodos *nombre* son alcanzables desde el nodo raíz siguiendo los caminos *proyecto.equipo.persona.nombre* y *proyecto.jefe.nombre*. Desde luego, también son alcanzables desde el nodo raíz siguiendo el camino *proyecto._*.nombre*, o el camino *_*.nombre*. Para $v \in V$, $v[\![Q]\!]$ denota el conjunto de nodos en T que son alcanzables desde v siguiendo un camino-Q, esto es, $v[\![Q]\!] = \{w \mid T \models Q(v, w)\}$. Se usará $[\![Q]\!]$ como abreviación de $r[\![Q]\!]$ donde r es el nodo raíz de T. Luego, $[\![Q]\!]$ denota el conjunto de todos los nodos en T alcanzables desde el nodo raíz r. Como ejemplo, sea v el segundo nodo *proyecto* en el orden determinado por el documento XML de la Figura 2.2. Luego $v[\![\_^*.nombre]\!]$ denota el conjunto de nodos *nombre* que son descendientes del segundo nodo *proyecto*, mientras que $[\![\_^*.nombre]\!]$ denota el conjunto de nodos *nombre* en todo el árbol XML.

El lenguaje de caminos PL_s es el sublenguaje de PL generado por la gramática obtenida eliminando el símbolo comodín "$\_^*$" de la gramática de PL. Por lo tanto, PL_s es simplemente el conjunto de palabras sobre el alfabeto $\mathcal{L}$, mientras que PL es el conjunto de palabras sobre el alfabeto $\mathcal{L} \cup \{\_^*\}$. Ambos lenguajes incluyen la palabra vacía ε.

Dado que todo nodo atributo o texto en un árbol XML T es una hoja, una palabra Q en PL es considerada *válida* si no posee etiquetas $\ell \in \mathbf{A}$ o $\ell = S$ en otra posición que no sea la última. Notar que cada prefijo de una palabra Q válida, es también una palabra válida.

Sean P, Q dos palabras de PL, se dice que P está *contenida* en Q, denotado por $P \subseteq Q$, si para todo árbol XML T y para todo nodo v de T, se tiene que $v[\![P]\!] \subseteq v[\![Q]\!]$. Como consecuencia directa de la definición de refinamiento, se tiene que $P \lesssim Q$ implica $P \subseteq Q$. En Buneman et al. (2003) se demuestra que, si P y Q son palabras válidas en PL, entonces $P \subseteq Q$ si y sólo si, P es un refinamiento de Q, y que para PL existe un algoritmo que decide este problema en tiempo cuadrático.

Una palabra de PL está en *forma normal* si ésta no tiene comodines consecutivos (es decir, $\_^*.\_^*$). Cada clase de congruencia contiene una única palabra en forma normal. Cada palabra de PL puede ser transformada a su forma normal en tiempo lineal, sólo eliminando los comodines

sobrantes. En particular, cada palabra de PL_s se encuentra en forma normal. El *largo* $|Q|$ de una expresión Q de PL es el número de etiquetas en Q, más el número de $\_^*$ en la forma normal de Q (Buneman et al., 2003). La expresión de camino vacío ε tiene largo 0.

Nótese que existe una sencilla conversión de expresiones PL a expresiones XPath (Clark & DeRose, 1999), sólo es necesario reemplazar "$\_^*$" por ".//." y "." por "/".

Se usará en este trabajo la noción de intersección en valor introducida en Buneman et al. (2003): para los nodos v y v' de un árbol XML T, la *intersección en valor* de $v[\![Q]\!]$ y $v'[\![Q]\!]$ está dada por,

$$v[\![Q]\!] \cap_v v'[\![Q]\!] = \{(w, w') \mid w \in v[\![Q]\!],\, w' \in v'[\![Q]\!],\, w =_v w'\} \tag{2.2}$$

esto significa que, $v[\![Q]\!] \cap_v v'[\![Q]\!]$ consiste de todos los pares de nodos en T que son iguales en valor, y que son alcanzables desde v y v' respectivamente, siguiendo caminos-Q.

CAPÍTULO 3. CLAVES XML, AXIOMATIZACIÓN E IMPLICACIÓN

En este capítulo se revisa el concepto más popular de restricción XML, las claves XML. Una clave XML permite identificar nodos de manera unívoca, ya sea en el árbol completo o relativo a algún subárbol seleccionado, en base a los valores que poseen los nodos descendientes. En este contexto las claves son definidas en términos de expresiones de camino. Dependiendo del tipo de expresiones de camino utilizadas, se tiene el nivel de expresividad de una clave XML. Se revisará además, la axiomatización válida y completa conocida para un fragmento dado de claves XML, introducida en Hartmann & Link (2009a).

Además, en este capítulo se presenta y analiza el algoritmo (también introducido en Hartmann & Link (2009a)) para decidir la implicación de claves en $\mathcal{K}^{PL}_{PL,PL^+_s}$. Este algoritmo se deriva de la técnica utilizada para comprobar la completitud del conjunto de reglas $\Re$ de la axiomatización, utilizando las herramientas *mini-tree* y *witness-graph*.

3.1 CLAVES XML

Las claves son un aspecto esencial del diseño de bases de datos, dan la capacidad de identificar un dato de manera unívoca. Como se dice en el Capítulo 2, los datos XML son típicamente representados como un árbol con nodos etiquetados (Neven, 2002). En este capítulo se muestra que una clave es definida en función de expresiones de camino. Así, las claves XML son más complejas que las restricciones sobre el modelo relacional.

Siguiendo la definición dada en Buneman et al. (2002), a continuación se define formalmente el concepto de clave XML.

Definición 3.1 Una *clave XML* φ en la clase $\mathcal{K}$ de claves XML, es una expresión de la forma

$$(Q, (Q', \{P_1, \ldots, P_k\})) \tag{3.1}$$

donde Q y Q' son expresiones en PL, y para $1 \leq i \leq k$, P_i es una expresión en PL_s, tal que $Q.Q'.P_i$ es una expresión PL válida. Desde ahora, Q será llamado *camino de contexto*, Q' será llamado *camino objetivo*, y los $P_1, \ldots, P_k$ serán llamados *caminos clave* de φ. Si $Q = \varepsilon$, se debe referir a φ como *clave absoluta*; en otro caso, se debe referir a φ como *clave relativa*.

Mientras que en las bases de datos relacionales las claves identifican de manera unívoca una tupla en una relación, una clave XML identifica de manera unívoca, mediante los valores asociados a los caminos claves a un nodo en el conjunto objetivo, dentro de un camino de contexto.

En este trabajo, se usará la notación $\mathcal{K}^{PL}_{PL,PL_s^+}$ para identificar la clase de claves XML con la cual se trabajará. El superíndice PL de $\mathcal{K}^{PL}_{PL,PL_s^+}$ indica que cualquier expresión en PL puede ser utilizada como camino de contexto. Los subíndices PL y PL_s de $\mathcal{K}^{PL}_{PL,PL_s^+}$ indican que cualquier expresión en PL puede ser utilizada como camino objetivo, y que sólo expresiones en PL_s pueden ser utilizadas como caminos clave. Se considera además, que el conjunto de caminos clave es finito y no vacío, lo cual queda indicado en el uso del símbolo "+" en el segundo subíndice (PL_s^+).

Dada una clave φ, se usará Q_φ para denotar su camino de contexto, Q'_φ para denotar su camino objetivo, y $P_1^\varphi, \ldots, P_{k_\varphi}^\varphi$ para denotar sus caminos clave, con k_φ como el número total de caminos clave en φ. Luego, el tamaño de una clave φ, denotado por $|\varphi|$, es definido como la suma de los largos de todas las expresiones de camino existentes en φ. El tamaño de una clave φ se obtiene según la Ecuación 3.2.

$$|\varphi| = |Q_\varphi| + |Q'_\varphi| + \sum_{i=1}^{k_\varphi} |P_i^\varphi| \tag{3.2}$$

Definición 3.2 [Buneman et al. (2003)] Se entenderá que un árbol XML T *satisface* una clave $(Q, (Q', \{P_1, \ldots, P_k\}))$ si y sólo si, para todo nodo $q \in [\![Q]\!]$ y todo par de nodos $q'_1, q'_2 \in q[\![Q']\!]$, tal que existen los nodos $x_i \in q'_1[\![P_i]\!]$, $y_i \in q'_2[\![P_i]\!]$, con $x_i =_v y_i$ para todo $i = 1, \ldots, k$, se tiene que $q'_1 = q'_2$. Más formalmente,

$$\forall q \in [\![Q]\!]\ \forall q'_1, q'_2 \in q[\![Q']\!] \left(\bigwedge_{1 \leqslant i \leqslant k} q'_1[\![P_i]\!] \cap_v q'_2[\![P_i]\!] \neq \emptyset \right) \Rightarrow q'_1 = q'_2 \tag{3.3}$$

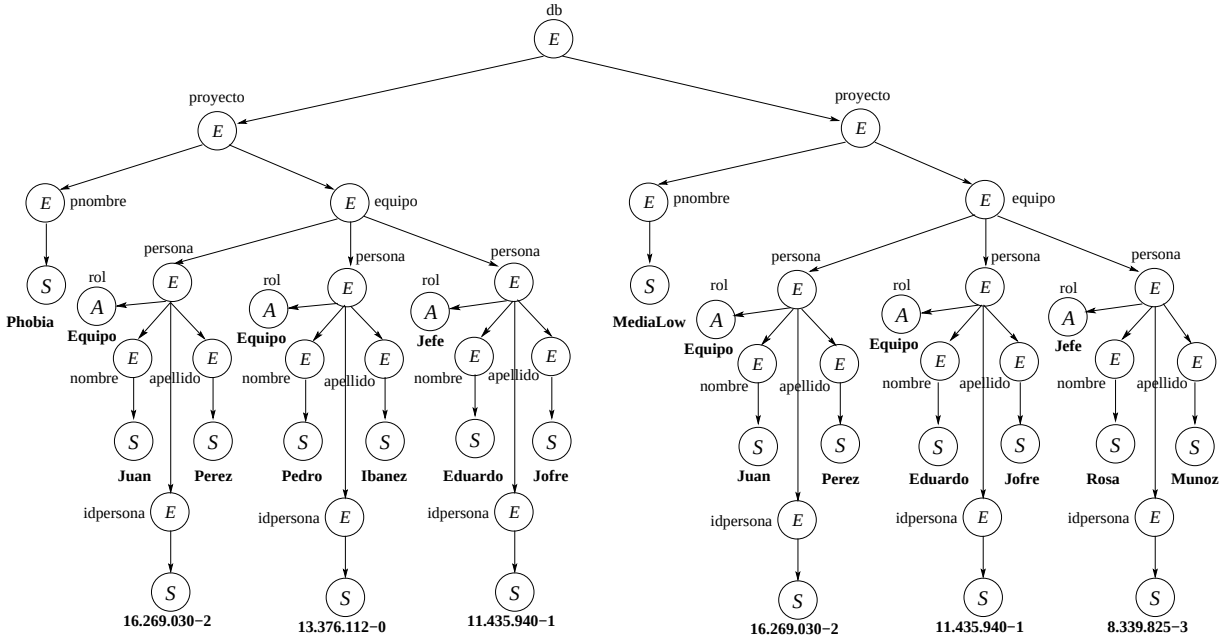


FIGURA 3.1: *Árbol XML alternativo al de la Figura 2.2.*

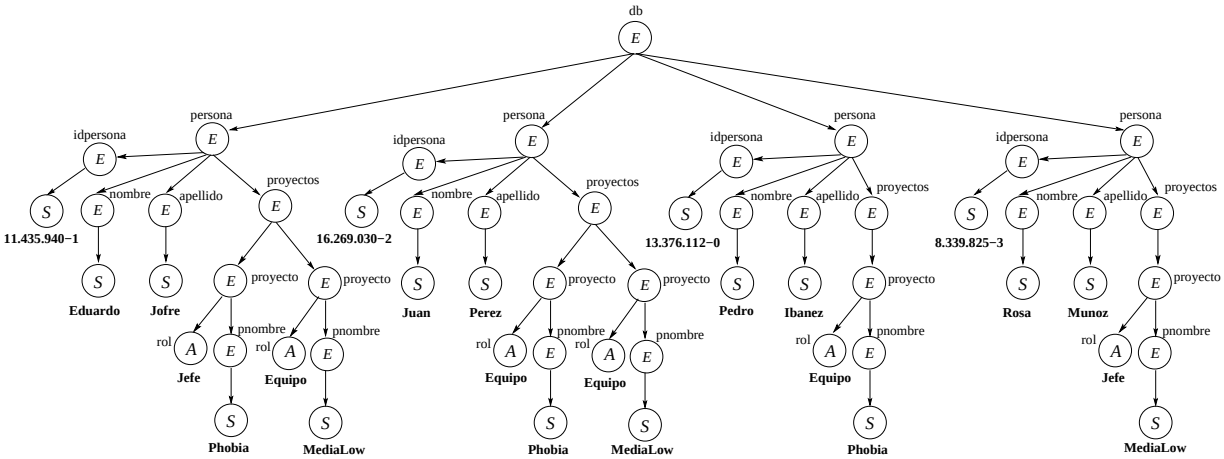


FIGURA 3.2: *Árbol XML alternativo al de la Figura 2.2.*

Ejemplo 3.1 Tomando los árboles XML presentados en las Figuras 2.2, 3.1, y 3.2, se presentan algunos ejemplos de claves XML absolutas y relativas, con su correspondiente interpretación semántica:

(a) $(\varepsilon, (\_^*.proyecto, \{pnombre.S\}))$. Esta clave XML absoluta expresa que dos nodos *proyecto* diferentes en un árbol T, deben diferir en el valor de sus descendientes *pnombre*. Esta clave es respetada, de manera particular, por los árboles de las Figuras 2.2 y 3.1, pero no, por el de la Figura 3.2.

(b) $(\varepsilon, (\_^*.persona, \{idpersona.S\}))$. Una persona puede ser identificada por su *idpersona* en todo el documento XML. Dos nodos *persona* diferentes no pueden tener el mismo valor en *idpersona*. Esta clave es respetada, en particular, por el árbol de la Figura 3.2, pero no por aquellos que tengan la forma de las Figuras 2.2 y 3.1.

(c) $(\_^*.proyecto.equipo, (persona, \{idpersona.S\}))$. Al igual que en (b), se desea definir que dos nodos *persona* no pueden tener el mismo valor en *idpersona*, pero esta vez en el contexto de un equipo dentro de un proyecto. Esta clave relativa es satisfecha, en particular, por árboles con la forma de los árboles de las Figuras 2.2 y 3.1, y es trivialmente satisfecha por cualquier árbol de la forma del árbol de la Figura 3.2.

(d) $(\_^*.proyecto.equipo, (\_^*, \{idpersona.S\}))$. En un árbol XML no se encontrarán dos descendientes de un nodo *equipo* cuya intersección en valor de su correspondiente *idpersona*, que sea no vacía. Esta clave es satisfecha por árboles con la forma de los presentados en las Figuras 2.2 y 3.1, y de manera trivial por un árbol con la forma de la Figura 3.2.

(e) $(\varepsilon, (\_^*.proyecto.\_^*, \{S\}))$. De manera simple, expresa que cualquier par de nodos descendientes del nodo *proyecto* en el árbol, deben diferir en los valores de todos sus nodos S (PCDATA) descendientes. Ninguno de los árboles presentados en las Figuras 2.2, 3.1, ni 3.2, satisfacen esta clave.

(f) $(\varepsilon, (\_^*.proyecto, \{\varepsilon\}))$. Esta clave expresa que no puede haber dos nodos *proyecto*: u y v diferentes, tal que ambos son hijos del nodo raíz, y los subárboles con raíz en u y v son isomorfos por un isomorfismo que es la identidad sobre cadenas de texto. Es satisfecha en particular por los árboles en las Figuras 2.2 y 3.1, y no es satisfecha por árboles como el de la Figura 3.2.

(g) $(\varepsilon, (proyecto, \{\emptyset\}))$. Esta clave es satisfecha por un árbol T, si hay a lo más un nodo *proyecto* que es un hijo del nodo raíz. Esta clave captura el concepto de restricción UNIQUE de XML *Schema*.

(h) $(\_^*.proyecto, (\_^*.persona, \{rol\}))$. En un árbol XML no se encontrarán dos nodos *persona* descendientes de un nodo *proyecto*, que sean iguales en valor de un nodo *rol*. Esta clave es satisfecha por árboles con la forma de las Figuras 2.2 y 3.2, pero no por aquellos que tengan la forma de la Figura 3.1.

□

Las claves aquí presentadas, son respetadas o no respetadas en particular por los árboles de las figuras mencionadas, pero también, por cualquier otro árbol que tenga la forma del árbol de la figura indicada. Además, se dice que un árbol respeta de manera *trivial* una clave, si la Definición 3.2 es satisfecha, pero el conjunto de nodos alcanzables siguiendo las expresiones de camino (camino de contexto, o camino objetivo) presentes en la clave es vacío.

Si bien las claves primarias y claves foráneas son soportadas por el esquema XML DTD (Fan & Libkin, 2002) a través del uso de atributos ID e IDREF, siendo éstas capaces de identificar de manera unívoca un elemento dentro de un documento XML, estas son más un "puntero" que una clave. Primero, sólo identifican elementos en el documento completo, no entre algún subconjunto determinado de elementos o subdocumento. Segundo, usando atributos ID como clave significa que se limita a claves unarias, y a usar atributos en lugar de elementos. Tercero, se puede especificar a lo más un atributo ID para un tipo elemento, mientras que en la práctica se puede querer una clave compuesta de varios elementos (Fan, 2005). Considerando, que no es obligatoria la existencia de un esquema asociado a un documento, y que la aplicación de este tipo de claves ligadas a un esquema particular –ID e IDREF en DTD y, UNIQUE y KEY en XML *Schema*– entregan una menor libertad de expresión, las claves aquí trabajadas no dependen de un esquema particular, y entregan una mayor expresividad.

3.1.1 Implicación de claves XML

Al igual que en el modelo relacional, es importante en XML saber cómo a partir de un conjunto de restricciones, aplicando determinadas reglas, es posible derivar restricciones que

estaban implícitas en el conjunto original. En el modelo relacional, existen los denominados axiomas de Armstrong (Armstrong, 1974), que permiten determinar las dependencias funcionales definidas implícitamente, a partir de un conjunto de dependencias definidas explícitamente. En este trabajo se estudiaran las claves como restricciones de integridad en XML. Para referencias de restricciones sobre XML revisar Fan (2005) y Fan & Siméon (2003).

Sea $\Sigma \cup \{\varphi\}$ un conjunto finito de claves XML en una clase $\mathcal{C}$. Se dice que Σ *(finitamente) implica* φ, denotado por $\Sigma \models_{(f)} \varphi$ si y sólo si, todo árbol XML (finito) que satisface todas las claves $\sigma \in \Sigma$ también satisface φ. El *problema de implicación (finita) para* $\mathcal{C}$ es decidir, dado cualquier conjunto finito $\Sigma \cup \{\varphi\}$ de claves en $\mathcal{C}$, si $\Sigma \models_{(f)} \varphi$. Sea Σ un conjunto de claves en $\mathcal{C}$, su clausura semántica (finita) $\Sigma^*_{(f)} = \{\varphi \in \mathcal{C} \mid \Sigma \models_{(f)} \varphi\}$ es el conjunto de todas las claves (finitamente) implicadas por Σ.

La noción de inferencia sintáctica ($\vdash_{\Re}$) con respecto a un conjunto $\Re$ de reglas de inferencia puede ser definida de manera análoga a la noción existente en el modelo de datos relacional. Esto es, una secuencia finita $\gamma = [\gamma_1, \ldots, \gamma_l]$ de claves XML es llamada una *inferencia a partir de* Σ *utilizando* $\Re$, si toda γ_i es, o bien un elemento de Σ, o es obtenida de la aplicación de una de las reglas en $\Re$ a elementos apropiados de $\{\gamma_1, \ldots, \gamma_{i-1}\}$. Se dice que la inferencia γ infiere γ_l, es decir, el último elemento de la secuencia γ, y se escribe $\Sigma \vdash_{\Re} \gamma_l$. Para un conjunto finito Σ de claves en $\mathcal{C}$, se denota $\Sigma^+_{\Re} = \{\varphi \mid \Sigma \vdash_{\Re} \varphi\}$ su *clausura sintáctica* bajo inferencias usando $\Re$.

Un conjunto $\Re$ de reglas de inferencia se dice que es *válido (completo)* para la implicación (finita) de claves en $\mathcal{C}$, si para todo conjunto finito Σ de claves XML en $\mathcal{C}$ se tiene que $\Sigma^+_{\Re} \subseteq \Sigma^*_{(f)}$ ($\Sigma^*_{(f)} \subseteq \Sigma^+_{\Re}$).

El conjunto $\Re$ de reglas se dice es que una *axiomatización* para la implicación (finita) de claves en $\mathcal{C}$, si es tanto válido como completo para la implicación (finita) de claves en $\mathcal{C}$. Finalmente, se dice que una *axiomatización* es *finita* si el conjunto $\Re$ es finito.

3.2 AXIOMATIZACIÓN DE CLAVES EN $\mathcal{K}^{PL}_{PL,PL^+_S}$

Una axiomatización finita para la implicación (finita) de claves en la clase de claves XML con conjunto no vacío de caminos clave simples $\mathcal{K}^{PL}_{PL,PL^+_s}$, fue establecida en Hartmann & Link

TABLA 3.1: *Una axiomatización de clave XML en* $\mathcal{K}^{PL}_{PL,PL^+_s}$.

$\dfrac{}{(Q,(\varepsilon,\mathcal{S}))}$ (epsilon)	$\dfrac{(Q,(Q',\mathcal{S}\cup\{\varepsilon,P\}))}{(Q,(Q',\mathcal{S}\cup\{\varepsilon,P.P'\}))}$ (prefijo-epsilon)	$\dfrac{(Q,(Q',\mathcal{S}))}{(Q,(Q',\mathcal{S}\cup\{P\}))}$ (superclave)
$\dfrac{(Q,(Q'.P,\{P'\}))}{(Q,(Q',\{P.P'\}))}$ (subnodos)	$\dfrac{(Q,(Q',\mathcal{S}))}{(Q'',(Q',\mathcal{S}))}\,Q''\subseteq Q$ (contención-camino-contexto)	$\dfrac{(Q,(Q',\mathcal{S}))}{(Q,(Q'',\mathcal{S}))}\,Q''\subseteq Q'$ (contención-camino-objetivo)
$\dfrac{(Q,(Q'.Q'',\mathcal{S}))}{(Q.Q',(Q'',\mathcal{S}))}$ (contexto-objetivo)	$\dfrac{(Q,(Q'.P,\{\varepsilon,P'\}))}{(Q,(Q',\{\varepsilon,P.P'\}))}$ (subnodos-epsilon)	$\dfrac{(Q,(Q',\{P.P_1,\ldots,P.P_k\}),\ (Q.Q',(P,\{P_1,\ldots,P_k\}))}{(Q,(Q'.P,\{P_1,\ldots,P_k\}))}$ (interacción)

(2007, 2009a). Recordar que "+" indica que el conjunto de caminos clave no puede ser vacío. Es importante notar que desde el punto de vista práctico, y como es ilustrado por los ejemplos dados en Buneman et al. (2002); Hartmann & Link (2007, 2009a) y en el presente trabajo, muchas claves XML útiles pertenecen a la clase $\mathcal{K}^{PL}_{PL,PL^+_s}$. En particular todas las claves en el Ejemplo 3.1, a excepción de la clave (g), pertenecen a esta clase.

En la Tabla 3.1, se muestra el conjunto de reglas de inferencia que componen la axiomatización de la clase $\mathcal{K}^{PL}_{PL,PL^+_s}$, presentada en Hartmann & Link (2009a). A continuación se analizara brevemente cada una de estas reglas.

Las pruebas formales respecto a la validez de las reglas *epsilon, prefijo-epsilon, superclave, contención-camino-contexto, contención-camino-objetivo, contexto-objetivo, e interacción* pueden encontrarse en Buneman et al. (2003), y respecto a las reglas *subnodos*, y *subnodos-epsilon* pueden encontrarse en Hartmann & Link (2009a).

- Regla *epsilon:* Notar que dada una clave con la forma $(Q,(\varepsilon,\mathcal{S}))$ con $Q\neq\varepsilon$, para todo nodo contexto $q\in[\![Q]\!]$, se sigue que $q[\![\varepsilon]\!]=\{q\}$. Entonces, para todo nodo contexto $q\in[\![Q]\!]$, existe un único nodo objetivo, y por lo tanto cualquier conjunto $\mathcal{S}$ de caminos clave constituye una clave para q.

- Regla *prefijo-epsilon:* Si para todo $q\in[\![Q]\!]$, un conjunto $\mathcal{S}\cup\{\varepsilon,P\}$ de caminos clave constituye una clave para el conjunto de nodos en $q[\![Q']\!]$, entonces se puede extender el camino clave P

concatenándolo con otro camino P', y el conjunto $\mathcal{S} \cup \{\varepsilon, P.P'\}$ resultante es también una clave para $q[\![Q']\!]$. Dado que para todo par de nodos $q_1, q_2 \in q[\![Q']\!]$, si $q_1[\![P.P']\!] \cap_v q_2[\![P.P']\!] \neq \emptyset$ y $q_1 =_v q_2$, entonces se tiene que $q_1[\![P]\!] \cap_v q_2[\![P]\!] \neq \emptyset$. Nótese que $q_1 =_v q_2$ si $q_1[\![\varepsilon]\!] \cap_v q_2[\![\varepsilon]\!] \neq \emptyset$. Así, por la definición de claves XML, $\mathcal{S} \cup \{\varepsilon, P.P'\}$ es también una clave para $q[\![Q']\!]$.

- Regla *superclave:* Si para todo $q \in [\![Q]\!]$ $\mathcal{S}$ es una clave para el conjunto de nodos en $q[\![Q']\!]$, entonces también lo es cualquier superconjunto de $\mathcal{S}$. Esta es la única regla que tiene su contraparte en la inferencia de claves en el modelo relacional.

- Regla *subnodos:* Dado que se trabaja con un modelo de árbol y P es una expresión PL_s, si $q \in [\![Q]\!]$ se sigue que para todo nodo $v \in q[\![Q'.P]\!]$, existe un único nodo v' en $q[\![Q']\!]$ tal que, $v \in v'[\![P]\!]$. Por lo tanto, si un camino clave P' identifica de manera unívoca a un nodo v en $q[\![Q'.P]\!]$, entonces $P.P'$ identifica de manera unívoca a un nodo v' en $q[\![Q']\!]$.

- Regla *contención-camino-contexto:* Dado que $Q'' \subseteq Q$ para todo árbol XML T, si $(\varepsilon, (Q', \mathcal{S}))$ se mantiene en todos los subárboles cuya raíz es un nodo en $[\![Q]\!]$, entonces también debe mantenerse para todos los subárboles cuya raíz es un nodo en el subconjunto $[\![Q']\!]$ de $[\![Q]\!]$.

- Regla *contención-camino-objetivo:* Dado que $Q'' \subseteq Q'$, para todo $q \in [\![Q]\!]$, una clave para el conjunto $q[\![Q']\!]$ es también una clave para cualquier subconjunto $q[\![Q'']\!]$ de $q[\![Q']\!]$.

- Regla *contexto-objetivo:* Si en un árbol T con raíz en un nodo $q \in [\![Q]\!]$, un conjunto $\mathcal{S}$ de caminos clave es una clave para $q[\![Q'.Q'']\!]$, entonces en todo subárbol de T con raíz en $q' \in q[\![Q']\!]$, $\mathcal{S}$ es una clave para $q'[\![Q'']\!]$. Nótese, que $q'[\![Q'']\!]$ se compone de los nodos que están tanto en $q[\![Q'.Q'']\!]$ y en el árbol con raíz q'. En particular, cuando $Q = \varepsilon$ esta regla dice que si $(\varepsilon, (Q'.Q'', \mathcal{S}))$ se mantiene, entonces también lo hace $(Q', (Q'', \mathcal{S}))$.

- Regla *interacción:* Esta es la única regla en $\Re$ que tiene más de una clave en su precondición. Por la primera clave en la precondición, en cada subárbol con raíz en un nodo q en $[\![Q]\!]$, $P.P_1, \ldots, P.P_k$ identifica de manera unívoca los nodos en $q[\![Q']\!]$. Intuitivamente, la segunda clave en la precondición previene la existencia de más de un nodo P bajo Q' que coincida en sus nodos $P_1, \ldots, P_k$. Por lo tanto, $P_1, \ldots, P_k$ identifican de manera unívoca un nodo en $q[\![Q'.P]\!]$ en cada subárbol con raíz q en $[\![Q]\!]$. Más formalmente, para todo $q \in [\![Q]\!]$ y $q_1, q_2 \in q[\![Q'.P]\!]$, debe existir un par de nodos v_1, v_2 en $q[\![Q']\!]$ tal que $q_1 \in v_1[\![P]\!]$, $q_2 \in v_2[\![P]\!]$ y para

todo $i = 1, \ldots, k$, se debe tener $q_1[\![P_i]\!] \subseteq v_1[\![P.P_i]\!]$, y $q_2[\![P_i]\!] \subseteq v_2[\![P.P_i]\!]$. Si $q_1[\![P_i]\!] \cap_v q_2[\![P_i]\!] \neq \emptyset$, entonces $v_1[\![P.P_i]\!] \cap_v v_2[\![P.P_i]\!] \neq \emptyset$, para todo $i = 1, \ldots, k$. Así, por la primera clave en la precondición, $v_1 = v_2$. Desde que $q_1, q_2 \in v_1[\![P]\!]$ y como resultado, $q_1 = q_2$ por la segunda clave en la precondición. Por lo tanto, $(Q, (Q'.P, \{P_1, \ldots, P_k\}))$ se mantiene.

- Regla *subnodos-epsilon:* Dado que se trabaja con un modelo de árbol y P es una expresión PL_s, si $q \in [\![Q]\!]$ se sigue que para todo nodo $v \in q[\![Q'.P]\!]$, existe un único nodo $v' \in q[\![Q']\!]$ tal que, $v \in v'[\![P]\!]$. Por lo tanto, si un conjunto $\{\varepsilon, P'\}$ de caminos clave constituye una clave para un nodo $v \in q[\![Q'.P]\!]$, entonces el conjunto $\{\varepsilon, P.P'\}$ identifica de manera unívoca a un nodo $v' \in q[\![Q']\!]$.

Ejemplo 3.2 Dada la siguiente inferencia, se puede decir que $\Sigma \models \varphi$. Sea $\Sigma = \{\sigma\}$, con $\sigma = (\varepsilon, (proyecto.equipo.persona._^*, \{rol\}))$ y $\varphi = (proyecto.equipo, (persona, \{rol, idpersona.S\}))$.

(1) $(\varepsilon, (proyecto.equipo.persona, \{rol\}))$ obtenida a partir de la aplicación de la regla *contención-camino-objetivo* a σ, considerando que $proyecto.equipo.persona \subseteq proyecto.equipo.persona._^*$.

(2) $(proyecto.equipo, (persona, \{rol\}))$ obtenida aplicando a (1) la regla *contexto-objetivo.*

(3) $(proyecto.equipo, (persona, \{rol, idpersona.S\}))$ obtenida de (2) aplicando la regla *superclave.* □

Para probar la completitud del conjunto $\Re$ para la implicación de claves en $\mathcal{K}^{PL}_{PL,PL^+_s}$, se requiere mostrar que para un conjunto finito arbitrario $\Sigma \cup \{\varphi\}$ de claves en $\mathcal{K}^{PL}_{PL,PL^+_s}$, si $\varphi \notin \Sigma^+_{\Re}$, entonces hay algún árbol XML T que es un contra-ejemplo para la implicación de φ por Σ.

La estrategia de prueba (Hartmann & Link, 2009a) es la siguiente: en primer lugar, se presenta la clave φ en términos de un árbol finito con nodos etiquetados $T_{\Sigma,\varphi}$, al cual se le llama *mini-tree*. Luego, se calcula el impacto de cada una de las claves $\sigma \in \Sigma$ en el árbol T de contra-ejemplo que se desea construir. Se mantiene el rastro de estos impactos mediante la inserción de aristas dirigidas ascendentes en $T_{\Sigma,\varphi}$. Esto resulta en un digrafo $G_{\Sigma,\varphi}$ al cual se le llama *witness-graph*. Finalmente, se aplica un algoritmo de alcanzabilidad sobre $G_{\Sigma,\varphi}$, para decidir que nodos serán duplicados en $T_{\Sigma,\varphi}$, con el fin de generar el árbol de contra-ejemplo T que no satisfaga φ, pero que satisfaga todas las claves en Σ.

En el árbol $T_{\Sigma,\varphi}$ se distinguen dos nodos a saber: q_φ y q'_φ, los cuales son alcanzables desde la raíz de $T_{\Sigma,\varphi}$ siguiendo un camino-Q_φ y un camino-$Q_\varphi.Q'_\varphi$, respectivamente. En $G_{\Sigma,\varphi}$, un nodo v

es *alcanzable* desde un nodo u cuando existe un camino desde u a v en $G_{\Sigma,\varphi}$, esto es, una secuencia $u = v_0, \ldots, v_m = v$ de nodos mutuamente distintos con una arista (v_{i-1}, v_i) para todo $i = 1, \ldots, m$. Se muestra que si $\varphi \notin \Sigma^+_{\Re}$, entonces q_φ no es alcanzable desde q'_φ en $G_{\Sigma,\varphi}$.

A continuación se presentan las nociones de *mini-tree* y *witness-graph*, utilizadas para la comprobación de completitud de $\Re$ para la implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$, esto es las herramientas usadas para probar el siguiente lema.

Lema 3.1 [Hartmann & Link (2009a)] Sea $\Sigma \cup \{\varphi\}$ un conjunto finito de claves en $\mathcal{K}^{PL}_{PL,PL^+_s}$. Si q_φ es alcanzable desde q'_φ en el *witness-graph* $G_{\Sigma,\varphi}$, entonces $(Q_\varphi, (Q'_\varphi, \{P^\varphi_1, \ldots, P^\varphi_{k_\varphi}\})) \in \Sigma$.

Este lema es la clave para el algoritmo que decide la implicación de claves en $\mathcal{K}^{PL}_{PL,PL^+_s}$ en tiempo cuadrático en el tamaño de la clave de entrada, algoritmo que fuera originalmente también presentado en Hartmann & Link (2009a).

3.2.1 *Mini-Trees* y *Witness-Graphs*

Estas herramientas permiten solucionar el problema de implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$, a través de su caracterización como un problema de alcanzabilidad de determinados nodos en un grafo dirigido dado. Esta caracterización resulta en un procedimiento elegante y eficiente para el problema de implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$.

Sea $\Sigma \cup \{\varphi\}$ un conjunto finito de claves en $\mathcal{K}^{PL}_{PL,PL^+_s}$, sea $\mathcal{L}_{\Sigma,\varphi}$ el conjunto de todas las etiquetas $\ell \in \mathcal{L}$ que ocurren en expresiones de camino de las claves en $\Sigma \cup \{\varphi\}$, y sea ℓ_0 una etiqueta en $\mathbf{E} - \mathcal{L}_{\Sigma,\varphi}$. Además, sea O_φ y O'_φ las expresiones PL_s obtenidas desde las expresiones PL Q_φ y Q'_φ, respectivamente, mediante el reemplazo de cada comodín de largo variable "_*" por ℓ_0. Finalmente, sea p un camino-O_φ desde un nodo r_φ a un nodo q_φ, sea p' un camino-O'_φ desde un nodo r'_φ a un nodo q'_φ y, para todo $i = 1, \ldots, k_\varphi$, sea p_i un camino-P^φ_i desde un nodo r^φ_i a un nodo x^φ_i, tal que los caminos $p, p', p_1, \ldots, p_{k_\varphi}$ no poseen nodos en común.

A partir de los caminos $p, p', p_1, \ldots, p_{k_\varphi}$ se obtiene el *mini-tree* $T_{\Sigma,\varphi}$ identificando el nodo r'_φ con q_φ, e identificando cada uno de los nodos r^φ_i con q'_φ. Nótese, que q_φ es el único nodo en $T_{\Sigma,\varphi}$ que satisface $q_\varphi \in [\![O_\varphi]\!]$, y q'_φ es el único nodo en $T_{\Sigma,\varphi}$ que satisface $q'_\varphi \in [\![O'_\varphi]\!]$.

El conjunto de *marcado* del *mini-tree* $T_{\Sigma,\varphi}$ es un subconjunto $\mathcal{M}$ del conjunto de nodos

de $T_{\Sigma,\varphi}$: si para todo $i = 1, \ldots, k_\varphi$ se tiene $P_i^\varphi \neq \varepsilon$, entonces $\mathcal{M}$ consiste de los nodos hoja (nodos sin hijos) de $T_{\Sigma,\varphi}$, y en otro caso $\mathcal{M}$ consiste de todos los nodos descendientes de q'_φ en $T_{\Sigma,\varphi}$.

Ejemplo 3.3 Sea Σ el conjunto compuesto por las siguientes claves en $\mathcal{K}^{PL}_{PL,PL_s^+}$,

$$\sigma_1 = (\varepsilon, (publico.\_^*, \{proyecto.pnombre.S, proyecto.año.S\}))$$

$$\sigma_2 = (publico, (\_^*.proyecto, \{ciudad.S\}))$$

En la Figura 3.3 se presenta el *mini-tree* $T_{\Sigma,\varphi}$ para la clave

$$\varphi = (\varepsilon, (publico.\_^*.proyecto, \{pnombre.S, año.S\}))$$

donde *nacional* es la etiqueta ℓ_0 escogida del conjunto $\mathbf{E}-\mathcal{L}_{\Sigma,\varphi}$. En este caso el conjunto de marcado $\mathcal{M}$ consiste de las hojas del árbol (las cuales se denotan con $\times$). El proceso de construcción de $T_{\Sigma,\varphi}$ es el siguiente:

Se define el conjunto de etiquetas,

$$\mathcal{L}_{\Sigma,\varphi} = \{\ell \in \mathcal{L}\text{: } \ell \text{ ocurre en alguna expresión de camino en } \Sigma \cup \{\varphi\}\}$$

$$= \{publico, proyecto, pnombre, año, cuidad, S\}$$

Se realiza el reemplazo de los comodines por la etiqueta ℓ_0,

$$Q_\varphi = \varepsilon \qquad O_\varphi = \varepsilon$$

$$Q'_\varphi = publico.\_^*.proyecto \qquad O'_\varphi = publico.nacional.proyecto$$

Sea p un camino-O_φ desde un nodo r_φ a un nodo q_φ:

$$p = r_\varphi = q_\varphi \text{ ; donde } lab(p) = \varepsilon$$

Sea p' un camino-O'_φ desde un nodo r'_φ a un nodo q'_φ:

$$p' = r'_\varphi, v'_1, v'_2, q'_\varphi \text{ ; donde } lab(p') = publico.nacional.proyecto$$

Se tiene que $k_\varphi = 2$, Sea p_1 un camino-P_1^φ desde un nodo r_1^φ a un nodo x_1^φ:

$$p_1 = r_1^\varphi, w_1, x_1^\varphi \text{ ; donde } lab(p_1) = pnombre.S$$

Sea p_2 un camino-P_2^φ desde un nodo r_2^φ a un nodo x_2^φ:

$$p_2 = r_2^\varphi, w'_1, x_2^\varphi \text{ ; donde } lab(p_2) = año.S$$

Nótese, que los caminos p, p', p_1, p_2 son disjuntos en sus nodos. Con estos caminos es posible construir el *mini-tree* de la Figura 3.3, identificando: (1) el nodo r_φ con el nodo raíz de etiqueta *db*, (2) el nodo r'_φ con el nodo q_φ, y (3) cada nodo r_i^φ con q'_φ para todo $i = 1, \ldots, k_\varphi$. Además, se puede ver que el camino p consiste sólo de un nodo (temporal) r_φ, el cual es el mismo nodo q_φ, y que es identificado como el nodo raíz del árbol. Entonces al identificar el nodo r'_φ con q_φ, se tiene que r_φ, q_φ, y r'_φ corresponden al mismo nodo. □

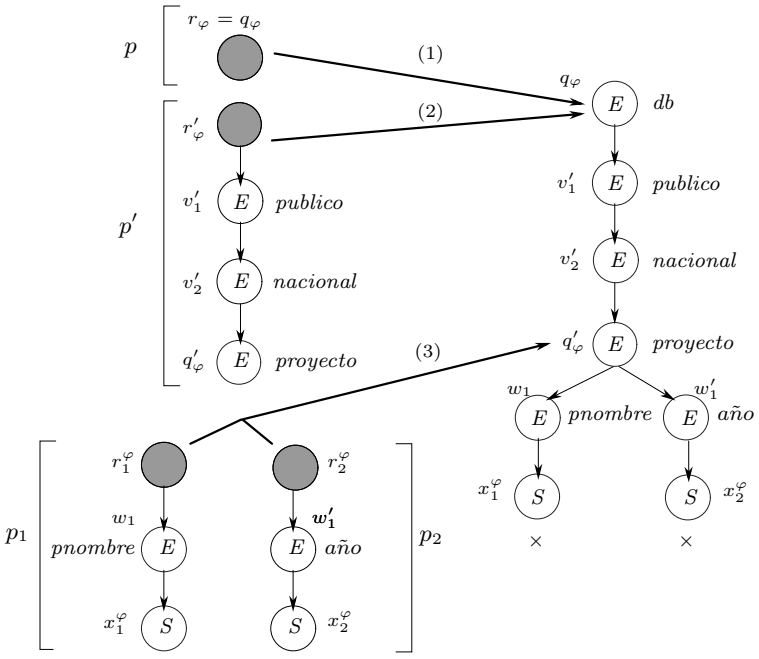


FIGURA 3.3: *Mini-tree resultante del Ejemplo 3.3.*

Los *mini-trees* son usados para calcular el impacto de una clave en Σ sobre un posible árbol de contra-ejemplo para la implicación de φ por Σ. Para distinguir las claves $\sigma \in \Sigma$ que tienen un impacto de aquellas que no lo tienen, se introduce la noción de *aplicabilidad*.

Definición 3.3 Sea $T_{\Sigma,\varphi}$ el *mini-tree* de la clave φ con respecto a Σ, y sea $\mathcal{M}$ su conjunto de marcado. Una clave σ se dice que es *aplicable* a φ si y sólo si, hay nodos $w_\sigma \in [\![Q_\sigma]\!]$ y $w' \in w_\sigma[\![Q'_\sigma]\!]$ en $T_{\Sigma,\varphi}$ tal que $w'_\sigma[\![P_i^\sigma]\!] \cap \mathcal{M} \neq \emptyset$ para todo $i = 1, \ldots, k_\sigma$. Se dice que w_σ y w'_σ *testifican* la aplicabilidad de σ a φ.

Ejemplo 3.4 Como ejemplo se considera Σ y φ del Ejemplo 3.3. Sea $\mathcal{M} = \{x_1^\varphi, x_2^\varphi\}$ el conjunto de marcado. La clave σ_1 es aplicable a φ, ya que existe un nodo w_{σ_1} en $[\![Q_{\sigma_1}]\!] = \{q_\varphi\}$ y un nodo w'_{σ_1} en $q_\varphi[\![Q'_{\sigma_1}]\!] = \{v'_1, v'_2, q'_\varphi, w_1, x_1^\varphi, w'_1, x_2^\varphi\}$ tal que, si se toma $w'_{\sigma_1} = v'_2$, $w'_{\sigma_1}[\![P_1^{\sigma_1}]\!] \cap \mathcal{M} \neq \emptyset$ y $w'_{\sigma_1}[\![P_2^{\sigma_1}]\!] \cap \mathcal{M} \neq \emptyset$. Por el contrario la clave σ_2 no es aplicable a φ ya que no existe un nodo w_{σ_2} en $[\![Q_{\sigma_2}]\!] = \{v'_1\}$ y un nodo w'_{σ_2} en $v'_1[\![Q'_{\sigma_2}]\!] = \{q'_\varphi\}$ tal que $w'_{\sigma_2}[\![P_1^{\sigma_2}]\!] \cap \mathcal{M} \neq \emptyset$. □

Se define el *witness-graph* $G_{\Sigma,\varphi}$ como el grafo dirigido (o dígrafo) de nodos etiquetados obtenido a partir de $T_{\Sigma,\varphi}$ mediante la inserción de aristas adicionales: para cada clave $\sigma \in \Sigma$ que es aplicable a φ, y para todo par de nodos $w_\sigma \in [\![Q_\sigma]\!]$ y $w'_\sigma \in w_\sigma[\![Q'_\sigma]\!]$ que testifican la aplicabilidad de σ a φ, $G_{\Sigma,\varphi}$ contiene la arista dirigida (w'_σ, w_σ) que va desde w'_σ hasta w_σ. A partir de ahora, para referirse a estas aristas se utilizará el nombre de *aristas testigo*, mientras a las aristas originales de

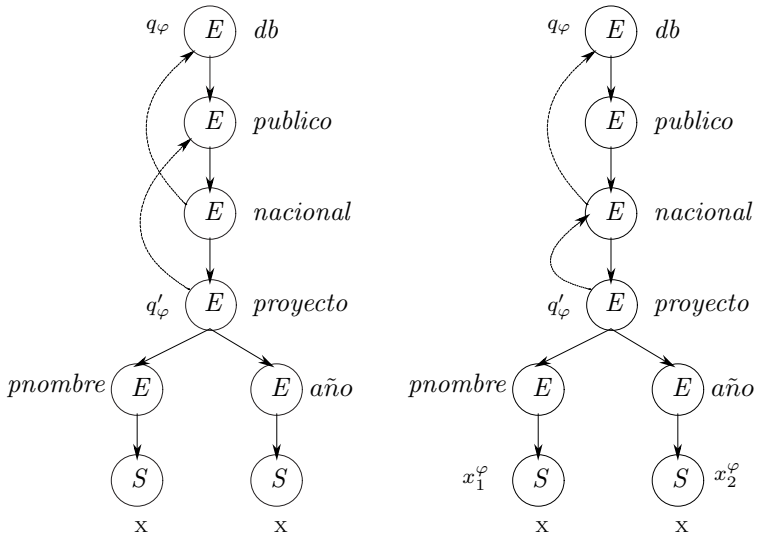


FIGURA 3.4: *Witness-graph resultante de los Ejemplos 3.5, 3.6.*

$T_{\Sigma,\varphi}$ se les llamará *aristas descendentes* de $G_{\Sigma,\varphi}$. Esto es motivado por el hecho de que para todos los nodos testigo w_σ y w'_σ, el nodo w'_σ es un nodo descendiente de w_σ en $T_{\Sigma,\varphi}$, y de este modo la arista testigo (w'_σ, w_σ) es una arista ascendente o un ciclo en $G_{\Sigma,\varphi}$.

Ejemplo 3.5 Sea φ como en el Ejemplo 3.3, y sea el conjunto $\Sigma = \{\sigma_1, \sigma_2\}$, con

$$\sigma_1 = (\varepsilon, (publico.\_^*, \{proyecto.pnombre.S, proyecto.año.S\}))$$

$$\sigma_2 = (publico, (\_^*.proyecto, \{pnombre.S, año.S\}))$$

Se obtiene el mismo *mini-tree* de la Figura 3.3 y, agregando las aristas testigo para las claves $\sigma \in \Sigma$ aplicables a φ, se obtiene el *witness-graph* a la izquierda en la Figura 3.4. Tanto σ_1 como σ_2 son aplicables a φ. De σ_1 se obtiene la arista testigo $(nacional, db)$, mientras de σ_2 se obtiene la arista testigo $(proyecto, publico)$.

Las aristas testigo resultantes de σ_1 y σ_2 se generan a partir de lo siguiente. Sea el nodo $w_{\sigma_1} \in [\![\varepsilon]\!] = \{q_\varphi\}$ y el nodo $w'_{\sigma_1} \in w_{\sigma_1}[\![publico.\_^*]\!] = \{v'_1, v'_2, q'_\varphi, w_1, w'_1, x_1^\varphi, x_2^\varphi\}$. Si $w'_{\sigma_1} = v'_2$, $w'_{\sigma_1}[\![proyecto.pnombre.S]\!] = \{x_1^\varphi\}$ y $w'_{\sigma_1}[\![proyecto.año.S]\!] = \{x_2^\varphi\}$, como $w'_{\sigma_1}[\![P_i^{\sigma_1}]\!] \cap \mathcal{M} \neq \emptyset$ (para $i = 1, 2$), se tiene la aplicabilidad de σ_1 a φ y la arista testigo $(w'_{\sigma_1}, w_{\sigma_1}) = (nacional, db)$. Luego, sea el nodo $w_{\sigma_2} \in [\![publico]\!] = \{v'_1\}$ y el nodo $w'_{\sigma_2} \in w_{\sigma_2}[\![\_^*.proyecto]\!] = \{q'_\varphi\}$. Si $w'_{\sigma_2} = q'_{\sigma_2}$, $w'_{\sigma_2}[\![pnombre.S]\!]$ $= \{x_1^\varphi\}$ y $w'_{\sigma_2}[\![año.S]\!] = \{x_2^\varphi\}$, como $w'_{\sigma_2}[\![P_i^{\sigma_2}]\!] \cap \mathcal{M} \neq \emptyset$ (para $i = 1, 2$), se tiene la aplicabilidad de σ_2 a φ, y la arista testigo $(w'_{\sigma_2}, w_{\sigma_2}) = (proyecto, publico)$. □

Nótese, que en el *witness-graph* (izquierdo) de la Figura 3.4, el nodo q_φ es alcanzable desde el nodo q'_φ. De acuerdo al Lema 3.1 esto significa que $\Sigma \vdash_\Re \varphi$. De hecho, aplicando la regla *contexto-*

objetivo a σ_2 se obtiene la clave $\sigma_2' = (publico._^*, (proyecto, \{pnombre.S, año.S\}))$ (el respectivo *witness-graph* para $\Sigma' = \{\sigma_1, \sigma_2'\}$ se muestra a la derecha en la Figura 3.4) y luego, aplicando la regla de *interacción* entre σ_1 y σ_2' se obtiene φ.

Ejemplo 3.6 Sea Σ el siguiente conjunto compuesto de las claves en $\mathcal{K}^{PL}_{PL,PL^+_s}$,

$$\sigma_1 = (\varepsilon, (departamento.equipo.proyecto.pnombre, \{\varepsilon, S\}))$$

$$\sigma_2 = (departamento, (equipo, \{proyecto, proyecto.pnombre.S, proyecto.año.S\}))$$

donde, σ_1 declara que no pueden haber dos nodos *pnombre*, siguiendo un camino-*departamento.equipo.proyecto.pnombre*, en todo el documento que sean isomorfos por algún isomorfismo que para cadenas de texto se corresponde con la función de identidad. Y σ_2 declara que en el contexto de un departamento, no pueden haber dos equipos que trabajen en un mismo proyecto, con mismo nombre y año. Sea $\varphi = (departamento, (equipo.proyecto, \{\varepsilon, pnombre.S, año.S\}))$ la clave que declare que no existen dos nodos *proyecto*: u y v diferentes, descendiente de un nodo *equipo* en el contexto de un nodo *departamento* dentro de un árbol XML T, tal que los subárboles con raíz en u y v sean isomorfos por un isomorfismo que es la identidad sobre cadenas de texto, destacando la igualdad en valor en sus descendientes *pnombre* y *año*.

A partir de φ, se tiene el *mini-tree* de la Figura 3.5(a), y de la aplicabilidad de las claves en Σ se tiene el *witness-graph* de la Figura 3.5(b). Aplicando la regla *subnodos-epsilon* a σ_1, se tiene la clave $\sigma_3 = (\varepsilon, (departamento.equipo.proyecto, \{\varepsilon, pnombre.S\}))$. Sea $\Sigma' = \{\sigma_3, \sigma_2\}$. Generando el nuevo *witness-graph* resultante de la aplicabilidad de las claves en Σ' a φ, se obtiene el grafo $G_{\Sigma',\varphi}$ de la Figura 3.5(c). Luego, aplicando la regla *contexto-objetivo* a σ_3 se obtiene $\sigma_4 = (departamento.equipo, (proyecto, \{\varepsilon, pnombre.S\}))$, y aplicando a ésta la regla *superclave* se obtiene $\sigma_5 = (departamento.equipo, (proyecto, \{\varepsilon, pnombre.S, año.S\}))$. Así, la arista testigo resultante de la aplicabilidad de σ_4 y σ_5 a φ es igual. Y generando el nuevo *witness-graph* $G_{\Sigma'',\varphi}$ se obtiene el grafo de la Figura 3.5(d). Nótese, ahora que si se aplica la regla de interacción entre σ_2 y σ_5, se obtiene la clave φ. □

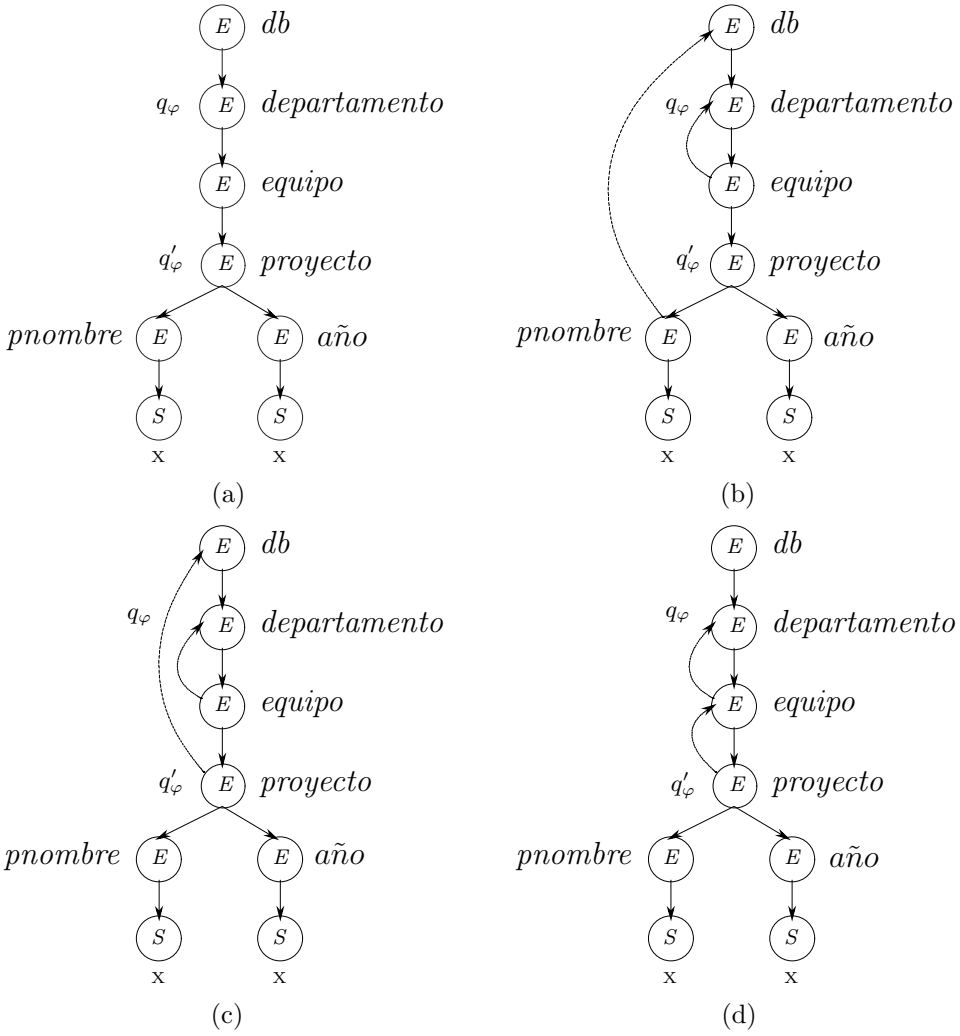


FIGURA 3.5: *Mini-tree y Witness-graphs resultantes del Ejemplo 3.6.*

3.3 ALGORITMO DE IMPLICACIÓN DE CLAVES XML EN $\mathcal{K}^{PL}_{PL,PL^+_S}$

Tal como se planteó en las secciones anteriores, el problema de implicación en $\mathcal{K}^{PL}_{PL,PL^+_s}$ puede ser caracterizado en términos del problema de alcanzabilidad de nodos en el *witness-graph*. La traducción del problema de implicación de claves XML a un problema de alcanzabilidad de nodos en un grafo, queda planteada en el Teorema 3.1.

Teorema 3.1 [Hartmann & Link (2009a)] Sea $\Sigma \cup \{\varphi\}$ un conjunto finito de claves en $\mathcal{K}^{PL}_{PL,PL^+_s}$. Se tiene que $\Sigma \models \varphi$ si y sólo si, q_φ es alcanzable desde q'_φ en $G_{\Sigma,\varphi}$.

En Hartmann & Link (2009a) se muestra la validez del teorema usando un contra-ejemplo. Recordando brevemente el contra-ejemplo: primero, a partir de φ se construye un árbol finito $T_{\Sigma,\varphi}$ llamado *mini-tree*. Luego, se calcula el impacto (aplicabilidad) de cada $\sigma \in \Sigma$, y se insertan las correspondientes aristas testigo en $T_{\Sigma,\varphi}$ que generan el grafo $G_{\Sigma,\varphi}$ llamado *witness-graph*. Finalmente, se aplica un algoritmo de alcanzabilidad sobre $G_{\Sigma,\varphi}$, para decidir que nodos serán duplicados en $T_{\Sigma,\varphi}$, con el fin de generar este árbol de contra-ejemplo T que no satisface φ, pero que satisface todas las claves en Σ. Se muestra que si $\varphi \notin \Sigma^+_{\Re}$, entonces q_φ no es alcanzable desde q'_φ en $G_{\Sigma,\varphi}$.

El Teorema 3.1 da paso al diseño del Algoritmo 3.1 para decidir implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$. El cual es resultado de las pruebas de completitud presentadas para el conjunto de reglas de inferencia en Hartmann & Link (2009a).

Algoritmo 3.1: $\Sigma \models \varphi$: *Implicación de claves XML en* $\mathcal{K}^{PL}_{PL,PL^+_S}$

Entrada: Conjunto finito de claves XML $\Sigma \cup \{\varphi\}$ en $\mathcal{K}^{PL}_{PL,PL^+_S}$
Salida: sí, si $\Sigma \models \varphi$; no, en otro caso
1: Construir $G_{\Sigma,\varphi}$ para Σ y φ;
2: **if** q_φ es alcanzable desde q'_φ en $G_{\Sigma,\varphi}$ **then**
3: **return** si;
4: **else**
5: **return** no;
6: **end if**

En el Ejemplo 3.5 ya que q_φ es alcanzable desde q'_φ en $G_{\Sigma,\varphi}$, según el Teorema 3.1 se sigue que el conjunto Σ de claves implica φ.

A continuación se discute el detalle de la estrategia usada en este trabajo para implementar el Algoritmo 3.1 de forma eficiente. Esta estrategia sigue la estrategia general presentada en Hartmann & Link (2009a) para obtener una implementación eficiente.

3.3.1 Guía para una implementación eficiente del algoritmo

Aquí se discute una implementación eficiente del Algoritmo 3.1 siguiendo los lineamientos de Hartmann & Link (2009a), y se realiza un análisis de su complejidad temporal. Primero se analiza la complejidad del problema de alcanzabilidad. El problema de decidir si q_φ es alcanzable desde q'_φ en $G_{\Sigma,\varphi}$ (paso 2, Algoritmo 3.1), puede ser resuelto aplicando un algoritmo de búsqueda en profundidad a $G_{\Sigma,\varphi}$ con raíz en q'_φ. Este algoritmo trabaja en tiempo lineal en el número de aristas de $G_{\Sigma,\varphi}$ (Jungnickel, 2007).

El algoritmo de búsqueda en profundidad (*Depth First Search, DFS*) es una técnica general para recorrer un grafo (digrafo); sigue caminos tanto como sea posible: desde un vértice v ya alcanzado, avanza por todo vértice w adyacente a v que no ha sido visitado; luego recursivamente continúa desde w a otro vértice (adyacente a w) no visitado aún, mientras esto sea posible. (Si no se puede continuar, retrocede en los vértices visitados tanto como sea necesario.) Un recorrido DFS de un grafo G, (i) visita todos los vértices y aristas de G, (ii) determina si G es conectado, (iii) calcula una foresta de expansión[1] de G.

Realizando una extensión (especialización) de DFS, se puede resolver el problema de encontrar y reportar un camino entre dos nodos dados en un grafo. En este caso específico, se trabajará DFS sobre grafos dirigidos (digrafos). La extensión aplicada a DFS considera agregar como parámetro, junto al digrafo G, un vértice de partida v, y un vértice de destino z. Luego, se desea encontrar (si existe) un camino desde v hasta z en G. (Nótese, que en este caso particular no es necesario reportar dicho camino, sino sólo determinar su existencia.)

El Algoritmo 3.2 determina la existencia de un camino entre un par de vértice v y z en un digrafo G, comenzando el camino en v y finalizando en z. Se considera que cada vértice en G posee una etiqueta, cuyo valor indica si el vértice ha sido visitado o no, de esta manera no se visita más de una vez un vértice y no se entra en ciclos. Esta etiqueta, para un nodo x, es consultada por la función

[1] Es un subgrafo acíclico con el mismo conjunto de nodos que G.

Algoritmo 3.2: *$CaminoDFS(G, v, z)$: Búsqueda en profundidad*

Entrada: Un grafo dirigido G, un nodo de partida v y un nodo destino z
Salida: sí, si existe un camino desde v hasta z; no, en otro caso

```
1: if getEtiqueta(v) == VISITADO then
2:    return no;
3: end if
4: if v = z then
5:    return si;
6: end if
7: setEtiqueta(v, VISITADO)
8: for all e ∈ G.aristasIncidentes(v) do
9:    w = opuesto(v, e)
10:   if CaminoDFS(G, w, z) then
11:      return si;
12:   end if
13: end for
14: return no; //No se puede alcanzar z
```

getEtiqueta(x), y modificada por *setEtiqueta*$(x, VALOR)$. La función *aristasIncidentes*(v), recibe como parámetro un vértice v, y retorna el conjunto de aristas de G, que van desde v hasta otro nodo w, donde $(v, w) \in E$. La función *opuesto*(v, e), recibe como parámetros un vértice v y una arista e que emerge de v, y retorna el vértice sobre el cual incide: si $e = (v, w) \in E$, *opuesto*(v, e) retorna w. Cada arista en el grafo conectado que es parte de v es usada exactamente una vez en cada dirección durante la ejecución del Algoritmo 3.2. Por lo anterior, el Algoritmo 3.2 posee una complejidad $\mathcal{O}(|E|)$ para grafos conectados (Jungnickel, 2007), es decir, su complejidad es lineal en el número de aristas en el grafo.

Volviendo al problema de alcanzabilidad sobre $G_{\Sigma,\varphi}$, este grafo posee $|\varphi|+1$ nodos, por lo cual se puede decidir alcanzabilidad en tiempo $\mathcal{O}(|\varphi|^2)$. Notar que en un grafo completo[2] $G = (V, E)$, si $n = |V|$ se tienen $n(n-1)$ aristas, siendo éste el peor caso.

La construcción del *witness-graph* $G_{\Sigma,\varphi}$ (paso 1, Algoritmo 3.1) se puede generar siguiendo los siguientes pasos:

(A) Inicializar $G_{\Sigma,\varphi}$ con $T_{\Sigma,\varphi}$

(B) Determinar el conjunto de marcado de $T_{\Sigma,\varphi}$

(C) Para cada $\sigma \in \Sigma$ agregar una arista dirigida (w'_σ, w_σ), desde w'_σ hasta w_σ a $G_{\Sigma,\varphi}$, si w_σ y w'_σ testifican la aplicabilidad de σ a φ (y la arista no existe actualmente);

[2]Es un grafo $G = (V, E)$ en el que $\forall u, v \in V, u \neq v, (u, v) \in E$.

Para el paso (A) se necesita construir el *mini-tree* $T_{\Sigma,\varphi}$ para Σ y φ. Una etiqueta ℓ_0 en $\mathbf{E}-\mathcal{L}$ apropiada puede ser encontrada en tiempo $\mathcal{O}(||\Sigma|| + |\varphi|)$, donde $||\Sigma||$ denota la suma de todos los tamaños $|\sigma|$ para σ en Σ (ver Ecuación 3.2). El número de nodos en $T_{\Sigma,\varphi}$ es $|\varphi| + 1$. Así, $T_{\Sigma,\varphi}$ puede ser generado en tiempo $\mathcal{O}(|\varphi|)$. En el paso (B) el conjunto de marcado del *mini-tree* $T_{\Sigma,\varphi}$ puede ser determinado en tiempo $\mathcal{O}(|\varphi|)$.

Una forma (semi-naïve) de ejecutar el paso (C) es realizar la evaluación en $T_{\Sigma,\varphi}$ de $w'_\sigma[\![P_i^\sigma]\!]$ para $i = 1, \ldots, k_\sigma$, para todo $w'_\sigma \in w_\sigma[\![Q'_\sigma]\!]$ y todo $w_\sigma \in [\![Q_\sigma]\!]$. Dado que una consulta de la forma $v[\![Q]\!]$ es una consulta Core XPath (Gottlob et al., 2005) y por lo tanto, puede ser evaluada en un árbol T de nodos etiquetados en tiempo $\mathcal{O}(|T| \times |Q|)$, se tiene que, se puede evaluar $w'_\sigma[\![P_i^\sigma]\!]$ para todo $i = 1, \ldots, k_\sigma$ en tiempo $\mathcal{O}(|\varphi| \times |\sigma|)$. Ya que $[\![Q_\sigma]\!]$ y $w_\sigma[\![Q'_\sigma]\!]$ contienen a lo sumo $|\varphi|$ nodos cada uno, este paso puede ser ejecutado en tiempo $\mathcal{O}(|\varphi|^3 \times |\sigma|)$ por cada σ. Por lo tanto, se puede generar $G_{\Sigma,\varphi}$ en tiempo $\mathcal{O}(||\Sigma|| \times |\varphi|^3)$. De todas formas, se presenta un algoritmo para la determinación del conjunto de nodos alcanzables desde algún nodo v en un árbol XML T, siguiendo un camino-Q, $v[\![Q]\!]$.

Se diseña el algoritmo *NodosAlcanzables* (ver Algoritmo 3.3). En este algoritmo, a partir de un nodo v, se recorre un árbol XML T (recorrido en profundidad), siguiendo las etiquetas presentes en el camino-Q. A partir del nodo v, se quiere encontrar todos los descendientes w alcanzables desde v siguiendo un camino-Q. Sea *pos* un puntero que indica la etiqueta actual en el camino-Q que está siendo evaluada. Cuando $pos = 1$, se comparan las etiquetas de cada uno de los hijos w de v, con la apuntada por *pos* ($Q[pos]$), si son iguales se agrega w a una pila o *stack*. Luego, si la pila no es vacía, se toma el elemento del tope y se obtienen sus hijos, comparando sus etiquetas con la siguiente posición (pos++) del camino-Q. Una vez que se analizan todos los hijos del nodo, este es sacado de la pila y no es vuelto a revisar. Si w no era el único hijo de v, se retrocede una posición (pos- -) en el camino-Q, para analizar los nodos hermanos de w. Así sucesivamente, hasta encontrar los nodos descendientes de v cuya etiqueta sea igual a la última etiqueta presente en el camino-Q, es decir, cuando $pos = |Q|$.

El Algoritmo 3.3 utiliza una pila *s_camino*, para almacenar los nodos que cuya etiqueta es igual a alguna en el camino-Q; y otra pila *s_hijos*, para contabilizar la cantidad de nodos de un mismo padre que poseen la misma etiqueta, presente en el camino-Q. De esta manera cada vez que se analice un nodo de un nivel superior se incrementará la posición, y cada vez que se analice

Algoritmo 3.3: *NodosAlcanzables*(T, v, Q)*: Alcanzabilidad de nodos - Parte 1*

Entrada: Un árbol XML T, un nodo de partida v y un camino Q en PL
Salida: Conjunto *Salida* con los nodos alcanzables desde v siguiendo un camino-Q

```
1: pos = −1; root = v
2: Stack s_camino, s_hijos
3: (1) pos++; //Puntero a la actual etiqueta dentro de Q
4: (2) hijos_root = Listado de hijos de root
5: if (hijos_root.size() > 1) then
6:    for i = 0; i < (hijos_root.size() - 1 ); i++ do
7:       s_camino.push(hijos_root[i]);
8:       s_camino.push(−1);
9:       s_hijos.push(pos);
10:   end for
11: else
12:   if (hijos_root.size() == 1) then
13:      s_camino.push(hijos_root[0])
14:   end if
15: end if
16: //Continúa en la siguiente página...
```

un nodo de nivel inferior, la posición disminuirá, manteniendo el puntero en la posición adecuada dentro del camino-Q. Con este mismo objetivo, cada vez que la etiqueta de un nodo w coincide con la etiqueta apuntada por *pos*, el nodo es agregado a *s_camino*. Si hay más de un nodo w (varios nodos hermanos hijos de v) con la misma etiqueta, cada vez que se agrega un nodo, se agrega un valor -1 como separador a *s_camino*. Además, se agrega el valor actual de *pos* a la pila *s_hijos*, tantas veces como cantidad de hermanos con la misma etiqueta se tenga. Finalmente, aquellos nodos cuya etiqueta sea igual a la última posición del camino-Q, son agregados a un conjunto *Salida*. Si no se cumple la igualdad de las etiquetas, se elimina el nodo del *stack s_camino*, y por ende, todos sus descendientes –realizando una poda del árbol de búsqueda y, reduciendo el espacio de búsqueda–.

El algoritmo diseñado para determinar el conjunto $v[\![Q]\!]$, acepta que Q sea una expresión PL de la forma $A._*.B$, donde A es una expresión PL_s y B es una expresión PL.

En realidad, como se ve en Hartmann & Link (2009a) no es necesario determinar *todas* las aristas testigo (w', w) para decidir si q_φ es alcanzable desde q'_φ en $G_{\Sigma,\varphi}$, basta con considerar si q_φ es alcanzable desde q'_φ en un grafo de expansión $H_{\Sigma,\varphi}$ de $G_{\Sigma,\varphi}$, en el cual ciertas aristas testigo son omitidas. Esto permite reducir la complejidad temporal de la estrategia delineada en el párrafo anterior. Se describe a continuación esta estrategia revisada para determinar las aristas testigo (w', w) que van desde w' hasta w, correspondientes a una clave $\sigma \in \Sigma$.

Algoritmo 3.3: *Alcanzabilidad de nodos - Parte 2*

```
1: (3) //Comparación etiquetas
2: if (s_camino.size() > 0) then
3:    if (s_camino.top() == -1) then
4:       s_camino.pop()
5:       pos = s_hijos.top()
6:       s_hijos.pop()
7:    end if
8:    if (s_camino.top() ≠ Q[pos]) then
9:       if (Q[pos == _*]) then
10:         if (Q[pos + 1] == ε) then
11:            root = s_camino.top()
12:            Salida.agregar(s_camino.top())
13:         else
14:            if (s_camino.top() != Q[pos + 1]) then
15:               root = s_camino.top()
16:            else
17:               Salida.agregar(s_camino.top())
18:               root = s_camino.top()
19:            end if
20:         end if
21:         s_camino.pop()
22:         pos–
23:         volver a (1)
24:      else
25:         s_camino.pop()
26:         s_hijos.pop()
27:         volver a (3)
28:      end if
29:   end if
30:   if (s_camino.top() == Q[pos] and pos < (Q.largo() - 1)) then
31:      root = s_camino.top()
32:      s_camino.pop()
33:      volver a (1)
34:   end if
35:   if (s_camino.top() == Q[pos] and pos == (Q.largo() - 1)) then
36:      Salida.agregar(s_camino.top())
37:      s_camino.pop()
38:      if (s_camino.size() > 0) then
39:         volver a (3)
40:      else
41:         return Salida
42:      end if
43:   end if
44: else
45:   return Salida
46: end if
```

Sea W'_σ el conjunto de todos los nodos w' en $T_{\Sigma,\varphi}$, para los cuales existe algún nodo w en $T_{\Sigma,\varphi}$ tal que, w y w' testifican la aplicabilidad de σ a φ. Además, para cada $w' \in W'_\sigma$, sea $W_\sigma(w')$ el conjunto de todos los nodos w en $T_{\Sigma,\varphi}$ tal que, w y w' testifican la aplicabilidad de σ a φ. Las aristas testigo son entonces los pares (w', w) con $w' \in W'_\sigma$ y $w \in W_\sigma(w')$. Como se demuestra en Hartmann & Link (2009a), no es necesario determinar por completo el conjunto $W_\sigma(w')$ para cada $w' \in W'_\sigma$, es suficiente con considerar sólo el ancestro más alto de q'_φ en $T_{\Sigma,\varphi}$, esto es aquél nodo con el menor camino desde la raíz, que pertenece a $W_\sigma(w')$, el cual será denotado por $w^{top}_\sigma(w')$ (si es que existe).

Por lo tanto, se necesita para aplicar esta estrategia, determinar el conjunto W'_σ, y luego, para cada $w' \in W'_\sigma$, determinar $w^{top}_\sigma(w')$. Por definición, W'_σ consiste de todos los nodos $w' \in [\![Q_\sigma.Q'_\sigma]\!]$ en $T_{\Sigma,\varphi}$ tal que, para cada $i = 1, \ldots, k_\sigma$, exista un nodo marcado en $w'[\![P^\sigma_i]\!]$. Nótese, que $[\![Q_\sigma.Q'_\sigma]\!]$ puede ser evaluado en $T_{\Sigma,\varphi}$ en tiempo $\mathcal{O}(|\varphi| \times |Q_\sigma.Q'_\sigma|)$.

$$W'_\sigma = \{w' \in [\![Q_\sigma.Q'_\sigma]\!] \text{ en } T_{\Sigma,\varphi} \mid \forall i = 1, \ldots, k_\sigma, \text{ existe un nodo marcado en } w'[\![P^\sigma_i]\!]\} \tag{3.4}$$

Luego, se selecciona algún $i \in \{1, \ldots, k_\sigma\}$. Sea v un nodo del conjunto $\mathcal{M}$ de marcado, y sea u el ancestro de v que se ubica $|P^\sigma_i|$ niveles sobre v en $T_{\Sigma,\varphi}$ (si es que existe). Recordar, que el *nivel* de un nodo n en un árbol es el largo del camino (único) desde la raíz del árbol al nodo n. Se puede entonces determinar si $v \in u[\![P^\sigma_i]\!]$, esto es, si el único camino desde u hasta v es un camino-P^σ_i. Esto puede ser realizado en tiempo $\mathcal{O}(min\{|P^\sigma_i|, |\varphi|\})$, ya que P^σ_i es una expresión PL_s. Revisando todos los nodos $v \in \mathcal{M}$, se obtiene el conjunto U^σ_i de todos los nodos u en $T_{\Sigma,\varphi}$ para los cuales $u[\![P^\sigma_i]\!] \cap \mathcal{M} \neq \emptyset$. El tiempo total para este pase es $\mathcal{O}(|\mathcal{M}| \times |P^\sigma_i|)$.

Por definición, W'_σ es la intersección de $[\![Q_\sigma.Q'_\sigma]\!]$ con el conjunto U^σ_i, $i = 1, \ldots, k_\sigma$. Con lo anterior, W'_σ se puede determinar en tiempo $\mathcal{O}(|\varphi| \times |Q_\sigma.Q'_\sigma| + |\mathcal{M}| \times \sum_{i=1}^{k_\sigma} |P^\sigma_i|)$, y por lo tanto en tiempo $\mathcal{O}(|\varphi| \times |\sigma|)$.

Ahora sólo resta determinar $w^{top}_\sigma(w')$ para cada $w' \in W'_\sigma$ (si es que existe). Para resolver esto, se presenta el Algoritmo 3.4. Nótese, que dicho algoritmo usa el hecho que si Q'_σ es de la forma $A.\_^*.B$, donde A es una expresión PL_s y B es una expresión PL, entonces $w^{top}_\sigma(w')$ es el ancestro más alto w de q'_φ en $T_{\Sigma,\varphi}$ que pertenece a $[\![Q_\sigma]\!]$ y para el cual $w[\![A]\!]$ es no vacío (ver Lema 4.5 en Hartmann & Link (2009a)). En particular, $w^{top}_\sigma(w')$ es independiente del w' en W'_σ que se elija.

En el paso 6 del Algoritmo 3.4, el conjunto $[\![Q_\sigma.A]\!]$ de nodos en $T_{\Sigma,\varphi}$ puede ser evaluado en tiempo $\mathcal{O}(|\varphi| \times |Q_\sigma.A|)$, y por lo tanto, el Algoritmo 3.4 toma un tiempo $\mathcal{O}(|\varphi| \times |Q_\sigma.A|)$.

Algoritmo 3.4: *Obtenerwtop(w'): Determinar el $w_\sigma^{top}(w')$*

Entrada: Mini-tree $T_{\Sigma,\varphi}$ y $w' \in W'_\sigma$
Salida: $w_\sigma^{top}(w')$

1: **if** Q'_σ es una expresión PL_s **then**
2: $w_\sigma^{top}(w')$ es el nodo ubicado $|Q'_\sigma|$ niveles arriba de w' en $T_{\Sigma,\varphi}$
3: **return** //Entregar $w_\sigma^{top}(w')$
4: **else**
5: Q'_σ contiene un $\_^*$, y tiene la forma $A.\_^*.B$ donde A es una expresión PL_s y B una expresión PL
6: Determinar el conjunto $[\![Q_\sigma.A]\!]$ de nodos en $T_{\Sigma,\varphi}$
//En particular, $w_\sigma^{top}(w')$ es independiente de la elección de w' en W'_σ
//Este único nodo se denota w_σ^{top}
//Pasos para determinar w_σ^{top}
7: **if** $[\![Q_\sigma.A]\!] \neq \emptyset$ **then**
8: Escoger el nodo v más arriba y considerar el nodo w que se ubica $|A|$ niveles sobre v en $T_{\Sigma,\varphi}$
9: **if** w es un ancestro de q'_φ **then**
10: **return** w //w es el nodo w_σ^{top} que se busca
11: **else**
12: w_σ^{top} no existe, y $w_\sigma^{top}(w')$ no existe para todo $w' \in W'_\sigma$
13: **end if**
14: **end if**
15: **end if**

En conclusión, cada uno de los dos pasos requiere un tiempo $\mathcal{O}(|\varphi| \times |\sigma|)$. Por lo tanto, se pueden determinar todas las aristas testigo derivadas de σ, que son necesarias para decidir la alcanzabilidad de q_φ desde q'_φ en $G_{\Sigma,\varphi}$, en tiempo $\mathcal{O}(|\varphi| \times |\sigma|)$.

Teorema 3.2 [Hartmann & Link (2009a)] Sea $\Sigma \cup \{\varphi\}$ un conjunto finito de claves XML en $\mathcal{K}^{PL}_{PL,PL_s^+}$. El problema de implicación $\Sigma \models \varphi$ se puede decidir en tiempo $\mathcal{O}(|\varphi| \times (||\Sigma|| + |\varphi|))$.

Demostración De la discusión anterior, se obtiene que la construcción de un subgrafo G del *witness-graph* $G_{\Sigma,\varphi}$, tal que q_φ es alcanzable desde q'_φ en G siempre que q_φ sea alcanzable desde q'_φ en $G_{\Sigma,\varphi}$, toma un tiempo $\mathcal{O}(|\varphi| \times ||\Sigma||)$. El problema de decidir si q_φ es alcanzable desde q'_φ en G, y por ende en $G_{\Sigma,\varphi}$ puede ser resuelto en tiempo lineal en el número de aristas de G; recordar Algoritmo 3.2 (ver Jungnickel (2007)). Ya que el número de nodos en G, como también en $G_{\Sigma,\varphi}$, es $|\varphi| + 1$, la alcanzabilidad puede ser decidida en tiempo $\mathcal{O}(|\varphi|^2)$. Esto da un tiempo total para el algoritmo de implicación de claves de $\mathcal{O}(|\varphi| \times (||\Sigma|| + |\varphi|))$. □

CAPÍTULO 4. IMPLEMENTACIÓN ALGORITMO DE IMPLICACIÓN

La prueba de completitud de la axiomatización de claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$ presentada en Hartmann & Link (2009a), es la base para la definición de un algoritmo de implicación de claves XML que trabaja en tiempo cuadrático en el tamaño de la clave de entrada. En el Capítulo 3, se analizó detalladamente dicho algoritmo y se presentó una pauta para su implementación. En este capítulo, se detalla la primera contribución de este trabajo, definiendo una metodología, una representación (estructuras de datos), y algunas funciones, que luego serán utilizadas en la implementación del Algoritmo 3.1. Usando esta implementación se analizará en el Capítulo 7 el comportamiento del algoritmo en la práctica. Hasta donde sabemos, según la revisión bibliográfica, ésta es la primera implementación de un algoritmo para la implicación de esta clase de claves XML.

4.1 DETALLES DE LA IMPLEMENTACIÓN

Se presentan a continuación los detalles de la implementación desarrollada del algoritmo de implicación de claves XML en la clase $\mathcal{K}^{PL}_{PL,PL^+_s}$. Recordar que la clase $\mathcal{K}^{PL}_{PL,PL^+_s}$ contiene aquellas claves con un conjunto no vacío de caminos claves en PL_s. El algoritmo de implicación de claves XML (ver Algoritmo 3.1 en Capítulo 3), traduce el problema de implicación de claves XML en un problema de alcanzabilidad de nodos en un grafo, utilizando dos herramientas: llamadas *mini-tree* y *witness-graph*. Como se vio en el Capítulo 3, el *mini-tree* $T_{\Sigma,\varphi}$ es una representación en forma de árbol de la clave φ, para la cual se desea averiguar si es implicada por las claves en Σ. Usando $T_{\Sigma,\varphi}$ como base, se calcula el impacto o aplicabilidad de cada una de las claves $\sigma \in \Sigma$ a φ, agregando aristas testigo a $T_{\Sigma,\varphi}$, y generando un grafo dirigido llamado *witness-graph* $G_{\Sigma,\varphi}$.

Recordar, que el algoritmo de implicación de claves XML posee básicamente dos pasos: (1) la construcción del *witness-graph* $G_{\Sigma,\varphi}$ a partir del *mini-tree* $T_{\Sigma,\varphi}$, y (2) determinar la alcanzabilidad de q_φ desde q'_φ en el *witness-graph* $G_{\Sigma,\varphi}$.

Para implementar el paso (1), primero se requiere utilizar una estructura de árbol para representar un *mini-tree*, y luego una estructura de grafo para representar un *witness-graph*. Nótese, que un árbol es un tipo especial de grafo conexo, con la restricción de no poseer ciclos[1] (ser acíclico), y en el cual hay exactamente un camino entre todo par de vértices. Un *árbol* es una estructura no lineal y homogénea, en la cual cada elemento puede tener un número indefinido de hijos (por lo que también se denomina estructura no rankeada), pero tan sólo puede tener un padre, a diferencia de un grafo que puede tener un número indeterminado tanto de padres como de hijos. De ésta manera, es suficiente con lograr la representación de un grafo, para tener la de árbol. Esta estructura debe permitir almacenar los correspondientes nodos con su información, y la interconexión entre estos, dada por las aristas. De este modo, una vez formado el árbol –*mini-tree*–, la estructura debe permitir agregar aristas testigo o aristas ascendentes para generar el grafo –*witness-graph*–. Luego, se cumple que el *mini-tree* $T_{\Sigma,\varphi}$ posee el mismo conjunto finito de vértices V que el *witness-graph* $G_{\Sigma,\varphi}$, y un subconjunto E' del conjunto de aristas E de $G_{\Sigma,\varphi}$, por lo tanto el *mini-tree* $T_{\Sigma,\varphi}$ se dice que se encuentra contenido en el *witness-graph* $G_{\Sigma,\varphi}$.

Luego, el paso (2) de verificación de alcanzabilidad de q_φ desde q'_φ en $G_{\Sigma,\varphi}$, puede ser resuelto utilizando el Algoritmo 3.2 de búsqueda en profundidad.

El primer paso a seguir es definir la representación de las estructuras de manera abstracta. Existen varias estructuras de datos que pueden ser utilizadas para representar grafos y digrafos (grafos dirigidos). La elección de la estructura de datos adecuada depende del tipo de operaciones que se desee realizar sobre ella, es decir, sobre sus conjuntos de vértices y aristas. Las representaciones más comunes son las matrices de adyacencia y las listas de adyacencia (Aho & Hopcroft, 1974; Aho et al., 1983). Cada una posee ventajas y desventajas, en aspectos como asignación de memoria, tiempos de inserción, eliminación y búsqueda, formas de recorrido, entre otras.

Dado un grafo $G = (V, E)$ con $V = \{1, \ldots, n\}$, la *matriz de adyacencia* de G es una matriz A de booleanos de tamaño $n \times n$ en la que $A[i][j]$ es 1 si y sólo si, la arista (i, j) que une al vértice i con el vértice j está en E. Ésta puede ser una representación adecuada en casos en que

[1]Un *ciclo* es un camino que comienza y acaba en el mismo vértice, en el cual todos los vértices del camino son diferentes, a excepción del primer y último.

sea necesario saber con mucha frecuencia si una determinada arista está presente en el grafo. Cosa que para el problema tratado aquí no es relevante. Además, la desventaja principal de utilizar una matriz de adyacencia para representar un grafo (digrafo) es el espacio, ya que se requiere un espacio n^2 incluso si el grafo (digrafo) es *esparso*, es decir, si tiene bastante menos de n^2 aristas, cosa que ocurre en este caso.

Por otro lado, dado un grafo $G = (V, E)$, la *lista de adyacencia* de un vértice i de G, es una lista, en un orden cualquiera, de todos los vértices adyacentes a i. Se puede representar G como un vector L en el que $L[i]$ es un puntero a la lista de adyacencia del vértice i. La cantidad de memoria requerida por esta representación es proporcional a la suma del número de vértices y el número de punteros (que corresponde al número de aristas). Es decir, el costo en memoria posee una cota ajustada $(n + m)$ con $n = |V|$ y $m = |E|$. Ya que las herramientas *mini-tree* y *witness-graph* son grafos esparsos, el coste de memoria de esta estructura es mucho menor que si se utiliza una representación matricial (matriz de adyacencia). La desventaja de esta representación es que el determinar si una arista está o no en el grafo puede tomar tiempo $\mathcal{O}(n)$, ya que el número máximo de vértices que puede haber en la lista de adyacencia de un vértice dado es n.

Por lo expuesto, se decide utilizar listas de adyacencia para representar los grafos utilizados por el algoritmo de implicación de claves XML. Las listas de adyacencia a utilizar tendrán algunas características particulares, requeridas para este trabajo. Estarán compuestas de un vector dinámico L con todos los vértices (nodos) del grafo, y cada nodo almacenará un puntero a información propia del nodo (tal como etiqueta, valor, etc.), un puntero a la lista de nodos adyacentes al nodo i, y un puntero al siguiente nodo de la lista L.

De esta forma se representa una estructura no lineal (árbol y grafo) a través de estructuras lineales (listas y colas). Las estructuras aquí utilizadas para implementar un grafo son dinámicas en su asignación de memoria, debido a que no se sabe de antemano la cantidad de nodos y aristas. Si bien, estas cantidades (nodos y aristas) no son muy grandes, se impone este requerimiento, para mantener la escalabilidad del sistema.

La Figura 4.1 muestra a la izquierda el *witness-graph* $G_{\Sigma,\varphi}$ obtenido en el Ejemplo 3.5, y a la derecha su correspondiente lista de adyacencia. (Nótese, que los nodos del grafo han sido enumerados y etiquetados para facilitar la comprensión de la representación.) En la lista de adyacencia, se puede ver que existe una lista principal de nodos L (vertical), en la cual se encuentran

todos los nodos $v \in V$ del grafo (esta lista no sigue ningún orden en especial más allá de comenzar con el nodo raíz). Cada uno de estos nodos posee un puntero a una estructura con información propia del nodo, y otro a una lista de nodos adyacentes. Así, el `Nodo 0` posee una adyacencia con el `Nodo 1` (arista dirigida desde el nodo 0 al 1), el `Nodo 1` posee una adyacencia con el `Nodo 2` (arista dirigida desde el nodo 1 al 2), y así sucesivamente. Nótese, que el `Nodo 2` y `Nodo 3`, además de poseer las aristas con los nodos inferiores (por la estructura de árbol $T_{\Sigma,\varphi}$), poseen aristas con nodos de nivel superior que representan las aristas testigo, obtenidas de la aplicabilidad de las claves $\sigma \in \Sigma$, presentes en $G_{\Sigma,\varphi}$.

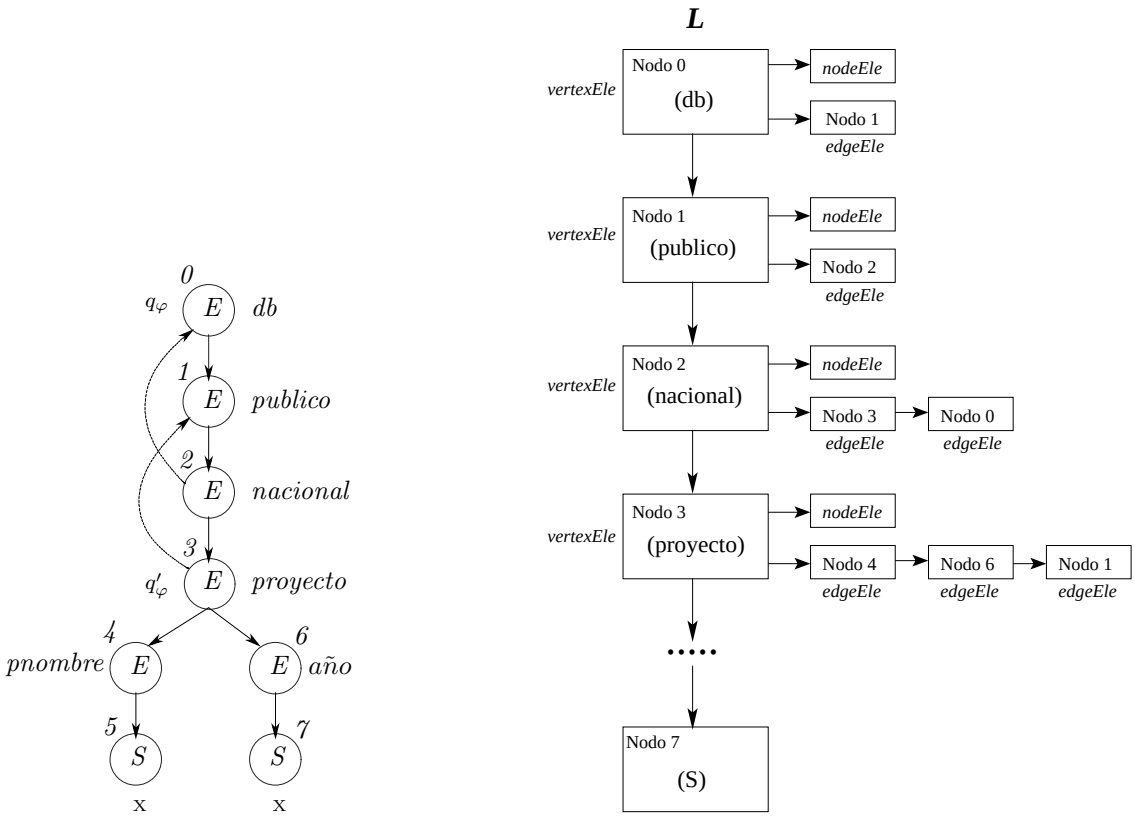


FIGURA 4.1: *Lista de adyacencia para el witness-graph del Ejemplo 3.5.*

Se utiliza la estructura `nodeEle` para almacenar la información necesaria de un nodo: el identificador del nodo, su etiqueta, su tipo (elemento, atributo o texto), y su valor en el caso de ser nodo atributo o texto.

Es conveniente ver una estructura de grafo, y en general las estructuras de datos, con un enfoque de Orientación a Objetos (OO) (Rumbaugh et al., 1991; Schach, 2007). Este enfoque es el utilizado en este trabajo para el diseño de las estructuras y funciones que actúan sobre ellas. El lenguaje escogido para realizar la implementación es C++, utilizando el compilador *g++* de GCC

(*GNU Compiler Collection*)[2] en su versión 4.4. La implementación fue realizada en una máquina Intel Core 2 Duo 2.0 GHz, 3 GB en memoria RAM, Sistema Operativo Linux con Kernel 2.6.32.

Una vez tomada la decisión sobre la forma en la cual se representarán los grafos, resta definir las estructuras de datos necesarias para la implementación.

4.2 ESTRUCTURAS DE DATOS

En esta sección, se presentan las estructuras de datos definidas para representar un grafo, lo cual permite el trabajo con las herramientas *mini-tree* y *witness-graph* en el problema de implicación de claves XML. Se define un conjunto de estructuras de datos, utilizando la notación del lenguaje de programación C++.

La primera estructura es la que permitirá almacenar la información propia del nodo.

Estructura de Nodo: Un nodo es la unidad básica de la representación utilizada. Esta estructura almacena la información propia del nodo, por lo cual posee las propiedades: identificador, etiqueta, tipo, y valor. Cada instancia de esta estructura debe ser única y poseer un único identificador.

```
1 struct nodeEle
2 {
3   int     m_idEle; /**< Identificador del nodo.*/
4   char*   m_label; /**< Etiqueta del nodo.*/
5   char    m_type; /**< Tipo del nodo. E (Elemento), A (Atributo), o S (PCDATA).*/
6   char*   m_val; /**< Valor del nodo.*/
7 };
```

Estructura de Aristas: Dado un nodo $v_i \in V$, esta estructura permite almacenar las referencias a todos los nodos en V hacia los cuales v_i tiene una arista, esto es permite almacenar la lista de adyacencia de v_i. Para esto, se usa una estructura llamada `edgeEle` que permite almacenar una referencia o puntero al primer nodo en la lista de nodos adyacentes a v_i (instancia de `nodeEle`), y un puntero a un siguiente nodo adyacente de v_i (si existe). Con esto se genera la lista de adyacencia de un nodo cualquiera del grafo.

[2]Sitio Web http://gcc.gnu.org/

```
1 struct edgeEle
2 {
3   struct vertexEle *m_connectsTo; /**< Puntero al vertice incidente.*/
4   struct edgeEle   *m_next;   /**< Puntero a la siguiente instancia de edgeEle
5                                    que contiene el siguiente nodo adyacente.*/
6 };
```

Como ejemplo, a la derecha en la Figura 4.1 se muestra que el `Nodo 3`(proyecto) posee dos punteros, el primero a una instancia de `nodeEle` que contiene su información; y el segundo a una instancia de `edgeEle`, que representa el inicio de su lista de adyacencia, la cual contiene punteros a los nodos 4, 6, y 1, que son también instancias de `edgeEle`. Por ejemplo, en la lista de adyacencia del nodo 3, `Nodo 4` es una instancia de `edgeEle`, que contiene: una referencia al `Nodo 4`(pnombre) de la lista principal L (lista vertical de la Figura 4.1), y una referencia al siguiente elemento de la lista de adyacencia `Nodo 6`, que se comporta del mismo modo. Sin embargo, en la lista de adyacencia del nodo 0, se tiene un solo elemento `Nodo 1`, que sólo hace referencia a `Nodo 1` en la lista principal L, y no posee un puntero a un siguiente elemento de la lista (apunta a `NULL`), ya que es el primer y último nodo de esa lista de adyacencia.

Con las estructuras ya explicadas, se puede almacenar la información y representar las aristas de un nodo en un grafo. Sólo resta definir la estructura que representará el grafo en sí, con todos sus nodos.

Estructura de Grafo: La lista principal L mencionada es creada a partir de la estructura `vertexEle`, la cual junto con las demás estructuras representan el grafo en sí, donde L posee todos los vértices del grafo. Cada vértice, instancia de `vertexEle`, representa un nodo distinto del grafo, posee un puntero a una instancia de `nodeEle` con la información del nodo, y otro a las aristas salientes de este nodo, instancias de `edgeEle`. Además, a cada vértice se le agrega una bandera o *flag* que será utilizada por el algoritmo de alcanzabilidad (con el cual se determinará si un nodo ha sido visitado), una bandera o *flag* que permitirá saber si un nodo pertenece al conjunto de marcado $\mathcal{M}$, y un puntero al siguiente elemento de la lista principal (el cual es también instancia de `vertexEle`). Ésta es la estructura principal usada para representar un grafo en la implementación, la cual puede ser recorrida a través del puntero llamado `m_next`, el cual apunta a la siguiente instancia de `vertexEle` dentro de L.

```
1  struct vertexEle
2  {
3    /**< Puntero al nodo con informacion del vertice.*/
4    struct nodeEle *m_node;
5    /**< Puntero a la lista de adyacencia del nodo.*/
6    struct edgeEle *m_edges;
7    /**< Flag que indica si el nodo ha sido visitado o no, en DFS.*/
8    bool visited;
9    /**< Flag que indica si el nodo pertenece al Marking del mini-tree.*/
10   bool marked;
11   /**< Puntero al siguiente vertice de la lista.*/
12   struct vertexEle *m_next;
13 };
```

Nótese, que la estructura anterior permite representar las herramientas *mini-tree* y *witness-graph* requeridas por el algoritmo de implicación de claves XML. Además, se puede ver cómo esta estructura sirve para representar tanto un árbol como un grafo (recordar que un árbol es un grafo conexo y acíclico con un único nodo raíz).

Luego, lo primero es definir una instancia de `vertexEle` y asignarle una instancia de `nodeEle` a `m_node` con la información del nodo. Antes de definir las adyacencias a cada nodo, nótese que debe primero existir la lista L con todas las instancias de `vertexEle`, ya que a estas instancias es a las cuales se hace referencia en dicha lista. Una vez que se tenga la lista con todos los nodos del grafo, recién se pueden definir las listas de adyacencia para cada nodo.

La Figura 4.2 muestra un conjunto de instancias de `vertexEle` que representan el grafo de la Figura 4.1. Se tiene una lista principal `Lista` en la cual se tienen todos los nodos del grafo, y cada nodo tiene los punteros y propiedades mencionadas anteriormente. El recorrido de la lista se realiza usando `m_next`. Por ejemplo, para acceder al nodo con `m_idEle` igual a 2 (en la posición 3 de la lista), se utiliza `Lista->m_next->m_next`, donde `Lista` apunta al primer elemento de la lista, y luego se avanza dos posiciones en esta lista para acceder a la posición 3 deseada. Del mismo modo, para acceder a la lista de adyacencia de este nodo 2, se utiliza `Lista->m_next->m_next->m_edges`, con lo cual se tiene un puntero a la primera instancia de `edgeEle` en la lista de adyacencia del nodo 2. Si a lo anterior, se le agrega otro puntero `m_next`, se obtiene la segunda instancia de `edgeEle` en la lista de adyacencia del nodo 2. A su vez, estas instancias apuntan mediante `m_connectsTo` a las instancias de `vertexEle` que representan el primer y el segundo nodo adyacente al nodo 2.

Con la representación y estructuras definidas, es posible definir la arquitectura de la implementación.

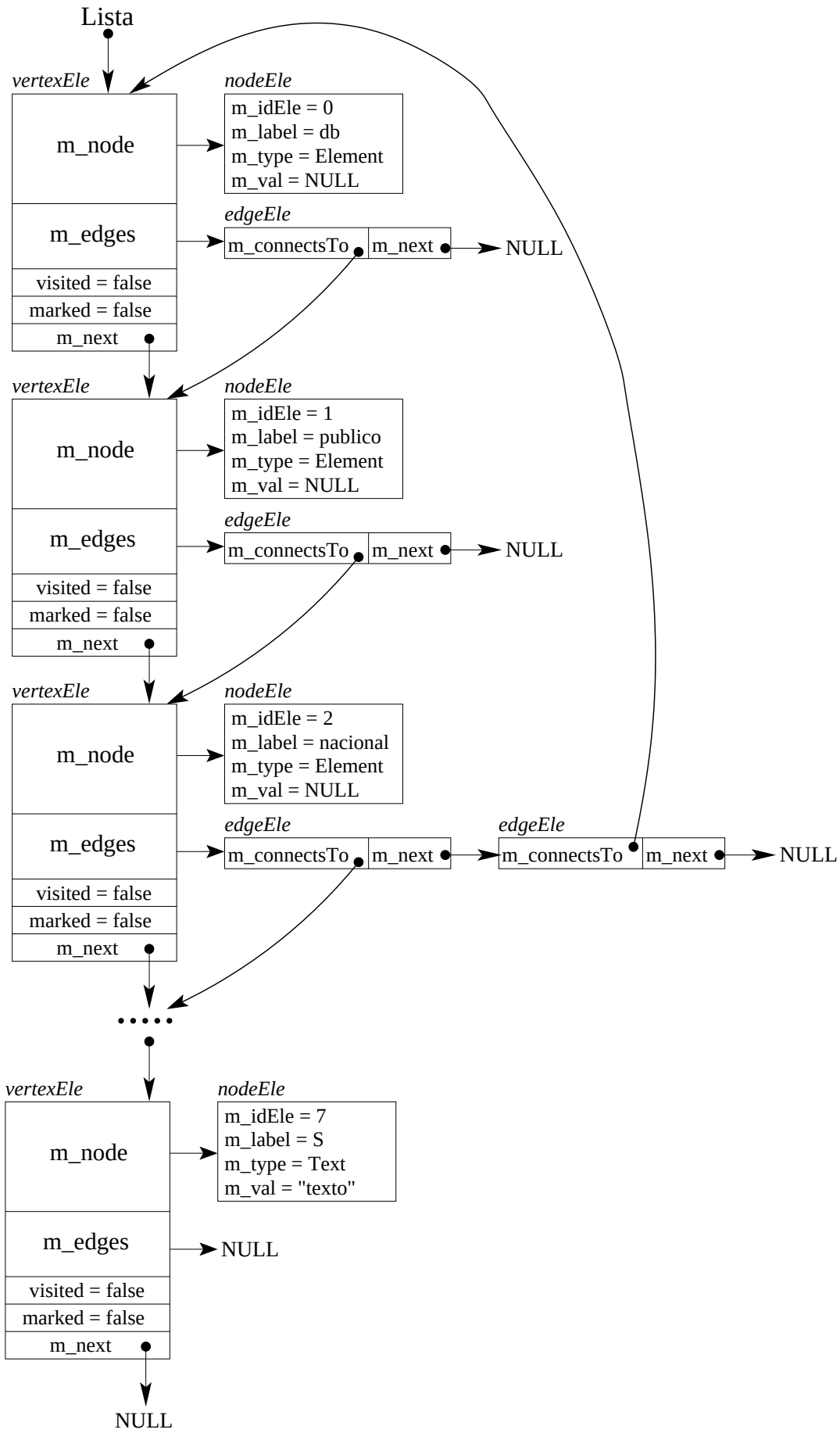


FIGURA 4.2: *Conjunto de instancias que representa el grafo de la Figura 4.1.*

4.3 CLASES DE LA IMPLEMENTACIÓN

La implementación del algoritmo de implicación de claves XML para $\mathcal{K}^{PL}_{PL,PL^+_s}$ presentará las estructuras definidas en la Sección 4.2. A partir de ahora, esas estructuras, siguiendo el paradigma de *Orientación a Objetos (OO)*, estarán contenidas en clases y se referirá a ellas como tal.

Se plantea un conjunto de cuatro clases y tres archivos de cabecera que capturan los principales conceptos definidos por la representación y estructura a utilizar, los cuales son necesarios para el funcionamiento e implementación del algoritmo de implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$. Dentro de este conjunto, se cuenta con una clase *Main* o *Principal* de la implementación, que es la primera clase en ser instanciada, y la que se encarga de instanciar a las demás; se cuenta además, con tres archivos denominados *Header files* o *Archivos cabecera*, los cuales contienen declaraciones directas de las clases, variables y funciones, que son utilizadas (referidas) por otras clases; y otras tres clases que definen las funciones (presentan el cuerpo de las funciones) declaradas en los archivos cabecera.

Para describir cada una de las clases de las cuales se compone la implementación, se presenta el siguiente diccionario de clases:

pathExpressions Clase que captura el concepto de expresiones de camino para selección de nodos en árboles XML. Esta clase, permite implementar la gramática para PL definida en la Ecuación 2.1. Para ello, permite almacenar cadenas de caracteres, más el elemento vacío ε, y el comodín de longitud variable $\_^*$, junto al operador binario "." usado para concatenar dos expresiones de camino. Aquí, se define que cada expresión de camino: posee un tipo que puede ser PL o PL_s (si no posee el símbolo comodín $\_^*$), posee un largo definido por su número de etiquetas, puede estar en forma normal o no, y se le pueden reemplazar sus comodines $\_^*$ por alguna otra cadena de texto (etiqueta).

xmlKey Esta clase captura el concepto de *clave XML*. Según las claves trabajadas aquí, esta clase contiene un camino de contexto, camino objetivo, y un conjunto de caminos clave, instancias de la clase `pathExpressions`. Esta clase, además de permitir la configuración de estos caminos, permite conocer su valor y, saber qué etiquetas componen la instancia de una clave.

graphTool Esta clase es la que define la estructura de un grafo, por lo cual captura los conceptos de *mini-tree* y *witness-graph* (de acuerdo a las estructuras antes definidas). Es en esta clase en donde se declaran y definen las estructuras de datos `nodeEle`, `edgeEle`, y `vertexEle` propuestas anteriormente, para lograr la representación de los *mini-trees* y *witness-graphs*. Además, es la encargada de almacenar la instancia de la clave XML φ, que dará paso a la construcción inicial del *mini-tree*, y luego del *witness-graph*. Por lo cual, mantiene referencias a instancias de las estructuras `pathExpressions` y `xmlKey`. Es la clase más importante ya que es la que implementa el algoritmo de implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$. En resumen, esta clase recibe el conjunto de claves $\Sigma \cup \{\varphi\}$ (codificadas en un archivo de entrada) y genera el *mini-tree* $T_{\Sigma,\varphi}$. Luego, identifica el conjunto de marcado de $T_{\Sigma,\varphi}$, y procede a determinar las aristas testigo, las cuales se agregan a $T_{\Sigma,\varphi}$ y generan el *witness-graph* $G_{\Sigma,\varphi}$. También, implementa una función que permite decidir la alcanzabilidad de dos nodos en el grafo.

main La clase main o principal de la implementación, que recibe las entradas (archivo con el conjunto de claves $\Sigma \cup \{\varphi\}$, y una etiqueta l_0 que pertenece a $\mathbf{E}$-$\mathcal{L}_{\Sigma,\varphi}$), para luego instanciar a las demás clases y ejecutar el algoritmo de implicación de claves XML.

Por cada una de las tres primeras clases descritas, se tiene tanto un archivo cabecera (archivo de extensión ".h") con las declaraciones de funciones, como un archivo C++ (archivo de extensión ".cpp") con la definición de dichas funciones. Así, si el archivo cabecera `pathExpressions.h` posee la declaración `void operator+(char*&)`, el archivo `pathExpressions.cpp` posee la siguiente definición.

```
1 void pathExp::operator +(char*& cExp)
2 {
3   char *cad;
4   cad = new char[50];
5   strcpy(cad, cExp);
6   path.push_back(cad);
7 }
```

Las clases descritas, forman parte de la arquitectura de la implementación del algoritmo de implicación de claves XML, presentada en la Figura 4.3. En esta figura es posible identificar las cuatro clases mencionadas más los tres archivos cabecera, y la relación existente entre estos. Es posible además ver en las definiciones dadas para las clases, que existen relaciones específicas entre ellas. Por ejemplo, la clase `graphTool` posee una relación estática con las clases `pathExpressions`

y `xmlkey`, es decir, el tiempo de vida de éstas últimas está condicionado por el tiempo de vida de `graphTool`. A esta relación estática o por valor se le denomina composición, y se dice que `graphTool` se compone de las clases `pathExpressions` y `xmlkey`. También, se dice que `xmlKey` se compone de `pathExpressions`.

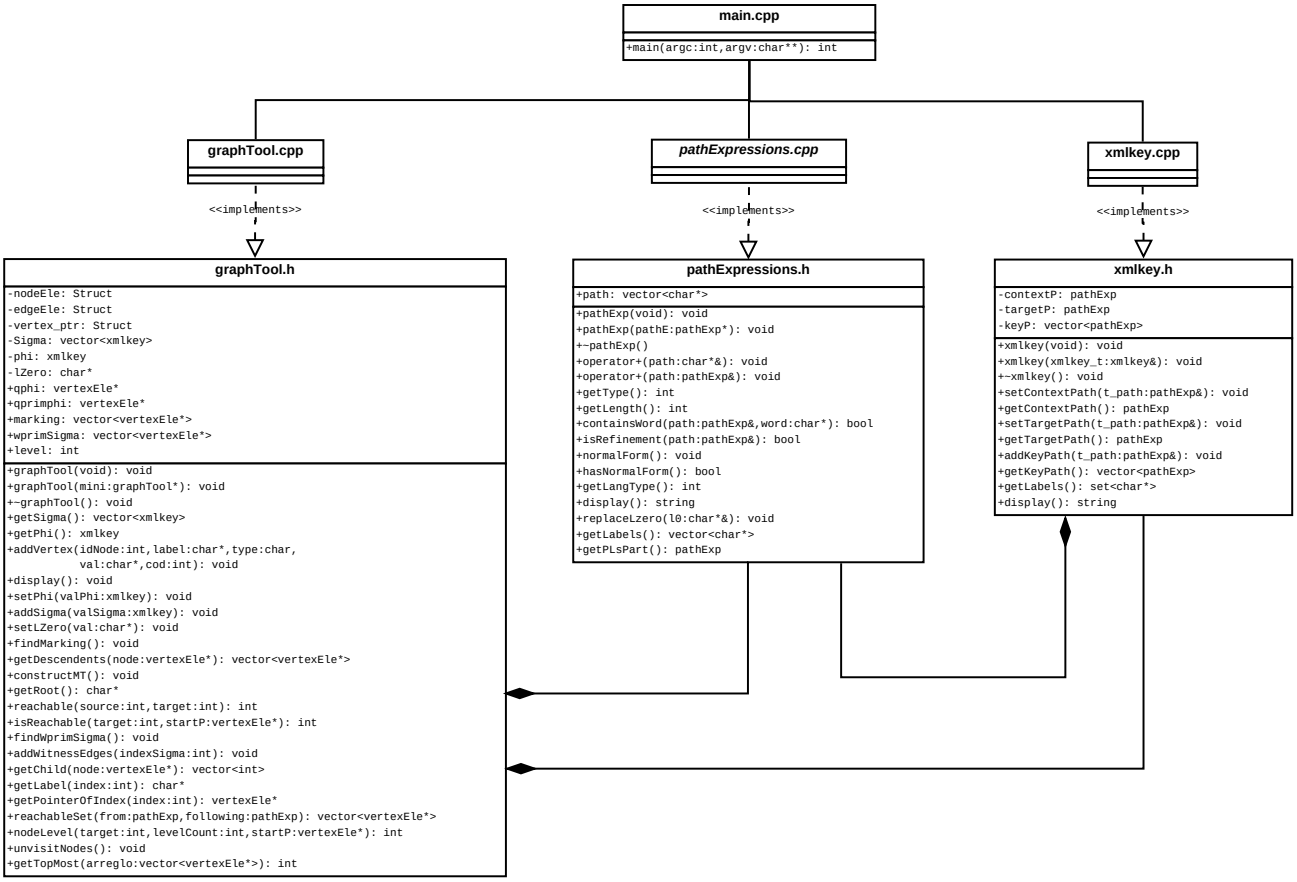


FIGURA 4.3: *Diagrama de clases para la implementación del algoritmo de implicación.*

4.4 DETALLES PARA LA UTILIZACIÓN DE LA IMPLEMENTACIÓN

Tanto los archivos ejecutables, como los códigos fuentes de la implementación desarrollados, se encuentran en el CD-Rom que se adjunta a esta tesis.

TABLA 4.1: *Estructura archivo de entrada implementación de implicación de claves.*

Formato Archivo	Clave XML φ Clave XML σ_1 ... Clave XML σ_n (línea vacía)
Ejemplo	$epsilon; proyecto; pnombre.S$ $\_^*.proyecto.equipo; persona; idpersona.S$ $epsilon; persona; idpersona.S$

4.4.1 Archivo de entrada

La implementación del algoritmo de implicación de claves, tal como indica el algoritmo, recibe como entrada un conjunto $\Sigma \cup \{\varphi\}$ de claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$. Estas claves deben ser pasadas como parámetros a la implementación, a través, de un archivo con una estructura definida.

La estructura definida para el archivo de entrada es mostrada en la Tabla 4.1. La notación definida para cada clave es la siguiente. El camino de contexto se separa del camino objetivo mediante un ";". El camino objetivo se separa del conjunto de caminos clave mediante un ";". Cada elemento de los caminos claves se separa de otro mediante ",". Si una clave contiene el símbolo comodín $\_^*$, este se escribe tal cual "_*" en el archivo de entrada, mientras el símbolo "ε" para la palabra vacía se escribe "epsilon".

Se requiere que al final del archivo de entrada, se deje una línea en blanco que represente el fin del archivo. Es decir, si el conjunto $\Sigma \cup \{\varphi\}$ contiene 5 claves, el archivo de entrada que representa este conjunto debe contener 6 líneas, una línea por cada clave más una línea vacía. Nótese, que la primera clave es la que se considera como φ, se considera que las otras pertenecen a Σ.

4.4.2 Ejecución de la implementación

Para la ejecución de esta implementación del algoritmo de implicación de claves XML, programada en el lenguaje C++, se debe tener instalada alguna distribución Linux con Kernel 2.6.32 (o posterior) sobre una máquina x86. En el caso de requerir recompilar el código fuente, se debe tener

instalada la versión 4.4 (o posterior) del compilador *g++* de GCC[3].

La ejecución de la implementación, puede ser iniciada en una consola con el siguiente comando: `./xmlkeys -f archivo -l etiqueta [-m] [-w]`, donde `archivo` debe ser reemplazado por el nombre del archivo que contenga el conjunto de claves $\Sigma \cup \{\varphi\}$ (en el formato indicado en la Sección 4.4.1); `etiqueta` debe ser reemplazado por el valor de la etiqueta $\ell_0 \in \mathbf{E}{-}\mathcal{L}_{\Sigma,\varphi}$; `-m` es opcional, y permite desplegar el conjunto de marcado; `-w` también es opcional, y permite desplegar una representación del *witness-graph* resultante.

4.5 DETALLE DE ALGUNAS FUNCIONES PRINCIPALES DE LA IMPLEMENTACIÓN

Las clases definidas en la Sección 4.3, cuentan con las funciones estándar de recorrido, inserción, y eliminación de etiquetas en un camino, o nodos en una estructura. También existen otras funciones que implementan los conceptos estudiados en el Capítulo 2. En esta sección se describen dos funciones fundamentales en este sentido. La primera función que se describe determina el conjunto de marcado $\mathcal{M}$ de un *mini-tree* $T_{\Sigma,\varphi}$. La segunda determina la alcanzabilidad desde un nodo u a otro v en el *witness-graph* $G_{\Sigma,\varphi}$.

4.5.1 Determinar el conjunto de marcado

Recordar, que el conjunto de marcado del *mini-tree* $T_{\Sigma,\varphi}$ es un subconjunto $\mathcal{M}$ del conjunto de nodos de $T_{\Sigma,\varphi}$ que se determina de la siguiente forma: si para todo $i = 1, \ldots, k_\varphi$ se tiene $P_i^\varphi \neq \varepsilon$, entonces $\mathcal{M}$ consiste de los nodos hoja (nodos sin hijos) de $T_{\Sigma,\varphi}$, y en otro caso $\mathcal{M}$ consiste de todos los nodos descendientes de q'_φ en $T_{\Sigma,\varphi}$.

Siguiendo la definición del conjunto de marcado, se diseña la siguiente función, que primero verifica la condición $P_i^\varphi \neq \varepsilon$, para saber en qué caso entrar de los dos posibles. Los casos son dos, en el primero "CASO 1" se tiene que el conjunto de marcado $\mathcal{M}$ se compone de aquellos nodos hoja del *mini-tree*, y en el segundo "CASO 2" se tiene que el conjunto de marcado $\mathcal{M}$ se compone de todos los descendientes de q'_φ en $T_{\Sigma,\varphi}$. Para el CASO 1, es necesario simplemente

[3]GNU Compiler Collection, http://gcc.gnu.org/

recorrer la lista en busca de aquellos nodos que tengan una lista de adyacencia vacía (es decir, no tengan hijos y sean considerados nodos hoja). Mientras en el CASO 2, se utiliza el algoritmo DFS (ver Algoritmo 3.2), para obtener los nodos descendientes de q'_{φ}.

```
/**
 * Funcion que define el conjunto de Marking del mini-tree.
 * @return Conjunto de nodos que componen el marking del mini-tree.
 */
void graphTool::findMarking()
{
  vector<graphTool::vertexEle*> marking_t;
  bool epsilonBool = false;

  /* Verificacion de los caminos clave */
  int count = 0; // contador de caminos clave iguales a epsilon
  vector<pathExp> keys;
  keys = phi.getKeyPath(); // Se recuperan los caminos clave
  for (int i = 0; i < keys.size(); i++)
  {
    if (keys[i].getLength() == 0) //getLength retorna 0 si keys[i] es epsilon
    {
      epsilonBool = true; //existe un camino clave igual a epsilon
      break;
    }
  }
  vertexEle *temp_vertex;
  //(CASO 1)
  if (!epsilonBool)
  {
    temp_vertex = this->vertex_ptr;
    // Se recorre todo el grafo, y se agregan las hojas al conjunto marking
    do
    {
      // Si el nodo no tiene edges salientes --> es hoja
      if (temp_vertex->m_edges == NULL)
      {
        marking_t.push_back(temp_vertex);
        temp_vertex->marked = true; //temp_vertex pertenece al Marking.
      }
      temp_vertex = temp_vertex->m_next;
    } while (temp_vertex != NULL);
  }
  //(CASO 2)
  else
  {
    /* Se utiliza el mismo principio de DFS, se recorren los nodos adyacentes
     * y los hijos de ellos. */
    /* Asegurarse que el origen no es nulo. */
    if (this->qprimphi != NULL)
      /* Se obtienen los descendientes del nodo q'_\varphi. */
      marking_t = getDescendents(this->qprimphi);
  }
  // Seteo el estado de la variable marking del mini-tree
  this->marking = marking_t;
}
```

La función utilizada para obtener los nodos descendientes, se basa en la estrategia utilizada por el algoritmo DFS. Dado que el *mini-tree* es un árbol, si un nodo v es alcanzable a partir de un nodo u, entonces v pertenece al conjunto de descendientes de u.

```
/**
 * Funcion que retorna todos los descendientes de un nodo dado.
 * @param node Nodo del cual se desean saber sus descendientes.
 * @return Vector con los nodos descendientes del nodo.
 */
vector<graphTool::vertexEle*> graphTool::getDescendents(graphTool::vertexEle* node)
{
  int start_node = node->m_node->m_idEle;
  vector<graphTool::vertexEle*> marking_out;
  vertexEle* temp_vertex;
  for (temp_vertex = this->vertex_ptr; temp_vertex != NULL;
          temp_vertex = temp_vertex->m_next)
  {
    /* Se llama a la funcion reachable que recibe el identificador
     * del nodo de partida y del nodo destino. Luego, recorre la lista
     * de nodos y obtiene el puntero del nodo de partida, y luego llama
     * a la funcion isReachable con estos parametros, que es la que
     * implementa DFS sobre el arbol.
     */
    if (reachable(start_node, temp_vertex->m_node->m_idEle))
    {
      marking_out.push_back(temp_vertex);
      temp_vertex->marked = true; //El nodo pertenece al marking
    }
  }
  return marking_out;
}
```

4.5.2 Determinar la alcanzabilidad

Otra función fundamental en el algoritmo de implicación de claves XML, es la que determina la existencia de un camino entre un par de nodos u, v, es decir determinar si v es alcanzable desde u. Esta función es llamada por la función `reachable`, y recibe como parámetros el identificador del nodo destino v, y el puntero al nodo de partida u. Retorna 1 si v es alcanzable desde u; y 0 en otro caso. Como un nodo no puede ser visitado más de una vez, se utiliza el atributo booleano `visited` declarado en cada nodo (según estructura `nodeEle`), que evita la formación de ciclos en el recorrido DFS. Luego, es posible determinar de manera recursiva si el nodo v es alcanzable desde el nodo u.

```
/**
 * Funcion que dado un vertice, verifica si existe un camino para alcanzar otro.
 * @param target El identificador del nodo destino.
 * @param source_ptr Un puntero al nodo de partida del camino.
 * @return 1 si es alcanzable, 0 en caso contrario.
 */
int graphTool::isReachable(const int target, graphTool::vertexEle * source_ptr)
{
  graphTool::edgeEle *edgeP;

  /* Se ha visitado este nodo? */
  if (source_ptr->visited)
    return 0;

  /* Este nodo es el destino? Si es asi, se ha alcanzado el target. */
```

```
16   if (source_ptr->m_node->m_idEle == target)
17     return 1;
18
19   /* No visitarlo nuevamente. */
20   source_ptr->visited = true;
21
22   /* Ver si es posible llegar desde cada uno
23    * de los vertices con los cuales se tiene adyacencia.
24    * Si se puede llegar a partir de al menos uno de ellos,
25    * el nodo target es alcanzable.
26    */
27   for (edgeP = source_ptr->m_edges; edgeP != NULL; edgeP = edgeP->m_next)
28   {
29     if (isReachable(target, (vertexEle*&) edgeP->m_connectsTo))
30       return 1;
31   }
32
33   /* No se puede llegar al target desde alguno de los vecinos, asi que
34    * el target no es alcanzable desde aqui.
35    */
36   return 0;
37 }
```

CAPÍTULO 5. VALIDACIÓN DE DOCUMENTOS XML CONTRA CLAVES

Mientras que la satisfacción de claves como restricciones sobre alguna instancia de un esquema ha sido un problema muy estudiado en el modelo relacional (Abiteboul et al., 1995; Gupta et al., 1994). Validar la satisfacción de claves como restricciones sobre documentos XML es un área de investigación abierta (Liu et al., 2005). Debido a que las aplicaciones de intercambio de datos basadas en XML, deben transmitir tanto la correcta estructura, como la información semántica de los datos, la validación juega un rol crucial en los entornos de intercambio y consistencia de datos (Arenas & Libkin, 2008).

5.1 SATISFACCIÓN DE CLAVES

Se recuerda brevemente la Definición 3.2 sobre satisfacción de claves XML. Un árbol XML T se dice que satisface una clave $(Q, (Q', \{P_1, \ldots, P_k\}))$ si y sólo si, para todo nodo $q \in [\![Q]\!]$ y todo par de nodos $q'_1, q'_2 \in q[\![Q']\!]$, tal que, existen los nodos $x_i \in q'_1[\![P_i]\!]$, $y_i \in q'_2[\![P_i]\!]$ con $x_i =_v y_i$ para todo $i = 1, \ldots, k$, se tiene entonces que $q'_1 = q'_2$.

Como ejemplo, se presenta el árbol XML T de la Figura 5.1 que representa un documento que contiene información de proyectos gestionados por diversas instituciones. En cada institución existe un responsable o jefe de proyecto del cual se conoce su nombre y la oficina que ocupa. Además, se presenta un conjunto de claves con su descripción, y se analiza su satisfacción por parte de dicho árbol. Este ejemplo será trabajado a lo largo del presente capítulo, para ejemplificar la validación de documentos XML contra claves.

Sobre árboles con la forma del árbol T de la Figura 5.1 se pueden definir entre otras las siguientes claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$:

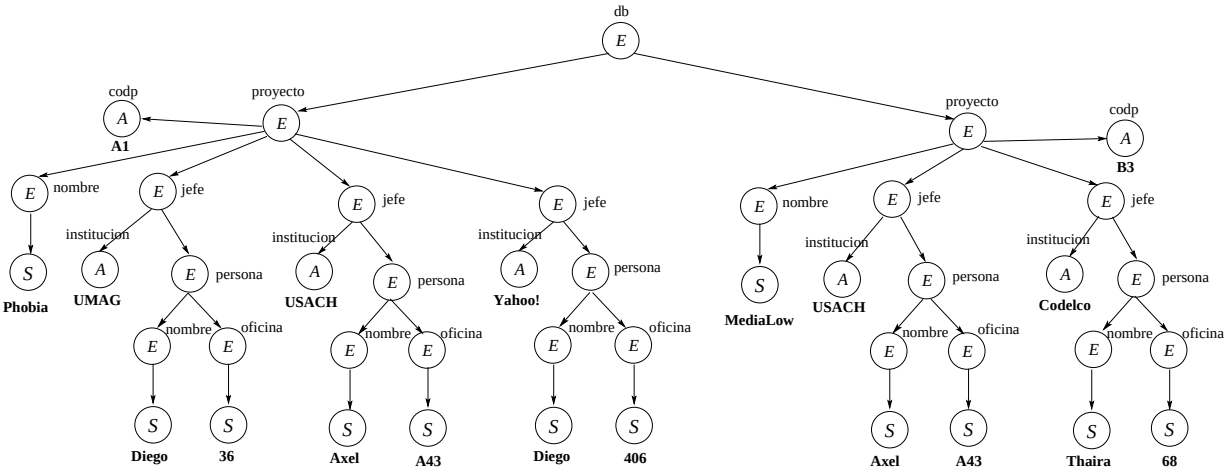


FIGURA 5.1: *Árbol XML de un documento XML con proyectos.*

(a) $(\varepsilon, (proyecto, \{codp\}))$, un árbol XML con la forma del árbol T en la Figura 5.1 satisface esta clave si y sólo si, no existen en el árbol dos nodos *proyecto* con el mismo valor en el atributo código (*codp*), en todo el árbol XML.

(b) $(\varepsilon, (proyecto, \{nombre.S, codp\}))$, no pueden existir dos nodos *proyecto* con el mismo valor en texto para *codp* y *nombre* en todo el árbol XML.

(c) $(proyecto, (jefe, \{institucion, persona.nombre.S, persona.oficina.S\}))$, en el contexto de un proyecto, un jefe se identifica por los valores de sus nodos *nombre*, *oficina*, e *institucion* que representa.

(d) $(proyecto, (jefe, \{persona\}))$, indica que un jefe de proyecto es identificado por el valor de su nodo hijo *persona*, por lo tanto, no pueden existir dos nodos *jefe* cuyos nodos *persona* descendientes coincidan en valor.

(e) $(\varepsilon, (\_^*.persona, \{nombre.S, oficina.S\}))$, un nodo *persona* se identifica en todo el árbol XML por los valores de sus nodos descendientes *nombre* y *oficina*.

(f) $(\varepsilon, (proyecto.\_^*.persona, \{nombre.S, oficina.S\}))$, sobre árboles con la forma del árbol T en la Figura 5.1 esta clave es equivalente a la clave anterior.

(g) $(proyecto, (jefe, \{persona.nombre.S\}))$, un jefe de proyecto, se puede identificar de manera unívoca, por su *nombre*, en el contexto de un *proyecto*.

Las claves (a), (b), (c), y (d) son claves XML satisfechas por el árbol XML T de la Figura 5.1. Mientras, las claves XML (e) y (f) no son satisfechas, ya que "Axel", utiliza la oficina "A43" en más de un proyecto. Tampoco es satisfecha la clave (g) ya que dentro del subárbol del primer proyecto "Phobia", el primer y tercer nodo *jefe* tienen como descendiente un nodo *nombre* con el texto "Diego" asociado. Sólo las claves (a), (b), (e) y (f) pueden ser expresadas como tal en XML *Schema* (Sperberg-McQueen & Thompson, 2000); y ya que no existe una noción directa de claves relativas en XML *Schema*, las claves (c), (d) y (g) no pueden ser expresadas. Nótese que a pesar que se reemplace la expresión de camino *proyecto* por ε en (d), la clave absoluta resultante no puede ser expresada utilizando XML *Schema*, ya que posee un camino clave que retorna un nodo elemento el cual es la raíz de un subárbol.

5.2 TRABAJOS RELACIONADOS

Los trabajos que han desarrollado el problema de validación de documentos XML contra un conjunto de claves XML, utilizan la sintaxis para definir claves XML presentada en Buneman et al. (2003, 2002), desde la cual XML *Schema* ha adoptado algunos aspectos. Según lo visto hasta ahora, las claves permitidas por XML *Schema* son: aquellas llamadas KEY, cuyos caminos claves se asocian a un valor en texto (PCDATA o atributos), y aquellas llamadas UNIQUE, en donde el conjunto de caminos claves puede ser vacío (restricción estructural). Por lo tanto, es fácil darse cuenta que la clase de claves permitidas por XML *Schema* está contenida en la clase general de claves $\mathcal{K}^{PL}_{PL,PL^*_s}$ presentada en Buneman et al. (2003, 2002). Mientras, la clase de claves XML $\mathcal{K}^{PL}_{PL,PL^+_s}$ aquí trabajada incluye sólo aquellas claves llamadas KEY, y no las llamadas UNIQUE cuyo conjunto de caminos claves es vacío, permitiendo además claves cuyos caminos claves retornen un nodo elemento el cual es la raíz de un subárbol.

La primera aproximación es este problema se encuentra en Chen et al. (2002), donde se presenta un validador de documentos XML contra claves basado en un analizador de acceso secuencial para XML utilizando *Simple API for XML* (SAX) (Megginson, 2004), basándose en una técnica de indexación de claves, y almacenando el documento XML en una base de datos relacional. Con lo cual son capaces de realizar la validación de un documento completo, y la validación incremental de

actualizaciones hechas al documento, con un rendimiento lineal en el tamaño del documento o de la actualización respectivamente. Es importante mencionar que este validador requiere de un esquema (DTD o XML *Schema*) asociado al documento que se desea validar. En Bouchou et al. (2003) utilizan un autómata árbol para validar un documento XML contra una clave, y además contra un esquema, en tiempo lineal en el tamaño del documento. Dicho trabajo, al igual que el anterior, considera que los caminos claves están asociados a un valor texto, y que el documento de entrada está asociado a un esquema DTD. Luego, en Abrão et al. (2004) se expande el enfoque anterior usando un autómata árbol para permitir la validación de un documento XML contra un conjunto de claves, incluyendo claves foráneas, y la validación incremental ante actualizaciones en el documento XML. Por otro lado, Liu et al. (2005, 2004) proponen otro enfoque realizando la validación de un documento XML contra una clave, mediante una caracterización al problema de verificación estructural (tal como la revisión de un documento XML contra un DTD). Esta caracterización les permite desarrollar un algoritmo polinomial, que permite una validación completa y también incremental de documentos XML contra claves. Generan un documento de valores de claves, que posee los valores de los elementos afectados por las claves, que son extraídos desde el documento XML mediante consultas XPath, y realizan una verificación estructural de este documento generado contra un DTD específico y predefinido creado por ellos. De esta forma, mediante la verificación estructural se verifica si existen ciertos elementos repetidos dentro del documento de valores de claves. Luego, si no existen elementos repetidos, se dice que la verificación es exitosa, y en consecuencia que el documento XML original es válido contra el conjunto de claves.

En todos los trabajos expuestos, junto con utilizar la sintaxis para claves dada en Buneman et al. (2003, 2002), se asegura utilizar el concepto de igualdad en valor propuesto en Buneman et al. (2003, 2002), el cual no está restringido a la igualdad en valor en texto, sino que se basa en isomorfismos. Pero, hasta donde se sabe los algoritmos presentados se restringen sólo a la igualdad de nodos texto (sólo permiten caminos clave asociados a un valor) (Davidson et al., 2008, 2007). Los enfoques tomados por los trabajos anteriores, difieren del realizado aquí ya que en primer lugar, ninguno considera claves sobre las cuales se pueda realizar un razonamiento eficiente. Segundo, algunos exigen que exista un esquema asociado al documento XML. Tercero, algunos utilizan estructuras indexadas, y almacenan los documentos XML sobre bases de datos relacionales, alejándose de los fundamentos del modelo XML.

En base a lo expuesto, en la segunda sección de este capítulo se presenta un algoritmo de validación de documentos XML contra claves pertenecientes a la clase $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$ con una axiomatización conocida, representando un documento XML como un árbol generado a partir de la propuesta de DOM (independiente de otros modelos como el relacional), realizando una traducción de las expresiones de camino que componen las claves a consultas XPath a realizarse sobre el árbol DOM, y utilizando el concepto de igualdad en valor no restringido a texto. Apoyándose en la axiomatización conocida, y considerando el hecho que el tiempo de validación de un documento XML depende, entre otras cosas, de la cantidad de claves a evaluar, en la tercera sección se introduce y analiza el concepto y la utilidad práctica de calcular *covers* no redundantes para conjuntos de claves XML en la clase $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$.

5.3 VALIDACIÓN DE DOCUMENTOS XML

5.3.1 Estrategia de validación

Para lograr la validación de un documento XML D contra un conjunto Σ de claves XML, se analiza cada clave $\psi \in \Sigma$, transformando las expresiones de camino –presentes en el camino de contexto, camino objetivo, y caminos clave– a expresiones XPath (Clark & DeRose, 1999), reemplazando "." por "/" y "_*" por ".//.". Esto toma tiempo lineal en el tamaño de las expresiones de camino. (La evaluación de consultas XPath ha sido ampliamente estudiada (Benedikt & Koch, 2009), su problema para consultas XPath generales pertenece a la clase PTIME.) Por cada nodo q alcanzable siguiendo la expresión XPath correspondiente al camino de contexto Q_{ψ}, se evalúa la expresión XPath correspondiente al camino objetivo Q'_{ψ} (obteniendo un conjunto de nodos), y por cada nodo q' en dicho conjunto, se evalúa cada una de las expresiones XPath para los caminos clave P^{ψ}_{i} obteniendo un conjunto de nodos $V^{q'}_{i}$, aplicando un algoritmo que verifique si la intersección en valor es o no vacía. Si para algún par de nodos objetivo q'_1 y q'_2, se tiene que $V^{q'}_{1} \cap_v V^{q'}_{2} \neq \emptyset$ para todo $i = 1, \ldots, k_{\psi}$, entonces la clave no es satisfecha por D, en otro caso la clave es satisfecha por D.

Para la evaluación de las consultas XPath, se requiere analizar el documento XML D y construir un árbol XML T_D. Todo documento XML puede ser representado como un árbol ordenado

y etiquetado, ya sea que éste sea válido o inválido de acuerdo a un esquema XML como DTD (Nierman & Jagadish, 2002). Claramente, el árbol XML T_D correspondiente a un documento D puede obtenerse en tiempo lineal en el tamaño de D. Por otro lado, esta transformación es necesaria para evaluar eficientemente las expresiones de camino que conforman las claves (Buneman et al., 2003). Por lo tanto, para abordar el problema de validación de un conjunto Σ de claves XML contra un documento D, se le transforma al problema de determinar si el correspondiente árbol T_D satisface todas las claves en Σ ($T \models \Sigma$).

Se debe recordar que la definición de una clave se sustenta sobre el concepto de igualdad en valor (Sección 2.3), muy diferente de la igualdad de valores atómicos en el modelo relacional. Nótese, que esta igualdad en valor toma en cuenta el orden del documento XML. De manera resumida, dos nodos u y v son iguales en valor ($u =_v v$) si y sólo si, los subárboles con raíz u y v son isomorfos por un isomorfismo que para cadenas de texto se corresponde con la función de identidad. Si bien el problema de grafos isomorfismos, es un problema clasificado como NP dentro del estudio de complejidad computacional, su generalización, el problema de subgrafos isomorfos es clasificado como NP-Completo (Aho & Hopcroft, 1974; Garey & Johnson, 1979). Dado que se trabaja con árboles ordenados, el problema se vuelve tratable (Campbell & Radford, 1991). Esto implica que para resolver si $u =_v v$, se puede utilizar el Algoritmo 5.1.

En el Algoritmo 5.1, se determinará si dos nodos son iguales en valor, si se cumplen las siguientes condiciones:

(i) las etiquetas de u y v son iguales,
(ii) si u y v son nodos texto o atributo, que sus valores sean iguales, y
(iii) si u y v son nodos elemento, que los hijos de u y v sean también iguales en valor.

La mayor complicación para resolver la igualdad en valor de nodos en un árbol XML T_D, se presenta cuando u y v son nodos elementos, con $att(u) \neq \emptyset$ y $att(v) \neq \emptyset$, y además $|att(u)| = |att(v)|$. Dado que $att(u)$ y $att(v)$ son conjuntos no ordenados de atributos, se les debe primero ordenar según la etiqueta de cada atributo, para así poder realizar una comparación de valor en texto. Mientras los nodos elemento descendientes de u y v se comparan según su posición en las listas $ele(u)$ y $ele(v)$. El Algoritmo 5.1 para igualdad en valor propone, al momento de comparar los nodos atributo, definir un orden alfabético en las etiquetas de los atributos, de manera tal de poder realizar la comparación según su posición en las listas resultantes, del mismo modo que con los nodos elemento.

Algoritmo 5.1: *IgualdadValor*(u, v)*: Igualdad en Valor de dos nodos* $(=_v)$

Entrada: Dos nodos u y v en T.
Salida: Verdadero si $u =_v v$, falso en otro caso.

```
1: if (lab(u) == lab(v)) then
2:   if (u, v ∈ A or u, v ∈ {S}) then
3:     if (val(u) == val(v)) then
4:       return verdadero;
5:     else
6:       return falso;
7:     end if
8:   else
9:     //Comparar nodos atributo hijos
10:    if (att(u) ≠ ∅ and att(v) ≠ ∅ and |att(u)| == |att(v)|) then
11:      attrListU = lista de nodos en attr(u) por orden alfabético de etiquetas;
12:      attrListV = lista de nodos en attr(v) por orden alfabético de etiquetas;
13:      for (i = 1 hasta attrListU.length) do
14:        if (!IgualdadValor(attrListU.item(i), attrListV.item(i))) then
15:          return falso;
16:        end if
17:      end for
18:    else
19:      return falso;
20:    end if
21:    //Comparar nodos elemento hijos
22:    eleListU = ele(u);
23:    eleListV = ele(v);
24:    if (eleListU.length == eleListV.length) then
25:      for (i = 1 hasta eleListU.length) do
26:        if (!IgualdadValor(eleListU.item(i), eleListV.item(i))) then
27:          return falso;
28:        end if
29:      end for
30:    else
31:      return falso;
32:    end if
33:    //Todos los pares de atributos y elementos son iguales en valor
34:    return verdadero;
35:  end if
36: else
37:   return falso;
38: end if
```

El Algoritmo 5.1 se diseñó como una función recursiva –realizando un barrido en preorden del árbol XML T_D–, retornando verdadero en el caso que la igualdad se cumpla, y falso en otro caso. El algoritmo utiliza como estrategia la búsqueda de la primera desigualdad en valor, caso en el cual se finaliza y retorna el valor falso. Por lo tanto, el peor caso en la ejecución del Algoritmo 5.1 se dará si los nodos analizados son iguales en valor. En el ejemplo de la Figura 5.1 existe una igualdad en valor en los nodos *jefe* descendientes de proyecto, donde la persona tiene nombre "Axel", utiliza la oficina "A43", y pertenece a la institución "USACH", con lo cual las claves definidas (e) y (f) no son satisfechas.

A continuación, se discute la complejidad temporal del Algoritmo 5.1. El caso más simple se da si los nodos u y v pertenecen a $\mathbf{A}$ o $\{S\}$ (se asume igualdad de etiquetas, paso 1), donde el algoritmo se ejecuta en tiempo constante (pasos 2-7). Por otro lado, si los nodos pertenecen a $\mathbf{E}$, se requiere analizar cada nodo presente en los subárboles con raíz u y v (pasos 8-35). Se analizan todos los descendientes que están en $\mathbf{A}$ y $\mathbf{E}$. (Como se definió en el Capítulo 2, los nodos atributo no poseen un orden específico dentro del árbol XML, en cambio sí lo poseen los nodos elemento.) El algoritmo propone extraer los nodos atributo de u y v, y realizar un orden alfabético según etiquetas (pasos 11 y 12). Luego, tanto atributos como elementos pueden ser comparados según su correspondiente orden. Para la ordenación de la lista de atributos en los pasos (11) y (12), se usa el algoritmo *Quicksort*, que en promedio es orden $\mathcal{O}(N_v \times log\ N_v)$, con N_v igual al largo de la lista $att(v)$ para determinar la igualdad en valor de u y v. En caso que todos los nodos descendientes de u y v pertenezcan a $\mathbf{A}$, el Algoritmo 5.1 tomará sólo $\mathcal{O}(N \times log\ N)$. En caso que todos los nodos descendientes de u y v pertenezcan a $\mathbf{E}$, se tiene que el Algoritmo 5.1 se ejecuta en tiempo $\mathcal{O}(M)$, donde M es la cantidad máxima de nodos descendientes de u y v. En el peor caso se tienen dos subárboles que son isomorfos y además cada elemento en esos subárboles tiene N atributos. Luego, se dice que el Algoritmo 5.1 puede ser ejecutado en el peor caso en tiempo $\mathcal{O}(M \times N \times log\ N)$.

La satisfacción de una clave ψ por un árbol XML T_D (ver Ecuación 3.3), se basa en que las intersecciones en valor de los nodos alcanzables desde un nodo objetivo siguiendo los caminos claves, sean vacías; en otro caso, T_D no satisface la clave ψ. Recordar que la intersección en valor, definida en la Ecuación 2.2, está basada en la igualdad en valor.

Se presenta el Algoritmo 5.2 que captura el concepto de intersección en valor. Dado que el algoritmo de validación diseñado no requiere conocer los elementos que componen la intersección,

Algoritmo 5.2: *InterseccionValor*($Lista_1, Lista_2$)*: Intersección en Valor* ($\cap_v$)

Entrada: Dos listados de nodos en T.
Salida: Verdadero si la intersección en valor no es vacía, falso en otro caso.

```
1: //Recorrer ambas listas comparando sus nodos en valor
2: for (i = 1 hasta Lista1.length) do
3:    for (j = 1 hasta Lista2.length) do
4:       if (IgualdadValor(Lista1.item(i), Lista2.item(j))) then
5:          //Se cumple la condición necesaria y suficiente, para determinar que la intersección no
            es vacía
6:          return verdadero;
7:       end if
8:    end for
9: end for
10: return falso;
```

el Algoritmo 5.2 sólo retorna el valor verdadero si la intersección en valor entre dos listados no es vacía, y falso en otro caso. Por lo tanto, la estrategia seguida es buscar la primera igualdad de nodos, para cumplir la condición necesaria y suficiente, para determinar que la intersección no es vacía.

Nótese, que la complejidad temporal del Algoritmo 5.2 depende de la complejidad del Algoritmo 5.1. Si se considera que P es el largo de la $Lista_1$, y R el de la $Lista_2$ se ejecuta el paso 4 del Algoritmo 5.2 $P \times R$ veces, y se obtiene tiempo $\mathcal{O}(P \times R \times M \times N \times log\ N)$.

5.3.2 Algoritmo de validación

La validación de un documento XML T_D contra una clave XML ψ, se reduce a determinar si T_D satisface ψ. Se presenta el Algoritmo 5.3 que sigue la estrategia trazada, traduciendo y evaluando las expresiones de camino como expresiones XPath. De esta manera se obtiene un listado de nodos alcanzables siguiendo el camino de contexto (paso 1). A partir de un nodo de contexto q (paso 3), se evalúa la expresión XPath correspondiente al camino objetivo, obteniendo un listado de nodos objetivo (paso 4). Si la lista de nodos objetivo posee sólo un elemento, se asumirá que T_D satisface ψ, y el algoritmo retorna el valor verdadero (paso 31). Si la lista de nodos contexto o objetivo es vacía, se dice que T_D *satisface trivialmente* a ψ.

Se evalúa en los pasos 5 al 29, la intersección en valor de las listas de nodos alcanzables desde los nodos objetivo, siguiendo los caminos clave. Se utiliza la variable *objetivo* como acumulador,

para determinar la cantidad de veces que la intersección no es vacía, si su valor es igual a la cantidad de caminos clave de ψ, se tiene que ψ no es satisfecha por T_D, y el algoritmo retorna el valor falso, en otro caso se retorna verdadero.

Notar que para la validación de un conjunto Σ de claves XML, se deben realizar tantas ejecuciones del Algoritmo 5.3 como claves se tenga en Σ.

Si se considera que una consulta $v[\![Q]\!]$ de tipo Core XPath puede ser ejecutada sobre el árbol XML T_D en tiempo $\mathcal{O}(|T_D| \times |Q|)$, se puede obtener la complejidad temporal del algoritmo de validación de la siguiente manera. La ejecución del algoritmo de validación de un documentos XML D contra un conjunto Σ de claves XML, depende primero de la cantidad de claves $|\Sigma|$. Luego, depende de la cantidad de nodos contexto, nodos objetivo y, nodos clave involucrados en cada clave $\sigma \in \Sigma$, obtenidos siguiendo los caminos Q_σ, Q'_σ y todos los P_i^σ respectivamente. Sea V el número máximo de nodos contexto, pertenecientes a $[\![Q_\sigma]\!]$; W el número máximo de nodos objetivo descendientes de los nodos contexto, pertenecientes a $q[\![Q'_\sigma]\!]$ con $q \in [\![Q_\sigma]\!]$; y C el número de caminos clave de una clave σ. El Algoritmo 5.3 para validación de un documento D contra una claves XML en $\mathcal{K}^{PL}_{PL,PL_s^+}$ se ejecuta en tiempo $\mathcal{O}((|T_D| \times |Q|) \times V \times W \times C \times (P \times R \times M \times N \times log\ N))$. De acuerdo a la definición dada para cada uno de los parámetros, y considerando la complejidad del algoritmo para determinar la intersección en valor, se obtiene que el algoritmo de validación es un algoritmo en tiempo polinomial.

5.4 *COVERS* PARA CLAVES XML EN $\mathcal{K}^{PL}_{PL,PL_S^+}$

El tiempo de validación de un documento XML depende, entre otras cosas, del número de claves en Σ. Por lo tanto, es conveniente analizar una forma de disminuir el tamaño de Σ. En este sentido, el algoritmo de implicación de claves XML presentado en el Capítulo 3, puede usarse para descartar claves en Σ que son implicadas por otras claves también en Σ. De esta forma se puede reducir Σ hasta que ninguna de las claves en el conjunto reducido sea implicada por las restantes. Siguiendo la denominación usada en bases de datos relacionales, se llama *cover no redundante* de Σ al subconjunto de claves de Σ obtenido mediante este proceso.

El principal análisis de *covers* mínimos de dependencias funcionales en el modelo

Algoritmo 5.3: *ValidaClaves*(T_D, ψ)*: Validación de docs XML contra claves XML*

Entrada: Árbol XML T_D, y una clave XML ψ en $\mathcal{K}^{PL}_{PL,PL^+_s}$.
Salida: verdadero si T_D satisface ψ; falso en otro caso.

```
1: listaNodosContexto = nodos en [[Q_ψ]];
2: for (j = 1 hasta listaNodosContexto.length) do
3:    q = listaNodosContexto[j];
4:    listaNodosObjetivo = nodos en q[[Q'_ψ]];
5:    if (listaNodosObjetivo.length > 1) then
6:       //Existe más de un nodo objetivo. Se analizan todos los nodos objetivo para determinar si
         el árbol satisface la clave.
7:       for (l = 1 hasta listaNodosObjetivo.length-1) do
8:          for (k = l + 1 hasta listaNodosObjetivo.length) do
9:             objetivo = 0;
10:            for (n = 1 hasta ψ.caminosClave.length) do
11:               q'_1 = listaNodosObjetivo[l];
12:               q'_2 = listaNodosObjetivo[k];
13:               listaClavesA = nodos en q'_1[[P^ψ_n]];
14:               listaClavesB = nodos en q'_2[[P^ψ_n]];
15:               if (InterseccionValor(listaClavesA, listaClavesB)) then
16:                  objetivo++;
17:               else
18:                  break; //Salir del ciclo for
19:               end if
20:            end for
21:            if (objetivo == ψ.caminosClave.length) then
22:               //Clave no satisfecha.
23:               return falso;
24:            end if
25:         end for
26:      end for
27:      //Clave satisfecha.
28:      return verdadero;
29:   else
30:      //Existe un solo nodo objetivo o ninguno. Clave satisfecha.
31:      return verdadero;
32:   end if
33: end for
34: return verdadero; //Clave satisfecha trivialmente.
```

relacional se encuentra en Maier (1980). Aparentemente, este tema no ha sido abordado todavía desde la perspectiva de XML. Por lo tanto, siguiendo el trabajo de Maier, se definen las nociones de *cover* y equivalencia. El objetivo es explorar métodos para representaciones sucintas de conjuntos de claves XML.

Ejemplo 5.1 Sea Σ un conjunto de claves XML compuesto de $\sigma_1 = (\varepsilon, (proyecto, \{codp\}))$ y $\sigma_2 = (\varepsilon, (proyecto, \{nombre.S, codp\}))$. Se puede ver fácilmente que σ_2 puede derivarse de σ_1 aplicando la regla *superclave*. El conjunto *cover* de Σ es $\Sigma' = \{\sigma_1\}$. Se puede ver también, que $|\sigma_1| < |\sigma_2|$, lo cual indica que σ_1 tiene un menor número de etiquetas que σ_2. (Recordar que la complejidad del algoritmo de implicación de claves XML se basa en el tamaño de las claves.) □

5.4.1 Nociones de *covers* y equivalencia

En el Capítulo 3, se revisaron los conceptos de clave XML y la noción de implicación de claves. También, se definió la clausura semántica, y sintáctica para un Σ dado. Recordar que, la clausura semántica de Σ, denotada Σ^*, es el conjunto de todas las claves XML implicadas por las claves en Σ. Tanto la clausura semántica, como la sintáctica, son conjuntos infinitos.

Se define la noción de *cover* para un conjunto de claves XML, siguiendo la definición usada para bases de datos relaciones en Maier (1980).

Definición 5.1 Dados los conjuntos Σ_1 y Σ_2 de claves XML se dice que Σ_1 y Σ_2 son equivalentes, denotado por $\Sigma_1 \equiv \Sigma_2$, si $\Sigma_1^* = \Sigma_2^*$. Si $\Sigma_1 \equiv \Sigma_2$, entonces Σ_2 es un *cover* para Σ_1. Esto es, Σ_1 y Σ_2 implican el mismo conjunto de claves XML.

Notar que para todo *cover* Σ_2 de Σ_1, si un árbol XML T_D satisface Σ_2 ($T_D \models \Sigma_2$), entonces también $T_D \models \Sigma_1$. Si $\Sigma_1 \equiv \Sigma_2$, entonces para toda clave XML φ en Σ_1^*, $\Sigma_2 \models \varphi$, ya que $\Sigma_2^* = \Sigma_1^*$. En particular, $\Sigma_2 \models \varphi$ para toda clave φ en Σ_1. Nótese, que el decir que Σ_2 es un *cover* de Σ_1 no dice nada respecto de los tamaños relativos de ninguno de los conjuntos.

Definición 5.2 Un conjunto Σ_2 de claves XML es *no redundante* si no hay un conjunto Σ_3 de claves propiamente contenido en Σ_2 con $\Sigma_3^* = \Sigma_2^*$. Si existiera Σ_3, Σ_2 es redundante. Σ_2 es un *cover no redundante* para Σ_1 si Σ_2 es un *cover* para Σ_1 y Σ_2 es no redundante.

Algoritmo 5.4: *CoverNoRedundante*(Σ)*: Cover no redundante para* Σ

Entrada: Conjunto finito Σ de claves XML en $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$.
Salida: Un cover no redundante para Σ.

```
1: Θ = Σ;
2: for cada clave ψ en Σ do
3:    if Θ − {ψ} ⊨ ψ then
4:       Θ = Θ − ψ;
5:    end if
6: end for
7: return Θ
```

Una caracterización de no redundancia, es que Σ es no redundante si no existe una clave ψ en Σ, tal que $\Sigma - \{\psi\} \models \psi$ (Maier, 1983). Se llama redundante a una clave ψ en Σ si $\Sigma - \{\psi\} \models \psi$.

5.4.2 Algoritmo *covers* no redundantes

Para cualquier conjunto Σ_1 de claves XML, existe un conjunto $\Theta \subseteq \Sigma_1$ tal que, Θ es un *cover* no redundante para Σ_1. Si Σ_1 es no redundante, entonces $\Theta = \Sigma_1$. Si Σ_1 es redundante, entonces hay al menos una clave XML $\psi \in \Sigma_1$ que es redundante en Σ_1, por lo tanto, podemos eliminar ψ de Σ_1 obteniendo $\Sigma_2 = \Sigma_1 - \{\psi\}$. Si a su vez Σ_2 es redundante, entonces existe una clave ϕ que es redundante en Σ_2, podemos eliminar ϕ de Σ_2 obteniendo $\Sigma_3 = \Sigma_2 - \{\phi\}$ (Nótese que se mantiene que $\Sigma^{*}_{3} = \Sigma^{*}_{2} = \Sigma^{*}_{1}$). Siguiendo este proceso, se pueden remover todas las claves redundantes existentes en Σ_1. Este proceso eventualmente alcanza una condición de parada, cuando no es posible descartar más claves. El resultado es un subconjunto *cover* no redundante Θ de Σ_1. El Algoritmo 5.4 formaliza el proceso expuesto, calculando el *cover* no redundante para un conjunto de claves XML.

Ejemplo 5.2 Sea Σ el conjunto que contiene las siete claves definidas al comienzo del capítulo. Aplicando el Algoritmo 5.4 a Σ, se obtiene Θ que contiene a las claves (a), (d), (e) y (g), que es un *cover* no redundante de Σ. Siempre se tendrá que $|\Theta| \leq |\Sigma|$, lo que indica que Θ posee un menor o igual número de claves que Σ. □

Ejemplo 5.3 Sea el conjunto Σ de claves XML de la Tabla 5.1, sobre el árbol XML T de la Figura 5.1. Un *cover* no redundante para Σ es el conjunto Θ formado por las claves $11, 12, 10, 14, 16, 4, 1$. □

TABLA 5.1: *Conjunto de claves XML para la Figura 5.1.*

No.	Clave XML
1	$(proyecto, (jefe, \{persona\}))$
2	$(proyecto, (jefe, \{persona.nombre.S\}))$
3	$(proyecto, (jefe.persona, \{nombre.S\}))$
4	$(proyecto, (jefe, \{institucion\}))$
5	$(proyecto, (jefe.persona, \{nombre.S, oficina.S\}))$
6	$(proyecto.jefe, (persona, \{nombre.S, oficina.S\}))$
7	$(proyecto, (\_^*.persona, \{nombre.S, oficina.S\}))$
8	$(proyecto, (jefe, \{institucion, persona.nombre.S, persona.oficina.S\}))$
9	$(\varepsilon, (proyecto.jefe, \{persona.nombre.S, persona.oficina.S\}))$
10	$(\varepsilon, (proyecto.jefe.persona.\_^*, \{S\}))$
11	$(\varepsilon, (\_^*.persona, \{nombre.S, oficina.S\}))$
12	$(\varepsilon, (proyecto.\_^*, \{S, institucion\}))$
13	$(\varepsilon, (proyecto.jefe.persona, \{nombre.S, oficina.S\}))$
14	$(\varepsilon, (proyecto, \{codp\}))$
15	$(\varepsilon, (proyecto, \{nombre.S, codp\}))$
16	$(\varepsilon, (proyecto, \{nombre.S\}))$
17	$(\varepsilon, (proyecto.\_^*.persona, \{nombre.S, oficina.S\}))$

Al obtenerse un *cover* con menor número de claves que el conjunto original, es válido y recomendable realizar la validación de documentos XML contra los *covers*. Para ver pruebas empíricas, revisar el Capítulo 7.

El análisis de complejidad temporal del Algoritmo 5.4 depende de la complejidad del algoritmo de implicación, visto en el Capítulo 3. Se ejecuta el algoritmo de implicación, tantas veces como claves ψ en Σ se tengan. En cada ejecución se escoge una de las claves $\psi \in \Sigma$, la que es considerada como la clave a saber si es implicada por el resto $\Sigma - \{\psi\}$. Se llama $|\Sigma|$ veces al algoritmo de implicación, y se obtiene rápidamente que la complejidad del algoritmo para calcular *covers* no redundantes. Así, obtenemos el siguiente resultado.

Teorema 5.1 Un *cover* no redundante para Σ puede ser calculado en tiempo $\mathcal{O}(|\Sigma| \times (max\{|\psi| : \psi \in \Sigma\})^2)$.

Un conjunto de claves XML Σ' es *mínimo* si no hay un conjunto Σ con menos claves que Σ', tal que $\Sigma' \equiv \Sigma$ (Maier, 1980). Esta definición es válida para el modelo relacional, pero en el caso de XML, teniendo en cuenta que las clausuras son infinitas, no es tan sencillo encontrar un algoritmo en tiempo polinomial que permita decidir un *cover* mínimo, ni siquiera es tan claro que el problema sea decidible para XML, y se deja como un problema abierto.

Se analiza a continuación, la posible existencia de más de un *cover* no redundante.

Definición 5.3 Dos claves φ y ψ en Σ son *equivalentes*, denotado $\varphi \equiv \psi$, si $\{\varphi\} \models \phi$ y $\{\psi\} \models \phi$.

De la Definición 5.3 surge claramente que:

(i). Dos claves distintas pueden implicar a una misma clave. $\exists \sigma_1, \sigma_2, \varphi$ con $\sigma_1 \neq \sigma_2$, tal que $\{\sigma_1\} \models \varphi$ y $\{\sigma_2\} \models \varphi$.

(ii). Una misma clave puede implicar a más de una clave. $\exists \sigma_1, \sigma_2, \psi$, tal que $\{\psi\} \models \sigma_1$, y $\{\psi\} \models \sigma_2$.

Respecto al punto (1), nótese que por ejemplo para $\sigma_1 = (a, (b.c.d.e, \{f\})$ y $\sigma_2 = (a, (b.c.d, \{e.f\}))$, aplicando la regla *subnodos* a cada una de estas claves, se puede inferir la misma clave $\psi = (a, (b.c, \{d.e.f\}))$.

Esta declaración indica que si dos claves σ_1 y σ_2 pueden implicar a una misma clave ψ, entonces $\sigma_1 \equiv \sigma_2$.

Ejemplo 5.4 Tomando las claves (14), (15) y (16) definidas en el Ejemplo 5.3. Se tiene que la clave (15) es implicada tanto por la clave (14), como por la (16), en ambos casos aplicando la regla de inferencia *superclave*. □

Respecto al punto (2), notar que por ejemplo para $\psi = (a.b, (c.d.e, \{f, g\}))$, si se aplica a ψ la regla de inferencia *superclave*, se obtiene $\sigma_1 = (a.b, (c.d.e, \{f, g, h\}))$. Si se le aplica a ψ la regla *contexto-objetivo*, se obtiene $\sigma_2 = (a.b.c, (d.e, \{f, g\}))$. Luego, se tiene que $\{\psi\} \models \sigma_1$ y $\{\psi\} \models \sigma_2$.

Claramente un conjunto Σ de claves XML puede contener más que un *cover* no redundante, y considerando el punto (1) también pueden existir *covers* no redundantes para Σ que no estén contenidos en el mismo Σ.

Ejemplo 5.5 Sea el conjunto Σ_1 igual al *cover* no redundante para un conjunto Σ obtenido en el Ejemplo 5.3. Considerando σ_1 como la clave (11) y σ_2 como la clave (12), se tiene que $\_^*.jefe.persona \subseteq Q'_{\sigma_1}$ y $proyecto.jefe.\_^* \subseteq Q'_{\sigma_2}$. Reemplazando σ_1 y σ_2 en Σ por $\sigma'_1 = (\varepsilon, (\_^*.jefe.persona, \{nombre.S, oficina.S\}))$ y $\sigma'_2 = (\varepsilon, (proyecto.jefe.\_^*, \{S, institucion\}))$ respectivamente, se obtiene Σ_2. Notar que se mantiene la equivalencia $\Sigma_2 \equiv \Sigma_1 \equiv \Sigma$ ($\Sigma_2^* = \Sigma_1^* = \Sigma^*$). Notas además que $\Sigma_2 \subsetneq \Sigma$, es decir, el *cover* no redundante Σ_2 posee claves que no pertenecen a Σ. □

CAPÍTULO 6. IMPLEMENTACIÓN VALIDADOR DE DOCUMENTOS XML

La validación de un documento XML sobre una clave (o conjunto de claves) es un área de investigación abierta. En el Capítulo 5 se propuso un algoritmo para realizar esta validación, el cual permite decidir la satisfacción de una clave (o conjunto de claves) en tiempo polinomial. También, se presentó un algoritmo para calcular *covers* no redundantes para claves en $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$, con complejidad temporal $\mathcal{O}(|\Sigma| \times (max\{|\psi| : \psi \in \Sigma\})^2)$. En este capítulo se presentan los detalles de las implementaciones de los algoritmos de validación y *covers* propuestos, analizando sus alcances y limitaciones.

6.1 DETALLES DE LA IMPLEMENTACIÓN

Las claves XML en $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$ son definidas sobre un modelo de árbol XML como el propuesto por DOM (Apparao et al., 1998) y XPath (Clark & DeRose, 1999). Para analizar la satisfacción de una clave, es necesario contar con este modelo de árbol, el cual debe ser obtenido a partir de la lectura del documento XML. Al proceso de lectura del documento XML se le denomina análisis o *parsing*, donde el documento es escaneado y divido en sus partes lógicas (elementos, atributos, comentarios, etc). Los dos métodos más populares para realizar esto son DOM (Apparao et al., 1998) y SAX (Megginson, 2004). En la propuesta de SAX, el análisis y lectura comienza en el inicio del documento XML. Mientras DOM trabaja sobre el documento completo, SAX trabaja el documento en "partes" de manera secuencial. SAX detecta cuando empieza y termina un elemento o documento, o conjunto de caracteres, etc., pasando cada "parte" del documento a la aplicación a medida son encontradas. No se almacena nada en memoria, por lo cual SAX se considera eficiente en el uso de memoria. La aplicación tiene acceso a las "partes" a medida estas aparecen en el análisis, no pudiendo hacer ninguna manipulación en memoria de los datos. Cuando se encuentra

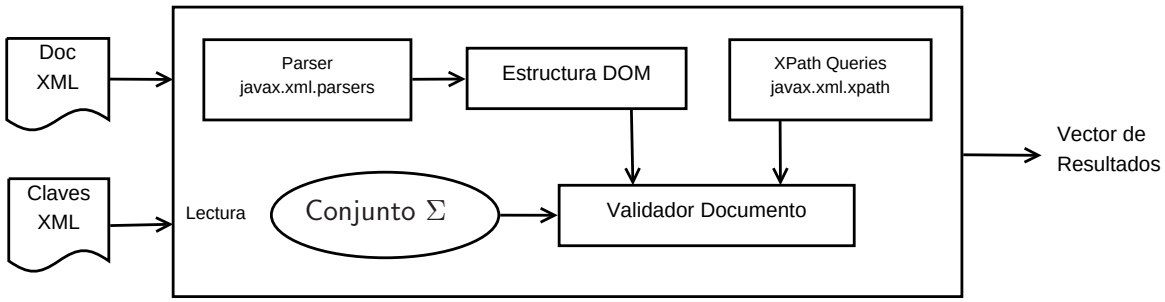


FIGURA 6.1: *Arquitectura del validador de documentos XML contra claves.*

algo significativo (lo que SAX denomina un "evento") ya sea una etiqueta de apertura o de cierre, o el texto contenido en la etiqueta, SAX deja disponibles los datos para la aplicación.

La propuesta de DOM, a partir del análisis del documento, crea un árbol de objetos que representa el contenido y la organización de los datos contenidos en el documento. Por lo tanto, en este caso el árbol es almacenado en memoria. Debido a esto, la aplicación puede realizar navegaciones a través del árbol (ir a los hijos, volver al padre, etc.) para obtener los datos a medida los necesite, o incluso manipularlos.

Es sabido que las aplicaciones que utilizan el enfoque basado en árboles como DOM, si bien son muy útiles en las operaciones, requieren un gran esfuerzo de los recursos del sistema, especialmente si se trabaja con documentos de gran tamaño. Es por esto, que se sabe de antemano que la aplicación a desarrollar tendrá impuesta limitaciones en el tamaño de documentos XML a trabajar, y presentará cierta desventaja ante las aplicaciones existentes (que utilizan índices, bases de datos relacionales, etc.).

Utilizando la biblioteca `javax.xml` que provee Java para el trabajo con documentos XML, se presenta una implementación simple, que ejecuta las instrucciones planteadas por el Algoritmo 5.3. La Figura 6.1, muestra la arquitectura que sigue el validador de documentos XML contra un conjunto de claves implementado.

La entrada del validador, consiste en dos archivos: el primero es el documento XML a validar contra un conjunto de claves, y el segundo un archivo de texto plano con las claves XML (ver Sección 6.2). Cada una de las entradas es analizada y transformada en estructuras almacenadas en memoria. Las claves son almacenadas en un conjunto (instancia de `LinkedHashSet`), como instancias de una clase `KeyClass`. Esta clase, almacena cada una de las expresiones de camino que componen una clave XML.

```
/**
 * Representacion de una clave XML
 */
public class KeyClass {
  private String contextPath;
  private String targetPath;
  private ArrayList<String> keyPaths;
}
```

Por otro lado, el documento XML es transformado en una estructura de árbol DOM, la cual es almacenada completamente en memoria. Para hacer esto, en la implementación se utiliza la biblioteca `javax.xml.parsers`, con las siguientes líneas de código.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(new InputSource(pathXMLdoc));
```

Con esto se pueden producir árboles de objetos DOM a partir de documentos XML. (Se requiere la ubicación el documento, dada por el valor de `pathXMLdoc`.)

Se habla de una implementación simple, ya que las bibliotecas y API's utilizadas, no son las más eficientes en el manejo de XML. (Este punto es justificado en la Sección 7.2.) Debido a esto, la implementación desarrollada posee algunas limitaciones.

La limitación más importante, es producto de la utilización de una estructura DOM, que es una representación del documento XML almacenada en memoria. Esta estructura posibilita la realización de consultas XPath sobre ella, permitiendo realizar las instrucciones dictadas por el algoritmo. Debido a esto, no es posible realizar la validación de documentos de gran tamaño. En los experimentos, el documento de mayor tamaño con el cual se ha trabajado correctamente pesa sólo 23 MB.

A grandes rasgos, una consulta XPath lo que hace es recorrer y procesar un árbol XML, de acuerdo a las etiquetas que la componen. Este lenguaje permite buscar y seleccionar nodos tomando en consideración la estructura jerárquica de XML. Las expresiones de camino que componen una clave, son traducidas a expresiones XPath.

En la implementación se realiza esta traducción, utilizando la siguiente función.

```
/**
 * Traduccion de lenguaje de camino a XPath.
 * @param path Expresion de lenguaje de camino a traducir.
 * @return Expresion XPath.
 */
public String getXPathOf(String path)
{
  String xpath = "";
  xpath = path.replace("epsilon", "");
```

```
10    xpath = xpath.replace("S", "text()");
11    xpath = xpath.replace(".", "/");
12    xpath = xpath.replace("_*", ".//.");
13    return xpath;
14  }
```

Todos los nodos del árbol construido son instancias de la clase `Node`. Esta clase posee métodos que permiten explorar el árbol: (1) se pueden obtener referencias a todos los nodos hijos, al primer hijo, al último, al siguiente hijo, o al padre de un nodo; (2) se puede obtener el tipo de un nodo (los requeridos son atributo, elemento, o texto); y (3) encontrar todos los nodos en un espacio de nombres particular. A partir de esto se tienen todas las operaciones deseadas.

De esta forma, se realizan las consultas XPath necesarias, y se calculan las intersecciones en valor correspondientes, para decidir si una determinada clave es satisfecha por la estructura (árbol XML T_D), y por ende por el documento XML D.

Notar que la implementación permite claves compuestas por expresiones de camino con etiquetas que no están en T_D. Estas claves son satisfechas de manera trivial. Por ejemplo, la clave XML $(\varepsilon, (aa.bb.c, \{ee.dd.f.S, pp.rr.S\}))$, cuyas expresiones de camino no se componen de etiquetas presentes en el árbol XML de la Figura 5.1, es satisfecha de manera trivial por el árbol, ya que cumple con las condiciones impuestas en la definición de satisfacción de claves XML (ver Definición 3.2).

6.2 ARCHIVO DE ENTRADA

La implementación del algoritmo de validación de documentos XML contra un conjunto de claves, recibe como entrada un documento XML, y un conjunto de claves XML. Estas claves deben ser pasadas como parámetros a la implementación, a través, de un archivo con una estructura definida. (En el caso de desear previamente aplicar el algoritmo para calcular un *cover* no redundante, se debe tener en cuenta que es necesario asegurar que las claves pertenezcan a $\mathcal{K}^{PL}_{PL,PL^+_s}$.)

Para el archivo que contiene el conjunto de claves contra las cuales se validará el documento XML, es mostrado en la Tabla 6.1. La notación definida para cada claves es la siguiente. El camino de contexto se separa del camino objetivo mediante un ";". El camino objetivo se separa del conjunto de caminos clave mediante un ";". Cada elemento de los caminos claves se separa de otro mediante ",". Si una clave contiene el símbolo comodín _*, este se escribe tal cual "_*" en el

TABLA 6.1: *Estructura del archivo de entrada del validador.*

Formato Archivo	Clave XML σ_1 Clave XML σ_2 ... Clave XML σ_n
Ejemplo	$epsilon; proyecto; @codp$ $proyecto; jefe; persona$ $epsilon; \_^*.persona; nombre.S, oficina.S$

archivo de entrada, mientras el símbolo "ε" para la palabra vacía se escribe "epsilon". Nótese, que en la tabla se muestran elementos con "@" como prefijo. Aquellos nodos que sean atributos, deben ser denotados agregándoles el prefijo "@" (requerido para realizar las consultas XPath). Esto último se exige sólo en la implementación de validación de documentos. La entrada a la implementación de *covers* no redundantes es la misma que para el caso de la implicación (ver Sección 4.4.1).

6.3 EJECUCIÓN DE LA IMPLEMENTACIÓN

La implementación del algoritmo de validación de documentos XML contra un conjunto de claves se realizó sobre Java versión 6. Para su ejecución se requiere el entorno de ejecución de Java (JRE)[1]. En el caso de querer recompilar el código fuente se requiere adicionalmente el entorno de desarrollo (JDK)[2].

Para ejecutar la implementación del validador de documentos XML, se debe utilizar la siguiente línea de comandos:

```
java [conf] -jar KeyValidator.jar -f archivoxml -k arhivoclaves -r nodoraiz [-s] [-v]
```

donde, `conf` es un argumento opcional, que corresponde a configuraciones para la máquina virtual Java, y puede ser reemplazado por los parámetros `-DentityExpansionLimit=1000000 -Xms512m -Xmx1024m`; estas configuraciones son requeridas para ampliar el límite de memoria disponible para la aplicación. `archivoxml` debe ser reemplazado: por el nombre del archivo XML si éste se encuentra en la misma carpeta, o por la ruta seguida del nombre del archivo, si se encuentra en otra carpeta.

[1]http://www.java.com/es/download/
[2]http://www.oracle.com/technetwork/java/javase/downloads/index.html

Lo mismo aplica a `archivoclaves`, el cual debe cumplir con el formato especificado. Seguido a `-r` se debe reemplazar `nodoraiz`, por la etiqueta del nodo raíz del documento XML. La opción `-s` se debe agregar sólo cuando se desee validar el documento contra un conjunto de claves. (Por defecto se realiza la validación contra la primera clave encontrada en el archivo de claves.) `-v` solicita a la aplicación entregar mayor detalle del proceso de validación. Los resultados obtenidos, son entregados mediante la misma línea de comandos. El resultado es un vector V de valores booleanos, donde $V[i]$ es *true*, si la clave i es satisfecha, y *false* si no lo es.

6.4 FUNCIONES PRINCIPALES DE LA IMPLEMENTACIÓN

El Algoritmo 5.3 para validación de documentos XML contra un conjunto de claves, requiere del algoritmo para determinar si existe o no intersección en valor (Algoritmo 5.2) entre dos listas de nodos. A su vez, el algoritmo de intersección requiere del algoritmo de igualdad en valor (Algoritmo 5.1). Estos algoritmos, junto con el algoritmo para calcular *covers* no redundantes de conjuntos de claves XML, componen la parte más novedosa de la implementación, comparado con Abrão et al. (2004); Bouchou et al. (2003); Chen et al. (2002); Liu et al. (2005, 2004) donde la igualdad en valor se restringe a la igualdad en valor de los nodos hoja del árbol XML (atributos y PCDATA). A continuación, se presentan las funciones que implementan estos algoritmos.

6.4.1 Función igualdad en valor

Esta función hace uso de los métodos que provee el trabajo con la estructura DOM, para determinar si dos nodos son iguales en valor. Primero, se realiza la comparación de etiquetas, y luego dependiendo del tipo de nodo: se determina su igualdad en valor en texto, para nodos atributo o texto; o se realizan llamados de manera recursiva a la misma función, para comparan los nodos en los subárboles, si se trata de nodos elemento.

```
1 /**
2  * Igualdad en valor entre dos nodos dados.
3  * @param node_u Primer nodo.
4  * @param node_v Segundo nodo.
5  * @return True si node_u es igual en valor a node_v, false en otro caso.
6  */
```

```
7  public boolean valueEqual(Node node_u, Node node_v)
8  {
9    if (node_u.getNodeName().equals(node_v.getNodeName()))
10   {
11     if ((node_u.getNodeType() == node_v.getNodeType()) &&
12         (node_u.getNodeType() == Node.ATTRIBUTE_NODE ||
13            node_u.getNodeType() == Node.TEXT_NODE))
14     {
15       if (node_u.getNodeValue().equals(node_v.getNodeValue()))
16       {
17         return true;
18       } else
19       {
20         return false;
21       }
22     } else if (node_u.getNodeType() == node_v.getNodeType() &&
23                node_u.getNodeType() == Node.ELEMENT_NODE)
24     {
25       // Compara nodos atributos hijos
26       NamedNodeMap attrMapU = node_u.getAttributes();
27       NamedNodeMap attrMapV = node_v.getAttributes();
28       if (attrMapU.getLength() == attrMapV.getLength())
29       {
30         for (int i = 0; i < attrMapU.getLength(); i++)
31         {
32           Node nodeU_aux = attrMapU.item(i);
33           Node nodeV_aux = attrMapV.item(i);
34           if (!valueEqual(nodeU_aux, nodeV_aux))
35           {
36             return false;
37           }
38         } //endFor
39       } else
40       {
41         // Los nodos no poseen la misma cantidad de atributos
42         return false;
43       } //fi
44
45       // Comparar nodos elemento hijos
46       /* La funcion getChildNodes() retorna los nodos atributo hijos,
47        * ordenados por orden alfabetico en las etiquetas de los nodos. */
48       NodeList eleListU = node_u.getChildNodes();
49       NodeList eleListV = node_v.getChildNodes();
50       if (eleListU.getLength() == eleListV.getLength())
51       {
52         for (int i = 0; i < eleListU.getLength(); i++)
53         {
54           Node nodeU_aux = eleListU.item(i);
55           Node nodeV_aux = eleListV.item(i);
56           if (!valueEqual(nodeU_aux, nodeV_aux))
57           {
58             return false;
59           }
60         } //endFor
61       } else
62       {
63         // Los nodos no tienen la misma cantidad de elementos hijo
64         return false;
65       } //fi
66       return true;
67     } //fi
68   } //fi
69   return false;
70 }
```

6.4.2 Función intersección en valor

Siguiendo la estrategia definida en el Algoritmo 5.2, se busca la primera ocurrencia de una intersección en valor, entre dos nodos, dentro de los listados pasados como parámetro. Ambos listados son instancias de la clase `NodeList` que provee DOM en Java, y son comparados según la posición de los ítems que los componen. Se compara cada ítem de la `lista1`, con cada ítem de la `lista2`.

```
1  /**
2   * Interseccion en valor de dos listas de nodos.
3   * @param lista1 Primer listado de nodos.
4   * @param lista2 Segundo listado de nodos.
5   * @return true si la interseccion es vacia; false en otro caso.
6   */
7  public boolean valueIntersection(NodeList lista1, NodeList lista2)
8  {
9    for (int i = 0; i < lista1.getLength(); i++)
10   {
11     for (int j = 0; j < lista2.getLength(); j++)
12     {
13       Node nodeLista1 = lista1.item(i);
14       Node nodeLista2 = lista2.item(j);
15       if (valueEqual(nodeLista1, nodeLista2))
16       {
17         /*
18          * Se cumple la condicion necesaria y suficiente,
19          * para determinar que la interseccion no es vacia
20          */
21         return false;
22       } //fi
23     } //endFor
24   } //endFor
25   return true;
26 }
```

6.5 IMPLEMENTACIÓN ALGORITMO *COVERS* NO REDUNDANTES

Al momento de realizar la validación de un documento XML contra un conjunto de claves Σ, tal como se describió en el Capítulo 5, es natural pensar en calcular primero un *cover* no redundante de Σ. Esto puede disminuir potencialmente el número de claves contra las cuales debe ser validado el documento, y consecuentemente el tiempo de la validación.

A continuación se presentan los detalles de la implementación del Algoritmo 5.4 para determinar *covers* no redundantes.

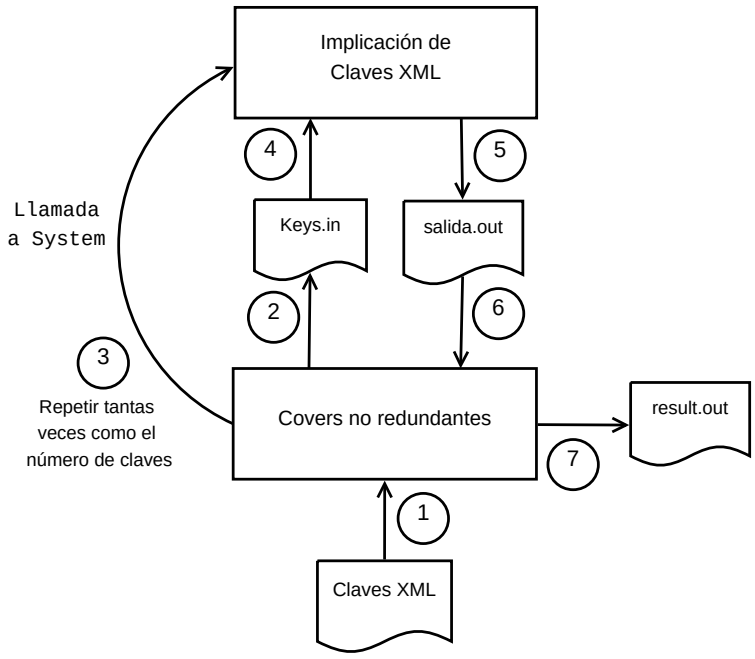


FIGURA 6.2: *Flujo de ejecución de la implementación del algoritmo de covers no redundantes.*

6.5.1 Detalles de la implementación

La implementación del algoritmo de *covers* no redundantes, requiere de la implementación realizada para la implicación de claves XML. Por lo tanto, siguiendo la línea de programación, también fue desarrollada en C++.

Básicamente, la implementación del algoritmo de *covers* no redundantes para claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$, consiste de siete pasos. Estos pasos se grafican, en la Figura 6.2, y son explicados a continuación:

1. En este paso, se entrega el archivo con el conjunto Σ de claves, del cual se desea obtener su *cover* no redundante.

2. La aplicación lee el archivo de entrada, y genera un archivo de entrada para la aplicación que realiza la implicación de claves. En este caso el archivo que se genera es llamado `keys.in`.

3. Se realiza una llamada a la aplicación que decide la implicación, por cada una de las claves en Σ.

4. La aplicación de implicación, realiza la lectura del archivo de entrada `keys.in`. Realiza el proceso para determinar la implicación de la clave, y obtiene el resultado.

5. El resultado es guardado en un archivo llamado `salida.out`. La salida es *true* si la clave es implicada, o *false* en otro caso.

6. La aplicación de *covers* retoma su ejecución, leyendo el archivo de salida de la implicación.

7. Si ya se han analizado todas las claves en Σ, se escribe el *cover* Θ obtenido en un archivo de salida `result.out`. En otro caso, si la clave es implicada es eliminada del conjunto, se analiza la implicación de la siguiente clave en Σ, y se genera un nuevo archivo de claves, volviendo al paso 2.

En el paso 3, se construye la variable `path` como cadena de caracteres, con valor `./xmlkeys -f keys.in -l valor_l`. Esta variable es la que luego será utilizada como comando para una ejecución de la implementación de implicación de claves, pasando como parámetro (permitiendo la comunicación) el archivo `keys.in` creado en el paso 2, a partir del archivo de claves XML de entrada. (Notar que esta comunicación se puede hacer mucho más eficiente, manteniendo las variables en memoria, pero para efectos de los experimentos, y para mantener la claridad de los mismos, se ha tomado la decisión de mantener la comunicación a través de archivos.) Esta variable es utilizada para realizar un llamado a sistema, a través de la siguiente instrucción en la aplicación, `int result_xmlkeys = system(path.c_str());`. Esta instrucción, solicita la ejecución de la implicación de claves (paso 3), y retorna un valor igual a cero, si la ejecución fue exitosa, o distinto a cero en otro caso. Luego de dicha ejecución (paso 4), se puede recuperar el resultado de la implicación desde el archivo `salida.out` (paso 5), y determinar si dicha clave pertenece o no al *cover* no redundante. La variable `valor_l`, toma su valor del parámetro `etiqueta`, entregado en la llamada a la ejecución de la implementación de *covers* no redundantes.

6.5.2 Archivo de entrada

La estructura del archivo de entrada para la implementación del algoritmo de *covers* no redundantes, es igual a la presentada en la Sección 4.4.1, para la implicación de claves XML.

Se resalta el requerimiento de que al final del archivo de entrada, se deje una línea en blanco.

6.5.3 Ejecución de la implementación

Si se desea realizar la ejecución de la implementación del algoritmo de *cover* no redundante para claves XML, programada en el lenguaje C++, se debe tener instalada alguna distribución Linux con Kernel 2.6.32 (o posterior) sobre una máquina x86. En el caso de requerir recompilar el código fuente, se debe tener instalada la versión 4.4 (o posterior) del compilador *g++* de GCC[3].

La ejecución de la implementación, puede ser iniciada en una consola con el siguiente comando: `./coverset -f archivo -l etiqueta`. Donde, `archivo` debe ser reemplazado por el nombre del archivo (o la ruta) que contenga el conjunto de claves Σ (en el formato indicado en el punto 6.5.2); `etiqueta` debe ser reemplazado por el valor de la etiqueta $\ell_0 \in$ a ser reemplazada en el proceso de implicación (`valor_l`).

Como resultado se obtiene un detalle de las llamadas a la implementación del algoritmo de implicación, para cada una de las claves en Σ, por línea de comandos. Además, se obtiene un archivo `result.out` el cual puede ser utilizado como entrada para la implementación del algoritmo de validación. Agregando previamente los prefijos "@" cuando corresponda (ver Sección 6.2).

[3]GNU Compiler Collection, http://gcc.gnu.org/

CAPÍTULO 7. EXPERIMENTACIÓN CON IMPLEMENTACIONES

Hasta ahora, junto con presentar cada uno de los algoritmos trabajados, se ha presentado un análisis de sus correspondientes complejidades temporales. Todas las complejidades presentadas consideran el llamado *peor caso*, en donde se consideran las peores instancias para una ejecución. A partir de cada uno de los algoritmos se realizó una implementación computacional, con el fin de poder realizar pruebas prácticas con cada uno. En este capítulo se evaluará el rendimiento de los algoritmos en la práctica a partir de las implementaciones desarrolladas.

Para iniciar, en la primera sección se analizan las ejecuciones de implementación del algoritmo de implicación de claves, revisando los casos posibles y, mostrando los tiempos que toma cada ejecución con diferentes entradas. En la segunda sección se estudia el comportamiento de la validación de documentos XML, evaluando diferentes claves, sobre diferentes documentos. Por último, se presenta un análisis del cálculo de *covers* no redundantes sobre un conjunto de claves. Se muestra también cuál es la disminución obtenida en el tiempo de validación de un documento XML, contra un *cover* no redundante, en lugar del conjunto completo de claves.

Todos los experimentos fueron ejecutados en una misma máquina Intel(R) Core(TM) 2 Duo T7250 de 2.0 GHz, 3 GB de memoria RAM, y un disco duro de 5400 RPM. El sistema operativo es una distribución Linux con kernel 2.6.32. Se utilizó C++ para la implementación de los algoritmos de implicación y *covers* no redundante, y Java para la validación de documentos XML contra claves XML.

7.1 EXPERIMENTOS CON IMPLICACIÓN DE CLAVES

Se evalúa el rendimiento del algoritmo de implicación de claves XML (Hartmann & Link, 2009a), utilizando conjuntos de claves definidas de manera aleatoria a partir de un alfabeto finito.

Las expresiones de camino utilizadas en la generación de las claves, se restringen a aquellas que son válidas, y estén en forma normal. Los conjuntos de etiquetas utilizados para generar las claves (considerando el comodín, $\_^*$, y la palabra vacía, ε), se obtuvieron de las instancias utilizadas en los trabajos de referencia y en la Sección 7.2 para validación de documentos.

7.1.1 Análisis de casos específicos

Se presentan los resultados obtenidos en experimentos con el algoritmo de implicación de claves XML, definidos por el tipo de clave φ y la composición del conjunto Σ de claves. φ puede ser absoluta o relativa, y Σ puede estar compuesto solamente de claves absolutas o relativas (homogéneo), o puede contener claves de ambos tipos (heterogéneo). Combinando lo anterior es posible generar seis casos para su análisis. Los conjuntos Σ utilizados son de tamaño igual a 100 claves en cada uno de los casos.

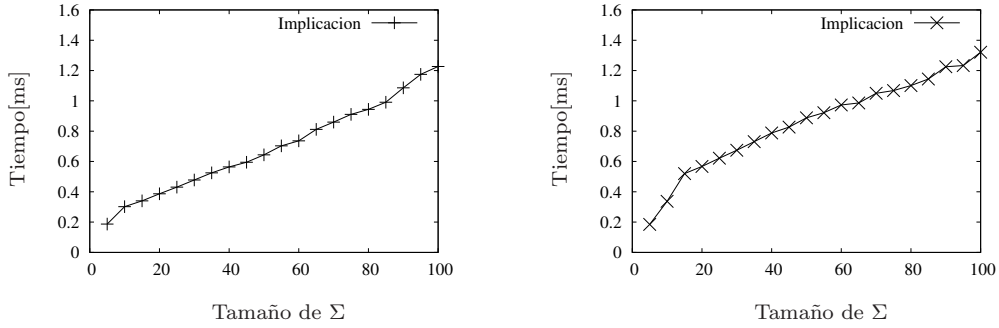
El eje x de las gráficas, corresponde al número de clave del conjunto Σ, denotado $|\Sigma|$. Aquí, se consideran incrementos diferenciales, con $\Delta = 5$ claves en cada ejecución, hasta alcanzar el máximo de 100. El eje y corresponde al tiempo requerido para decidir si $\Sigma \models \varphi$; sin considerar el tiempo de lectura de los archivos de entrada, sólo el tiempo requerido por las instrucciones que indica el algoritmo.

Cabe resaltar que debido a que, hasta donde se ha estudiado el problema, no existen otras implementaciones del algoritmo de implicación trabajado, no se pueden realizar comparaciones con otras implementaciones.

En los primeros dos casos se considera que el conjunto Σ se compone solamente de claves absolutas, es decir, el camino de contexto es la palabra vacía. A la izquierda en la Figura 7.1 se muestra el caso donde φ es absoluta, y a la derecha cuando φ es relativa.

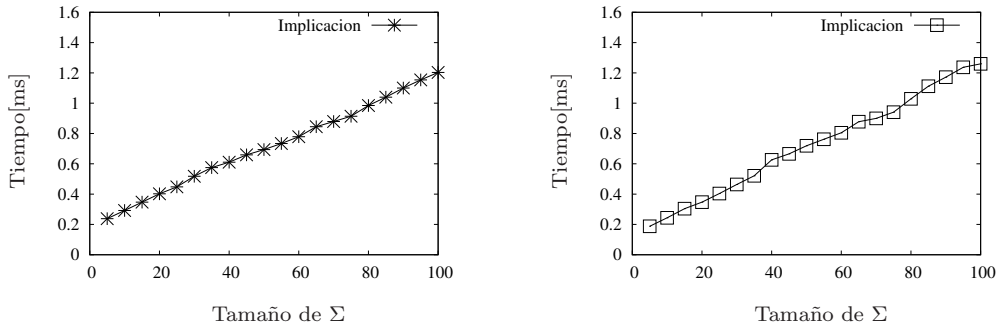
La Figura 7.2 presenta las gráficas obtenidas para la implicación de claves, absolutas y relativas, ante un conjunto Σ de claves relativas.

A medida que se incrementa el número de claves en el conjunto Σ, en el intervalo analizado, ambas gráficas siguen una tendencia casi lineal. Queda demostrado empíricamente que los tiempos que requiere la ejecución de la implicación de claves son ínfimos, menores a 1,5 milisegundos para un Σ con 100 claves. Así, el peor caso teórico considerado para el proceso de implicación de claves XML,



(a) Clave φ absoluta. (b) Clave φ relativa.

FIGURA 7.1: *Implicación de claves con* Σ *compuesto de claves absolutas.*



(a) Clave φ absoluta. (b) Clave φ relativa.

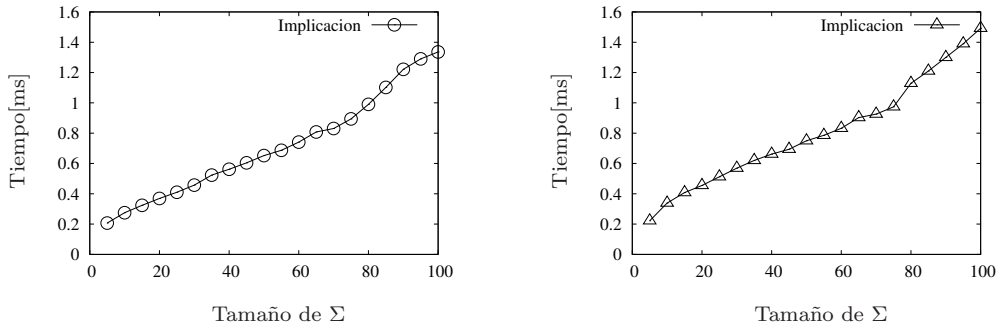
FIGURA 7.2: *Implicación de claves con* Σ *compuesto de claves relativas.*

$\mathcal{O}(n^2)$, no se cumple en la práctica hasta donde se puede ver, según los experimentos realizados.

En todos los casos la primera ejecución, con $|\Sigma|$ igual a 5 claves, toma un tiempo cercano a los 0,2 milisegundos; mientras la última ejecución, con un $|\Sigma|$ igual a 100 claves, toma un tiempo que bordea entre 1,2 y 1,4 milisegundos. Estos primeros cuatro casos, donde Σ se compone sólo de claves relativas o absolutas (Σ homogéneo), muestran la eficiencia del algoritmo en la práctica, y la tendencia casi lineal que siguen los tiempos en los experimentos realizados con el algoritmo de implicación de claves.

La Figura 7.3, muestra las gráficas obtenidas tomando un conjunto Σ que contiene claves tanto absolutas como relativas, presentándose un comportamiento similar en eficiencia al mostrado cuando Σ tiene una composición homogénea.

En todos los experimentos, se destacan los mejores tiempos obtenidos en el proceso de implicación, cuando φ es una clave absoluta, frente a aquellos en que φ es relativa. La explicación para esto presentada aquí, es que si $Q_\varphi = \varepsilon$, es más rápida la obtención de los conjuntos $[\![Q_\varphi]\!]$ y $[\![Q_\varphi.Q'_\varphi]\!]$, cada vez que se requiere de ellos. Además, siempre que la clave φ sea absoluta, el nodo



(a) Clave φ absoluta. (b) Clave φ relativa.

FIGURA 7.3: *Implicación de claves con* Σ *compuesto de claves absolutas y relativas.*

q_φ corresponde a la raíz, ahorrándose el tiempo de su búsqueda (como ocurre en el caso que φ es relativa).

Considerando los seis casos analizados (tipo de claves en Σ - tipo de φ), la Figura 7.4 muestra la superposición de todas las gráficas anteriores. Se puede observar que se mantiene una tendencia casi lineal en las ejecuciones, aunque se debe considerar que los tiempos de ejecución son muy pequeños. Se destacan algunos hechos: con un $|\Sigma| = 5$ se obtienen tiempos cercanos a 0,2 milisegundos en todos los casos; mientras, para un $|\Sigma| = 100$, los tiempos bordean entre 1,2 y 1,5 milisegundos. Los tiempos más altos se obtienen al considerar un Σ heterogéneo o mixto, que contiene claves relativas y absolutas.

Si se analiza el comportamiento que tiene la instancia en que Σ se compone de claves absolutas, y φ es una clave relativa (en la Figura 7.4). El excesivo incremento para $5 \leq |\Sigma| \leq 15$, es debido a que, en esas ejecuciones el conjunto Σ poseía el mayor número de claves que sí son aplicables a φ, y por ende se realizan todos los pasos para obtener las aristas testigo. Mientras, para claves que no son aplicables, estos pasos requieren menos tiempo (finalizan tempranamente). Además, mientras más aristas testigo existan más tiempo tomará decidir si q_φ es alcanzable desde q'_φ.

La presencia de comodines o *wildcards* en las instancias, también es importante e influyente en los tiempos obtenidos. A medida que aumenta la presencia de comodines en las claves que componen Σ, aumenta el tiempo de implicación. Para corroborar esto, se utiliza la instancia del caso en que Σ es heterogéneo y φ es relativa, duplicando la presencia de comodines. Se reemplazan algunas de las etiquetas presentes en las claves de la instancia por "_*", de manera aleatoria, procurando dejar claves que pertenezcan a $\mathcal{K}^{PL}_{PL,PL^+_s}$.

La Figura 7.5 presenta las gráficas del comportamiento de la implicación de claves ante

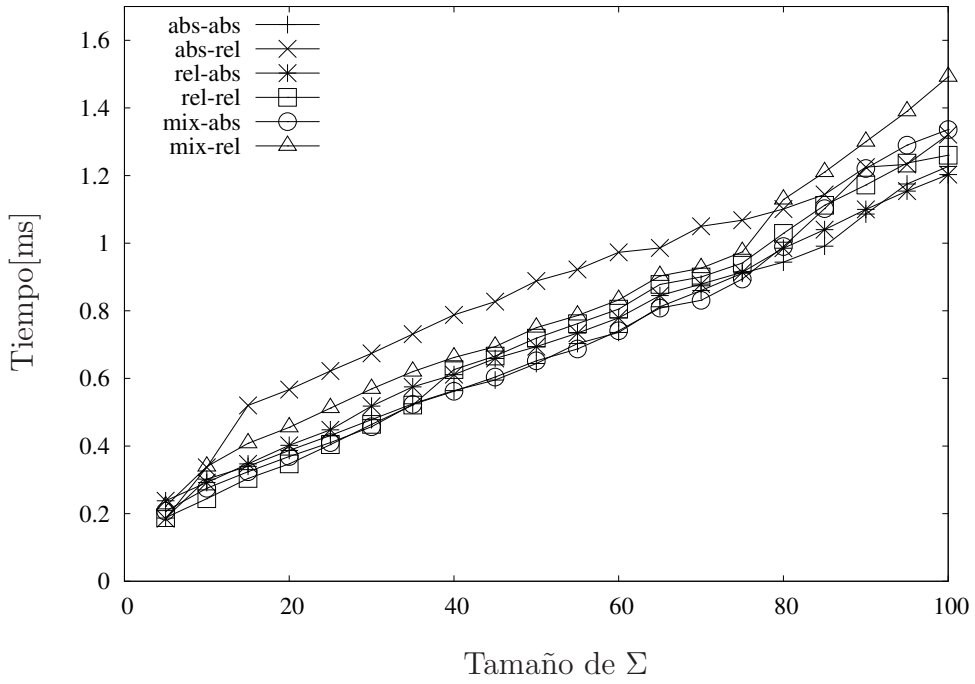


FIGURA 7.4: *Implicación de claves XML, todos los casos.*

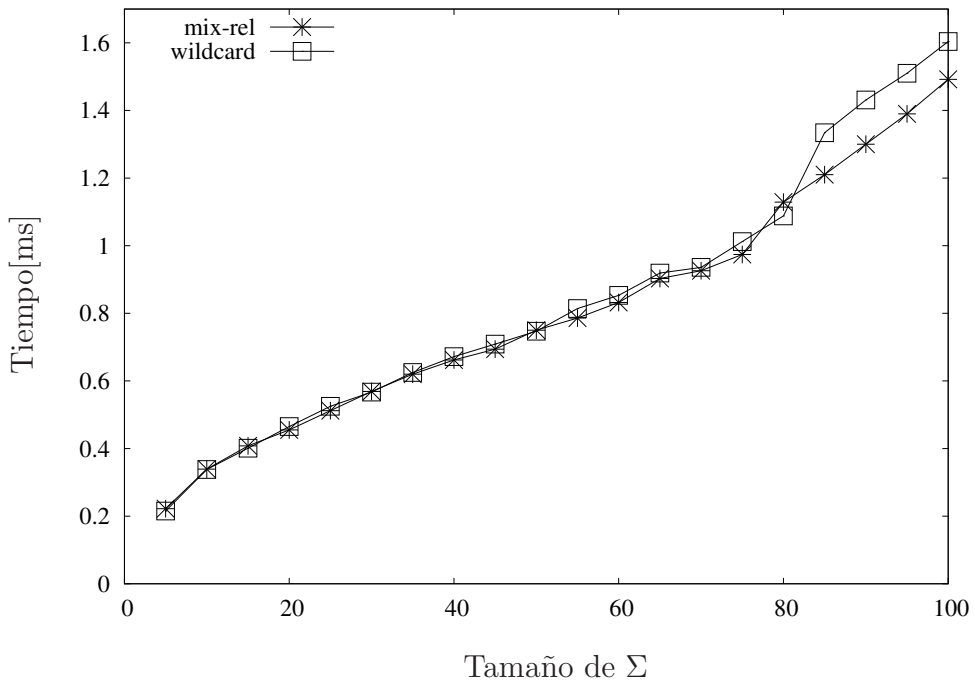


FIGURA 7.5: *Efecto de la presencia de comodines en las claves.*

la presencia excesiva de comodines en las claves. Durante la primera mitad, el comportamiento de ambas gráficas es similar. Pasado ese punto, la excesiva presencia de claves con comodín comienza a producir un aumento del tiempo, alcanzando los 1,6 milisegundos; mientras ante el mismo conjunto de claves, con menor cantidad de comodines, el tiempo de implicación alcanza los 1,5 milisegundos. Si bien, el cambio es insignificante tomando en cuenta la unidad utilizada, se resalta la existencia de un aumento en el tiempo de implicación en presencia de claves con comodines.

Lo expuesto muestra que el algoritmo de implicación de claves es eficiente en la práctica, con un comportamiento lineal en todos los casos, más eficiente que el peor caso teórico $\mathcal{O}(n^2)$.

7.1.2 Análisis del caso general

En esta sección se presenta un análisis del caso general para ejecuciones del algoritmo de implicación de claves, esto es, dado que la complejidad teórica en el peor caso del algoritmo de implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$ es cuadrática, es interesante analizar qué ocurre la mayoría de las veces que se aplica este algoritmo sobre un conjunto de claves.

Se trabajará con conjuntos específicos de claves, con características deseadas, que permitirán vislumbrar el comportamiento del algoritmo de implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^+_s}$ en el caso general. Se utilizarán las claves XML presentadas en la Tabla 7.1. Estas claves, están divididas en cuatro conjuntos: (1) las claves 1.x son claves absolutas de tamaño incremental, con presencia de comodines en algunos de los caminos objetivo; (2) las claves 2.x son claves relativas de tamaño incremental, que no presentan comodines en ninguno de los caminos; (3) las claves 3.x son las mismas claves 2.x modificadas, concatenando la palabra comodín al inicio de los caminos objetivo; y (4) las claves σ_i que componen el conjunto Σ.

Considerando los tres primeros conjuntos, se generan cuatro pruebas a analizar. Tomando el primer conjunto, y un $\ell_0 = personal$ se obtiene la primera prueba, y con un $\ell_0 = agenda$ se obtiene la segunda prueba. La tercera prueba se obtiene tomando el segundo conjunto (sin palabras comodín a reemplazar). Tomando el tercer conjunto, y un $\ell = personal$ se obtiene la cuarta prueba. Cada una de estas pruebas consiste en determinar el tiempo de implicación de una clave en el conjunto correspondiente, por el conjunto Σ compuesto de cinco claves. Las gráficas resultantes para cada prueba son mostradas en la Figura 7.6, considerando en el eje x el tamaño de las claves de entrada,

TABLA 7.1: *Conjuntos de claves XML para análisis del algoritmo de implicación.*

Id	$\|\varphi\|$	Clave
1.1	1	$(\varepsilon, (contacto, \{\varepsilon\}))$
1.2	2	$(\varepsilon, (contacto, \{nombre\}))$
1.3	3	$(\varepsilon, (contacto, \{nombre.S\}))$
1.4	4	$(\varepsilon, (contacto, \{nombre.S, telefono\}))$
1.5	5	$(\varepsilon, (contacto, \{nombre.S, telefono.S\}))$
1.6	6	$(\varepsilon, (contacto.\_^*, \{nombre.S, telefono.S\}))$
1.7	7	$(\varepsilon, (\_^*.contacto.\_^*, \{nombre.S, telefono.S\}))$
1.8	8	$(\varepsilon, (\_^*.contacto.\_^*, \{nombre.S, telefono.S, direccion\}))$
1.9	9	$(\varepsilon, (\_^*.contacto.\_^*, \{nombre.S, telefono.S, direccion.S\}))$
1.10	10	$(\varepsilon, (\_^*.contacto.\_^*, \{nombre.S, telefono.S, direccion.S, relacion\}))$
1.11	11	$(\varepsilon, (\_^*.contacto.\_^*, \{nombre.S, telefono.S, direccion.S, relacion.S\}))$
2.1	2	$(agenda, (contacto, \{\varepsilon\}))$
2.2	3	$(agenda, (contacto, \{nombre\}))$
2.3	4	$(agenda, (contacto, \{nombre.S\}))$
2.4	5	$(agenda, (contacto, \{nombre.S, telefono\}))$
2.5	6	$(agenda, (contacto, \{nombre.S, telefono.S\}))$
2.6	7	$(agenda, (contacto, \{nombre.S, telefono.S, direccion\}))$
2.7	8	$(agenda, (contacto, \{nombre.S, telefono.S, direccion.S\}))$
2.8	9	$(agenda, (contacto, \{nombre.S, telefono.S, direccion.S, relacion\}))$
2.9	10	$(agenda, (contacto, \{nombre.S, telefono.S, direccion.S, relacion.S\}))$
3.1	3	$(agenda, (\_^*.contacto, \{\varepsilon\}))$
3.2	4	$(agenda, (\_^*.contacto, \{nombre\}))$
3.3	5	$(agenda, (\_^*.contacto, \{nombre.S\}))$
3.4	6	$(agenda, (\_^*.contacto, \{nombre.S, telefono\}))$
3.5	7	$(agenda, (\_^*.contacto, \{nombre.S, telefono.S\}))$
3.6	8	$(agenda, (\_^*.contacto, \{nombre.S, telefono.S, direccion\}))$
3.7	9	$(agenda, (\_^*.contacto, \{nombre.S, telefono.S, direccion.S\}))$
3.8	10	$(agenda, (\_^*.contacto, \{nombre.S, telefono.S, direccion.S, relacion\}))$
3.9	11	$(agenda, (\_^*.contacto, \{nombre.S, telefono.S, direccion.S, relacion.S\}))$
σ_1		$(\varepsilon, (agenda.contacto, \{nombre.S\}))$
σ_2		$(\varepsilon, (agenda.contacto, \{nombre.S, telefono.S\}))$
σ_3		$(\varepsilon, (agenda.\_^*, \{nombre.S\}))$
σ_4		$(agenda, (contacto, \{nombre.S\}))$
σ_5		$(agenda, (contacto, \{nombre.S, telefono.S\}))$

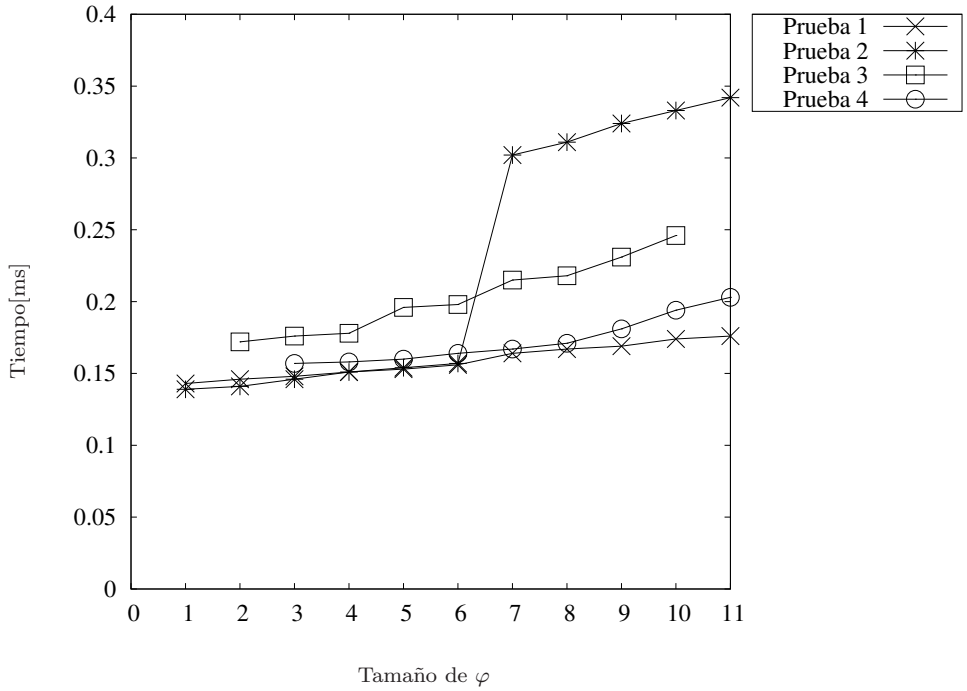


FIGURA 7.6: *Pruebas con la implicación de claves XML.*

y en el eje y, el tiempo requerido por el algoritmo de implicación.

A partir de las gráficas obtenidas para cada prueba (ver Figura 7.6) se realizan las siguientes descripciones.

Prueba 1: En esta prueba, ninguna de las claves del conjunto Σ es aplicable a la clave φ. Luego, no se agrega ninguna arista testigo al grafo. En esta prueba se puede visualizar que el efecto del tamaño de la clave φ sobre el tiempo de implicación es muy insignificante, es decir, a medida incrementa el tamaño de φ casi no incrementa el tiempo de implicación.

Prueba 2: En esta prueba se tiene un comportamiento similar a la prueba 1, hasta la clave 1.7. A partir de esta clave, el tiempo se incrementa casi el doble. Es particularmente debido al primer comodín presente en el camino objetivo de las claves 1.7 a la 1.11 y al $\ell_0 = agenda$ escogido, lo que incrementa el tiempo de implicación en estas claves absolutas. Ya que la presencia de comodines en los caminos objetivo, provoca que las consultas tipo XPath resultantes sean más complejas, y se tenga una demora en su respuesta. Nótese, que las claves $\sigma \in \Sigma$ no generan ninguna arista testigo, y el incremento del tiempo se debe sólo a la presencia del comodín. Realizando la misma prueba con otro valor para ℓ_0 como por ejemplo personal (Prueba 1), se obtienen tiempos mucho menores.

Prueba 3: En esta prueba se tienen los peores casos promedio. Ya que, casi todas las claves en Σ son aplicables a las φ, por lo cual se generan varias aristas testigo. Por lo tanto, determinar la alcanzabilidad también toma más tiempo. A partir de la clave 2.3 la implicación de la clave φ por Σ es verdadera, pero esto no implica un incremento excesivo en el tiempo.

Prueba 4: Aquí no existen claves σ que apliquen a φ, por lo que no se generan aristas testigo. A pesar de esto, los tiempos son un poco más elevados, debido a que las claves φ son claves relativas (igual que en la Prueba 3).

Con todos los experimentos realizados con el algoritmo de implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$, es válido considerar que en ningún caso el *witness-graph* resultante será un grafo completo, y que la cantidad de aristas de éste grafo dirigido es mucho menor a $n(n-1)$ con $n = |V|$, sino que por lo general se tienen $n \cdot k$ aristas, donde $k \geq 1$ y $k << (n-1)$. Debido a lo anterior, el peor caso $\mathcal{O}(|\varphi|^2)$ considerado para la alcanzabilidad no se alcanza nunca en el problema de implicación de claves XML. Esto resulta en que el tiempo requerido para decidir la implicación sea mucho menor al cuadrático en la práctica. También, se observa que los tiempos de implicación no dependen mayormente del tamaño de la clave de entrada. Sí dependen de la presencia de comodines en las expresiones de camino que componen las claves en Σ. La presencia excesiva de comodines hace que las expresiones tipo XPath resultantes sean más complejas de responder, y que tomen más tiempo.

7.1.3 Conclusiones

Una vez revisados los experimentos realizados con el algoritmo de implicación de claves, y dejando de lado casos específicos, se extraen los hechos observados que influyen en los tiempos de implicación de todas las entradas:

- El tamaño de las claves involucradas, sobre todo la cantidad de caminos claves presentes, no provocan un mayor incremento en el tiempo de implicación.
- El peor caso cuadrático teórico, no se da en ningún caso en la práctica. Siempre se tendrá una cantidad muy inferior de aristas, lo que reduce la complejidad temporal cuadrática de la alcanzabilidad.

- Mientras mayor sea el número de claves $\sigma \in \Sigma$ aplicables a φ, mayor será el tiempo requerido para decidir la implicación.
- El tiempo de implicación de una clave φ cuando $\sigma \in \Sigma$ es aplicable, es mayor a que si σ no fuese aplicable.
- La excesiva presencia de expresiones de camino con comodines en las claves, aumenta el tiempo requerido en decidir la implicación.
- Es eficiente el uso del algoritmo de implicación de clave en la práctica. Se comprueba empíricamente lo planteado en Hartmann & Link (2009a).

7.2 EXPERIMENTOS CON VALIDACIÓN DE DOCUMENTOS XML

En esta sección se realizan pruebas con la implementación del algoritmo de validación de documentos XML contra un conjunto de claves.

Todos los archivos utilizados en los experimentos pueden ser encontrados en *XML Data Repository* (Suciu, 2002b). Sólo el archivo correspondiente a DBLP ha sido modificado, descartando algunos elementos con el fin de disminuir su tamaño. Las instancias se limitan a documentos de tamaño menor a 23 MB, debido a limitaciones impuestas por el *Framework* y la API utilizados.

7.2.1 Descripción de las instancias

A continuación se presenta una breve descripción de los datos contenidos en cada uno de los archivos de prueba.

Auction Data Información de subastas electrónicas, convertida a un documento XML desde fuentes de la Web. Archivos 321gone.xml y yahoo.xml.

DBLP Computer Science Bibliography El servidor DBLP provee de información bibliográfica de los principales *journals* y *proceedings* de Informática. Archivo dblp.xml.

Nasa Conjuntos de datos convertidos a partir de archivos de texto plano en XML, y puesto a disposición del público. Archivo nasa.xml.

SIGMOD Record Índice de artículos de la revisar *SIGMOD Record*. Archivo SigmodRecord.xml.

Mondial Base de datos con la geografía mundial, integración del *CIA World Factbook*, el *International Atlas*, y la base de datos TERRA, entre otras fuentes (May, 1999). Archivo mondial-3.0.xml.

La Tabla 7.2 muestra una descripción de las instancias que se utilizan en los experimentos con el algoritmo de validación de documentos XML contra claves. El tamaño de las instancias no supera los 23 MB, debido a la limitación de memoria impuesta por la API y el *Framework* utilizados en la implementación.

Características relevantes para la definición de claves XML sobre un documento XML, son la profundidad máxima y promedio. Estas son determinadas por el correspondiente árbol XML T_D generado a partir del documento XML D. Si un árbol XML T_D posee poca altura (y probablemente pocas etiquetas), implica que la cantidad de claves válidas que se pueden definir es menor: se requiere de las etiquetas de los nodos para generar las expresiones de camino que componen una clave (camino de contexto, camino objetivo, y caminos clave).

Las características como número de elementos y atributos, dan cuenta de la complejidad al momento de resolver consultas de tipo XPath sobre el árbol XML. Un mayor número de nodos implica, que los tamaños de los conjuntos retornados por una consulta de este tipo, son mayores. Así, siguiendo el camino de contexto, se pueden tener muchos nodos contexto, que llevarán a muchos conjuntos de nodos objetivo siguiendo el camino objetivo, y a muchos nodos clave siguiendo los caminos clave.

No se realiza una clasificación de los documentos como la expuesta por Qureshi & Samadzadeh (2005), basada en la complejidad de las expresiones regulares obtenidas a partir de su DTD, ya que las claves con las cuales se trabaja no exigen que los documentos XML cumplan con restricciones estructurales como las impuestas por los DTD o XML *Schema*.

Las instancias utilizadas son de acceso público, y han sido utilizadas por muchos autores en otros experimentos.

TABLA 7.2: *Caracterización de las instancias para validación de documentos.*

Documento	No. de elementos	No. de atributos	Tamaño	Profundidad Máxima	Profundidad Promedio
321gone.xml	311	0	23 KB	5	3.76527
yahoo.xml	342	0	24 KB	5	3.76608
dblp.xml	29.494	3.247	1.6 MB	6	2.90228
nasa.xml	476.646	56.317	23 MB	8	5.58314
SigmodRecord.xml	11.526	3.737	476 KB	6	5.14107
mondial-3.0.xml	22.423	47.423	1 MB	5	3.59274

7.2.2 Claves a evaluar

A partir de cada una de las instancias descritas, se obtuvieron las claves XML en $\mathcal{K}^{PL}_{PL,PL_s^+}$ mostradas en la Tabla 7.3. Las claves fueron obtenidas de manera aleatoria, respetando el vocabulario existente, y el orden de los elementos en el árbol. Todas las expresiones de camino que componen las claves presentadas son válidas y se encuentran en su forma normal. Se tienen tanto claves que son satisfechas por el documento, como claves que no lo son. Sólo para este caso, y estos experimentos, se realiza la distinción de los nodos atributo, agregándoles el prefijo "@".

Se realiza la validación de los distintos documentos, primero contra cada una de las claves individualmente. Luego, contra el conjunto completo de claves.

Las Tablas 7.4-7.9, muestran el resultado y los tiempo de la validación de cada uno de los documentos contra las respectivas claves. Se distingue entre el tiempo utilizado para la construcción de la estructura DOM, utilizada como representación del documento en memoria, y el tiempo utilizado para decidir la satisfacción de una clave sobre dicha estructura. La estructura se logra a partir del proceso de análisis o *parsing* del documento XML, y es cargada completamente en memoria.

Al realizar la validación de los dos documentos de *Auction Data*, que poseen características similares, se obtienen tiempos similares. El tiempo de construcción de la estructura para el documento yahoo.xml, en el caso del experimento con la clave 1.3, es 14 veces el tiempo que toma decidir la satisfacción de la clave (ver Tabla 7.5). Mientras en la Tabla 7.4 para 321gone.xml, es casi 12 veces. En promedio, se tiene que este proceso en estos dos documentos, toma 9 veces el tiempo de decidir si una clave es satisfecha. De aquí la relevancia de realizar una separación entre los tiempos.

TABLA 7.3: *Claves utilizadas en la validación de documentos XML.*

Id	Clave
1.1	$(\varepsilon, (listing.seller\_info, \{seller\_name.S, seller\_rating.S\}))$
1.2	$(\varepsilon, (listing.auction\_info, \{current\_bid.S\}))$
1.3	$(listing, (auction\_info.high\_bidder, \{bidder\_name.S, bidder\_rating.S\}))$
1.4	$(listing, (item\_info, \{memory.S, hard\_drive.S, cpu.S, brand.S\}))$
1.5	$(\varepsilon, (listing.seller\_info, \{seller\_rating.S\}))$
1.6	$(\varepsilon, (listing.auction\_info.high\_bidder, \{bidder\_rating.S\}))$
1.7	$(listing, (auction\_info.num\_items, \{S\}))$
2.1	$(\varepsilon, (\_*.book, \{isbn\}))$
2.2	$(\varepsilon, (\_*.article.author, \{first.S, last.S\}))$
2.3	$(\_*.article, (author, \{first.S, last.S\}))$
2.4	$(\_*.article, (editor, \{name.S\}))$
2.5	$(\_*.book, (chapter, \{name.S\}))$
2.6	$(\varepsilon, (\_^*.article, \{title\}))$
2.7	$(\varepsilon, (\_^*.inproceedings, \{title\}))$
2.8	$(\varepsilon, (\_^*.phdthesis, \{title\}))$
2.9	$(\varepsilon, (\_^*.mastersthesis, \{title\}))$
2.10	$(\varepsilon, (\_^*.incollection, \{title\}))$
2.11	$(\varepsilon, (\_^*.proceedings, \{title\}))$
2.12	$(\varepsilon, (\_^*.book, \{title\}))$
2.13	$(\varepsilon, (\_^*.www, \{title\}))$
2.14	$(\varepsilon, (\_^*.book, \{publisher\}))$
3.1	$(\varepsilon, (dataset, \{@subject\}))$
3.2	$(\varepsilon, (dataset.reference.source.other.date, \{year.S\}))$
3.3	$(dataset, (reference.source.journal.author, \{lastName.S\}))$
3.4	$(\varepsilon, (dataset.reference.source.journal, \{title.S\}))$
3.5	$(dataset, (reference.source.journal, \{title.S\}))$
3.6	$(dataset.reference, (source.journal, \{title.S\}))$
3.7	$(dataset.reference.source, (journal, \{title.S\}))$
3.8	$(dataset.reference.source, (\_^*, \{title.S\}))$
3.9	$(dataset.\_^*.source, (journal, \{title.S\}))$
4.1	$(\varepsilon, (issue, \{volume.S, number.S\}))$
4.2	$(issue, (articles.article, \{title.S\}))$
4.3	$(issue.articles, (article, \{title.S\}))$
4.4	$(issue, (articles.article, \{initPage.S, endPage.S\}))$
4.5	$(\varepsilon, (issue, \{volume.S\}))$
4.6	$(\varepsilon, (issue.articles.article.authors.author, \{@position\}))$
5.1	$(\varepsilon, (country, \{@goverment\}))$
5.2	$(country, (city, \{name.S\}))$
5.3	$(country, (city, \{@country\}))$
5.4	$(\varepsilon, (country, \{language.S\}))$
5.5	$(country, (religious, \{@percentage, S\}))$
5.6	$(country, (ethnicgroups, \{@percentage, S\}))$
5.7	$(\varepsilon, (country.city, \{name.S\}))$
5.8	$(\varepsilon, (country.city, \{population.S\}))$
5.9	$(\varepsilon, (country, \{encompased.@continent\}))$
5.10	$(\varepsilon, (country.province.city, \{name.S\}))$
5.11	$(\varepsilon, (country.province, \{city\}))$
5.12	$(country, (religious, \{S\}))$
5.13	$(country, (province.city, \{name.S\}))$

Se puede observar que generalmente los casos en que las validaciones toman mayor tiempo, se dan cuando las claves son absolutas (1.1, 1.2, 1.5, 1.6), ya que la validación abarca todo el árbol, y comienza desde el nodo raíz. Las claves 1.3 y 1.4, que son relativas, son además, las de mayor tamaño $|\varphi|$, por lo que cada una de sus consultas XPath asociadas debiese retornar conjuntos reducidos de elementos (debido a que la validación comienza en subárboles del documento); por lo tanto, el tiempo de validación contra estas claves debiese ser menor que en los otros casos, cosa que se cumple en ambos documentos.

TABLA 7.4: *Tiempos de validación del documento 321gone.xml.*

IdClave	$T_D \models \Sigma$	Tiempo DOM	Tiempo Val.
1.1	false	0s 719ms	0s 248ms
1.2	true	0s 712ms	0s 283ms
1.3	true	0s 709ms	0s 60ms
1.4	true	0s 712ms	0s 59ms
1.5	false	0s 717ms	0s 61ms
1.6	false	0s 709ms	0s 72ms
1.7	true	0s 718ms	0s 60ms
Validación Conjunto:		0s 716ms	0s 379ms
Suma Tiempo Val.:			0s 843ms

TABLA 7.5: *Tiempos de validación del documento yahoo.xml.*

IdClave	$T_D \models \Sigma$	Tiempo DOM	Tiempo Val.
1.1	true	0s 718ms	0s 414ms
1.2	true	0s 716ms	0s 333ms
1.3	true	0s 720ms	0s 51ms
1.4	true	0s 711ms	0s 64ms
1.5	false	0s 716ms	0s 118ms
1.6	false	0s 713ms	0s 75ms
1.7	true	0s 714ms	0s 63ms
Validación Conjunto:		0s 717ms	0s 546ms
Suma Tiempo Val.:			1s 118ms

Al final de cada tabla se presentan además, los tiempos de validación del documento contra el conjunto de claves en una sola ejecución, y luego la suma de los tiempos de validación individual obtenidos.

En la Tabla 7.6 se mantiene la tendencia de las claves absolutas, las que requieren un

TABLA 7.6: *Tiempos de validación del documento dblp.xml.*

IdClave	$T_D \models \Sigma$	Tiempo DOM	Tiempo Val.
2.1	true	0s 765ms	8s 924ms
2.2	true	0s 764ms	1s 199ms
2.3	true	0s 770ms	0s 748ms
2.4	true	0s 739ms	0s 727ms
2.5	true	0s 734ms	0s 770ms
2.6	true	0s 752ms	0s 915ms
2.7	true	0s 746ms	1s 2ms
2.8	true	0s 762ms	8s 389ms
2.9	true	0s 744ms	0s 998ms
2.10	true	0s 744ms	0s 802ms
2.11	TE	–	–
2.12	true	0s 773ms	9s 119ms
2.13	true	0s 753ms	4s 295ms
2.14	false	0s 762ms	1s 258ms
Validación Conjunto:		0s 762ms	26s 224ms
Suma Tiempo Val.:			39s 146ms

mayor tiempo en la validación del documento. Además, del uso de comodines en todas las claves. La sigla TE utilizada en la clave 2.11, indica que el tiempo de validación excede los 5 minutos. (Las claves que exceden el tiempo de validación, no fueron consideradas en los procesos de validación.) En el caso específico de la clave 2.11, se tiene que la consulta XPath `'/dblp/proceedings/title'` resultante retorna 3.008 elementos título, número excesivo para la sencilla implementación realizada.

TABLA 7.7: *Tiempos de validación del documento nasa.xml.*

IdClave	$T_D \models \Sigma$	Tiempo DOM	Tiempo Val.
3.1	false	2s 183ms	3s 342ms
3.2	false	2s 193ms	41s 335ms
3.3	true	2s 204ms	2s 876ms
3.4	TE	–	–
3.5	true	2s 191ms	2s 847ms
3.6	true	2s 213ms	3s 163ms
3.7	true	2s 218s	3s 185ms
3.8	true	2s 199ms	5s 683ms
3.9	true	2s 191ms	3s 934ms
Validación Conjunto:		2s 220ms	44s 552ms
Suma Tiempo Val.:			1m 6s 365ms

Según los valores mostrados en la Tabla 7.7, el tiempo de validación excede los 5 minutos para la clave 3.4. Se tiene en este caso que la consulta XPath correspondiente `'/datasets/dataset/reference/source/journal/title/text()'` retorna 3.317 elementos.

TABLA 7.8: *Tiempos de validación del documento SigmodRecord.xml.*

IdClave	$T_D \models \Sigma$	Tiempo DOM	Tiempo Val.
4.1	true	1s 459ms	50s 361ms
4.2	true	1s 492ms	0s 600ms
4.3	true	1s 443ms	0s 456ms
4.4	true	1s 412ms	0s 698ms
4.5	false	1s 428ms	0s 687ms
4.6	false	1s 410ms	0s 719ms
Validación Conjunto:		1s 481ms	51s 692ms
Suma Tiempo Val.:			53s 521ms

En la Tabla 7.8, la clave 4.1 presenta el mayor tiempo de validación, debido a que esta consulta, al ser absoluta, abarca todo el documento, e involucra a todos los nodos *issue* (67 nodos), que contienen a casi todos los nodos del árbol. Además, posee dos caminos clave, lo que eleva el tiempo en la determinación de la igualdad en valor de los nodos.

TABLA 7.9: *Tiempos de validación del documento mondial-3.0.xml.*

IdClave	$T_D \models \Sigma$	Tiempo DOM	Tiempo Val.
5.1	false	1s 14ms	1s 45ms
5.2	true	1s 26ms	0s 916ms
5.3	false	1s 43ms	0s 877ms
5.4	false	1s 55ms	9s 588ms
5.5	true	1s 18ms	0s 763ms
5.6	true	1s 43ms	0s 811ms
5.7	TE	–	–
5.8	false	1s 55ms	8s 914ms
5.9	false	1s 50ms	0s 971ms
5.10	TE	–	–
5.11	TE	–	–
5.12	true	1s 48ms	0s 863ms
5.13	true	1s 36ms	0s 818ms
Validación Conjunto:		1s 31ms	18s 171ms
Suma Tiempo Val.:			25s 566ms

La Tabla 7.9 muestra que las claves 5.7, 5.10, y 5.11, exceden los 5 minutos como tiempo

TABLA 7.10: *Comparación de los tiempos de validación de documentos.*

	Validación Individual			Validación Conjunta		
Documento	Σt_{k_i}	$\overline{t}_{DOM}$	(+)[ms]	t_K	t_{DOM}	(+)[ms]
321gone.xml	843	714	1.557	379	716	1.095
yahoo.xml	1.118	715	1.833	546	717	1.263
dblp.xml	39.146	754	39.900	26.224	762	26.986
nasa.xml	66.365	2.199	68.564	44.552	2.220	46.772
SigmodRecord.xml	53.521	1.441	54.962	51.692	1.481	53.173
mondial-3.0.xml	25.566	1.040	26.606	18.171	1.031	19.202

de validación. Las tres claves corresponden a claves absolutas. Según la semántica definida por las claves (la restricción), las tres claves son satisfechas por el documento. No pueden existir dos ciudades de países con el mismo nombre, o ciudades de provincia con el mismo nombre, y dos subárboles de provincias no pueden tener igualdad en valor en sus nodos ciudad (*city*). Esto indica, que aquellas claves para las cuales el documentos XML es válido, requieren de mayor tiempo en la validación.

Como se vio en las tablas anteriores, la mayor parte del tiempo de validación de un documento XML contra una clave (o conjunto de claves), depende del proceso de *parsing* y construcción de la estructura, propio de la biblioteca utilizada para la implementación. Para profundizar y resaltar este hecho, se presenta la Tabla 7.10.

Si se considera que la validación de un documento contra un conjunto de claves XML Σ, debiese consistir de la aplicación múltiples veces del Algoritmo 5.3 para cada $\sigma \in \Sigma$. Se podría inferir que el tiempo de la validación de D contra el conjunto Σ, debiese tomar el mismo tiempo (o similar) que la validación de D contra cada una de las claves por separado. En ninguno de los casos se cumple esto. Luego de realizar un análisis con mayor detención, se puede observar que: cada vez que se analiza una clave, existe una cota mínima de tiempo necesaria para realizar la lectura del archivo XML (*parsing*) y crear la estructura DOM, que corresponde al árbol XML T con el cual se trabaja.

Sea $n = |\Sigma|$, α el tiempo gastado en realizar el *parsing* y construcción de la estructura, y t_{k_i} el tiempo que demora decidir la satisfacción de la clave k_i para todo $1 \leq i \leq n$. El tiempo total de validación de un documento XML contra un conjunto de claves debiese seguir la siguiente ecuación.

$$\alpha + t_{k_1} + t_{k_2} + \cdots + t_{k_n} = \alpha + \sum_{i=1}^{n} t_{k_i}$$

TABLA 7.11: *Correlación tamaño documento-construcción árbol.*

Documento	Tamaño	Tiempo $\overline{x}$ Construcción[ms]
321gone.xml	23 KB	713,7
yahoo.xml	24 KB	715,4
SigmodRecord.xml	476 KB	1.440,7
mondial-3.0.xml	1 MB	1.038,8
dblp.xml	1.6 MB	754,5
nasa.xml	23 MB	2.199,0

En la Tabla 7.10, se realiza la distinción entre validación individual, y validación conjunta. Como se dijo antes, el tiempo de validación conjunta debiese ser similar a la suma de los tiempos de la validación individual. Esta tabla muestra que esto no se cumple. Si bien, el tiempo requerido por el proceso de *parsing* y construcción de la estructura, es regular para cada una de las ejecuciones de la validación de documentos, este proceso se lleva gran parte del tiempo total. En el mejor caso corresponde al 3 % del tiempo total, mientras, en el peor al 94 %. La implementación realizada es sencilla, utiliza la biblioteca `javax.xml` provista por Java para trabajar con documentos XML. No es objetivo de este trabajo realizar o buscar interfaces de trabajo eficiente con documentos XML. Pero sí, se muestra la limitación de la implementación desarrollada en este punto, y la importancia de mejorar esto en futuras implementaciones.

Sí se resalta el hecho que el tiempo de *parsing* y construcción de la estructura, posee una alta correlación con las características del documento, que determinan la complejidad: su tamaño, su profundidad máxima y promedio, la cantidad de líneas, etc. La Tabla 7.11 muestra los documentos ordenados por tamaño, con sus respectivos tiempos de construcción de la estructura. Si el documento es de tamaño reducido, el proceso de *parsing* y construcción de la estructura, es relativamente rápido. El caso del documento dblp.xml, que a pesar de ser uno de los más grandes en tamaño, posee un tiempo de construcción reducido. Esto puede ser explicado recordando el valor de su profundidad promedio, que es el menor dentro de todos los documentos. A menor profundidad, se puede tener una estructura de árbol más ancha, pero no tan alta.

7.2.3 Conclusiones

Se ha mostrado que la validación de documentos XML contra claves, es prácticamente eficiente. Pero, aún se requiere un análisis más profundo respecto a las bibliotecas utilizadas para el tratamiento de documentos XML en la implementación, tarea ingenieril que permitiría hacer un uso más eficiente de memoria, y obtener resultados más rápidos a consultas XPath.

Se nota en los experimentos, que la validación de documentos XML contra claves absolutas es más lenta que contra claves relativas. Esto debido al ámbito de acción de cada una de las claves, es decir, el contexto sobre el cual actúan.

Es notoria la proporción significativa del tiempo total, necesaria para realizar el *parsing* del documento, y la construcción de la estructura, necesaria para realizar las consultas XPath. Al no ser un objetivo de este trabajo, se deja abierto el análisis y desarrollo de bibliotecas que permitan un trabajo eficiente sobre documentos XML.

7.3 EXPERIMENTOS CON *COVERS* NO REDUNDANTES

Debido a la relevancia que tiene el tamaño del conjunto Σ en el tiempo del proceso de validación de documentos XML contra claves, se presentó un algoritmo que permite obtener un conjunto Σ', denominado *cover* no redundante. Este conjunto, posee igual o menor cantidad de elementos, $|\Sigma'| \leq |\Sigma|$, manteniéndose la igualdad de sus clausuras semánticas, $(\Sigma')^* = (\Sigma)^*$.

Tomando los documentos XML y los conjuntos de claves utilizados en la Sección 7.2, se calcularán los respectivos *covers* no redundantes, y se determinará la disminución de tiempos que se obtienen en la validación de documentos.

La Tabla 7.12 muestra los *covers* no redundantes obtenidos de cada uno de los conjuntos de claves vistos, y el tiempo utilizado. En el primer caso, para el conjunto de claves para *auction data*, se obtiene un *cover* no redundante que es dos claves menor que el conjunto original. Del conjunto de claves para *DBLP*, no se extrae ninguna clave, siendo el *cover* no redundante igual al conjunto inicial. El mejor caso se obtiene con los conjuntos de claves para *SIGMOD Record* y *Mondial*, de los cuales se extraen 4 claves, quedando un *cover* no redundante con sólo 2 y 10 claves, respectivamente.

TABLA 7.12: *Covers no redundantes para claves XML.*

Documento	*Cover* no redundante	Tiempo[ms]	$\|\Sigma\|$	$\|\Sigma'\|$
Auction Data	{1.2, 1.4, 1.5, 1.6, 1.7}	0,64	7	5
DBLP	Mismo conjunto	3,3	14	14
Nasa	{3.1, 3.2, 3.3, 3.4, 3.9}	0,901	9	5
SIGMOD Record	{4.2, 4.4, 4.5, 4.6}	0,574	6	2
Mondial	{5.1, 5.3, 5.4, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12}	1,361	14	10

TABLA 7.13: *Validación de documentos XML contra covers no redundantes.*

	Validación Σ			Validación Σ'		
Documento	t_Σ	$\bar{t}_{DOM}$	(+)[ms]	$t_{\Sigma'}$	t_{DOM}	(+)[ms]
(Doc1) 321gone.xml	379	716	1.095	326	719	1.045
(Doc2) yahoo.xml	546	717	1.263	376	722	1.098
(Doc3) dblp.xml	26.224	762	26.986	26.250	749	26.999
(Doc4) nasa.xml	44.552	2.220	46.772	41.658	2.165	43.823
(Doc5) SigmodRecord.xml	51.692	1.481	53.173	952	1.470	2.422
(Doc6) mondial-3.0.xml	18.171	1.031	19.202	17.038	1.028	18.066

Ya que en la mayoría de los casos, se obtuvieron *covers* no redundantes de menor tamaño, se realiza nuevamente la validación de los documentos, ahora contra estos nuevos conjuntos. La Tabla 7.13 muestra una comparación, entre los tiempos de validación de documentos obtenidos para los conjuntos originales de claves, y sus respectivos *covers* no redundantes obtenidos.

Como lo muestra la Tabla 7.13 y la Figura 7.7, se obtienen disminuciones de tiempo en todos los casos en que $|\Sigma'| < |\Sigma|$. Mientras más claves son descartadas, mayor es la disminución de tiempo obtenida. También, influye en la magnitud del tiempo disminuido, el tipo de la clave descartada. Si ésta es absoluta, o posee comodines, el tiempo disminuido será mayor. Este caso se da en el conjunto de claves para *SIGMOD Record*. De este conjunto, se elimina la clave 4.1 (ya que es implicada por la clave 4.5), que corresponde a una clave absoluta, y para la cual, la validación del documento SigmodRecord.xml tomaba 50s 361ms. En consecuencia, el tiempo de validación del documento contra el *cover* no redundante es 50s 751ms menor.

La Figura 7.8 muestra una gráfica con los tiempos utilizados en obtener un *cover* no redundante, para un conjunto heterogéneo de 150 claves XML, obtenidas de las referencias utilizadas, y de la Sección 7.2. En el eje x de la gráfica, se tiene el tamaño del conjunto de claves, utilizando un incremento diferencial, con $\Delta = 5$ claves.

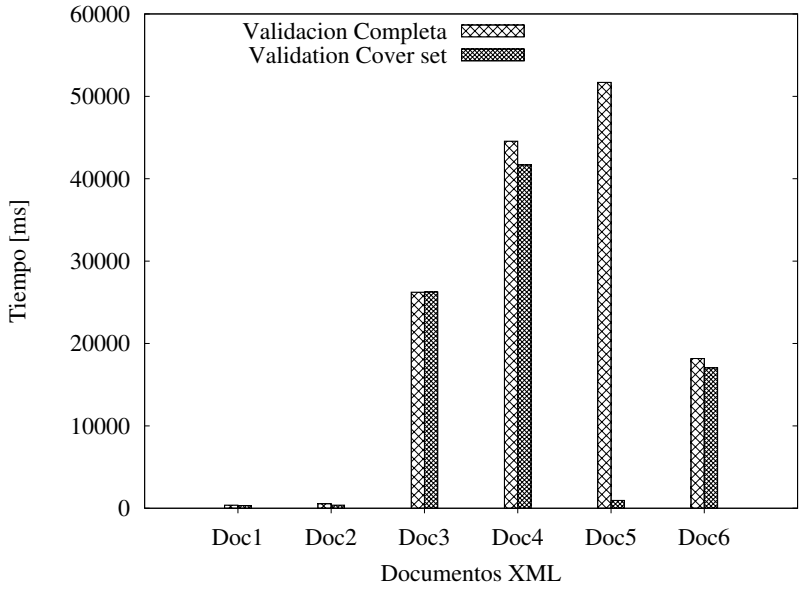


FIGURA 7.7: *Tiempos de validación de documentos contra covers no redundantes.*

También en este caso, se tiene una tendencia casi lineal para los tiempos de obtención de *covers* no redundantes. Del conjunto de 150 claves utilizado, el *cover* no redundante resultante posee sólo 82, descartándose 68 claves, y fue obtenido en 103,072 milisegundos.

7.3.1 Conclusiones

Se ha demostrado con experimentos, que el obtener un *cover* no redundante de un conjunto de claves XML en $\mathcal{K}^{PL}_{PL,PL_s^+}$, requiere de un tiempo reducido, del orden de los milisegundos. Para un conjunto de 150 claves, el *cover* no redundante se obtuvo en 103,072 milisegundos. Además, se concluye que precalcular un conjunto *cover* no redundante de claves, y realiza la validación de un documento XML contra este conjunto, disminuye el tiempo de dicha validación. La magnitud de la disminución, depende de las características de las claves descartadas.

Se obtuvo que el tiempo de obtención de un conjunto *cover* no redundante, se ve afectado por las mismas variables aplicables al proceso de implicación de claves XML, es decir, a variables como: el tipo de claves –absolutas o relativas–, la presencia de comodines en las expresiones de camino que componen la clave, la cantidad de caminos claves, y el tamaño de la clave.

Además, se puede ver que el tiempo que se requiere para calcular un *cover* no redundante, es sólo una pequeña parte del tiempo total de validación de un documento XML una única clave XML.

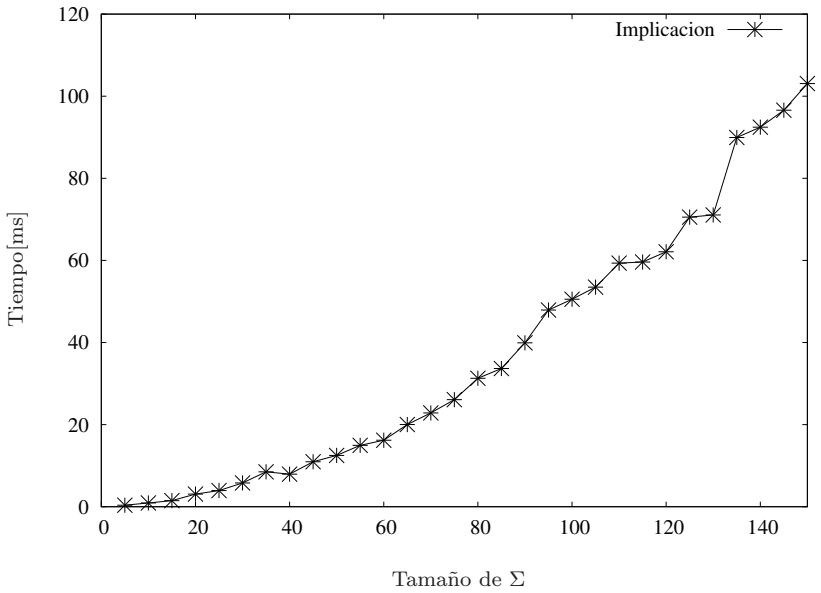


FIGURA 7.8: *Tiempo obtención covers no redundantes.*

Se enfatiza, que el comportamiento del algoritmo presentado para calcular *covers* no redundantes, es lineal en la práctica.

Finalmente, a raíz de los resultados obtenidos es altamente recomendable precalcular el conjunto *cover* no redundante al momento de realizar la validación de un documento XML contra un conjunto de claves XML. Debido a la gran disminución de tiempo que se puede alcanzar en la práctica (ver Figura 7.7).

CAPÍTULO 8. CONCLUSIONES Y TRABAJOS FUTUROS

Los resultados de este trabajo de tesis tienen la siguiente aplicación. Dado un conjunto de documentos en formato XML tratados como una base de datos, interesa saber si dichos documentos satisfacen un conjunto de claves relevantes para la aplicación objetivo. La tesis propone un algoritmo para realizar este proceso, y la evaluación experimental, considerando entradas significativas, muestra tiempos de ejecución del orden de la fracción de segundos por cada clave.

Un aspecto clave en la eficiencia del algoritmo de validación de documentos XML propuesto, lo cual es parte de la innovación al estado del arte introducida por el algoritmo, proviene del uso del llamado conjunto *cover* no redundante para determinar un conjunto reducido de claves. Para este propósito fue necesario utilizar el algoritmo de implicación de claves XML propuesto por Hartmann & Link (2009a). Para el problema de implicación, el último trabajo publicado en la literatura, propone un algoritmo teórico de complejidad cuadrática Hartmann & Link (2009a). En esta línea, parte relevante de la contribución de este trabajo de tesis consistió en diseñar y evaluar una implementación práctica de este algoritmo para el problema de implicación de claves XML (Hartmann & Link, 2009a). Esto requirió del diseño de estructuras de datos y algoritmos de apoyo para hacer que el algoritmo de implicación de claves XML sea eficiente en tiempo de ejecución. Los resultados experimentales, basados en un conjunto completo de casos para claves, muestran que en la práctica el caso cuadrático tiene una probabilidad muy pequeña de ocurrencia y que los tiempos de ejecución resultan ser muy bajos. En la literatura del área no se conocen trabajos de ingeniería de algoritmos como el realizado para este algoritmo de implicación de claves XML.

El aporte de este trabajo, se puede resumir en los siguientes 3 puntos: (1) una implementación eficiente de un algoritmo de implicación sobre un fragmento con capacidades semánticas deseables, propuesto en (Hartmann & Link, 2009a), (2) el diseño e implementación de un algoritmo para la validación de documentos XML contra claves XML, y (3) el diseño

e implementación de un algoritmo para calcular *covers* no redundantes. En particular, la implementación de (3) está basada en la implementación formulada para (1), y (3) es usado para reducir significativamente el tiempo total de ejecución de (2). A continuación, se presenta una discusión técnica de estos aportes.

8.1 DISCUSIÓN TÉCNICA

El objetivo de este trabajo fue diseñar, implementar y evaluar experimentalmente métodos que hagan factible la explotación de las capacidades semánticas que entregan las claves presentes en colecciones de documentos XML. Para esto se analizaron y diseñaron algoritmos con los cuales se pudo construir una implementación para decidir el problema de implicación de claves XML, y otra para decidir el problema de validación de documentos contra claves XML. En el análisis de los algoritmos, se pudo ver que a diferencia del modelo relacional cuya estructura está dada por relaciones, en XML el complejo anidamiento de los datos en una estructura tipo árbol, hace que el trabajo con restricciones como claves XML sea todo un desafío.

Se ha demostrado a través de experimentos con las implementaciones desarrolladas en este trabajo, que es posible razonar de forma eficiente sobre claves XML en el fragmento $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$. El problema de implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$ puede generalmente ser decidido en un tiempo del orden de la fracción de segundos, dándose un comportamiento inferior al cuadrático en la mayoría de los casos. En este problema, se ha notado que el tamaño de la clave no es lo más influyente en los tiempos de decisión; sí lo es la existencia de comodines en las expresiones de camino, y la aplicabilidad de las claves, lo que produce un incremento considerable en los tiempos de decisión, los cuales en la mayoría de los experimentos se mantuvieron en un orden inferior al milisegundo.

Revisando los trabajos que desarrollan el problema de validación de documentos XML contra claves, se ha observado que ninguno de ellos respeta el concepto de igualdad en valor definido por Buneman et al. (2003, 2002), y hacen uso de otros modelos de datos. Por lo cual, se ha desarrollado un validador de documentos XML asumiendo que la noción de igualdad en valor no está restringida a la igualdad en texto, permitiéndose caminos clave que retornan conjuntos de nodos elemento, y consecuentemente alcanzando una mayor expresividad. Además, utilizando el

algoritmo de implicación de claves XML y la noción de *covers* no redundantes de claves XML, se logró optimizar los tiempos de validación de documentos XML, realizando ésta contra conjuntos reducidos de claves.

Se observa la importancia del proceso de *parsing* de documentos XML en la validación contra claves, ya que éste puede alcanzar el 94 % del tiempo total utilizado en la validación de documentos XML contra una clave. Por lo tanto, a futuro se requiere profundizar el análisis de los métodos y bibliotecas existentes para realizar el *parsing* de documentos XML de manera eficiente en el uso de memoria.

Finalmente, se ha dado cumplimiento a los principales objetivos propuestos al comienzo de este trabajo, desarrollando un trabajo teórico-práctico en base al uso de claves como restricciones para documentos XML, trabajando un fragmento sobre el cual es posible razonar de forma eficiente en la práctica. Como resultado, se tienen nuevos métodos e implementaciones que permiten avanzar a futuro en aplicaciones XML tales como, validación de consistencia, diseño de esquemas, integración de datos, intercambio y limpieza de datos, optimización y reescritura de consultas, indexación, y respuesta consistente a consultas (Chomicki, 2007; Davidson et al., 2007; Fan, 2005).

8.2 TRABAJOS FUTUROS

Para futuros trabajos, las implementaciones desarrolladas pueden ser extendidas para soportar nuevas clases de claves XML que entreguen una mayor expresividad. Cuando se definen claves XML existen al menos dos factores que determinan su expresividad: (i) los operadores del lenguaje navegacional para acceder a los nodos (es decir, expresiones de camino para descendientes/ancestros) (Benedikt et al., 2005; Deutsch & Tannen, 2005; Miklau & Suciu, 2004; Neven & Schwentick, 2006; Wood, 2002), y (ii) la noción de igualdad en valor (es decir, igualdad en texto de los nodos hoja, árboles isomorfos, etc.) (Buneman et al., 2003, 2002; Davidson et al., 2008, 2007). Para reutilizar las implementaciones desarrollada en este trabajo, se requiere que se mantenga la caracterización del problema de implicación de claves XML como un problema de alcanzabilidad de nodos en un grafo. Por ejemplo en Hartmann & Link (2009b), se analiza la clase $\mathcal{K}^{PL}_{PL,PL^*_s}$, donde el conjunto de caminos clave simples puede ser vacío (denotado por símbolo "*"). Esta clase es

más expresiva y contiene a la trabajada aquí, pero no se ha podido caracterizar su problema de implicación como un problema de alcanzabilidad en un grafo (Hartmann & Link, 2009b). La clave (g) del Ejemplo 3.1 pertenece a esta clase. Nótese, que un árbol XML T satisface una clave llamada estructural $(Q, (Q', \emptyset))$ si y sólo si, para todo nodo $q \in [\![Q]\!]$ existe a lo más un nodo alcanzable desde q siguiendo un camino-Q', esto es, el conjunto $q[\![Q']\!]$ contiene a lo más un elemento. Por ejemplo, la clave $(proyecto, (encargado, \emptyset))$ pertenece a esta clase $\mathcal{K}^{PL}_{PL,PL^*_s}$, y expresa que para un proyecto puede haber a lo más un encargado.

Ferrarotti et al. (2010) sigue la estrategia utilizada en Hartmann & Link (2009a) caracterizando el problema de implicación como un problema de alcanzabilidad de nodos en un grafo, y realiza el estudio de la clase $\mathcal{K}(PL^{\{.,\_,\_^*\}}, PL^{\{.,\_,\_^*\}}, PL^{\{.,\_\}}_+)$ de claves XML, en la cual se extienden los lenguaje PL y PL_s permitiendo comodines simples ("$\_$") que pueden ser reemplazados por una palabra del alfabeto, en el camino de contexto, objetivo y caminos clave, además del operador binario de concatenación ".", y el comodín de largo variable "$\_^*$" (que pueden ser reemplazados por ninguna, una o varias palabras del alfabeto). Por ejemplo, a excepción de la clave (g) del Ejemplo 3.1, todas las otras claves definidas en dicho ejemplo pertenecen a esta clase. Esta clase de claves XML claramente es más expresiva que la clase estudiada en Hartmann & Link (2009a, 2007), ya que incluye estrictamente todas las claves XML que pertenecen a $\mathcal{K}^{PL}_{PL,PL^+_s}$ y, al mismo tiempo, incluye claves como por ejemplo $(\_^*.proyecto.equipo, (\_, \{idpersona.S\}))$, y $(\varepsilon, (\_, \{\_\}))$, para las que no hay claves XML equivalentes en $\mathcal{K}^{PL}_{PL,PL^+_s}$. Se puede ver que esta nueva clase es ortogonal a la clase $\mathcal{K}^{PL}_{PL,PL^*_s}$ estudiada en Hartmann & Link (2009b). Si se toma por ejemplo la clave XML $(\varepsilon, (\_, \{\_\}))$ que especifica que no podrán existir dos nodos hijos de la raíz diferentes, para los cuales la intersección en valor de su correspondiente conjunto de nodos hijos sea vacía; es fácil ver que no hay una clave XML en $\mathcal{K}^{PL}_{PL,PL^*_s}$, la cual sea satisfecha por exactamente aquellos árboles XML que satisfacen $(\varepsilon, (\_, \{\_\}))$. Así, el conjunto de reglas de inferencia usado en Hartmann & Link (2009b) para la axiomatización finita de la clase $\mathcal{K}^{PL}_{PL,PL^*_s}$ es considerablemente diferente del conjunto de reglas de inferencia propuesto en Ferrarotti et al. (2010) para la clase $\mathcal{K}(PL^{\{.,\_,\_^*\}}, PL^{\{.,\_,\_^*\}}, PL^{\{.,\_\}}_+)$. A pesar que la clase de claves XML presentada en Ferrarotti et al. (2010) permite expresar capacidades semánticas no presentes en $\mathcal{K}^{PL}_{PL,PL^+_s}$, el algoritmo propuesto para determinar la implicación de estas claves posee una complejidad temporal $\mathcal{O}(n^4)$ mayor al algoritmo existente para $\mathcal{K}^{PL}_{PL,PL^+_s}$ –a pesar de utilizar la misma estrategia. De todas formas a la vista de los experimentos realizados en este trabajo

con la implementación del algoritmo para el problema de implicación de claves XML en $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$, y la mejor complejidad para la clase $\mathcal{K}^{PL}_{PL,PL^{+}_{s}}$, se conjetura que no se vería afectada la utilidad de la clase $\mathcal{K}(PL^{\{.,\_,-,^{*}\}}, PL^{\{.,\_,-,^{*}\}}, PL^{\{.,-\}}_{+})$ en la práctica.

REFERENCIAS

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of Databases: The Logical Level*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed.

Abrão, M. A., Bouchou, B., Ferrari, M. H., Laurent, D., & Musicante, M. A. (2004). Incremental Constraint Checking for XML Documents. In Z. Bellahsène, T. Milo, M. Rys, D. Suciu, & R. Unland (Eds.) *Database and XML Technologies*, vol. 3186 of *Lecture Notes in Computer Science*, (pp. 358–379). Springer Berlin / Heidelberg. 10.1007/978-3-540-30081-6_9.
URL http://dx.doi.org/10.1007/978-3-540-30081-6_9

Aho, A. V., & Hopcroft, J. E. (1974). *The Design and Analysis of Computer Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed.

Aho, A. V., Ullman, J. D., & Hopcroft, J. E. (1983). *Data structures and algorithms*. Addison-Wesley, Reading, Mass.

Alon, N., Milo, T., Neven, F., Suciu, D., & Vianu, V. (2001). XML with data values: typechecking revisited. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, (pp. 138–149). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/375551.375570

Alon, N., Milo, T., Neven, F., Suciu, D., & Vianu, V. (2003). Typechecking XML views of relational databases. *ACM Trans. Comput. Logic*, *4*, 315–354.
URL http://doi.acm.org/10.1145/772062.772065

Apparao, V., Byrne, S., Champion, M., Isaacs, S., Hors, A. L., Nicol, G., Robie, J., Sharpe, P., Smith, B., Sorensen, J., Sutor, R., Whitmer, R., & Wilson, C. (1998). Document object model (DOM) level 1 specification. http://www.w3.org/TR/REC-DOM-Level-1/. Extraído el 19 de Octubre de 2010.

Arenas, M., Fan, W., & Libkin, L. (2002). What's Hard about XML Schema Constraints? In *Proceedings of the 13th International Conference on Database and Expert Systems Applications*, DEXA '02, (pp. 269–278). London, UK, UK: Springer-Verlag.
URL `http://portal.acm.org/citation.cfm?id=648315.756182`

Arenas, M., & Libkin, L. (2004). A normal form for XML documents. *ACM Trans. Database Syst.*, *29*, 195–232.

Arenas, M., & Libkin, L. (2008). XML data exchange: Consistency and query answering. *J. ACM*, *55*, 7:1–7:72.
URL `http://doi.acm.org/10.1145/1346330.1346332`

Armstrong, W. W. (1974). Dependency Structures of Data Base Relationships. In *IFIP Congress*, (pp. 580–583).

Benedikt, M., Chan, C.-Y., Fan, W., Freire, J., & Rastogi, R. (2003). Capturing both types and constraints in data integration. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, (pp. 277–288). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/872757.872792`

Benedikt, M., Fan, W., & Kuper, G. (2005). Structural properties of XPath fragments. *Theor. Comput. Sci.*, *336*, 3–31.
URL `http://portal.acm.org/citation.cfm?id=1085304.1085306`

Benedikt, M., & Koch, C. (2009). XPath leashed. *ACM Comput. Surv.*, *41*, 3:1–3:54.
URL `http://doi.acm.org/10.1145/1456650.1456653`

Boehm, B. (1986). A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, *11*, 14–24.
URL `http://doi.acm.org/10.1145/12944.12948`

Bouchou, B., Alves, M. H. F., & Musicante, M. A. (2003). Tree Automata to Verify XML Key Constraints. In V. Christophides, & J. Freire (Eds.) *WebDB*, (pp. 37–42).

Bray, T., Paoli, J., Sperberg-Queen, C., Maler, E., & Yergeau, F. (2006). eXtensible Markup Language (XML). `http://www.w3.org/TR/2006/REC-xml-20060816/`. Extraído el 21 de Marzo de 2010.

Buneman, P., Davidson, S. B., Fan, W., Hara, C. S., & Tan, W. C. (2002). Keys for XML. *Computer Networks*, *39*(5), 473–487.

Buneman, P., Davidson, S. B., Fan, W., Hara, C. S., & Tan, W. C. (2003). Reasoning about keys for XML. *Inf. Syst.*, *28*(8), 1037–1063.

Buneman, P., Fan, W., Siméon, J., & Weinstein, S. (2001). Constraints for Semi-structured Data and XML. *SIGMOD Record*, *30*(1), 47–45.

Buneman, P., Fan, W., & Weinstein, S. (2000). Path Constraints in Semistructured Databases. *J. Comput. Syst. Sci.*, *61*(2), 146–193.

Campbell, D. M., & Radford, D. (1991). Tree Isomorphism Algorithms: Speed vs. Clarity. *Mathematics Magazine*, *64*(4), 252–261.

Chen, Y., Davidson, S. B., & Zheng, Y. (2002). XKvalidator: A Constraint Validator For XML. In *Proceedings of the eleventh international conference on Information and knowledge management*, CIKM '02, (pp. 446–452). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/584792.584866`

Chomicki, J. (2007). Consistent Query Answering: Five Easy Pieces. In T. Schwentick, & D. Suciu (Eds.) *ICDT*, vol. 4353 of *Lecture Notes in Computer Science*, (pp. 1–17). Springer.

Clark, J., & DeRose, S. (1999). XML Path Language (XPath). `http://www.w3.org/TR/xpath`. Extraído el 21 de Marzo de 2010.

Davidson, S., Fan, W., & Hara, C. (2007). Propagating XML constraints to relations. *J. Comput. Syst. Sci.*, *73*, 316–361.
URL `http://portal.acm.org/citation.cfm?id=1223810.1223864`

Davidson, S., Fan, W., & Hara, C. (2008). Erratum: Erratum to "propagating xml constraints to relations"[j. comput. system sci. 73 (2007) 316–361]. *J. Comput. Syst. Sci.*, *74*, 404–405.
URL `http://portal.acm.org/citation.cfm?id=1332131.1332249`

Deutsch, A., & Tannen, V. (2005). XML Queries and constraints, containment and reformulation. *Theor. Comput. Sci.*, *336*, 57–87.
URL http://portal.acm.org/citation.cfm?id=1085304.1085308

Fagin, R., & Vardi, M. Y. (1984). The Theory of Data Dependencies - An Overview. In *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, (pp. 1–22). London, UK: Springer-Verlag.
URL http://portal.acm.org/citation.cfm?id=646238.683349

Fan, W. (2005). XML Constraints: Specification, Analysis, and Applications. In *DEXA Workshops*, (pp. 805–809). IEEE Computer Society.

Fan, W., & Libkin, L. (2002). On XML integrity constraints in the presence of DTDs. *J. ACM*, *49*(3), 368–406.

Fan, W., & Siméon, J. (2003). Integrity constraints for XML. *J. Comput. Syst. Sci.*, *66*(1), 254–291.

Ferrarotti, F., Hartmann, S., Link, S., & Wang, J. (2010). Promoting the Semantic Capability of XML Keys. In M. Lee, J. Yu, Z. Bellahsène, & R. Unland (Eds.) *Database and XML Technologies*, vol. 6309 of *Lecture Notes in Computer Science*, (pp. 144–153). Springer Berlin / Heidelberg. 10.1007/978-3-642-15684-7_12.
URL http://dx.doi.org/10.1007/978-3-642-15684-7_12

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co.

Goldfarb, C. F. (1990). *The SGML handbook*. New York, NY, USA: Oxford University Press, Inc.

Gottlob, G., Koch, C., & Pichler, R. (2005). Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, *30*(2), 444–491.

Gupta, A., Sagiv, Y., Ullman, J. D., & Widom, J. (1994). Constraint checking with partial information. In *Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '94, (pp. 45–55). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/182591.182597

Hartmann, S., & Link, S. (2007). Unlocking Keys for XML Trees. In T. Schwentick, & D. Suciu (Eds.) *Database Theory – ICDT 2007*, vol. 4353 of *Lecture Notes in Computer Science*, (pp. 104–118). Springer Berlin / Heidelberg. 10.1007/11965893_8.
URL http://dx.doi.org/10.1007/11965893_8

Hartmann, S., & Link, S. (2009a). Efficient Reasoning about a Robust XML Key Fragment. *ACM Trans. Database Syst.*, *34*(2).

Hartmann, S., & Link, S. (2009b). Expressive, yet tractable XML keys. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, (pp. 357–367). New York, NY, USA: ACM.

Hartmann, S., & Trinh, T. (2006). Axiomatising Functional Dependencies for XML with Frequencies. In J. Dix, & S. J. Hegner (Eds.) *FoIKS*, vol. 3861 of *Lecture Notes in Computer Science*, (pp. 159–178). Springer.

Jungnickel, D. (2007). *Graphs, Networks and Algorithms*. Springer Publishing Company, Incorporated.

Liu, Y., Yang, D., Tang, S., Wang, T., & Gao, J. (2004). Extracting Key Value and Checking Structural Constraints for Validating XML Key Constraints. In Q. Li, G. Wang, & L. Feng (Eds.) *Advances inWeb-Age Information Management*, vol. 3129 of *Lecture Notes in Computer Science*, (pp. 399–408). Springer Berlin / Heidelberg. 10.1007/978-3-540-27772-9_40.
URL http://dx.doi.org/10.1007/978-3-540-27772-9_40

Liu, Y., Yang, D., Tang, S., Wang, T., & Gao, J. (2005). Validating key constraints over XML document using XPath and structure checking. *Future Generation Computer Systems*, *21*(4), 583–595. High-Speed Networks and Services for Data-Intensive Grids: the DataTAG Project.

Maier, D. (1980). Minimum Covers in the Relational Database Model. *J. ACM*, *27*, 664–674.
URL http://doi.acm.org/10.1145/322217.322223

Maier, D. (1983). *The Theory of Relational Databases (Online book)*. http://web.cecs.pdx.edu/~maier/TheoryBook/TRD.html.

May, W. (1999). Information Extraction and Integration with FLORID: The MONDIAL Case Study. Tech. Rep. 131, Universität Freiburg, Institut für Informatik. Available from `http://dbis.informatik.uni-goettingen.de/Mondial`.

Megginson, D. (2004). Simple API for XML (SAX). `http://sax.sourceforge.net/about.html`. Extraído el 18 de Octubre de 2010.

Miklau, G., & Suciu, D. (2004). Containment and equivalence for a fragment of XPath. *J. ACM*, *51*, 2–45.
URL `http://doi.acm.org/10.1145/962446.962448`

Neven, F. (2002). Automata theory for XML researchers. *SIGMOD Rec.*, *31*(3), 39–46.

Neven, F., & Schwentick, T. (2006). On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. In *Logical Methods in Computer Science*, (p. 2006).

Nierman, A., & Jagadish, H. V. (2002). Evaluating Structural Similarity in XML Documents. In *WebDB*, (pp. 61–66).

Qureshi, M. H., & Samadzadeh, M. H. (2005). Determining the Complexity of XML Documents. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, ITCC '05, (pp. 416–421). Washington, DC, USA: IEEE Computer Society.
URL `http://dx.doi.org/10.1109/ITCC.2005.126`

Ramakrishnan, R., & Gehrke, J. (2002). *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 3 ed.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-oriented modeling and design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Schach, S. R. (2007). *Object-Oriented and Classical Software Engineering*. New York, NY, USA: McGraw-Hill, Inc., 7 ed.

Sperberg-McQueen, C. M., & Thompson, H. (2000). XML Schema. `http://www.w3.org/XML/Schema`. Extraído el 19 de Octubre de 2010.

Suciu, D. (2001). On database theory and XML. *SIGMOD Rec.*, *30*(3), 39–45.

Suciu, D. (2002a). The XML typechecking problem. *SIGMOD Rec.*, *31*, 89–96.
URL http://doi.acm.org/10.1145/507338.507360

Suciu, D. (2002b). XML Data Repository, University of Washington. http://www.cs.washington.edu/research/xmldatasets/www/repository.html. Extraído el 10 de Diciembre de 2010.

Thalheim, B. (1991). *Dependencies in Relational Databases*. Teubner.

Thompson, H., Beech, D., Maloney, M., & Mendelsohn, N. (2004). XML Schema Part 1: Structures Second Edition. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028. Extraído el 19 de Octubre de 2010.

Vianu, V. (2003). A Web odyssey: from codd to XML. vol. 32, (pp. 68–77).

Vincent, M. W., Liu, J., & Liu, C. (2004). Strong functional dependencies and their application to normal forms in XML. *ACM Trans. Database Syst.*, *29*(3), 445–462.

Widom, J. (1999). Data Management for XML: Research Directions. *IEEE Data Engineering Bulletin*, *22*, 44–52.

Wood, P. T. (2002). Containment for XPath Fragments under DTD Constraints. In *Proceedings of the 9th International Conference on Database Theory*, ICDT '03, (pp. 300–314). London, UK: Springer-Verlag.
URL http://portal.acm.org/citation.cfm?id=645505.656434

APÉNDICE A. INSTANCIAS DE PRUEBA IMPLICACIÓN DE CLAVES XML

TABLA A.1: *Archivo prueba implicación - Σ absoluto y φ absoluta.*

Claves
epsilon;agenda.contacto;nombre.S
epsilon;agenda.contacto;nombre.S,telefono.S
epsilon;_*.empleado;empId
epsilon;_*.empleado;nombre.S,telefono.S
epsilon;_*.gerente;nombre.S,oficina.S,telefono.S
epsilon;_*.person;name.S,last.S,address.S
epsilon;_*.persona;nombre.S,oficina.S
epsilon;agenda._*;nombre.S
epsilon;banco._*.oficinista;nombre.S,cargo.S
epsilon;banco._*.oficinista;nombre.S,oficina.S
epsilon;banco._*.oficinista;oficina.S,cargo.S
epsilon;banco.departamento.oficinista;nombre.S,cargo.S
epsilon;banco.departamento.oficinista;nombre.S,oficina.S
epsilon;banco.departamento.oficinista;oficina.S,cargo.S
epsilon;banco;nombre.S
epsilon;biblioteca.estante.repisa.libro;nombre.S
epsilon;carpeta.archivo;nombre.S,tamano.S
epsilon;computador.usuario._*.archivo;nombre.S,tamano.S
epsilon;country.city;population.S
epsilon;country;encompassed.continent
epsilon;country;government
epsilon;country;languages.S
epsilon;curso.alumno;nombre.S,apellido.S
epsilon;curso.alumno;rut.S,matricula.S
epsilon;dataset.reference.source.other.date;year.S
epsilon;dataset;subject
epsilon;disco._*.archivo;nombre.S,tamano.S,fecha.S
epsilon;disco.carpeta.archivo;nombre.S,tamano.S,fecha.S
epsilon;employee.person;name.S,last.S
epsilon;employee;id
epsilon;employee;person.S
epsilon;empresa._*.gerente;nombre.S,oficina.S,telefono.S
epsilon;empresa.departamento.gerente;nombre.S,oficina.S,telefono.S
epsilon;empresa.gerente;departamento.S
epsilon;empresa.gerente;nombre.S,apellido.S
epsilon;issue.articles.article.authors.author;position
epsilon;issue;volume.S
epsilon;issue;volume.S,number.S

Continúa en la página siguiente...

TABLA A.1 – Continuación

Claves
epsilon;libro.autor;rut.S,fecha_nacimiento.S
epsilon;libro.autor;rut.S,nombre.S,apellido.S
epsilon;libro.capitulo;nombre.S,numero.S
epsilon;libro.seccion;titulo.S,ano.S,editorial.S
epsilon;oficinista;nombre.S,oficina.S
epsilon;parts.color;S
epsilon;parts;color.S,partId.S
epsilon;politicPos.college;year.S
epsilon;_*.college;year.S
epsilon;politicPos;title.S
epsilon;proyecto._*.persona;idpersona.S,nombre.S
epsilon;proyecto._*.persona;nombre.S,oficina.S
epsilon;proyecto._*;S,institucion
epsilon;proyecto._*;persona
epsilon;proyecto.equipo.persona;idpersona.S,nombre.S
epsilon;proyecto.equipo.persona;idpersona.S,nombre.S,apellido.S
epsilon;proyecto.jefe.persona._*;S
epsilon;proyecto.jefe.persona;nombre.S,oficina.S
epsilon;proyecto.jefe;idpersona.S
epsilon;proyecto.jefe;nombre.S,apellido.S
epsilon;proyecto.jefe;persona
epsilon;proyecto.jefe;persona.nombre.S,persona.oficina.S
epsilon;proyecto;codp
epsilon;proyecto;jefe
epsilon;proyecto;nombre.S,codp
epsilon;restaurant._*.wine;name.S,year.S,price.S
epsilon;restaurant.menu.ditches;name.S
epsilon;restaurant.menu.meals.wine;name.S,year.S,price.S
epsilon;restaurant;name.S
epsilon;restaurant;name.S,address.S
epsilon;_*.wine;name.S,year.S,price.S
epsilon;student;name.S,grade.S
epsilon;universidad.empleado;@empId
epsilon;universidad;nombre.S
epsilon;user._*.file;name.S,size.S
epsilon;user._*.file;name.S,size.S,date.S
epsilon;user._*;file.name.S,file.size.S
epsilon;user.directory.file;name.S,size.S
epsilon;user.directory.file;name.S,size.S,date.S
epsilon;user.directory.file;name.S,size.S,fecha.S
epsilon;user.directory;file.name.S,file.size.S
epsilon;library.paper;title.S,year.S
epsilon;mochila.cuaderno;marca.S,titulo.S
epsilon;a.b;c
epsilon;a;b,c
epsilon;a.c;b
epsilon;b.a;c
epsilon;b;a,c
epsilon;b.c;a
epsilon;_*;a
epsilon;_*;a,b
epsilon;_*;a,b,c
epsilon;a._*;b,c

Continúa en la página siguiente...

TABLA A.1 – Continuación

Claves
epsilon;a._*.b;c
epsilon;_*.aa.bb;b.c,d.e
epsilon;aa.bb;b,c.d.e
epsilon;aa._*.b;bb.c,dd.e
epsilon;_*;a.b,c.d
epsilon;aa;b,c
epsilon;bb;d,e
epsilon;cc;d,e
epsilon;dd.ee;e,f
epsilon;oo.p.o.r;s,p.d

TABLA A.2: *Archivo prueba implicación - Σ absoluto y φ relativa.*

Claves
banco;_*.oficinista;nombre.S,oficina.S
epsilon;agenda.contacto;nombre.S
epsilon;agenda.contacto;nombre.S,telefono.S
epsilon;_*.empleado;empId
epsilon;_*.empleado;nombre.S,telefono.S
epsilon;_*.gerente;nombre.S,oficina.S,telefono.S
epsilon;_*.person;name.S,last.S,address.S
epsilon;_*.persona;nombre.S,oficina.S
epsilon;agenda._*;nombre.S
epsilon;banco._*.oficinista;nombre.S,cargo.S
epsilon;banco._*.oficinista;nombre.S,oficina.S
epsilon;banco._*.oficinista;oficina.S,cargo.S
epsilon;banco.departamento.oficinista;nombre.S,cargo.S
epsilon;banco.departamento.oficinista;nombre.S,oficina.S
epsilon;banco.departamento.oficinista;oficina.S,cargo.S
epsilon;banco;nombre.S
epsilon;biblioteca.estante.repisa.libro;nombre.S
epsilon;carpeta.archivo;nombre.S,tamano.S
epsilon;computador.usuario._*.archivo;nombre.S,tamano.S
epsilon;country.city;population.S
epsilon;country;encompassed.continent
epsilon;country;government
epsilon;country;languages.S
epsilon;curso.alumno;nombre.S,apellido.S
epsilon;curso.alumno;rut.S,matricula.S
epsilon;dataset.reference.source.other.date;year.S
epsilon;dataset;subject
epsilon;disco._*.archivo;nombre.S,tamano.S,fecha.S
epsilon;disco.carpeta.archivo;nombre.S,tamano.S,fecha.S
epsilon;employee.person;name.S,last.S
epsilon;employee;id
epsilon;employee;person.S
epsilon;empresa._*.gerente;nombre.S,oficina.S,telefono.S
epsilon;empresa.departamento.gerente;nombre.S,oficina.S,telefono.S
epsilon;empresa.gerente;departamento.S

Continúa en la página siguiente...

TABLA A.2 – Continuación

Claves
epsilon;empresa.gerente;nombre.S,apellido.S
epsilon;issue.articles.article.authors.author;position
epsilon;issue;volume.S
epsilon;issue;volume.S,number.S
epsilon;libro.autor;rut.S,fecha_nacimiento.S
epsilon;libro.autor;rut.S,nombre.S,apellido.S
epsilon;libro.capitulo;nombre.S,numero.S
epsilon;libro.seccion;titulo.S,ano.S,editorial.S
epsilon;oficinista;nombre.S,oficina.S
epsilon;parts.color;S
epsilon;parts;color.S,partId.S
epsilon;politicPos.college;year.S
epsilon;_*.college;year.S
epsilon;politicPos;title.S
epsilon;proyecto._*.persona;idpersona.S,nombre.S
epsilon;proyecto._*.persona;nombre.S,oficina.S
epsilon;proyecto._*;S,institucion
epsilon;proyecto._*;persona
epsilon;proyecto.equipo.persona;idpersona.S,nombre.S
epsilon;proyecto.equipo.persona;idpersona.S,nombre.S,apellido.S
epsilon;proyecto.jefe.persona._*;S
epsilon;proyecto.jefe.persona;nombre.S,oficina.S
epsilon;proyecto.jefe;idpersona.S
epsilon;proyecto.jefe;nombre.S,apellido.S
epsilon;proyecto.jefe;persona
epsilon;proyecto.jefe;persona.nombre.S,persona.oficina.S
epsilon;proyecto;codp
epsilon;proyecto;jefe
epsilon;proyecto;nombre.S,codp
epsilon;restaurant._*.wine;name.S,year.S,price.S
epsilon;restaurant.menu.ditches;name.S
epsilon;restaurant.menu.meals.wine;name.S,year.S,price.S
epsilon;restaurant;name.S
epsilon;restaurant;name.S,address.S
epsilon;_*.wine;name.S,year.S,price.S
epsilon;student;name.S,grade.S
epsilon;universidad.empleado;@empId
epsilon;universidad;nombre.S
epsilon;user._*.file;name.S,size.S
epsilon;user._*.file;name.S,size.S,date.S
epsilon;user._*;file.name.S,file.size.S
epsilon;user.directory.file;name.S,size.S
epsilon;user.directory.file;name.S,size.S,date.S
epsilon;user.directory.file;name.S,size.S,fecha.S
epsilon;user.directory;file.name.S,file.size.S
epsilon;library.paper;title.S,year.S
epsilon;a.b;c
epsilon;a;b,c
epsilon;a.c;b
epsilon;b.a;c
epsilon;b;a,c
epsilon;b.c;a
epsilon;_*;a

Continúa en la página siguiente...

TABLA A.2 – Continuación

Claves
epsilon;_*;a,b
epsilon;_*;a,b,c
epsilon;a._*;b,c
epsilon;a._*.b;c
epsilon;_*.aa.bb;b.c,d.e
epsilon;aa.bb;b,c.d.e
epsilon;aa._*.b;bb.c,dd.e
epsilon;_*;a.b,c.d
epsilon;aa;b,c
epsilon;bb;d,e
epsilon;cc;d,e
epsilon;dd.ee;e,f
epsilon;oo.p.o.r;s,p.d

TABLA A.3: *Archivo prueba implicación - Σ relativo y φ absoluta.*

Claves
epsilon;agenda.contacto;nombre.S,telefono.S
_*.book;chapter;number.S
_*.chapter;verse;number.S
_*;college.person.name;first.S,last.S
agenda;contacto;nombre.S
agenda;contacto;nombre.S,telefono.S
banco;_*.oficinista;nombre.S,oficina.S
banco;departamento;depId.S
bible._*.chapter;verse;number.S
bible.book.chapter;verse;number.S
bible.book;chapter.figure;figId.S
bible.book;chapter;number.S
bible;book;name.S
biblioteca;seccion.libro;nombre.S
carpeta;archivo;nombre.S,tamano.S
computador;_*.archivo;nombre.S,tamano.S
computador;usuario._*.archivo;nombre.S,tamano.S
computador;usuario.disco.carpeta.archivo;nombre.S,tamano.S
country;city;country
country;city;name.S
country;ethnicgroups;percentage,S
country;religious;percentage,S
curso;alumno;nombre.S,apellido.S
curso;alumno;rut.S,matricula.S
dataset;reference.source.journal.other.date;lastName.S
disco._*;archivo;nombre.S,tamano.S,fecha.S
disco.carpeta;archivo;nombre.S,tamano.S,fecha.S
employee;person;name.S,last.S
employee;person;name.S,last.S,address.S
empresa.oficina;empleado;oficina.S,escritorio.S
empresa;departamento.bodega.producto;prodId.S,rack.S
empresa;departamento.bodega.producto;prodId.S,rack.S,palet.S

Continúa en la página siguiente...

TABLA A.3 – Continuación

Claves
empresa;departamento.cliente;rut.S
empresa;departamento.gerente;nombre.S,oficina.S
empresa;departamento.gerente;nombre.S,oficina.S,telefono.S
empresa;gerente;departamento.S
empresa;gerente;nombre.S,apellido.S
issue.articles;article;title.S
issue;articles.article.author;name.S,last.S
issue;articles.article;initPage.S,endPage.S
issue;articles.article;title.S
libro;autor;rut.S
libro;autor;rut.S,fecha_nacimiento.S
libro;autor;rut.S,nombre.S,apellido.S
libro;autores.autor;nombre.S,posicion.S
libro;capitulo;nombre.S,numero.S
libro;seccion;titulo.S,ano.S,editorial.S
politicPos;_*.person.name;first.S,last.S
politicPos;college.person.name;first.S,last.S
politicPos;college.person;name
proyecto.equipo;persona;idpersona.S,nombre.S
proyecto.equipo;persona;idpersona.S,nombre.S,apellido.S
proyecto.jefe;persona;nombre.S,oficina.S
proyecto;_*.persona;nombre.S,oficina.S
proyecto;jefe.persona;nombre.S,oficina.S
proyecto;jefe;idpersona.S
proyecto;jefe;institucion
proyecto;jefe;nombre.S,apellido.S
proyecto;jefe;persona
proyecto;jefe;persona.nombre.S
proyecto;jefe;persona.nombre.S,persona.oficina.S
restaurant;menu.drinks.wine;name.S,year.S
restaurant;menu.drinks.wine;name.S,year.S,price.S
universidad;_*.empleado;empId
universidad;_.departamento;nombre.S
universidad;facultad.departamento;nombre.S
user.directory;file;name.S,size.S
user;_*.file;name.S,size.S
user;directory._*.file;name.S,size.S,date.S
user;directory.file;name.S,size.S
user;directory.file;name.S,size.S,date.S
user;directory;file.name.S,file.size.S
restaurant;_*.wine;name.S,year.S,price.S
restaurant;_*.drinks.wine;name.S,year.S,price.S
universidad;_*.departamento;nombre.S
library;book;name.S,author.S
library;book;name.S
bookstore;book;name.S
_*.menu;drinks.wine;name.S,year.S,price.S
_*.equipo;persona;idpersona.S,nombre.S,apellido.S
_*.departamento;gerente;nombre.S,oficina.S
_*.a.b.c;d.e.f.g;h.i,j.k,l.m
_*.a.b;c.d.e.f.g.h.i;j,k
a._*.c;e._*.f.g.h.i;l,m
a.b.c;d,e;f,g

Continúa en la página siguiente...

TABLA A.3 – Continuación

Claves
a;b.c.d.e.f;g
a.c.d._*;f.g.h.i.j._*;m,n
aa;bb;c.d,e
ab.c;d;e,f
aj.b;cc.d;ee,ff,gg
aa;_*.bb;c,d
aa;bb._*.c;d,e
bb;cd.e;f,g
ab.ba;bab.bb.ac;b,c
la;as._*.lm.c;ed.cd,lm
ha._*;a.as;cd,c
lk.k;ll.mm;cd,cd.g.k
kj._*.km;kn.df;e,er.es.as
ca._*;cb.cd.ce;cf.cd,ch.ck.cl
ca._*;cb._*.ce;cf.cd,ch.ck.cl
ca._*;cb.cd._*;cf.cd,ch.ck.cl

TABLA A.4: *Archivo prueba implicación - Σ relativo y φ relativa.*

Claves
empresa;_*.gerente;nombre.S,oficina.S
_*.book;chapter;number.S
_*.chapter;verse;number.S
_*;college.person.name;first.S,last.S
agenda;contacto;nombre.S
agenda;contacto;nombre.S,telefono.S
banco;_*.oficinista;nombre.S,oficina.S
banco;departamento;depId.S
bible._*.chapter;verse;number.S
bible.book.chapter;verse;number.S
bible.book;chapter.figure;figId.S
bible.book;chapter;number.S
bible;book;name.S
biblioteca;seccion.libro;nombre.S
carpeta;archivo;nombre.S,tamano.S
computador;_*.archivo;nombre.S,tamano.S
computador;usuario._*.archivo;nombre.S,tamano.S
computador;usuario.disco.carpeta.archivo;nombre.S,tamano.S
country;city;country
country;city;name.S
country;ethnicgroups;percentage,S
country;religious;percentage,S
curso;alumno;nombre.S,apellido.S
curso;alumno;rut.S,matricula.S
dataset;reference.source.journal.other.date;lastName.S
disco._*;archivo;nombre.S,tamano.S,fecha.S
disco.carpeta;archivo;nombre.S,tamano.S,fecha.S
employee;person;name.S,last.S
employee;person;name.S,last.S,address.S

Continúa en la página siguiente...

TABLA A.4 – Continuación

Claves
empresa.oficina;empleado;oficina.S,escritorio.S
empresa;departamento.bodega.producto;prodId.S,rack.S
empresa;departamento.bodega.producto;prodId.S,rack.S,palet.S
empresa;departamento.cliente;rut.S
empresa;departamento.gerente;nombre.S,oficina.S
empresa;departamento.gerente;nombre.S,oficina.S,telefono.S
empresa;gerente;departamento.S
empresa;gerente;nombre.S,apellido.S
issue.articles;article;title.S
issue;articles.article.author;name.S,last.S
issue;articles.article;initPage.S,endPage.S
issue;articles.article;title.S
libro;autor;rut.S
libro;autor;rut.S,fecha_nacimiento.S
libro;autor;rut.S,nombre.S,apellido.S
libro;autores.autor;nombre.S,posicion.S
libro;capitulo;nombre.S,numero.S
libro;seccion;titulo.S,ano.S,editorial.S
politicPos;_*.person.name;first.S,last.S
politicPos;college.person.name;first.S,last.S
politicPos;college.person;name
proyecto.equipo;persona;idpersona.S,nombre.S
proyecto.equipo;persona;idpersona.S,nombre.S,apellido.S
proyecto.jefe;persona;nombre.S,oficina.S
proyecto;_*.persona;nombre.S,oficina.S
proyecto;jefe.persona;nombre.S,oficina.S
proyecto;jefe;idpersona.S
proyecto;jefe;institucion
proyecto;jefe;nombre.S,apellido.S
proyecto;jefe;persona
proyecto;jefe;persona.nombre.S
proyecto;jefe;persona.nombre.S,persona.oficina.S
restaurant;menu.drinks.wine;name.S,year.S
restaurant;menu.drinks.wine;name.S,year.S,price.S
universidad;_*.empleado;empId
universidad;_.departamento;nombre.S
universidad;facultad.departamento;nombre.S
user.directory;file;name.S,size.S
user;_*.file;name.S,size.S
user;directory._*.file;name.S,size.S,date.S
user;directory.file;name.S,size.S
user;directory.file;name.S,size.S,date.S
user;directory;file.name.S,file.size.S
restaurant;_*.wine;name.S,year.S,price.S
restaurant;_*.drinks.wine;name.S,year.S,price.S
universidad;_*.departamento;nombre.S
library;book;name.S,author.S
library;book;name.S
bookstore;book;name.S
_*.menu;drinks.wine;name.S,year.S,price.S
_*.equipo;persona;idpersona.S,nombre.S,apellido.S
_*.departamento;gerente;nombre.S,oficina.S
_*.a.b.c;d.e.f.g;h.i,j.k,l.m

Continúa en la página siguiente...

TABLA A.4 – Continuación

Claves
_*.a.b;c.d.e.f.g.h.i;j,k
a._*.c;e._*.f.g.h.i;l,m
a.b.c;d,e;f,g
a;b.c.d.e.f;g
a.c.d._*;f.g.h.i.j._*;m,n
aa;bb;c.d,e
ab.c;d;e,f
aj.b;cc.d;ee,ff,gg
aa;_*.bb;c,d
aa;bb._*.c;d,e
bb;cd.e;f,g
ab.ba;bab.bb.ac;b,c
la;as._*.lm.c;ed.cd,lm
ha._*;a.as;cd,c
lk.k;ll.mm;cd,cd.g.k
kj._*.km;kn.df;e,er.es.as
ca._*;cb.cd.ce;cf.cd,ch.ck.cl
ca._*;cb._*.ce;cf.cd,ch.ck.cl
ca._*;cb.cd._*;cf.cd,ch.ck.cl

TABLA A.5: *Archivo prueba implicación - Σ heterogéneo y φ absoluta.*

Claves
epsilon;agenda.contacto;nombre.S,telefono.S
_*.book;chapter;number.S
_*;college.person.name;first.S,last.S
agenda;contacto;nombre.S,telefono.S
banco;_*.oficinista;nombre.S,oficina.S
banco;departamento;depId.S
epsilon;banco.departamento.oficinista;oficina.S,cargo.S
bible._*.chapter;verse;number.S
epsilon;_*.gerente;nombre.S,oficina.S,telefono.S
epsilon;_*.person;name.S,last.S,address.S
bible.book.chapter;verse;number.S
bible.book;chapter;number.S
biblioteca;seccion.libro;nombre.S
carpeta;archivo;nombre.S,tamano.S
computador;_*.archivo;nombre.S,tamano.S
computador;usuario._*.archivo;nombre.S,tamano.S
computador;usuario.disco.carpeta.archivo;nombre.S,tamano.S
country;city;country
epsilon;country;encompassed.continent
country;city;name.S
country;ethnicgroups;percentage,S
country;religious;percentage,S
curso;alumno;nombre.S,apellido.S
curso;alumno;rut.S,matricula.S
dataset;reference.source.journal.other.date;lastName.S
disco._*;archivo;nombre.S,tamano.S,fecha.S

Continúa en la página siguiente...

TABLA A.5 – Continuación

Claves
disco.carpeta;archivo;nombre.S,tamano.S,fecha.S
employee;person;name.S,last.S
employee;person;name.S,last.S,address.S
epsilon;employee.person;name.S,last.S
empresa.oficina;empleado;oficina.S,escritorio.S
empresa;departamento.bodega.producto;prodId.S,rack.S
empresa;departamento.cliente;rut.S
empresa;departamento.gerente;nombre.S,oficina.S
empresa;departamento.gerente;nombre.S,oficina.S,telefono.S
empresa;gerente;departamento.S
empresa;gerente;nombre.S,apellido.S
issue.articles;article;title.S
issue;articles.article.author;name.S,last.S
issue;articles.article;initPage.S,endPage.S
issue;articles.article;title.S
libro;autor;idAutor.S,fecha_nacimiento.S
libro;autor;rut.S,nombre.S,apellido.S
libro;autores.autor;nombre.S,posicion.S
libro;capitulo;nombre.S,numero.S
libro;seccion;titulo.S,ano.S,editorial.S
epsilon;issue.articles.article.authors.author;position
politicPos;_*.person.name;first.S,last.S
politicPos;college.person.name;first.S,last.S
politicPos;college.person;name
proyecto.equipo;persona;idpersona.S,nombre.S
proyecto.equipo;persona;idpersona.S,nombre.S,apellido.S
proyecto.jefe;persona;nombre.S,oficina.S
proyecto;_*.persona;nombre.S,oficina.S
proyecto;jefe.persona;nombre.S,oficina.S
proyecto;jefe;idpersona.S
proyecto;jefe;institucion
proyecto;jefe;nombre.S,apellido.S
epsilon;proyecto._*.persona;nombre.S,oficina.S
proyecto;jefe;persona
proyecto;jefe;persona.nombre.S
proyecto;jefe;persona.nombre.S,persona.oficina.S
restaurant;menu.drinks.wine;name.S,year.S,price.S
universidad;_*.empleado;empId
universidad;_.departamento;nombre.S
universidad;facultad.departamento;nombre.S
user.directory;file;name.S,size.S
user;_*.file;name.S,size.S
user;directory._*.file;name.S,size.S,date.S
user;directory.file;name.S,size.S
epsilon;user.directory.file;name.S,size.S,date.S
user;directory.file;name.S,size.S,date.S
user;directory;file.name.S,file.size.S
restaurant;_*.wine;name.S,year.S,price.S
epsilon;restaurant.menu.ditches;name.S
restaurant;_*.drinks.wine;name.S,year.S,price.S
epsilon;_*.wine;name.S,year.S,price.S
universidad;_*.departamento;nombre.S
library;book;name.S,author.S

Continúa en la página siguiente...

TABLA A.5 – Continuación

Claves
library;book;name.S
bookstore;book;name.S
epsilon;_*;a,b,c
epsilon;a._*;b,c
epsilon;a._*.b;c
epsilon;_*.aa.bb;b.c,d.e
epsilon;aa.bb;b,c.d.e
epsilon;aa._*.b;bb.c,dd.e
_*.menu;drinks.wine;name.S,year.S,price.S
_*.equipo;persona;idpersona.S,nombre.S,apellido.S
_*.departamento;gerente;nombre.S,oficina.S
_*.a.b.c;d.e.f.g;h.i,j.k,l.m
_*.a.b;c.d.e.f.g.h.i;j,k
a._*.c;e._*.f.g.h.i;l,m
a.b.c;d,e;f,g
a;b.c.d.e.f;g
a.c.d._*;f.g.h.i.j._*;m,n
aa;bb;c.d,e
ab.c;d;e,f
ha._*;a.as;cd,c
epsilon;dd.ee;e,f
epsilon;oo.p.o.r;s,p.d

TABLA A.6: *Archivo prueba implicación - Σ heterogéneo y φ relativa.*

Claves
banco;_*.oficinista;nombre.S,oficina.S
_*.book;chapter;number.S
_*;college.person.name;first.S,last.S
agenda;contacto;nombre.S,telefono.S
banco;_*.oficinista;nombre.S,oficina.S
banco;departamento;depId.S
epsilon;banco.departamento.oficinista;oficina.S,cargo.S
bible._*.chapter;verse;number.S
epsilon;_*.gerente;nombre.S,oficina.S,telefono.S
epsilon;_*.person;name.S,last.S,address.S
bible.book.chapter;verse;number.S
bible.book;chapter;number.S
biblioteca;seccion.libro;nombre.S
carpeta;archivo;nombre.S,tamano.S
computador;_*.archivo;nombre.S,tamano.S
computador;usuario._*.archivo;nombre.S,tamano.S
computador;usuario.disco.carpeta.archivo;nombre.S,tamano.S
country;city;country
epsilon;country;encompassed.continent
country;city;name.S
country;ethnicgroups;percentage,S
country;religious;percentage,S
curso;alumno;nombre.S,apellido.S

Continúa en la página siguiente...

TABLA A.6 – Continuación

Claves
curso;alumno;rut.S,matricula.S
dataset;reference.source.journal.other.date;lastName.S
disco._*;archivo;nombre.S,tamano.S,fecha.S
disco.carpeta;archivo;nombre.S,tamano.S,fecha.S
employee;person;name.S,last.S
employee;person;name.S,last.S,address.S
epsilon;employee.person;name.S,last.S
empresa.oficina;empleado;oficina.S,escritorio.S
empresa;departamento.bodega.producto;prodId.S,rack.S
empresa;departamento.cliente;rut.S
empresa;departamento.gerente;nombre.S,oficina.S
empresa;departamento.gerente;nombre.S,oficina.S,telefono.S
empresa;gerente;departamento.S
empresa;gerente;nombre.S,apellido.S
issue.articles;article;title.S
issue;articles.article.author;name.S,last.S
issue;articles.article;initPage.S,endPage.S
issue;articles.article;title.S
libro;autor;idAutor.S,fecha_nacimiento.S
libro;autor;rut.S,nombre.S,apellido.S
libro;autores.autor;nombre.S,posicion.S
libro;capitulo;nombre.S,numero.S
libro;seccion;titulo.S,ano.S,editorial.S
epsilon;issue.articles.article.authors.author;position
politicPos;_*.person.name;first.S,last.S
politicPos;college.person.name;first.S,last.S
politicPos;college.person;name
proyecto.equipo;persona;idpersona.S,nombre.S
proyecto.equipo;persona;idpersona.S,nombre.S,apellido.S
proyecto.jefe;persona;nombre.S,oficina.S
proyecto;_*.persona;nombre.S,oficina.S
proyecto;jefe.persona;nombre.S,oficina.S
proyecto;jefe;idpersona.S
proyecto;jefe;institucion
proyecto;jefe;nombre.S,apellido.S
epsilon;proyecto._*.persona;nombre.S,oficina.S
proyecto;jefe;persona
proyecto;jefe;persona.nombre.S
proyecto;jefe;persona.nombre.S,persona.oficina.S
restaurant;menu.drinks.wine;name.S,year.S,price.S
universidad;_*.empleado;empId
universidad;_.departamento;nombre.S
universidad;facultad.departamento;nombre.S
user.directory;file;name.S,size.S
user;_*.file;name.S,size.S
user;directory._*.file;name.S,size.S,date.S
user;directory.file;name.S,size.S
epsilon;user.directory.file;name.S,size.S,date.S
user;directory.file;name.S,size.S,date.S
user;directory;file.name.S,file.size.S
restaurant;_*.wine;name.S,year.S,price.S
epsilon;restaurant.menu.ditches;name.S
restaurant;_*.drinks.wine;name.S,year.S,price.S

Continúa en la página siguiente...

TABLA A.6 – Continuación

Claves
epsilon;_*.wine;name.S,year.S,price.S
universidad;_*.departamento;nombre.S
library;book;name.S,author.S
library;book;name.S
bookstore;book;name.S
epsilon;_*;a,b,c
epsilon;a._*;b,c
_*.menu;drinks.wine;name.S,year.S,price.S
_*.equipo;persona;idpersona.S,nombre.S,apellido.S
epsilon;a._*.b;c
epsilon;_*.aa.bb;b.c,d.e
_*.departamento;gerente;nombre.S,oficina.S
_*.a.b.c;d.e.f.g;h.i,j.k,l.m
_*.a.b;c.d.e.f.g.h.i;j,k
a._*.c;e._*.f.g.h.i;l,m
a.b.c;d,e;f,g
a;b.c.d.e.f;g
epsilon;aa.bb;b,c.d.e
a.c.d._*;f.g.h.i.j._*;m,n
aa;bb;c.d,e
epsilon;aa._*.b;bb.c,dd.e
ab.c;d;e,f
ha._*;a.as;cd,c
epsilon;dd.ee;e,f
epsilon;oo.p.o.r;s,p.d

TABLA A.7: *Archivo prueba implicación - Σ heterogéneo y φ relativa más comodines.*

Claves
banco;_*.oficinista;nombre.S,oficina.S
_*.book;chapter;number.S
_*;college.person.name;first.S,last.S
agenda;contacto;nombre.S,telefono.S
banco;_*.oficinista;nombre.S,oficina.S
banco;departamento;depId.S
epsilon;banco.departamento.oficinista;oficina.S,cargo.S
bible._*.chapter;verse;number.S
epsilon;_*.gerente;nombre.S,oficina.S,telefono.S
epsilon;_*.person;name.S,last.S,address.S
bible._*.chapter;verse;number.S
bible.book;chapter;number.S
biblioteca;_*.libro;nombre.S
carpeta;archivo;nombre.S,tamano.S
computador;_*.archivo;nombre.S,tamano.S
computador;usuario._*.archivo;nombre.S,tamano.S
computador;usuario.disco._*.archivo;nombre.S,tamano.S
country;city;country
epsilon;country;encompassed.continent
country;city;name.S

Continúa en la página siguiente...

TABLA A.7 – Continuación

Claves
country;ethnicgroups;percentage,S
country;religious;percentage,S
curso;alumno;nombre.S,apellido.S
curso;alumno;rut.S,matricula.S
dataset;reference._*.journal.other.date;lastName.S
disco._*;archivo;nombre.S,tamano.S,fecha.S
disco.carpeta;archivo;nombre.S,tamano.S,fecha.S
employee;person;name.S,last.S
employee;person;name.S,last.S,address.S
epsilon;employee.person;name.S,last.S
empresa.oficina;empleado;oficina.S,escritorio.S
empresa;departamento.bodega.producto;prodId.S,rack.S
empresa;_*.producto;prodId.S,rack.S
empresa;departamento.cliente;rut.S
empresa;departamento.gerente;nombre.S,oficina.S,telefono.S
empresa;gerente;departamento.S
empresa;gerente;nombre.S,apellido.S
issue.articles;article;title.S
issue;articles.article.author;name.S,last.S
issue;_*.article;initPage.S,endPage.S
issue;articles.article;title.S
libro;autor;idAutor.S,fecha_nacimiento.S
libro;autor;rut.S,nombre.S,apellido.S
libro;_*.autor;nombre.S,posicion.S
libro;capitulo;nombre.S,numero.S
libro;seccion;titulo.S,ano.S,editorial.S
epsilon;issue._*.article.authors.author;position
politicPos;_*.person.name;first.S,last.S
politicPos;college.person.name;first.S,last.S
politicPos;college.person;name
proyecto.equipo;_*;idpersona.S,nombre.S
proyecto.equipo;persona;idpersona.S,nombre.S,apellido.S
proyecto.jefe;persona;nombre.S,oficina.S
proyecto;_*.persona;nombre.S,oficina.S
proyecto;jefe.persona;nombre.S,oficina.S
_*;jefe;idpersona.S
proyecto;jefe;institucion
proyecto;jefe;nombre.S,apellido.S
epsilon;proyecto._*.persona;nombre.S,oficina.S
proyecto;jefe;persona
proyecto;jefe;persona.nombre.S
proyecto;jefe;persona.nombre.S,persona.oficina.S
restaurant;menu.drinks.wine;name.S,year.S,price.S
universidad;_*.empleado;empId
universidad;_.departamento;nombre.S
universidad;facultad.departamento;nombre.S
user.directory;file;name.S,size.S
user;_*.file;name.S,size.S
user;directory._*.file;name.S,size.S,date.S
user;directory.file;name.S,size.S
epsilon;user.directory.file;name.S,size.S,date.S
_*;directory.file;name.S,size.S,date.S
user;directory;file.name.S,file.size.S

Continúa en la página siguiente...

TABLA A.7 – Continuación

Claves
restaurant;_*.wine;name.S,year.S,price.S
epsilon;restaurant.menu.ditches;name.S
restaurant;_*.drinks.wine;name.S,year.S,price.S
epsilon;_*.wine;name.S,year.S,price.S
universidad;_*.departamento;nombre.S
library;book;name.S,author.S
_*;book;name.S
bookstore;book;name.S
epsilon;_*;a,b,c
epsilon;a._*;b,c
_*.menu;drinks.wine;name.S,year.S,price.S
_*.equipo;persona;idpersona.S,nombre.S,apellido.S
epsilon;a._*.b;c
epsilon;_*.aa.bb;b.c,d.e
_*.departamento;gerente;nombre.S,oficina.S
_*.a.b.c;d.e.f.g;h.i,j.k,l.m
_*.a.b;c.d.e.f.g.h.i;j,k
a._*.c;e._*.f.g.h.i;l,m
a.b.c;d,e;f,g
a;b.c.d.e.f;g
epsilon;aa.bb;b,c.d.e
a.c.d._*;f.g.h.i.j._*;m,n
_*;bb;c.d,e
epsilon;aa._*.b;bb.c,dd.e
ab.c;d;e,f
ha._*;a.as;cd,c
epsilon;dd.ee;e,f
epsilon;oo.p.o.r;s,p.d

APÉNDICE B. INSTANCIAS DE PRUEBA VALIDACIÓN DE DOCUMENTOS

TABLA B.1: *Archivo prueba validación - Auction.*

Claves
epsilon;listing.seller_info;seller_name.S,seller_rating.S
epsilon;listing.auction_info;current_bid.S
listing;auction_info.high_bidder;bidder_name.S,bidder_rating.S
listing;item_info;memory.S,hard_drive.S,cpu.S,brand.S
epsilon;listing.seller_info;seller_rating.S
epsilon;listing.auction_info.high_bidder;bidder_rating.S
listing;auction_info.num_items;S

TABLA B.2: *Archivo prueba validación - DBLP.*

Claves
epsilon;_*.book;isbn
epsilon;_*.article.author;first.S,last.S
_*.article;author;first.S,last.S
_*.article;editor;name.S
_*.book;chapter;name.S
epsilon;_*.article;title
epsilon;_*.inproceedings;title
epsilon;_*.phdthesis;title
epsilon;_*.mastersthesis;title
epsilon;_*.incollection;title
epsilon;_*.proceedings;title
epsilon;_*.book;title
epsilon;_*.www;title
epsilon;_*.book;publisher

TABLA B.3: *Archivo prueba validación - Nasa.*

Claves
epsilon;dataset;@subject
epsilon;dataset.reference.source.other.date;year.S
dataset;reference.source.journal.author;lastName.S
epsilon;dataset.reference.source.journal;title.S
dataset;reference.source.journal;title.S
dataset.reference;source.journal;title.S
dataset.reference.source;journal;title.S
dataset.reference.source;_*;title.S
dataset._*.source;journal;title.S

TABLA B.4: *Archivo prueba validación - SIGMOD Record.*

Claves
epsilon;issue;volume.S,number.S
issue;articles.article;title.S
issue.articles;article;title.S
issue;articles.article;initPage,endPage
epsilon;issue;volume.S
epsilon;issue.articles.article.authors.author;@position

TABLA B.5: *Archivo prueba validación - Mondial.*

Claves
epsilon;country;@government
country;city;name.S
country;city;@country
epsilon;country;languages.S
country;religious;@percentage,S
country;ethnicgroups;@percentage,S
epsilon;country.city;name.S
epsilon;country.city;population.S
epsilon;country;encompassed.@continent
epsilon;country.province.city;name.S
epsilon;country.province;city
country;religious;S
country;province.city;name.S

APÉNDICE C. INSTANCIAS DE PRUEBA *COVERS* NO REDUNDANTES

TABLA C.1: *Archivo prueba cover no redundante - Σ heterogéneo.*

Claves
agenda;contacto;nombre.S
agenda;contacto;nombre.S,telefono.S
banco;departamento;depId.S
carpeta;archivo;nombre.S,tamano.S
country;city;country
country;city;name.S
country;ethnicgroups;percentage,S
country;religious;percentage,S
curso;alumno;nombre.S,apellido.S
curso;alumno;rut.S,matricula.S
dataset;reference.source.journal.other.date;lastName.S
disco._*;archivo;nombre.S,tamano.S,fecha.S
disco.carpeta;archivo;nombre.S,tamano.S,fecha.S
employee;person;name.S,last.S
employee;person;name.S,last.S,address.S
empresa.oficina;empleado;oficina.S,escritorio.S
empresa;departamento.bodega.producto;prodId.S,rack.S
empresa;departamento.bodega.producto;prodId.S,rack.S,palet.S
empresa;departamento.cliente;rut.S
empresa;gerente;departamento.S
empresa;gerente;nombre.S,apellido.S
epsilon;_*.empleado;empId
epsilon;_*.empleado;nombre.S,telefono.S
epsilon;_*.person;name.S,last.S,address.S
epsilon;_*.persona;nombre.S,oficina.S
epsilon;agenda._*;nombre.S
epsilon;agenda.contacto;nombre.S
epsilon;agenda.contacto;nombre.S,telefono.S
epsilon;banco;nombre.S
epsilon;carpeta.archivo;nombre.S,tamano.S
epsilon;country.city;population.S
epsilon;country;encompassed.continent
epsilon;country;government
epsilon;country;languages.S
epsilon;curso.alumno;nombre.S,apellido.S
epsilon;curso.alumno;rut.S,matricula.S
epsilon;dataset.reference.source.other.date;year.S
epsilon;dataset;subject

Continúa en la página siguiente...

TABLA C.1 – Continuación

Claves
epsilon;disco._*.archivo;nombre.S,tamano.S,fecha.S
epsilon;disco.carpeta.archivo;nombre.S,tamano.S,fecha.S
epsilon;employee.person;name.S,last.S
epsilon;employee;id
epsilon;employee;person.S
epsilon;empresa.gerente;departamento.S
epsilon;empresa.gerente;nombre.S,apellido.S
epsilon;issue.articles.article.authors.author;position
epsilon;issue;volume.S
epsilon;issue;volume.S,number.S
epsilon;libro.autor;rut.S,fecha_nacimiento.S
epsilon;libro.autor;rut.S,nombre.S,apellido.S
epsilon;libro.capitulo;nombre.S,numero.S
epsilon;libro.seccion;titulo.S,ano.S,editorial.S
epsilon;oficinista;nombre.S,oficina.S
epsilon;proyecto._*.persona;idpersona.S,nombre.S
epsilon;proyecto._*.persona;nombre.S,oficina.S
epsilon;proyecto._*;S,institucion
epsilon;proyecto._*;persona
epsilon;proyecto.equipo.persona;idpersona.S,nombre.S
epsilon;proyecto.equipo.persona;idpersona.S,nombre.S,apellido.S
epsilon;proyecto.jefe.persona._*;S
epsilon;proyecto.jefe.persona;nombre.S,oficina.S
epsilon;proyecto.jefe;idpersona.S
epsilon;proyecto.jefe;nombre.S,apellido.S
epsilon;proyecto.jefe;persona
epsilon;proyecto.jefe;persona.nombre.S,persona.oficina.S
epsilon;proyecto;codp
epsilon;proyecto;jefe
epsilon;proyecto;nombre.S,codp
epsilon;universidad;nombre.S
epsilon;universidad.empleado;@empId
epsilon;user._*.file;name.S,size.S
epsilon;user._*.file;name.S,size.S,date.S
epsilon;user._*;file.name.S,file.size.S
epsilon;user.directory.file;name.S,size.S
epsilon;user.directory.file;name.S,size.S,date.S
epsilon;user.directory.file;name.S,size.S,fecha.S
epsilon;user.directory;file.name.S,file.size.S
issue.articles;article;title.S
issue;articles.article;initPage.S,endPage.S
issue;articles.article;title.S
libro;autor;rut.S
libro;autor;rut.S,fecha_nacimiento.S
libro;autor;rut.S,nombre.S,apellido.S
libro;capitulo;nombre.S,numero.S
libro;seccion;titulo.S,ano.S,editorial.S
proyecto.equipo;persona;idpersona.S,nombre.S
proyecto.equipo;persona;idpersona.S,nombre.S,apellido.S
proyecto.jefe;persona;nombre.S,oficina.S
proyecto;_*.persona;nombre.S,oficina.S
proyecto;jefe.persona;nombre.S,oficina.S
proyecto;jefe;idpersona.S

Continúa en la página siguiente...

TABLA C.1 – Continuación

Claves
proyecto;jefe;institucion
proyecto;jefe;nombre.S,apellido.S
proyecto;jefe;persona
proyecto;jefe;persona.nombre.S
proyecto;jefe;persona.nombre.S,persona.oficina.S
universidad;_*.empleado;empId
universidad;_departamento;nombre.S
universidad;facultad.departamento;nombre.S
user.directory;file;name.S,size.S
user;_*.file;name.S,size.S
user;directory._*.file;name.S,size.S,date.S
user;directory.file;name.S,size.S
user;directory.file;name.S,size.S,date.S
user;directory;file.name.S,file.size.S
libro;autores.autor;nombre.S,posicion.S
issue;articles.article.author;name.S,last.S
biblioteca;seccion.libro;nombre.S
epsilon;biblioteca.estante.repisa.libro;nombre.S
computador;usuario.disco.carpeta.archivo;nombre.S,tamano.S
epsilon;computador.usuario._*.archivo;nombre.S,tamano.S
computador;usuario._*.archivo;nombre.S,tamano.S
computador;_*.archivo;nombre.S,tamano.S
epsilon;banco.departamento.oficinista;nombre.S,cargo.S
epsilon;banco._*.oficinista;nombre.S,cargo.S
epsilon;banco.departamento.oficinista;nombre.S,oficina.S
epsilon;banco._*.oficinista;nombre.S,oficina.S
epsilon;banco.departamento.oficinista;oficina.S,cargo.S
epsilon;banco._*.oficinista;oficina.S,cargo.S
banco;_*.oficinista;nombre.S,oficina.S
empresa;departamento.gerente;nombre.S,oficina.S,telefono.S
empresa;departamento.gerente;nombre.S,oficina.S
epsilon;empresa.departamento.gerente;nombre.S,oficina.S,telefono.S
epsilon;_*.gerente;nombre.S,oficina.S,telefono.S
epsilon;empresa._*.gerente;nombre.S,oficina.S,telefono.S
epsilon;restaurant;name.S
epsilon;restaurant;name.S,address.S
restaurant;menu.drinks.wine;name.S,year.S,price.S
restaurant;menu.drinks.wine;name.S,year.S
epsilon;restaurant.menu.meals.wine;name.S,year.S,price.S
epsilon;restaurant._*.wine;name.S,year.S,price.S
epsilon;_*.wine;name.S,year.S,price.S
epsilon;politicPos;title.S
epsilon;politicPos.college;year.S
politicPos;college.person.name;first.S,last.S
politicPos;_*.person.name;first.S,last.S
politicPos;college.person;name
epsilon;_*.college;year.S
politicPos;_*.person.name;first.S,last.S
_*;college.person.name;first.S,last.S
epsilon;student;name.S,grade.S
bible.book.chapter;verse;number.S
_*.chapter;verse;number.S
bible._*.chapter;verse;number.S

Continúa en la página siguiente...

TABLA C.1 – Continuación

Claves
bible.book;chapter;number.S
_*.book;chapter;number.S
bible;book;name.S
bible.book;chapter.figure;figId.S
epsilon;parts;color.S,partId.S
epsilon;parts.color;S