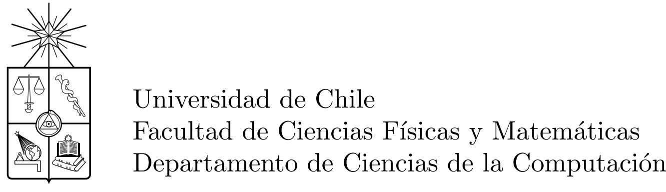
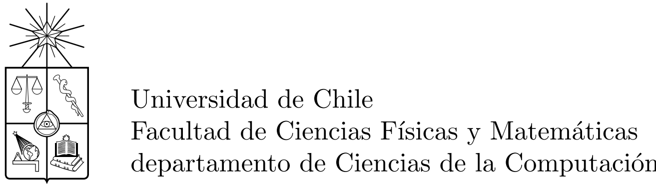

Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

EDITOR DE DOCUMENTOS XML
USANDO PLANTILLAS Y TRANSFORMACIONES

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

DANIEL RICARDO HERNÁNDEZ HERNÁNDEZ

SANTIAGO DE CHILE



Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
departamento de Ciencias de la Computación

EDITOR DE DOCUMENTOS XML
USANDO PLANTILLAS Y TRANSFORMACIONES

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

DANIEL RICARDO HERNÁNDEZ HERNÁNDEZ

PROFESOR GUÍA:
CLAUDIO GUTIÉRREZ GALLARDO

INTEGRANTES DE LA COMISIÓN
CECILIA BASTARRICA PIÑEYRO
MARCELA CALDERÓN CORAIL

SANTIAGO DE CHILE
ENERO 2011

Resumen

La edición de datos semi-estructurados, en particular de documentos *XML*, es habitualmente abordada con formularios e interfaces de edición para tablas (*scaffolds*), herramientas diseñadas para trabajar con datos estructurados. Otras herramientas que sí pueden abordar modelos semi-estructurados (como *XForms* y *XTiger*) no poseen un lenguaje completamente declarativo y están limitadas a modelos no recursivos. Estas limitaciones implican un mayor costo en el desarrollo de aplicaciones que trabajan con datos semi-estructurados.

En esta memoria se propone un lenguaje de plantillas, que bautizamos Queule, basado en *XTiger*, para la edición de datos semi-estructurados. Queule reduce los costos de implementar interfaces de edición al permitir la definición declarativa de interfaces para los modelos y al incluir el soporte a modelos recursivos. Las plantillas, al igual que los esquemas de datos (*DTD*, *XML Schema*, *Schematron*, *RELAX NG*, etc.), permiten delimitar el universo de las instancias que satisfacen un modelo. A diferencia de los esquemas, en los que se habitualmente se provee de algoritmos para decidir si se satisface un esquema, las plantillas proveen mecanismos para editar los datos definiendo de manera constructiva las restricciones del modelo.

La idea de usar transformaciones, unidireccionales o bidireccionales, mencionada en el título de esta memoria, estaba motivada por el almacenamiento de los datos en un documento distinto al usado como plantilla. Al inicio del desarrollo de esta memoria, las transformaciones cumplían un rol central en el sistema que, no obstante, perdió relevancia cuando se descubrió la posibilidad, y conveniencia, de que una plantilla no estuviera reflejada necesariamente en un sólo documento de una base documental, si no más bien, en una vista de una base de datos cualquiera. De este modo, los cambios no esperarían al término de la edición para ser enviados en la forma del documento editado, sino que podrían enviarse de manera asíncrona a medida que se edita.

En el desarrollo de esta memoria se implementó un intérprete para la versión 1.0 del lenguaje Queule que se limita a la definición de estructuras de árbol que consideran el orden entre los nodos hermanos. Futuras versiones de Queule, podrían extender la edición a un modelo de grafos más general.

El funcionamiento de Queule 1.0 es ejemplificado con varias plantillas que pueden ser editadas usando los navegadores Web más populares debido a que el intérprete fue programado como una biblioteca *JavaScript*.

Agradecimientos

A mis padres, a Natalia y a Sonia
por su cariño y paciencia.

A Claudio
por todo su apoyo.

Índice general

I Introducción **8**

1. Antecedentes **9**

1.1. Documentos XML 9
1.2. Marcado de documentos con RDFa 11
1.3. Edición de documentos XML 13
1.4. Edición de datos estructurados 14
1.5. Edición de datos semi-estructurados 15
1.6. Edición con formularios 16
1.6.1. Modelos XForms 17
1.6.2. Instancias de datos XForms 18
1.6.3. Interfaz de usuario 19
1.6.4. Recursividad en XForms 19
1.7. Edición guiada por plantillas 19
1.7.1. Tipos XTiger 21
1.7.2. Campos XTiger 22
1.7.3. Recursividad en XTiger 23
1.8. Conclusiones 24

2. Motivación y propuesta **26**

2.1. Casos de estudio 27
2.1.1. Editor de cuestionarios 27
2.1.2. Editor de organizaciones 28
2.1.3. Editor de segmentación de locales 29
2.2. Diagnóstico para herramientas existentes 30
2.3. Lineamientos del diseño de Queule 33

2.4. Incorporación de transformaciones 34
2.5. Asociación de elementos 36
2.6. Estructuras recursivas por construcción 39
2.7. Sistema de tipos 41

3. Objetivos y metodología **42**
3.1. Objetivo general 42
3.2. Objetivos específicos 42
3.3. Metodología 43
3.3.1. Casos de estudio 43
3.3.2. Estado del arte 43
3.3.3. Especificación del lenguaje propuesto Queule 43
3.3.4. Implementación 44

II Desarrollo **45**

4. Especificación de Queule 1.0 **46**
4.1. Tipos básicos 46
4.2. Elemento `library` 47
4.3. Elemento `import` 47
4.4. Elemento `yield` 48
4.5. Elemento `use` 49
4.6. Elemento `component` 49

5. Ejemplos de plantillas Queule **51**
5.1. Ejemplo de uso con RDFa 51
5.2. Ejemplo recursivo 53

6. Implementación **54**
6.1. Diseño de clases 54
6.2. Uso de Queule 56
6.3. Interfaz de usuario 57

III Conclusiones **60**

Índice de códigos

1.1. Ejemplo de microformatos 11
1.2. Ejemplo de RDFa 12
1.3. Ejemplo modelo de XForms 17
1.4. Ejemplo de un documento XML generado por XForms 18
1.5. Ejemplo de un XForms Control 18
2.6. Documento posible para la edición de segmentos 30
2.7. Interfaz para escoger valores desde un conjunto en XForms 32
5.8. Colección de libros marcada con RDFa 51
5.9. Tipo para agregar autores 52
5.10. Tipo para agregar libros 52
5.11. Plantilla para la colección de libros 52
5.12. Tipo recursivo 53
6.13. Líneas para incluir Queule 57

Índice de figuras

1.1. Arboles XML . . . 10
1.2. Mecanismos de Amaya . . . 21
1.3. Tipos compuestos en XTiger . . . 22
1.4. Recursividad infinita en XTiger . . . 23
1.5. Recursividad no necesariamente infinita en XTiger . . . 24

2.1. Diagrama de red semántica para los cuestionarios . . . 28
2.2. Diagrama de red semántica para organizaciones . . . 28
2.3. Diagrama de red semántica para la segmentación de locales . . . 29
2.4. Un documento puede ser visto y editado por Amaya. . . . 34
2.5. Transformaciones para editar y guardar . . . 35

6.1. Diagrama de clases de la biblioteca JavaScript para Queule . . . 55
6.2. Captura de pantalla 1 de la interfaz de Queule . . . 58
6.3. Captura de pantalla 2 de la interfaz de Queule . . . 59

Índice de cuadros

1.1. Tripletas extraídas del conjunto de 13
1.2. Diseño y objetivos de XForms 17

2.1. Modelos plantillas y por esquemas 33
2.2. CCS Selectors 3.0 versus XPath 38
2.3. Elementos instanciados y no instanciados en Queule 41

6.1. Comparación número términos de las especificaciones 61

Parte I

Introducción

Capítulo 1

Antecedentes

1.1. Documentos XML

Un documento es un registro de algo, por ejemplo, un poema, un decreto emitido por un organismo público, un libro de contabilidad, o una lista de precios de productos. La aparición de las computadoras cambió la forma de producir y almacenar documentos. El papel dejó de ser el medio de almacenamiento e intercambio, dando paso a los documentos digitales y a numerosas aplicaciones informáticas para asistir la tarea de registrar información.

La necesidad de operar con la información, en un nivel de granularidad más fino que el documento, dio origen a las bases de datos. En ellas la información ya no es guardada como una colección de documentos, sino como una colección de datos. Una definición intuitiva que podemos hacer para diferenciar entre documentos y datos es definir a un documento digital como una secuencia de bytes y a un dato como la afirmación de un hecho. De este modo, los dumps de las bases de datos, el resultado de una consulta y el formulario que un sitio web envía para que el usuario lo llene, son también documentos. En general, considerar a los documentos como secuencias de bits implica que toda la información se almacena o transmite mediante documentos, y de estos documentos, cuando se encuentran en un lenguaje adecuado, podemos extraer datos.

Con el surgimiento de la Web, XML (eXtensible Markup Language) se convirtió en el estándar más usado para la codificación de documentos y el intercambio de datos. La ventaja de XML ha sido su simplicidad y su flexibilidad para modelar la estructura de los diversos tipos de documentos. Permitiendo codificar, desde textos estructurados (como poemas, libros o decretos de ley), hasta conjuntos de datos (como un libro de contabilidad

o una lista de precios de productos).

XML es una forma de codificar un árbol de nodos "rotulados" y ordenado. La figura 1.1 presenta dos árboles XML que codifican los mismos datos, el radio y las coordenadas del centro que determinan un círculo.

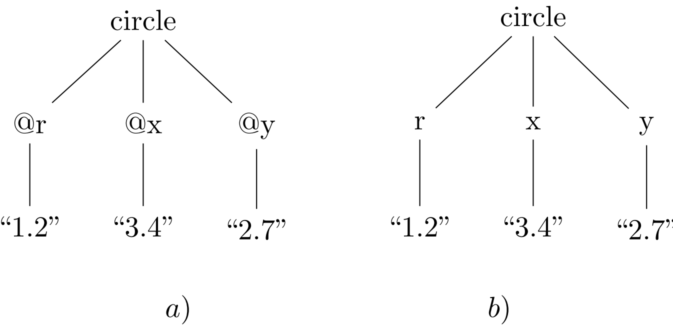


Figura 1.1: Dos árboles XML. Ambos codifican un círculo con los mismos datos, pero sus estructuras difieren en que *a*) usa atributos mientras que *b*) usa elementos.

En XML se reconocen tres tipos de nodos:

1. Los nodos elemento pueden tener como hijos a varios nodos de cualquiera de los otros tipos. En el ejemplo corresponden a los nodos `circle` en el árbol *a*) y `circle`, `r`, `x` e `y` en el árbol *b*).

2. Los nodos atributo, como `@r`, `@x` e `@y` en el árbol *a*), son identificados por el prefijo `@` y siempre deben tener como hijos a un único nodo de texto.

3. Los nodos de texto son identificados por el uso de comillas y no pueden tener hijos. Por ejemplo, los nodos "1.2", "3.4" y "2.7" presentes en ambos árboles de la figura 1.1.

Los árboles XML se codifican en documentos usando un sistema de etiquetas anidadas secuencialmente de izquierda a derecha. Por ejemplo, la representación de los árboles de la figura 1.1 es la siguiente:

a) `<circle r="1.2" x="3.4" y="2.7"/>`

b) `<circle><r>1.2</r><x>3.4</x><y>2.7</y></circle>`

Para codificar la información de un área mediante documentos, el universo de los documentos se acota mediante restricciones a la sintaxis de XML. Estas restricciones limitan los elementos, los atributos y la forma en la que estos pueden anidarse o aparecer en un documento, de modo que no se violen las reglas observadas en el área que se está modelando. Por ejemplo, en el caso de describir círculos, como se hace con el árbol a) de la figura 1.1 se podría exigir que los atributos `r`, `x` e `y` sean hijos obligatorios y únicos de los nodos `circle` y que sus valores sean siempre numéricos.

Para definir los conjuntos de documentos que modelan un tema específico existen lenguajes como DTD, XML Schema, RELAX NG o Schematron, lenguajes que se diferencian por su capacidad expresiva y su complejidad. Se llama un esquema a la definición de un conjunto de documentos definida en uno de estos lenguajes. Se dice que un documento satisface un esquema cuando pertenece al conjunto que el esquema define. Para que un lenguaje pueda usarse en la práctica, debe haber un algoritmo que permita decidir si un documento cualquiera satisface un esquema definido en el lenguaje.

1.2. Marcado de documentos con RDFa

Las ideas centrales de esta memoria nacen del lenguaje de plantillas XTiger [19] y de sus propiedades para editar documentos de manera estructurada e incorporando microformatos [20]. Los microformatos son patrones construidos con los elementos de HTML con el objetivo de explicitar la semántica de la información desplegada [10]. El fragmento de código 1.1 muestra cómo, usando microformatos, la información de una persona puede ser codificada y extraída de un documento HTML usando el microformato vCard [11].

Código 1.1: Ejemplo de microformatos

```
1 <div class="vcard">
2   <div class="fn">Joe Doe</div>
3   <div class="org">The Example Company</div>
4   <div class="tel">604-555-1234</div>
5   <a class="url" href="http://example.com/">http://example.com/</a>
6 </div>
```

Con el desarrollo de las bases de datos de grafos [12], la Web Semántica [13] y el proyecto Linked Data [14], el uso de tripletas sujeto/predicado/objeto para la codificación

de la información ha ido ganando terreno. Junto con los formatos específicamente dedicados a codificar tripletas como RDF/XML, N3, Turtle y N-Triples, el Consorcio de la Web ha propuesto una manera de integrar los datos RDF en un documento HTML. Este sistema de marcado funciona de manera similar a los microformatos, salvo que la semántica ahora cuenta con un modelo uniforme de datos: las tripletas. Por ejemplo, los datos presentados en el fragmento de código 1.1 pueden ser codificados mediante RDFa [15] de la manera que se muestra en el fragmento 1.2.

Código 1.2: Ejemplo de RDFa

```
<div typeof="foaf:Person" about="#joe-doe">
  <div property="foaf:name">Joe Doe</div>
  <div typeof="foaf:Group foaf:Organization">
    <div property="foaf:name">The Example Company</div>
    <div property="foaf:member"
         content="#joe-doe"
         style="display:none">
  </div>
  <div property="foaf:phone" content="tel:+604-555-1234">
    604-555-1234
  </div>
  <a rel="foaf:homepage" href="http://example.com/">
    http://example.com/
  </a>
</div>
```

La información contenida en el fragmento de código 1.2 corresponde al conjunto de las siguientes tripletas que se muestran en el cuadro 1.1.

Lo que se muestra con este ejemplo es que, si bien la estructura del documento puede tener forma de árbol, la información que éste representa puede estar expresada en forma de tripletas (o grafos). Existen varias formas de transformar datos contenidos en una estructura de árbol a otra estructura, como usar código procedural o usar lenguajes funcionales o declarativos específicamente diseñados para ello, como XSLT [31], XDuce [32] y biXid [33]. XSLT es probablemente el lenguaje más usado para transformaciones unidireccionales. También existen lenguajes como biXid que definen transformaciones bidireccionales.

En general, las transformaciones son usadas para relacionar modelos distintos. En algunos casos, también se usan las transformaciones para extraer información dentro de un

Cuadro 1.1: Tripletas extraídas del conjunto de

Sujeto	Predicado	Objeto
_:b1	rdf:type	foaf:Person
_:b1	foaf:name	"Joe Doe"
_:b2	rdf:type	foaf:Group
_:b2	rdf:type	foaf:Organization
_:b2	rdf:name	"The Example Company"
_:b2	foaf:member	_:b1
_:b1	foaf:phone	<tel:+604-555-1234>
_:b1	foaf:homepage	<http://example.com/>

documento codificada en un formato embebido en la sintaxis de este, como es el caso de la transformación que convierte documentos XHTML+RDFa en documentos RDF/XML [21]. Este tipo de transformaciones permiten usar XHTML como un soporte genérico para árboles que tienen una semántica expresable en términos de RDF. De la misma forma en que RDFa permite anidar tripletas en documentos XHTML, el lenguaje de plantillas propuesto en esta memoria, Queule[1], permite editar datos semi-estructurados usando como base la estructura de árbol y la representación gráfica de los documentos XHTML.

1.3. Edición de documentos XML

Editar documentos XML es una tarea para la cual aún no hay una solución genérica que sea suficientemente simple, tanto para quienes deben implementar los editores como para quienes deben ingresar los datos. Se entiende por solución genérica a aquella que se adapta a cualquier esquema XML. Por el contrario, las soluciones particulares son aquellas diseñadas sólo para editar instancias de un sólo esquema. La creación de facturas electrónicas, implementada actualmente por el Servicio de Impuestos Internos, es un ejemplo de una solución particular. Las facturas electrónicas son datos semi-estructurados al poseer cada factura un número variable de campos indicando el detalle de la venta. El usuario puede agregar más líneas de detalle si es que las que vienen inicialmente en la factura no son

[1]El nombre Queule no viene de ninguna sigla, tan sólo es un nombre común para el *Gomortega keule*, un árbol siempre verde nativo de las regiones VII y VIII de Chile y cuya conservación se encuentra en peligro.

suficientes. Esa agregación es realizada mediante un procedimiento implementado proceduralmente de manera particular para el caso de la factura. Una solución genérica podría simplemente declarar que las facturas pueden llevar un número indeterminado de líneas de detalle y permitir que la interfaz de edición se haga cargo de permitir que los usuarios agreguen más líneas cuando lo necesitan, sin necesidad de programar para cada caso la función que modifica el DOM del documento.

La edición de XML directamente a través del código o mediante la visualización y manipulación del árbol XML no es viable cuando los encargados de ingresar los datos son personas sin la capacitación suficiente para trabajar directamente con la estructura de los datos. Quienes ingresan datos de fichas médicas o redactan el decreto de un organismo público, necesitan una aplicación que les muestre el documento de un modo similar a como lo ven cuando escriben sobre un papel, es decir, necesitan algo que se vea como un formulario.

1.4. Edición de datos estructurados

Probablemente la interfaz por la cual los usuarios acceden a la mayoría de las bases de datos es la Web. Por otra parte, gran parte de la Web es generada por aplicaciones que se apoyan en bases de datos para lograr la persistencia y el acceso a la información. De este modo, la edición de datos por un público no especializado se convierte en un punto central para el desarrollo de aplicaciones. Los frameworks de desarrollo web, junto con facilitar el manejo de los permisos de acceso y escritura de la información, deben entregar herramientas flexibles para generar vistas e interfaces de edición. Las interfaces de edición deben ser simples, para minimizar la carga que significa la capacitación de los usuarios, y deben garantizar cierto nivel de consistencia de los datos.

Varios frameworks permiten generar interfaces de edición desde la definición de los modelos de datos. El framework de desarrollo web Ruby on Rails [2], en su versión 2.0, posee generadores que construyen scaffolds al momento de definir una clase de datos. Un scaffold consiste en una serie de plantillas para generar dinámicamente documentos HTML que constituyen la interfaz para las funcionalidades CRUD de una tabla en un modelo relacional. Los scaffolds se convierten en el código base sobre el cual los programadores pueden diseñar sus interfaces a medida. Lamentablemente, como el generador de scaffolds no analiza la existencia de llaves foráneas, resulta obligatorio agregar interfaces apropiadas para vincular los distintos conceptos.

ActiveScaffold [3], un plugin desarrollado para Ruby on Rails, permitió subsanar la

falencia de los generadores de scaffolds, al definir interfaces que permitían manipular asociaciones entre distintos objetos. Con ayuda de AJAX, ActiveScaffold permite, por ejemplo, agregar o borrar escritores, asociarlos a libros y editar sus títulos u otras de sus propiedades, todo ello en una interfaz intuitiva que no requiere recargar páginas al editar.

Hoy frameworks web como Django [4], TurboGears [5] y CakePHP [6], entre muchos otros, disponen de modos para generar scaffolds. En general todos ellos construyen interfaces CRUD para modelos relacionales que son mapeados a objetos mediante bibliotecas como ActiveRecord, SQLAlchemy, SQLObject, entre otras.

A pesar de que la generación rápida de interfaces que implementan las funcionalidades CRUD puede resultar útil, en muchos casos sólo corresponde a una porción muy pequeña del trabajo total que hay que realizar antes de lanzar una aplicación a producción. Las interfaces de usuario de las aplicaciones web suelen necesitar una interfaz más amigable que una orientada sólo a manipular objetos en forma de tablas. A pesar de que, para hacer calzar la información con el modelo relacional, ésta termina en forma de datos estructurados, en muchas ocasiones para los usuarios suele ser más intuitivo y fácil manejar la información en forma de datos semi-estructurados, tal como ésta era por naturaleza.

1.5. Edición de datos semi-estructurados

A diferencia de los frameworks diseñados para el uso de bases de datos relacionales, el manejador de contenidos Plone [7] presenta una manera de definir tipos de datos semi-estructurados, conocidos como arquetipos. Las instancias de los arquetipos son almacenadas en la base de datos orientada a objetos Zope Object Database [8], que permite agregar persistencia a objetos del lenguaje Python [9].

Si bien las herramientas que posee Plone para definir nuevos tipos parecieran proveer facilidades en la creación de interfaces usuarias para la edición de datos semi-estructurados, las herramientas se encuentran tan ligadas a Plone, que resulta difícil utilizarlas en aplicaciones desarrolladas con otras herramientas. En cambio, Queule es un lenguaje que define plantillas que pueden ser utilizadas con independencia de las herramientas usadas para construir la aplicación y del protocolo usado para transmitir los datos editados.

1.6. Edición con formularios

Los formularios son una parte importante de la Web al constituir la base de la mayoría de las aplicaciones interactivas en ella. Los primeros formularios, que aún se usan, permiten a los usuarios llenar una serie de campos dentro de un documento HTML. Cuando el usuario presiona un botón, se envía una serie de pares nombre-valor a un servidor.

Los formularios HTML disponen de un conjunto de interfaces que un usuario puede llenar: campos para un texto de una línea, áreas de texto para varias líneas, campos para ingresar una contraseña, menús de alternativas, botones de radio para alternativas y cajas de check para ingresar variables booleanas. No obstante, toda esa variedad de interfaces cumple el mismo rol de capturar un valor como una cadena de texto y asociarlo a un rótulo (o nombre del dato) que también es una cadena de texto. Los formularios HTML por sí solos no entregan ninguna forma de validar el tipo de los datos que se ingresan y la estructura de los datos ingresados mediante el formulario no permiten ingresar estructuras de árbol como los documentos XML.

XForms es el sucesor de los formularios HTML. La no validación de tipos en los formularios HTML incentivó la introducción de scripting para evitar los inconvenientes de que los datos ingresados fueran validados sólo en el servidor. XForms evita el uso de scripting para la validación al utilizar variables fuertemente tipadas en los campos de los formularios, codificar restricciones entre los valores e incorporar eventos. Además, los datos no se envían como una colección de pares rótulo-valor, sino que se envían como un documento XML. En el abstract de la definición de XForms 1.0 se dice lo siguiente:

> XForms is an XML application that represents the next generation of forms for the Web. By splitting traditional XHTML forms into three parts —XForms model, instance data, and user interface— it separates presentation from content, allows reuse, gives strong typing —reducing the number of round-trips to the server, as well as offering device independence and a reduced need for scripting.
>
> XForms is not a free-standing document type, but is intended to be integrated into other markup languages, such as XHTML or SVG.

Abstract de la especificación 1.0 de XForms.

El cuadro 1.2 muestra las dos decisiones de diseño mencionadas en el abstract de la especificación de XForms y los objetivos que ellas persiguen.

Cuadro 1.2: Diseño y objetivos de XForms

Diseño	**Objetivo**
División en modelo, instancias e interfaz de usuario	Separa la presentación del contenido, permitiendo la reutilización.
Datos fuertemente tipados	Evita el número de peticiones y respuestas con el servidor, ofrece independencia de dispositivos y reduce la necesidad de escribir código procedural (scripting).

Las secciones 1.6.1, 1.6.2 y 1.6.3 presentan las tres componentes de XForms: modelo, instancias de datos e interfaz usuaria. La descripción de sistema de tipos se omitirá puesto que, si bien XTiger y Queule proponen y deberían ser usados con un sistema de tipos para los datos terminales, ese no es el nudo central de lo desarrollado en esta memoria.

1.6.1. Modelos XForms

El fragmento de código 1.3 muestra cómo puede definirse un modelo en XForms a través del elemento `xf:instance`. En este caso, el modelo describe información de una persona. Además, el ejemplo muestra cómo se han restringido los campos del modelo, al asociarlos a tipos específicos mediante el atributo `@xsi:type`. Estos tipos corresponden a los que se definen para XML Schema [35], siendo el nombre (fname) y el apellido (lname), definidos de tipo `xsd:string`, y la fecha de nacimiento (born), de tipo `xsd:date`.

Código 1.3: Ejemplo modelo de XForms

```
1  <html
2     xmlns:xf="http://www.w3.org/2002/xforms"
3     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5  ...
6    <xf:instance>
7      <person>
8        <fname xsi:type="xsd:string"/>
9        <lname xsi:type="xsd:string"/>
10       <born xsi:type="xsd:date">
```

```
      </person>
    </xf:instance>
...
</html>
```

1.6.2. Instancias de datos XForms

Para crear instancias de modelos y permitir que estas sean editables existen varios elementos de control, conocidos también como XForms Controls. Los elementos de control son: input, submit, textarea, secret, output, trigger, upload, select1, select y range. Probablemente el más usado es input, que permite definir un campo de texto donde ingresar el valor de una de las variables terminales de un modelo. El fragmento de código 1.4 ejemplifica el uso de un elemento de control para editar el nombre de una persona según el modelo visto en el fragmento de código 1.3.

Código 1.4: Ejemplo de un documento XML generado por XForms

```
<person>
  <fname xsi:type="xsd:string"/>
  <lname xsi:type="xsd:string"/>
  <born xsi:type="xsd:date">
</person>
```

El atributo `@ref` permite relacionar el campo editable con el atributo `fname` del modelo que a representa las personas. El valor "fname" corresponde a una sentencia en el lenguaje XPath que permite identificar el nodo en el árbol XML, que se presenta en el fragmento de código 1.5, asociado al documento emitido al enviar el formulario. También se podría haber puesto una ruta absoluta como "/person/fname/".

Código 1.5: Ejemplo de un XForms Control

```
<xf:input ref="fname">
  <label>First Name</label>
</xf:input>
```

1.6.3. Interfaz de usuario

La interfaz usuaria se logra mediante un intérprete, típicamente programado proceduralmente, que se encarga de analizar los modelos e instancias de datos definidas de un documento para, en base a ello, entregar una interfaz que permita editar los datos de acuerdo a los modelos propuestos y se haga cargo del procesamiento y envío de ellos.

Inicialmente, se proponía que el intérprete de XForms debería ser nativo de los navegadores, no obstante, debido a las diferencias de soporte a XForm por los distintos navegadores, varias de las implementaciones hacen uso, tanto de procesamiento por el lado del servidor, como de JavaScript por el lado del cliente.

1.6.4. Recursividad en XForms

A pesar de que la salida de un formulario XForms es un documento XML, los documentos XForms aún no permiten capturar todas las posibilidades de un esquema lo suficientemente complejo. El problema es que XForms no provee recursividad en sus modelos de datos [16]. En efecto, para editar documentos XML con XForms, se debe proveer el modelo del documento de salida. Por ejemplo, es posible hacerlo mediante la siguiente estructura: `<A><B><B></B></B></A>`. Esta estructura indica que un elemento `A` puede contener elementos `B`. Que un elemento `B` contenido dentro de `A` puede, a su vez, contener otros elementos `B`. Pero no indica que uno de estos elementos `B` pueda contener otro elemento `B` nuevamente. Es decir, la profundidad del documento está determinada por la altura del árbol del modelo. Esto es una limitante para los documentos generables mediante XForms que impide editar cualquier documento XML de una determinada gramática (o esquema).

En el segundo caso de estudio 2, descrito en la sección 2, existe una relación recursiva de ser sub-unidad de otra unidad perteneciente al mismo organismo público. Un esquema para los organigramas de los organismos públicos podría expresar esa recursividad entre elementos y ello requeriría documentos de profundidad arbitraria, lo que se ha mostrado no ser factible usando sólo XForms.

1.7. Edición guiada por plantillas

Las plantillas permiten determinar la estructura y presentación de un documento definiendo áreas editables y secciones que pueden repetirse. Para el caso de estudio 1, descrito en la sección 2, el uso de plantillas resulta una opción habitual.

La mayoría de los procesadores de texto permiten almacenar documentos como plantillas. Editores de HTML, como Dreamweaver y FrontPage poseen lenguajes de plantillas con la funcionalidad de facilitar la edición. También existen varios lenguajes con la finalidad opuesta, es decir, desplegar datos ya disponibles sobre un documento (Tenjin, Mako, Smarty, Haml, Zope, etc.). El framework para aplicaciones web Zope entrega un lenguaje de plantillas que resulta especialmente interesante en el uso de la sintaxis de XML.

No obstante, la mayoría de estos sistemas de plantilla, cuya funcionalidad es asistir la edición de un humano, sólo permiten identificar áreas editables y áreas no editables. De este modo, su expresividad resulta muy pobre como para permitir editar datos estructurados como un árbol XML.

El navegador y editor de documentos Amaya [17], creado por el Consorcio de la Web y el Inria, aparte de permitir la edición de documentos XHTML de manera estructurada [18], incorpora un lenguaje de plantillas, XTiger [19], que resuelve el problema de generar datos estructurados como un árbol. La flexibilidad de las plantillas XTiger lo convierten en una herramienta útil para editar documentos con datos semánticos o microformatos [20].

XTiger es la base de la idea que se propone en este documento. En lo que sigue de está sección (1.7) se mostrará que XTiger permite editar documentos con la estructura requerida por el caso de estudio 1, pero no es suficiente para resolver los casos de estudio 2 y 3. En cambio, el sistema de plantillas propuesto en este documento, sí es capaz de abordar los tres casos de estudio.

El lenguaje XTiger se basa en una serie de elementos que se incorporan como nodos dentro del árbol de un documento XHTML cualquiera y que definen las áreas editables y la estructura de éstas. Tal como lo muestra la figura 1.2, el camino entre plantillas y documentos posee dos direcciones.

Una plantilla puede ser construida desde un documento cualquiera mediante un análisis de sus componentes. Amaya asiste este análisis permitiendo que los usuarios seleccionen un área que contiene datos y la asocien a alguno de los elementos de edición. Este proceso lleva a la inclusión de nodos que declaran las posibilidades de edición al árbol del documento. Aunque no necesario, también es posible hacer el proceso inverso de remoción de estos nodos, que es indicado en la figura 1.2 como síntesis. El siguiente paso en la generación de una plantilla es la generalización, es decir, eliminar valores propios del documento original y guardar el documento como una plantilla. El camino inverso consiste en la instanciación de una plantilla en un documento. En la práctica la instanciación se realiza simplemente copiando la plantilla con el nombre del documento que se va a editar. Amaya ofrece soporte

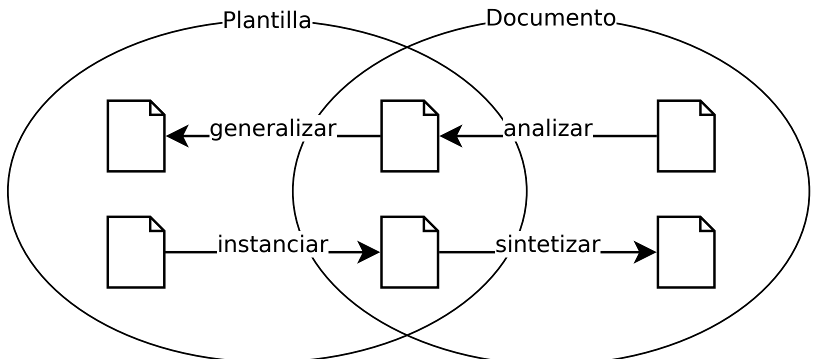


Figura 1.2: Amaya provee, tanto un mecanismo para instanciar una plantilla en un documento, como un mecanismo para generar plantillas seleccionando áreas editables en un documento.

a la instanciación al permitir la búsqueda del archivo de la plantilla en la que se va a basar el nuevo documento a crear.

En el estado intermedio, entre plantilla y documento, que se muestra en la figura 1.2, los documentos XHTML contienen elementos XTiger en la estructura del árbol XML que indican qué elementos pueden ser editados y cómo se editan. Amaya interpreta esos nodos y ofrece una interfaz de edición, siguiendo restricciones que éstos definen, de manera declarativa, sobre la estructura del documento. Dado que sólo Amaya posee soporte para los elementos de XTiger, los demás navegadores ven este documento intermedio tal cual como si se hubieran retirado los elementos de XTiger.

El funcionamiento de las plantillas XTiger se sustenta en dos grupos de elementos: una serie de formas para definir tipos e incorporar bibliotecas de tipos a una plantilla (detalladas en la sección 1.7.1) y una serie de maneras de definir campos de entrada, donde introducir variables tipadas (detalladas en la sección 1.7.2).

1.7.1. Tipos XTiger

XTiger provee cuatro clases de tipos: los tipos básicos (`number`, `boolean` y `string`), los tipos que son elementos del lenguaje (elementos tales como `h1`, `h2`, `p`, `span` o `strong` en el caso de XHTML), los tipos definidos como una unión de tipos y los tipos compuestos.

Tanto para la unión de tipos como para los tipos compuestos, XTiger provee de cons-

tructores que son estructuras XML que permiten asociar un nombre a la definición de un tipo, mediante el atributo `name`.

En el caso de la unión de tipos, el constructor es un nodo de `union` que tiene dos atributos, `include` y `exclude`, para listar los nombres de los tipos incluídos y excluidos de la unión, separados por espacios.

Para los tipos compuestos existe el elemento constructor `component`, que tal como se muestra en la figura 1.3 permite definir variables (mediante el elemento `use` que se describe en la sección 1.7.2) dentro de un fragmento de XML.

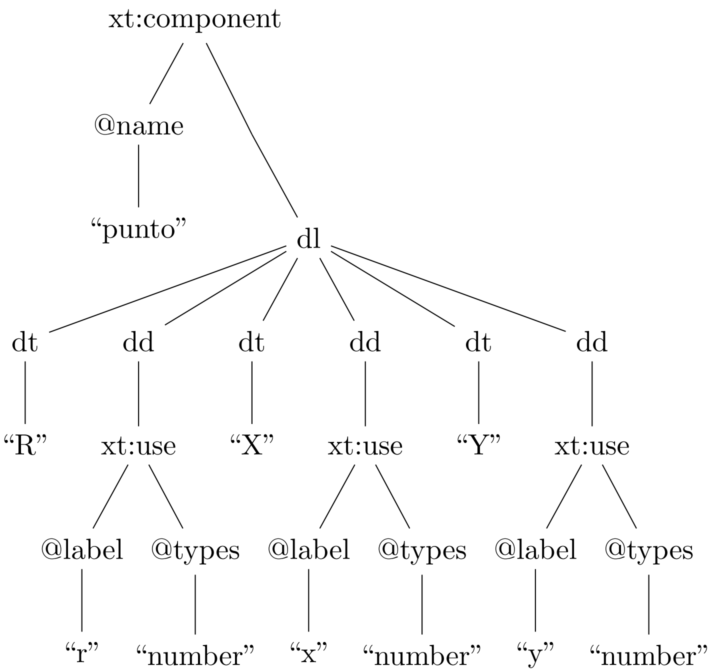


Figura 1.3: Ejemplo de construcción de tipos compuestos. Los elementos en el espacio de nombres `xt` corresponden a constructores de XTiger, mientras que los que están en el espacio de nombres por omisión son nodos de XHTML.

1.7.2. Campos XTiger

Para permitir la modificación del contenido sobre la plantilla, XTiger provee cuatro elementos de control: `use`, `bag`, `repeat` y `attribute`.

El elemento de control `use` sirve para insertar un campo de entrada para algunos de los tipos definidos previamente. Para ello se debe agregar en el lugar correspondiente del documento un nodo `use` que tenga en el atributo `types` los nombres de los atributos que queremos usar separados por espacios.

El elemento de control `bag` indica que dentro de él puede editarse el documento de manera libre, respetando el DTD del documento. Además, `bag` posee los atributos `types`, `include` y `exclude` para indicar, respectivamente, los tipos de elementos que pueden pueden ser hijos de `bag`, los que pueden ser descendientes de `bag` y los que no pueden ser descendientes de bag.

El elemento de control `repeat` permite insertar elementos de un mismo tipo varias veces. El número de veces debe estar en el rango que va de `monOccur` a `maxOccur`. El contenido de un elemento `repeat` debe ser siempre un elemento `use`.

Por último, el elemento de control `attribute` permite editar un atributo de un nodo. El atributo a editar pertenece al nodo dentro del cual se inserta en elemento de control `attribute` y su nombre debe ser igual al atributo `name` del elemento de control `attribute`.

1.7.3. Recursividad en XTiger

Al igual que XForms, XTiger no permite la recursividad. En efecto, en la especificación se indica que está prohibido incorporar un elemento creado con `component` dentro de sí mismo mediante el elemento `use`. Es decir, la definición de la figura 1.4 sería incorrecta.

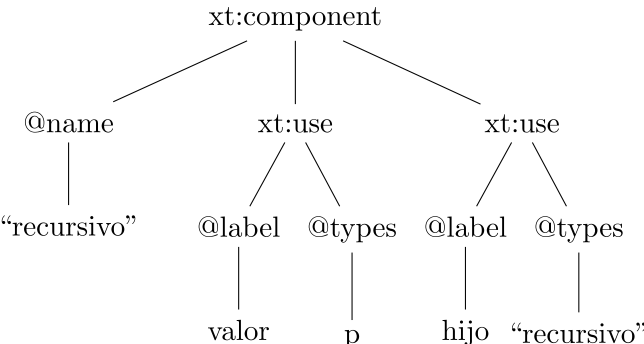


Figura 1.4: Ejemplo de recursividad prohibida en XTiger, porque produce un documento infinito.

La recursividad en muchos casos es útil, por ejemplo, en el caso de estudio 2, es necesaria para modelar la jerarquía del organigrama. Un tipo compuesto como el que se presenta en la figura 1.5 no genera necesariamente documentos infinitos, porque el valor "0" en el atributo `minOccur` abre la posibilidad de que haya un elemento de la jerarquía donde ésta termine.

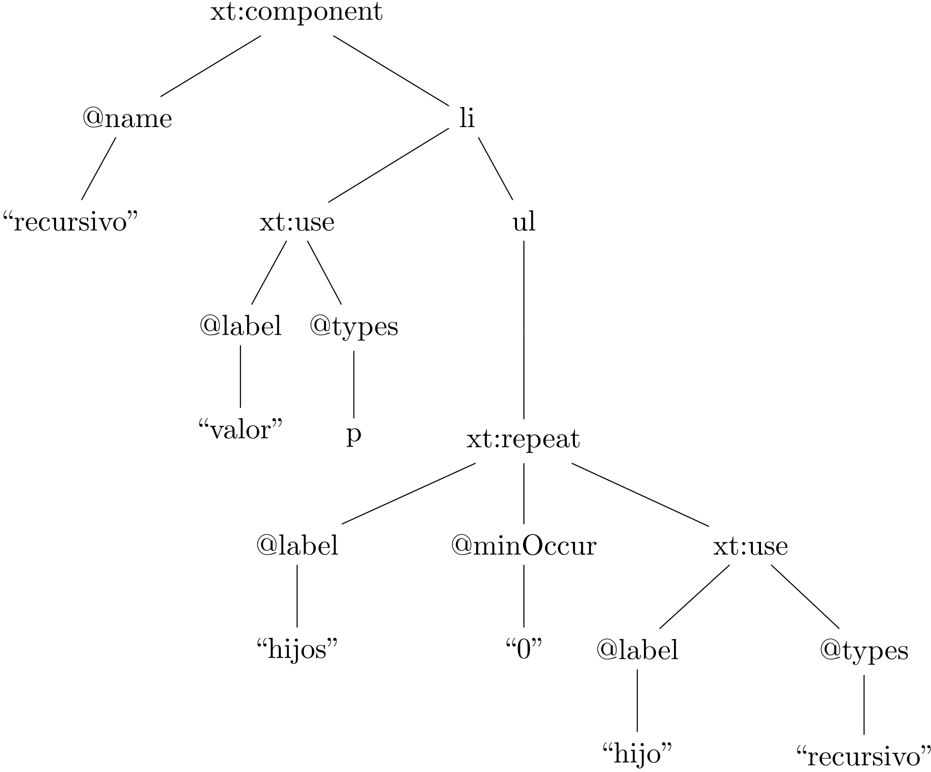


Figura 1.5: Ejemplo de recursividad prohibida en XTiger, pero que no necesariamente produce documentos infinitos.

1.8. Conclusiones

La edición de datos semi-estructurados, en particular de documentos XML, es habitualmente abordada con formularios HTML y scaffolds provistos por varios frameworks de desarrollo Web. Estas herramientas están diseñadas para trabajar con datos estructurados, por lo que no cubren completamente las funcionalidades necesarias para diseñar interfaces

amigables para los usuarios. Por consiguiente, en la práctica los formularios y los scaffolds son enriquecidos con técnicas de scripting que significan un costo extra en el desarrollo y mantención del software.

Otras herramientas para abordar modelos semi-estructurados, XForms y XTiger, tampoco poseen un lenguaje completamente declarativo para trabajar con modelos semi-estructurados y están limitadas a modelos no recursivos. Por ello, al igual que con los formularios y scaffolds, estas limitaciones implican un mayor costo en el desarrollo y mantención para aplicaciones que trabajan con datos semi-estructurados.

Las limitaciones mencionadas justifican la definición de un lenguaje declarativo para modelar la información mediante plantillas como el propuesto en el desarrollo de esta memoria.

Capítulo 2

Motivación y propuesta

Hoy en día la Web concentra un segmento importante de la edición de datos. Los frameworks de desarrollo web, junto con facilitar el manejo de los permisos de acceso y escritura de la información, deben entregar herramientas flexibles para generar vistas e interfaces de edición de los datos. Las interfaces de edición deben ser simples, para minimizar la carga que significa la capacitación de los usuarios, y deben garantizar cierto nivel de consistencia de los datos. La popularización del scripting se debe a la necesidad de generar interfaces más amigables y a evitar el flujo de ida y vuelta producido por la validación de los datos editados. Mantener la calidad del scripting, frente a los cambios frecuentes que sufren los modelos de datos, resulta una tarea costosa.

XForms fue desarrollado como una extensión a los formularios de HTML, con el objetivo de simplificar la implementación de interfaces de edición en la Web. No obstante, si bien XForms se propone como una alternativa para evitar el scripting, su lenguaje no es completamente declarativo, lo que implica que en una parte importante de las interfaces requiere ser desarrollada mediante programación procedural. Además, la complejidad de XForms, no ha permitido su amplia adopción.

La motivación del desarrollo del lenguaje Queule es construir un lenguaje simple, fácil de aprender y de usar, que sea capaz de definir interfaces de edición de manera declarativa para datos semi-estructurados a través de la Web. Se trata de definir un lenguaje que cubra las características centrales de los modelos de datos semi-estructurados, con el fin de evitar el scripting en un amplio número de casos.

En este capítulo se ejemplificarán los desafíos que exigen los modelos de datos semi-estructurados a la edición, se hará un diagnóstico de cómo estos desafíos son abordados por

herramientas como XForms y XTiger y luego se propondrá una serie soluciones que serán incorporadas en el lenguaje Queule.

2.1. Casos de estudio

Los siguientes tres casos de estudio, presentados en las secciones 2.1.1, 2.1.3 y 2.1.2, servirán como referencia para mostrar las cualidades de Queule y compararlo con las herramientas ya existentes.

Para la descripción de los casos de estudio se utilizará el lenguaje de modelado conceptual conocido como redes semánticas e introducido en [23].

2.1.1. Editor de cuestionarios

En varios ámbitos se usan cuestionarios para obtener datos con múltiples motivaciones. Aplicaciones como Google Spreadsheets y Drupal ya incluyen sus propias herramientas para definir formularios. La definición de cuestionarios o formularios no es una novedad, sino una tarea para la cual ya existen varias aplicaciones. Por ello, no hay que perder la perspectiva de que lo que se está proponiendo en esta memoria no es una herramienta para definir cuestionarios, sino una herramienta para crear un programa para definir cuestionarios de manera declarativa a través de plantillas Queule.

La figura 2.1 describe el modelo de los cuestionarios utilizando un diagrama de red semántica. Cada cuestionario (entidad Form) posee una o más secciones. Las secciones (entidad Section) se componen de preguntas (entidades RadioQuestion, CheckQuestion y OpenQuestion). RadioQuestion representa a preguntas donde sólo se puede escoger una alternativa, CheckQuestion representa preguntas donde se pueden escoger cero o más respuestas simultáneas y OpenQuestion representa preguntas abiertas, donde alguien que responda el cuestionario debiera ingresar texto. Tanto los cuestionarios, como las secciones y las preguntas poseen un título (atributo title) y una descripción (atributo description). Las preguntas poseen un peso (atributo weight) que, en conjunto con el puntaje de las respuestas (atributo score de la entidad Alternative) se usa para ponderar los resultados de las cuestionarios. Sólo dos tipos de preguntas poseen alternativas (entidad Alternative), cada una de las cuales se forma de una etiqueta (atributo label) y un puntaje (atributo score). El orden en el que se presentan las componentes de los cuestionarios importa, por ello las relaciones entre las entidades del modelo son siempre marcadas como ordenadas.

2.1.2. Editor de organizaciones

Los organismos públicos de Chile hoy están obligados a publicar sus organigramas. Se supondrá que ello significa nombrar las unidades dentro del organismo, agregar una breve descripción por cada una de ellas, y enumerar las sub-unidades que componen cada unidad (que a su vez podrían incluir recursivamente otras sub-unidades).

La figura 2.2 muestra como las organizaciones pueden componerse de sub-organizaciones. El desafío central de este modelo es la recursividad, por ello sólo se han agregado los atributos name, code y description para darle sentido al modelo.

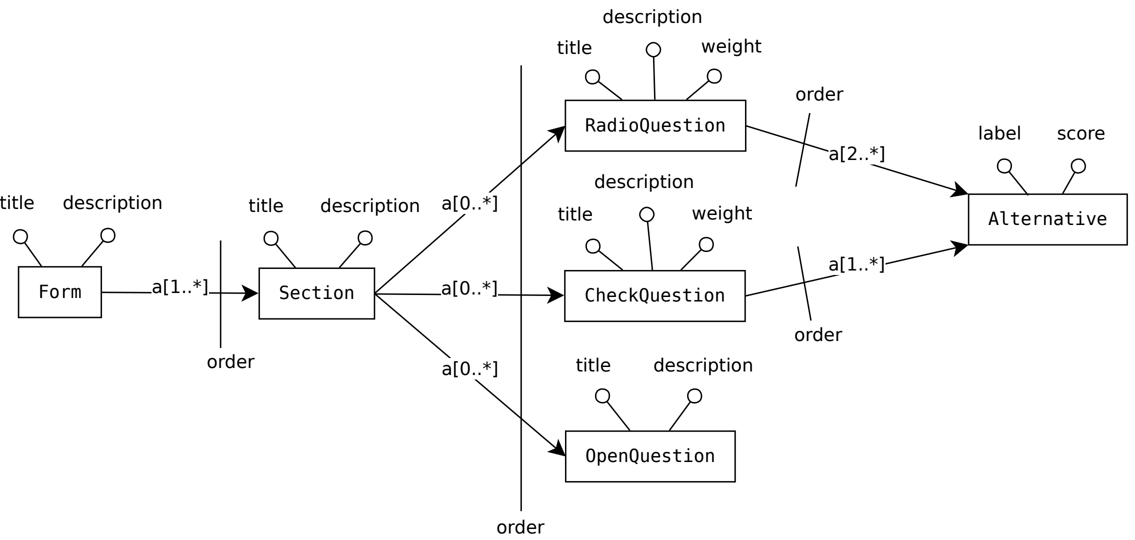


Figura 2.1: Diagrama de red semántica para los cuestionarios

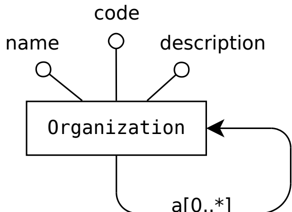


Figura 2.2: Diagrama de red semántica para organizaciones

2.1.3. Editor de segmentación de locales

La medición de cumplimientos de estándares de calidad en locales de una cadena de supermercados[1] requiere estos sean segmentados en clases no necesariamente disjuntas. Estas clases permiten a los directivos y al personal encargado, visualizar el estado de los locales en general en reportes que utilizan dichas segmentaciones.

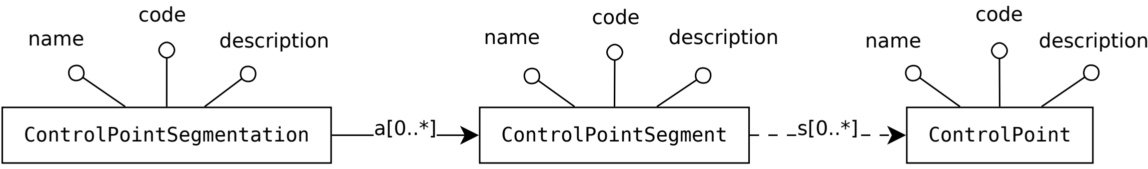


Figura 2.3: Diagrama de red semántica para la segmentación de locales

La figura 2.3 muestra las tres componentes del modelo de segmentación: las segmentaciones (entidad ControlPointSegmentation), los segmentos dentro de cada segmentación (entidad ControlPointSegment) y los locales (entidad ControlPoint). Cada una de las tres entidades es caracterizada por un nombre (atributo name), un código (atributo code) y una descripción (atributo description).

El desafío que impone la segmentación es que el modelo posee dos entidades en las que no están anidadas una dentro de la otra, como en los casos anteriores. La no anidación implica que podemos agregar o borrar instancias de cada una de las clases al mismo tiempo que debe existir un mecanismo por el cual éstas se puedan asociar.

En la práctica se supone que los locales son asociados a uno y no más de uno de los segmentos dentro de cada segmentación, o dicho de otro modo, que cada segmentación debe ser una partición completa y disjunta del conjunto de locales. No obstante, este tipo de restricciones no puede ser expresado por un lenguaje de modelado conceptual como el usado en el diagrama de redes semánticas. En general los lenguajes conceptuales no suelen tener herramientas para modelar restricciones tan finas, por lo cual, el lenguaje Queule tampoco las aborda.

[1] La motivación de este caso de uso surge de una aplicación desarrollada por el estudiante, aplicada a la medición de estándares en terreno y que, entre otros usos, está el de la evaluación de locales de cadenas de supermercados.

2.2. Diagnóstico para herramientas existentes

Los casos de estudio descritos en la sección 2, presentan funcionalidades que se les puede exigir a los dos sistemas presentados, XForms y XTiger. Se trata de casos particulares para modelos que pueden ser propuestos utilizando lenguajes de modelado conceptual. En esta ocasión, se escogió el lenguaje de redes semánticas, pero podría haberse escogido otro cualquiera que permita modelar datos semi-estructurados. El más conocido de los lenguajes de modelado conceptual es el Entidad-Relación, ampliamente usado en el modelado de datos estructurados, como es el modelo relacional. Una serie de lenguajes de modelado conceptual para datos semi-estructurados puede encontrarse en [24] (que es un resumen de [25]), tales como ER-B [26], EReX [27], X-Entity [29], ORA-SS [28] y las redes semánticas [23], entre otros.

El caso de estudio "editor de cuestionarios" (sección 2.1.1) presenta como desafío el poder agregar repeticiones de elementos compuestos y con la posibilidad de escoger entre más de un tipo de elementos (los distintos tipos de pregunta). Tanto XForms como XTiger son capaces de editar ese tipo de datos. XForms lo permite hacer mediante el elemento `repeat` y triggers que ejecuten acciones como `delete` e `insert`, tal como se muestra en [22] para editar una jerarquía de bookmarks. XTiger tan sólo requiere del elemento `repeat`.

El desafío del segundo caso de estudio, "Editor de organizaciones", es la incorporación de estructuras recursivas. Tal como se presentó en las secciones 1.6.4 y 1.7.3, ni XForms ni XTiger permiten la recursividad. Luego, no permiten abordar ese modelo.

En el tercer caso de estudio, "Editor de segmentación de locales", se requiere que para cada local se pueda elegir un conjunto de segmentos, definidos en el mismo documento. Es decir, los usuarios debieran poder agregar segmentos, agregar locales y asignar locales a los segmentos. Se podría esperar que el resultado de dicha edición sea un documento como el mostrado en el fragmento de código 2.6, donde el local 234 es referenciado por el segmento de locales con góndolas usando el atributo code del local como llave de asociación.

Código 2.6: Documento posible para la edición de segmentos

```
1 <Document>
2   <ControlPoint>
3     <name>Local 234</name>
4     <code>L234</code>
5     <description>Local con góndolas ubicado en la avenida...</description>
```

```
6      </ControlPoint>
7      ...
8      <ControlPointSegmentation>
9        <name>Tipo</name>
10       <code>Tipo</code
11       <description>Segmenta entre locales con y sin góndolas.</description>
12       <ControlPointSegment>
13         <name>Locales con góndolas</name>
14         <code>locales-con-gondolas</code
15         <description>Locales con góndolas.</desctiption>
16         <ControlPoint>L234</ControlPoint>
17         ...
18       </ControlPointSegment>
19       <ControlPointSegment>
20         <name>Locales sin góndolas</name>
21         <code>locales-sin-gondolas</code
22         <description>Locales sin góndolas.</desctiption>
23         ...
24       </ControlPointSegment>
25     </ControlPointSegmentation>
26     ...
27   </Document>
```

La edición sería posible, tanto con XTiger como con XForms, si es que los locales se mantuvieran constantes, pudiéndose editar sólo las segmentaciones, o si las segmentaciones se mantuvieran constantes, pudiéndose editar sólo los locales.

En XTiger, asociar un valor con el otro significa anidar uno dentro del tipo del otro. Por ejemplo, dentro del tipo que define los locales se podría seleccionar el segmento al que pertenecen. Ello requiere que la definición del tipo que define los locales contenga una estructura para seleccionar segmentos, pero el modelo implica que si se agrega un nuevo segmento al documento, esa estructura debiera cambiar, lo que no es posible, porque en XTiger los tipos son inmutables y libres de contexto.

En primera instancia se podría pensar que para XForms el problema de asociar locales a un segmento puede ser resuelto mediante un elemento del tipo select, como el que muestra el fragmento de código 2.7, en el cual cada componente representa un local que puede estar o no dentro del segmento. Sin embargo, esta solución no funciona, porque si se agrega un local en la lista de locales, entonces las listas de selección para cada segmento no serían

actualizadas, al menos en forma declarativa.

Código 2.7: Interfaz para escoger valores desde un conjunto en XForms

```
<select ref="ControlPoint">
  <label>Seleccione los locales en el segmento:</label>
  <item>
    <label>Locak 234</label>
    <value>L234</value>
  </item>
  <item>
  ...
  </item>
  ...
</select>
```

En resumen, se ha visto que ni XTiger ni XForms permiten trabajar satisfactoriamente los dos últimos casos de estudio planteados. Dado que XTiger es la herramienta de plantillas motivadora de la solución propuesta (el lenguaje Queule), en lo que sigue de esta memoria, las comparaciones tenderán a hacerse entre XTiger y Queule.

A pesar de que Amaya y XTiger entregan una manera sencilla de simultáneamente escribir documentos e ingresar datos, existen las otras limitaciones para su uso, aparte de las ya descritas con respecto a casos de estudio:

- No se puede crear un documento XML arbitrario, dado que no se cuenta con una manera de visualizarlo. Aunque XTiger está diseñado para trabajar con múltiples tipos de documentos, en la práctica sólo trabajamos con XHTML.
- Se deben introducir atributos de una manera distinta a la que usamos para ingresar el texto dentro de un nodo, lo que resulta una complejidad innecesaria para alguien sin los suficientes conocimientos de XML.
- Los elementos extra (que no corresponden a los datos) se desligan de la plantilla al momento que ésta es instanciada en el documento, lo que implica que si más tarde se cambia la plantilla, los documentos ya creados no sufrirán los cambios deseados. En cambio, cuando usamos un sistema de plantillas para generar documentos a partir de los datos, cambiar la plantilla es suficiente para modificar la presentación de todos los recursos.

2.3. Lineamientos del diseño de Queule

Tal como se vio en la sección 1, XTiger es un lenguaje que presenta una simpleza que justifica la incorporación de modificaciones con el fin de cumplir con los requerimientos propuestos en los tres casos de estudio de la sección 2. El cuadro 2.1 muestra cómo el uso de lenguajes de plantillas puede ser una alternativa al uso de esquemas, comparable a lenguajes tales como XML Schema [35], RELAX NG [36] y Schematron [37], en la expresividad para definir familias de documentos XML. El lenguaje XForms ha sido catalogado en la columna de los lenguajes de plantillas debido a que los modelos guardan una estrecha relación con la definición de tipos compuestos de XTiger y los XForms Controls permiten incorporar estas estructuras en una interfaz de edición de la misma forma que XTiger lo hace con el elemento `use`.

Cuadro 2.1: Modelos plantillas y por esquemas

Modelo por plantillas orientado a la edición	Modelo por esquemas orientado al almacenamiento	
Documentos XHTML	Documentos XML, JSON, YAML	Bases de datos Relacionales, de grafos, de documentos
Plantillas Queule, XTiger, XForms	Esquemas XML Schema, RELAX NG, Schematron	Esquemas Tablas, vocabularios RDF, esquemas de documentos

El objetivo va más allá de sólo facilitar la edición de documentos XML. Queule debiera ser un lenguaje que permita editar, tanto otros tipos de documentos que contengan datos, como YAML [38] y JSON [39], como bases de datos en general. En resumen, dichas modificaciones deberían permitir:

1. Trabajar con estructuras recursivas.

2. Generar documentos que no necesariamente sean XHTML.

3. Evitar que el usuario tenga que diferenciar entre nodos de texto y atributos.

4. Separar los contenidos de la presentación.

5. Permitir la asociación de elementos editables dentro de la plantilla.

6. Contar con un conjunto de tipos básicos atómicos y la posibilidad de extenderlos, de una manera similar a la de XML Schema y XForms.

2.4. Incorporación de transformaciones

En los principios de la Web, el editor WWW permitía tanto navegar por los documentos como editarlos [40]. Amaya también permite guardar documentos cuando interactúa con un servidor WebDAV. La figura 2.4 diagrama esta forma de edición.

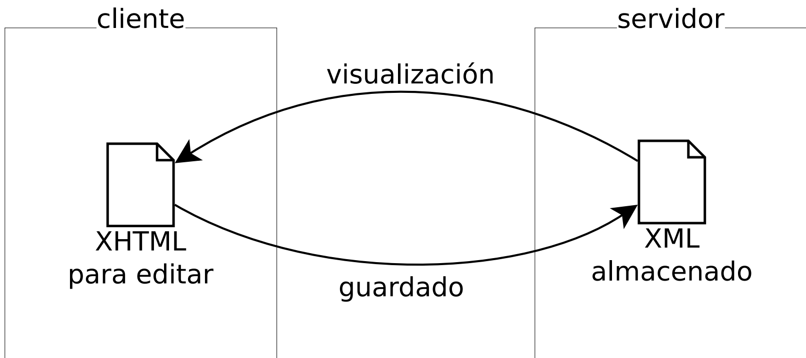


Figura 2.4: Un documento puede ser visto y editado por Amaya.

El uso de Amaya y sus plantillas XTiger, si bien es una alternativa para la edición de microformatos [20], no resulta completamente viable para el marcado de documentos con RDFa. Junto a las limitaciones vistas en la sección 2.2 se agrega que RDFa muchas veces exige a los usuarios ingresar datos como atributos. Esta diferenciación entre atributos y componentes de texto agrega una complejidad extra a los usuarios.

Como solución a este problema se planteó la posibilidad de introducir dos modificaciones al esquema presentado en la figura 2.4: reducir los elementos editables en el lenguaje de plantillas y aplicar transformaciones para transformar los datos contenidos en un documento XML cualquiera a un documento para editar y desde el documento a editar hacia el documento a salvar. De este modo, a diferencia del diagrama 2.4, las flechas del diagrama 2.5

representan transformaciones. Estas transformaciones permitirían convertir un documento XML almacenado en un documento XHTML con elementos del lenguaje de plantillas Queule incorporados para poder editar sobre él y, de igual modo, llevar el documento ya editado, de vuelta al espacio original.

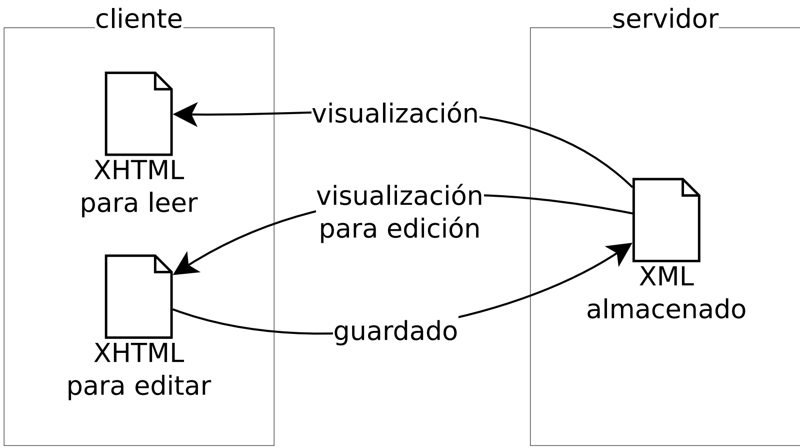


Figura 2.5: Transformaciones para guardar un documento en un formato distinto al que se está editando

La introducción de transformaciones permite satisfacer los puntos 2, 3 y 4 de la motivación. En efecto, a través de la transformación los documentos almacenados no son necesariamente XHTML, no es necesario lidiar con atributos, debido a que lo que se codifica como atributos en el documento XML puede codificarse como nodos de texto el documento XHTML y no es necesario guardar los elementos de presentación, pues basta incorporarlos al momento de la visualización.

Existen dos lugares donde las transformaciones pueden ser realizadas: en el lado del servidor o en el lado del cliente. Hoy la mayoría de los navegadores soportan transformaciones XSLT [31] (Amaya no), lo que permite visualizar directamente documentos XML. Además de XSLT, han surgido otros lenguajes para transformaciones unidireccionales entre documentos XML, como XDuce [32]. El uso de JavaScript permite implementar otros tipos de transformaciones, distintas a las soportadas nativamente por los navegadores, y permite que estos estén disponibles en una amplia gama de navegadores.

Lenguajes bidireccionales como biXid [33] son una alternativa a usar dos transformaciones unidireccionales. El beneficio es ahorrarse la escritura del código dos veces.

Al incluir la posibilidad de que los documentos editados no se guardasen necesariamente como el documento XML resultante de una transformación aplicada al documento editado, abriéndose la posibilidad de utilizar otros formatos, como JSON o YAML, o incluso bases de datos, la definición de una forma de transformar documentos perdió importancia en el desarrollo de esta memoria. Existen demasiadas formas de transformar los documentos por lo que el intentar definir una no tiene sentido. Además, las transformaciones constituyen una problemática independiente a la edición de datos semi-estructurados utilizando plantillas.

2.5. Asociación de elementos

El caso de estudio descrito en la sección 2.1.3, "editor de segmentación de locales", presenta como desafío la asociación de componentes que son editadas en distintas partes del documento (véase también la sección 2.2). Sea cual sea la interfaz diseñada para asociar los elementos, ésta deberá ofrecer una lista con los elementos asociables al usuario, ya sean locales o segmentos. Como estos elementos podrían cambiar a medida que el usuario edita el documento, el sistema debería estar atento a ello y actualizar las listas de selección.

El problema de asociar elementos está parcialmente resuelto en las planillas de cálculo, donde se puede indicar que ciertas celdas deben tomar sus valores escogiendo un valor de entre los contenidos en otros conjuntos de celdas. El uso de las planillas de cálculo es bastante extendido, aunque las tablas no se adaptan bien a gran parte de los modelos que allí se representan y aunque los archivos no entregan información sobre su estructura, para que ésta pueda ser analizada de manera automática.

El uso de las planillas de cálculo motiva la solución propuesta en el caso de las plantillas sobre XHTML. Para referenciar un conjunto de celdas las planillas usan una cadena como A3:C24;D2:E21;G4, que representa el conjunto de celdas resultante de unir los conjuntos asociados a las áreas rectangulares indicadas por cada una de las expresiones separadas por punto y coma. La expresión G4 representa un área rectangular que sólo contienen la celda de columna G y fila 4. De igual manera, A3:C24 representa el conjunto de celdas con columna entre A y C y filas entre 3 y 24.

Para seleccionar un conjunto de elementos en XML disponemos de dos tipos de herramientas: los selectores y los lenguajes de consulta. Los selectores son lenguajes como XPath [41] y CCS Selectors [42], que permiten referenciar nodos dentro de un documento, mientras que los lenguajes de consulta (o procesamiento) son lenguajes como XQuery [43], XSL [30], XQL [44], entre otros, que agregan expresividad para seleccionar elementos y procesarlos.

La asociación de elementos es sin duda una funcionalidad deseable para un lenguaje de plantillas, sin embargo, por falta de tiempo y para liberar la primera versión de Queule lo más pronto posible, la versión 1.0 de la especificación (ver sección 4) no considerará la asociación entre elementos. No obstante, para la siguiente especificación se recomienda hacer uso de selectores y en especial de los selectores CSS, debido a que los selectores son más sucintos, lo que facilita el ser incorporados como atributos, y porque los selectores CSS han ido ganando terreno en varias bibliotecas JavaScript, como JQuery [45] y Prototype [46], que han pasado de soportar XPath a soportar CSS Selectors 3.0.

A pesar del mayor poder expresivo de XPath para la selección de elementos de un documento, éste no se recomienda para el uso en documentos XHTML por ser más complejo en varias de las tareas más comunes (ver cuadro 2.2) porque algunas de sus funcionalidades, como la de buscar nodos dados sus hijos, imponen una carga de cómputo inconveniente para los navegadores. Además los selectores CSS proveen métodos para capturar eventos que no tienen un símil en XPath, como los selectores `E:link`, `E:visited`, `E:active`, `E:hover` y `E:focus`, que son necesarios para definir cómo interactuar con los documentos durante la navegación.

Cuadro 2.2: CCS Selectors 3.0 versus XPath

Función	CSS 3	XPath
Todos los elementos	`*`	`//*`
Todos los elementos `p`	`p`	`//p`
Todos los hijos de un elemento `p`	`p > *`	`//p/*`
Todos los descendientes `a` de un elemento `p`	`p a`	`//p//a`
Elementos con un ID específico	`\#foo`	`//*[@id=’foo’]`
Elementos por clase	`.foo`	`//*[contains(@class,’foo’)]` [a]
Elementos por atributo	`*[title]`	`//*[@title]`
Primer hijo de cada elemento `p`	`p > *:first-child`	`//p/*[0]`
Todos los elementos `p` que tengan un hijo `a`	no es posible	`//p[a]`
Elemento siguiente	`p + *`	`//p/following-sibling::*[0]`
Elementos `p` con contenido en español	`p:lang(es)`	`//p[ @xml:lang = "es" ]`
Selecciona cualquier elemento `p` inmediatamente precedido de un elemento `h1`	`h1 + p`	`//h1/following-sibling::*[1]/self::p`
Selecciona elementos `p` con el atributo `foo` asignado, cualquiera sea su valor.	`p[foo]`	`//p[ @foo ]`

[a] Esto no es completamente cierto porque también puede seleccionar “foobar” cuando sólo queremos seleccionar elementos de la clase “foo”. Una consulta exacta requiere expresiones algo más complejas para obtener ese valor. Dado que el uso de las clases es muy extendido en XHTML, esto resulta una razón fuerte en favor de los selectores CSS.

2.6. Estructuras recursivas por construcción

A pesar de que los modelos, como el del caso de estudio "editor de organismos" (ver sección 2.1.2), son de naturaleza recursiva, XForms y XTiger no soportan la recursividad. El problema es que, en algunos casos la recursividad obliga a desplegar formularios o plantillas infinitas (ver secciones 1.6.4 y 1.7.3). Para evitar estas estructuras infinitas, que colapsarían los navegadores, existen dos estrategias:

1. Detectar la infinitud de una estructura para detener el despliegue del formulario o plantilla.
2. No permitir el despliegue de estructuras por construcción.

En XTiger el tipo de estructura recursiva más simple es una donde se define una componente que se usa directamente a si misma. Es decir, una estructura de la forma

$$\text{component}(A) \text{ ——— } \text{use}(A)$$

donde component(A) representa a la definición del tipo compuesto A mediante el elemento component de XTiger y use(A) representa un campo donde A puede usarse dentro del árbol que define al tipo compuesto A. La línea representa la anidación en uno o más niveles del árbol XML.

También pueden haber recursividades infinitas construidas a lo largo de la definición de varias componentes, por ejemplo:

$$\text{component}(A) \text{ ——— } \text{use}(B)$$
$$\text{component}(B) \text{ ——— } \text{use}(A)$$

La única forma de que la estructura permita documentos que no sean infinitos, es que en el camino entre la definición de la componente y el nodo de uso de la componente haya al menos un nodo de repetición, que tenga como propiedad que la mínima cantidad de ocurrencias sea 0. Es decir, que la estructura sea realmente así:

$$\text{component}(A) \text{ —— } \text{repeat(minOccurr=0)} \text{ ——— } \text{use}(A)$$

Luego un algoritmo que evalúe si una plantilla XTiger es finita o infinita, deberá recorrer todos estos caminos para buscar ciclos problemáticos. Es necesario remarcar que esta búsqueda no necesita ser exhaustiva sobre todas las combinaciones posibles que se abren con los elementos unión. En efecto, basta con encontrar una de las alternativas que nacen de una unión con una valor repeat(minOcurr=0) para que exista la posibilidad de darle un fin al documento.

component(A) —— use(D)

union(D) —— use(C)
union(D) —— use(B)

component(B) —— repeat(minOccurr=0) —— use(A)

component(C) —— repeat(minOccurr=1) —— use(A)

En el ejemplo anterior, la posibilidad de siempre escoger la componente B permite al documento finalizar, a pesar de que si siempre se escoge C resulte obligatorio volver a A, una y otra vez.

En Queule se ha optado por la segunda estrategia: evitar las estructuras infinitas por construcción. Para ello se debe tener en cuenta que cada tipo compuesto está definido por tipos terminales y por otros tipos compuestos. En esa definición los tipos pueden aparecer instanciados o no instanciados (ver el cuadro 2.3). Por ejemplo, si tenemos que A se incorpora directamente sí mismo, deberá haber un árbol T que defina cómo se estructura A y en cual aparecerá nuevamente A. Sin embargo, aunque el A anidado vuelva a contener otra vez a A, esto no puede continuar indefinidamente porque T es un árbol finito. Así, deberá haber un conjunto finito de elementos A anidados que no han sido instanciados y que por ende corresponden a terminales en la recursión.

Cuando en una plantilla Queule se cita un elemento A, al agregarlo lo que se hará es copiar el árbol T que define a A dentro del elemento que lo cita. Como T es finito, el documento resultante continuará siendo finito, no obstante el usuario podrá indefinidamente anidar nuevos elementos A.

Cuadro 2.3: Elementos instanciados y no instanciados en Queule

<table>
<tr><th>Modo</th><th>Código</th></tr>
<tr><td>Instanciado</td><td><pre>[frame=single,numbers=left]
<use types="A" /></pre></td></tr>
<tr><td>No instanciado</td><td><pre>[frame=single,numbers=left]
<use types="A">
  <yield type="A">
    <use types="A"/>
  </yield>
</use></pre></td></tr>
</table>

2.7. Sistema de tipos

La ventaja de utilizar atributos fuertemente tipados es que ello permite la validación de los datos editados en el cliente y, por ende, disminuye el flujo de datos con el servidor para la validación. Para cumplir con este objetivo, es recomendable que el sistema de tipos sea tan expresivo como el de XForms. XTiger posee también un sistema de tipos, pero que sólo tiene tres tipos básicos: string, number y boolean. En Queule, además querríamos que el conjunto de tipos sea extensible, como lo es en XForms. Sin embargo, la complejidad de definir un sistema de tipos más amplio pone fuera esa característica de la especificación de la versión 1.0 de Queule, que es la que se desarrolla en esta memoria.

Junto con la definición de tipos, Queule propone que debe haber una manera de especificar la forma en que estos tipos sean capturados, es decir, por ejemplo, si se quiere usar un área de texto o si basta con un campo de texto. Para la especificación de Queule 1.0, se proponen sólo dos tipos: (string, textfield) y (string, textarea). Dada la modularidad de la implementación la extensión no será difícil para las próximas versiones de Queule.

Capítulo 3

Objetivos y metodología

3.1. Objetivo general

Este trabajo busca definir un sistema para editar documentos XML, basado en plantillas y transformaciones. Además, se pretende implementar un editor como prueba de concepto para un conjunto razonable de las funcionalidades discutidas.

3.2. Objetivos específicos

- Especificar los elementos e interpretación del lenguaje de plantillas a usar.
- Implementar una aplicación para interpretar, manipular y generar documentos a partir de las plantillas.
- Plantear métricas para comparar la solución implementada con otros enfoques existentes, en especial con XForms y XTiger.
- Utilizar esta implementación en al menos una aplicación real que pueda ser usada por un público amplio.

3.3. Metodología

La metodología seguida en el desarrollo de esta memoria se compone de las siguientes partes:

1. Plantear un conjunto de casos de estudio que sirvan para clarificar las características desafiantes que se desean implementar en el lenguaje de plantillas a proponer.

2. Estudiar el estado del arte en la edición de datos y distinguir los competidores más cercanos de la solución propuesta.

3. Especificar el lenguaje propuesto, Queule.

4. Implementar un conjunto de plantillas para los casos de estudio y un intérprete del lenguaje Queule que permita editar datos usando las plantillas.

3.3.1. Casos de estudio

El conjunto de casos de estudio a desarrollar fue descrito en la sección 2.1. Las características desafiantes que incluye cada uno de los casos de estudio son: edición de datos semi-estructurados (árboles de altura acotada), edición de estructuras recursivas (árboles que pueden crecer ilimitadamente) y asociación de elementos definidos en partes distintas de un documento (grafos dirigidos).

3.3.2. Estado del arte

Se hizo una amplia investigación que va desde herramientas para editar datos estructurados como los scaffolds, descrita en la sección 1.4, a la edición de datos semi-estructurados utilizando formularios y plantillas, descritas en las secciones 1.5, 1.6 y 1.7.

Del estudio realizado se concluyó que los competidores más cercanos a la solución propuesta son XForms y XTiger.

3.3.3. Especificación del lenguaje propuesto Queule

El capítulo 4 presenta la especificación del lenguaje propuesto, Queule, para su versión 1.0. La metodología empleada en la especificación fue la especificación de cada uno de los elementos del lenguaje usando DTD y describiendo informalmente los usos de cada uno de los elementos y sus atributos.

3.3.4. Implementación

La implementación de las plantillas para los casos de estudio y la implementación del intérprete del lenguaje Queule pueden encontrarse en el repositorio del proyecto [1]. En ese sitio pueden además encontrarse punteros a ejemplos del uso del lenguaje Queule.

Para la implementación del lenguaje Queule se decidió usar el lenguaje JavaScript, con la intención de que la información pueda ser editada usando una amplia gama de navegadores y sin necesidad de que los usuarios deban instalar algún software extra en sus computadores. Esta estrategia es similar a la seguida en varias de las implementaciones de XForms, como por ejemplo, la que usa betterForm al basarse en JavaScript. Además, se eligió la biblioteca Prototype [46], que permite un acceso simplificado al DOM de los documentos y la definición de clases y métodos para la aplicación de técnicas de orientación a objetos.

Parte II

Desarrollo

Capítulo 4

Especificación de Queule 1.0

El lenguaje Queule está inspirado en XTiger, por lo que toma varios de sus elementos, los simplifica y cambia levemente sus funciones.

En cada una de las siguientes secciones de esta implementación se presentará un elemento del lenguaje (excepto en la que describe los tipos básicos), se definirá la estructura del elemento mediante DTD y luego se precederá a describir las funcionalidades y usos de cada uno de los atributos y del contenido.

4.1. Tipos básicos

XTiger define los tipos `number`, `boolean` y `string` como tipos básicos de XTiger, no obstante, a diferencia de XTiger, Queule está diseñado para entregar una interfaz rica a los usuarios, por lo que el conjunto de tipos básicos debiera ser extensible. Sin embargo, para la implementación propuesta en esta memoria sólo se van a considerar los tipos básicos `text` y `textarea`, asociados a las interfaces de XHTML: textfield y textarea.

4.2. Elemento `library`

Este elemento sirve para colocar definiciones de tipos compuestos usados en la plantilla. En XHTML se recomienda poner este elemento dentro del elemento `head`.

La resolución de conflictos de nombres de componentes se hace mediante la siguiente regla: si un tipo es redefinido (se crea otro con el mismo atributo `@name`) el segundo será ignorado.

```
<!ELEMENT library ((component | import)*) >
<!ATTLIST library
    version         CDATA    #REQUIRED>
```

Atributos

`@version`
: Versión del lenguaje Queule usado en esta plantilla. El lenguaje de un atributo de esta versión debe ser "1.0". Este atributo es obligatorio.

Contenido
: El elemento `head` puede contener cero o más elementos `component` o `import` pero no puede contener otros elementos.

4.3. Elemento `import`

Cuando una plantilla o una librería es construida usando tipos definidos en otra librería, esta librería debe ser explícitamente importada. La semántica de import es la de incluir los contenidos de la librería importada en el mismo lugar donde se inserta el elemento `import`. Esto además, permite la resolución del conflicto de nombres de componentes, es decir, cuando dos poseen el mismo nombre, se procede siguiendo la práctica descrita en la sección 4.2.

Para evitar las inclusiones cíclicas de una librería en sí misma se establece la siguiente regla: toda librería debe ser insertada sólo una vez. Es responsabilidad del intérprete de la plantilla el controlar que una biblioteca no sea insertada más de una vez, lo que puede ser resuelto llevando una lista de las bibliotecas ya importadas. Si una biblioteca es importada más de una vez, la regla para definir cuándo será importada es la misma de la

que regula los conflictos de nombres de las componentes, es decir, sólo la primera aparición será considerada.

```
<!ELEMENT import EMPTY >
<!ATTLIST import
    src    CDATA    #REQUIRED>
```

Atributos

`@src`
: URI de la biblioteca importada. Este atributo es obligatorio.

Contenido
: Vacío.

4.4. Elemento `yield`

El elemento `yield` se usa para indicar un campo editable dentro del elemento `use` para indicar el lugar en la estructura donde se inserta el tipo a usar. Ver las sección 4.5 para una descripción más detallada.

```
<!ELEMENT yield ANY>
<!ATTLIST yield
    type   NMTOKEN    #REQUIRED
    delete CDATA      #IMPLIED "true">
```

Atributos

`@type`
: El atributo `@type` contiene el nombre del tipo del elemento actual que se ha insertado en un campo.

`@delete`
: Indica si un elemento puede ser borrado (true) o no (false).

Contenido
: En el contenido del nodo puede ir un valor que debe ser instancia del tipo indicado en el atributo @type, ya sea un tipo básico o un tipo compuesto.

4.5. Elemento `use`

El elemento `use` sirve para definir campos que se pueden llenar usando uno o más (dependiendo del atributo `@repat`) de entre un conjunto de tipos posibles.

```
<!ELEMENT use ((yield)*)>
<!ATTLIST use
    types   CDATA    #REQUIRED
    repeat  CDATA    #IMPLIED "1"
    label   CDATA    #IMPLIED>
```

`@types`
: El atributo `@types` debe contener una lista de uno o más nombres separados por espacios. Cada nombre debe identificar a un tipo básico o a un tipo compuesto. De este modo, esta lista de tipos define los tipos posibles a usar para llenar este campo.

`@repeat`
: Número máximo de instancias que pueden ser ingresadas en el campo. Por omisión toma el valor "1" y se usa el valor "*" para expresar que la cantidad no se encuentra restringida.

`@label`
: Texto que aparece como mensaje para indicar al usuario la posibilidad de agregar un elemento. De no indicarse, el intérprete podrá agregar un mensaje genérico como "Agregar un nuevo elemento".

Contenido
: Debe contener cero o más elementos `yield`.

4.6. Elemento `component`

El elemento `component` es un constructor que crea un nuevo tipo construido por una estructura XML que combina otros tipos. El atributo `@name` de este elemento es obligatorio y debe ser único dentro de la plantilla, pues es el que permite que el tipo definido sea invocado en los elementos `use`.

La estructura XML contenida por un elemento `component` puede contener varios elementos `use` en su interior. A diferencia de XTiger, la recursividad es permitida, es decir, los elementos `use` pueden contener también los nombres del tipo compuesto que se está definiendo.

```
<!ELEMENT component ANY>
<!ATTLIST component
    label   NMTOKEN    #REQUIRED>
```

`name`

El nombre del tipo compuesto que se define. Es un atributo obligatorio que permite identificar los tipos definidos dentro de una plantilla. Si había un tipo previamente definido, ya sea básico o compuesto, esta componente debiera ser ignorado (ver también sección 4.2).

Contenido

El contenido de una componente es cualquier estructura XML, que podría a su vez contener elementos `use`. A diferencia de XTiger, la recursividad de componentes es permitida en Queule. Es decir, un elemento `use` podría incluir un valor del mismo tipo compuesto que se está definiendo.

Capítulo 5

Ejemplos de plantillas Queule

5.1. Ejemplo de uso con RDFa

RDFa es un sistema que permite marcar datos en forma de tripletas dentro de un documento XHTML, permitiendo que estos puestan ser extraídos del documento independientemente de su estructura. El código 5.8 ejemplifica una colección de libros.

Código 5.8: Colección de libros marcada con RDFa

```
<ul>
 <li typeof="Book">
  <dl>
   <dt>Título</dt>
   <dd property="dc:title">Don Quijote</dd>
   <dt>Autor</dt>
   <dd property="dc:creator">
    <p typeof="Person">
     <span property="foaf:name">
      José Miguel de Cervantes
     </span>
    </p>
   </dd>
  </dl>
 </li>
 <li>...</li>
</ul>
```

Una plantilla que permita editar una colección de libros, tal como se muestra en el código 5.8, puede usar los tipos `Person` y `Book` definidos, usando el lenguaje Queule, en los fragmentos de código 5.9 y 5.10. Se ha escogido el prefijo "q" para el espacio de nombres asociados al lenguaje Queule.

Código 5.9: Tipo para agregar autores

```
1 <q:component name="Person">
2  <p typeof="Person">
3   <span property="foaf:name"><q:use types="text"/></span>
4  </p>
5 </q:component>
```

Código 5.10: Tipo para agregar libros

```
1  <q:component name="Book">
2   <li typeof="book">
3    <dl>
4     <dt>Título</dt>
5     <dd property="dc:title"><q:use types="text"/></dd>
6     <dt>Autor</dt>
7     <dd property="dc:creator">
8      <q:use repeat="*" types="Person"/>
9     </dd>
10   </dl>
11  </li>
12 </q:component>
```

A partir de la definición de los tipos compuestos "Person" y "Book" se puede definir la plantilla que se muestra en el fragmento de código 5.11.

Código 5.11: Plantilla para la colección de libros

```
1 <ul><q:repeat types="Book"/></ul>
```

El ejemplo presentado muestra cómo, con pocas líneas de código declarativo, es posible definir una interfaz que permite editar una colección de libros y que ésta incorpore información semántica a través del marcado con RDFa.

5.2. Ejemplo recursivo

Uno de los objetivos del lenguaje Queule es permitir la edición de estructuras recursivas. El código expresado en el fragmento 5.12 muestra la definición de un tipo de datos compuesto que resuelve el problema presentado en el caso de estudio visto en la sección 2.1.2, "Editor de organizaciones".

Código 5.12: Tipo recursivo

```
<q:component name="Organization">
  <li>
    <h2>
      <q:use types="text">
        <q:yield type="text" delete="false">Nombre</q:yield>
      </q:use>
    </h2>
    <p>Código:
      <span>
        <q:use types="text">
          <q:yield type="text" delete="false">Código</q:yield>
        </q:use>
      </span>
    </p>
    <p><q:use types="textarea" label="descripción"/></p>
    <h3>Sub organizaciones</h3>
    <ul><q:use repeat="*" types="Organization"></ul>
  </li>
</q:component>
```

En este ejemplo, junto con presentar la manera en que Queule permite editar estructuras recursivas, se muestra cómo es posible agregar elementos instanciados que ayuden al usuario a comprender la naturaleza de los datos que está editando. En este caso, los atributos correspondientes al nombre de la organización y su código aparecen instanciados desde un inicio, mientras que la descripción es un atributo que el usuario puede opcionalmente agregar. Además, al definir el atributo @delete como "false" en los dos elementos instanciados de las organizaciones, se está indicando que estos no pueden ser borrados, y por ende, que son atributos obligatorios.

Capítulo 6

Implementación

6.1. Diseño de clases

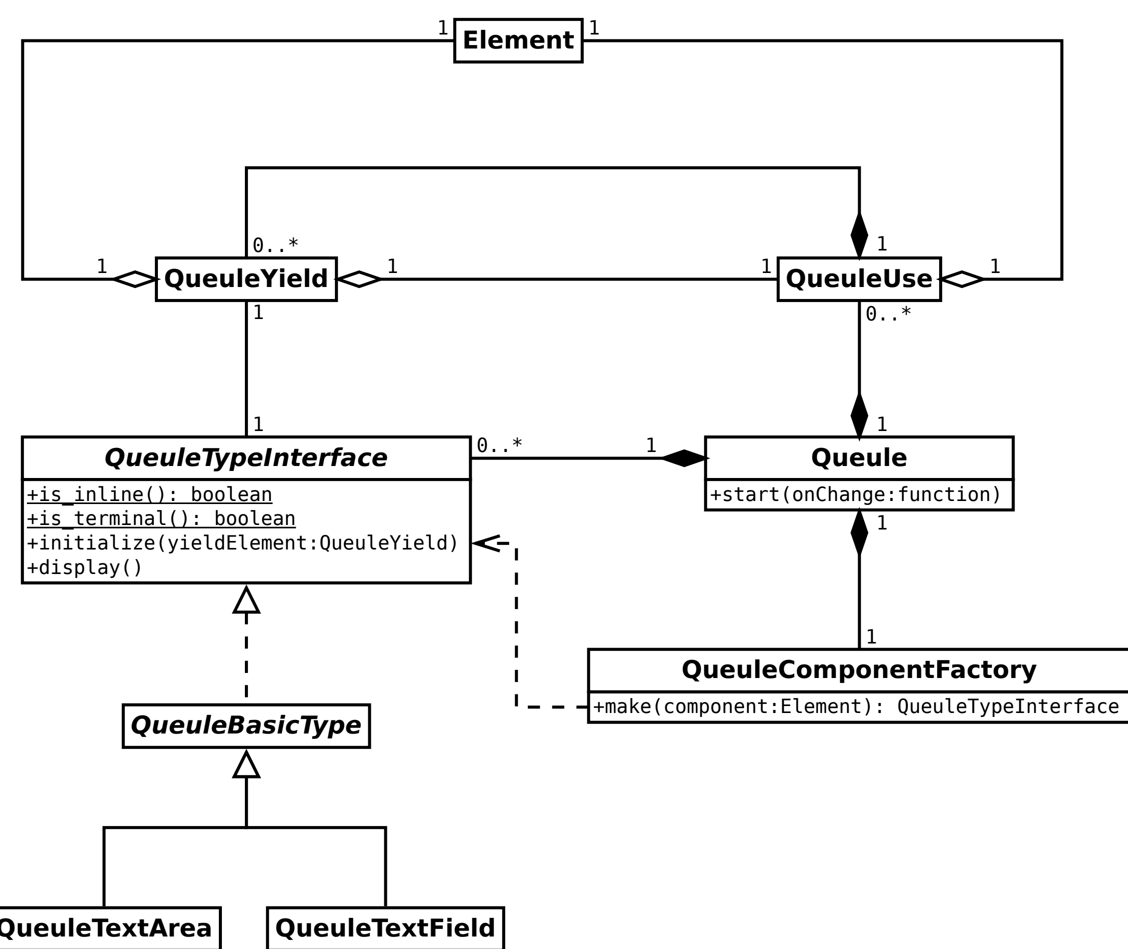
El intérprete para el lenguaje Queule fue implementado utilizando la biblioteca Prototype [46] para JavaScript que, entre otras cosas, facilita la definición de clases con el fin de utilizar técnicas de programación orientada a objetos.

La figura 6.1 presenta el diagrama de clases del diseño del intérprete. La única clase que necesita ser invocada en una plantilla es la clase Queule, que posee un método start que inicializa el intérprete sobre el documento. La clase Queule cumple el rol de manejar las componentes internas de la biblioteca y entregar una interfaz fácil de usar. Este patrón de diseño es conocido como Facade. La clase Queule guarda las referencias de todos los tipos disponibles (interfaz QueuleTypeInterface), de todos los campos editables de la plantilla (clase QueuleUse) y de una clase que permite generar tipos compuestos (clase QueuleComponentFactory).

Las clases QueuleUse y QueuleComponent son en realidad singletons, es decir, poseen una sola instancia. Cada instancia de la clase QueuleUse está asociada a un conjunto ordenado de instancias QueuleYield y, recíprocamente, las instancias QueuleYield conocen a la instancia QueuleUse que las anida. Las instancias QueuleYield y QueuleUse están asociadas a elementos específicos dentro del árbol DOM del documento. Estos elementos corresponden a instancias de la clase Element que proporciona la biblioteca Prototype.

La interfaz QueuleTypeInterface no corresponde a un elemento implementado, sino a una convención que los tipos deben satisfacer para ser utilizados por las instancias de la clase QueuleYield. Las dos maneras posibles de generar instancias de la interfaz QueuleYield

Figura 6.1: Diagrama de clases de la biblioteca JavaScript para Queule



son: generar instancias de las subclases concretas de la clase QueuleBasicType o generar instancias de tipos compuestos. Los tipos compuestos son clases que son generadas durante la ejecución por la clase QueuleComponentFactory a partir de las componentes declaradas en la plantilla. Los tipos deben poseer los métodos de clase is_inline y is_terminal, que permiten saber, respectivamente, si el tipo debe desplegarse dentro de un párrafo o como un bloque y si el tipo es terminal o compuesto. Los tipos básicos responden siempre verdadero al método is_terminal, mientras que los generados por QueuleComponentFactory responden siempre falso.

Todos los tipos deben incorporar los métodos initialize y display. El primero construye una instancia del tipo, recibiendo como parámetro el elemento yield que indica dónde se inserta la variable y el segundo se encarga de generar la interfaz de edición para capturar el valor de la variable.

Las clases QueuleYield y QueuleUse poseen una serie de métodos que implementan la mayor parte de las funcionalidades de la plantilla, tales como cambiar el orden de los elementos en una repetición, agregar nuevos elementos, borrar elementos insertados, abrir u ocultar las interfaces de edición de los elementos terminales y guardar los cambios realizados en la estructura del documento.

El parámetro onChange de la función Queule.start corresponde a una función que es ejecutada cada vez que el documento cambia. Esta función permite, por ejemplo, enviar los cambios en el modelo de manera asíncrona. Esto no deja fuera la posibilidad de enviar el documento completo luego de terminar la edición, dado que siempre es posible agregar un botón en la plantilla que envíe el documento, o una transformación de éste, a un servicio que se encargue de la persistencia de la información.

6.2. Uso de Queule

Para usar Queule en una plantilla basta con agregar las correspondientes líneas para la adición de las bibliotecas Prototype y Queule (en ese orden) y luego inicializar Queule utilizando el método start, El fragmento de código 6.13 presenta las líneas que deben incluirse.

Código 6.13: Líneas para incluir Queule

```
<script type="text/javascript" src="prototype.js"></script>
<script type="text/javascript" src="queule.js"></script>
<script type="text/javascript">
  window.onload = function() { new Queule( function(changed) {...} ); }
</script>
```

6.3. Interfaz de usuario

La interfaz de usuario de Queule es completamente configurable a partir del estilo (archivo CSS) asociado al documento. La biblioteca Queule viene también con una hoja de estilo que puede ser utilizada como base para construir una interfaz personalizada.

Las figuras 6.2 y 6.3 corresponden a capturas de pantalla de la interfaz de Queule. Las plantillas usadas corresponden a la solución implementada para el caso de estudio descrito en la sección 2.1.1. Los números encerrados en círculos son para referenciar las componentes y facilitar la descripción de las figuras.

- Los campos marcados con 1 y 5 corresponden a los títulos del cuestionario y de la sección. Si el usuario puede pincha sobre ellas, se abrirá la interfaz de edición de los campos respectivos.
- Los números 2 y 6 marcan enlaces que permiten agregar descripciones tanto al cuestionario como a la primera sección. Estas descripciones corresponden a tipos terminales.
- El número 3 marca un enlace que permite agregar nuevas secciones al cuestionario. Estas secciones corresponden a tipos compuestos.
- El número 4 indica un botón que permite borrar una sección ya agregada.
- El número 7 indica el selector que permite escoger el tipo de pregunta a insertar.
- El número 8 representa una interfaz de tipo textarea, que se ha desplegado luego de pinchar en el mensaje que estaba marcado con el número 2. De igual manera, el número 9 despliega una interfaz de tipo textfield que aparece luego de pinchar sobre la descripción de una alternativa.

- Los números 10, 11 y 12 señalan las flechas que permiten cambiar las alternativas de posición dentro de la pregunta.

Figura 6.2: Captura de pantalla 1 de la interfaz de Queule

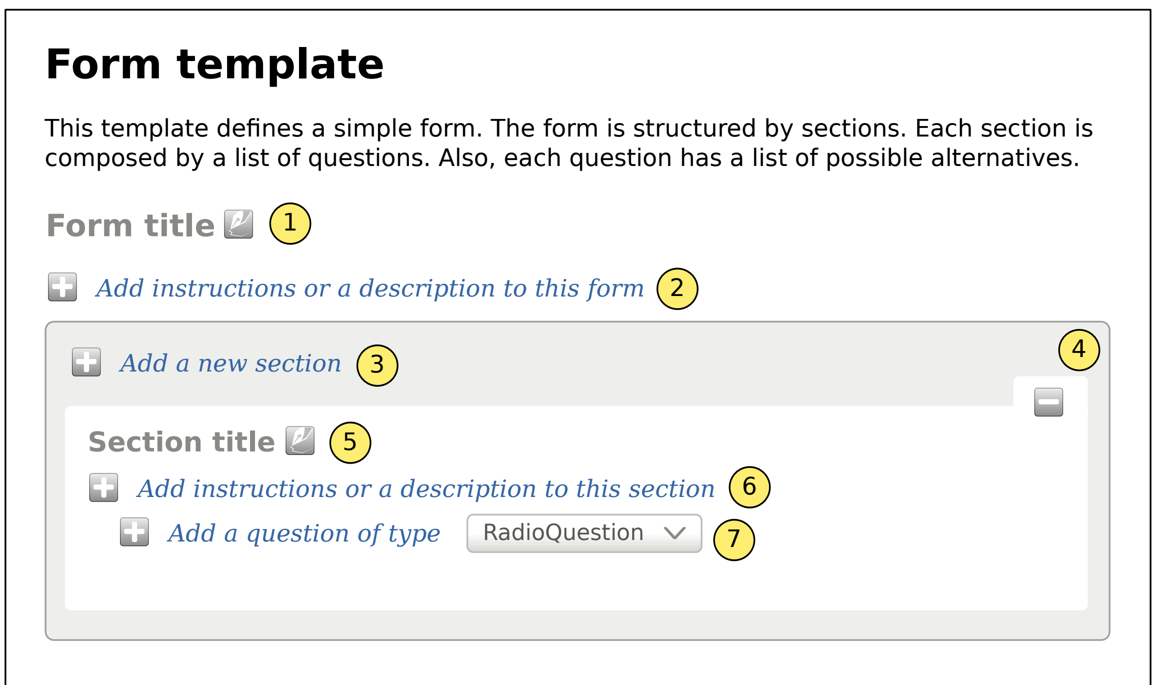


Figura 6.3: Captura de pantalla 2 de la interfaz de Queule

Add a new section

Section title

This section...

8

Save Cancel

Add a question of type RadioQuestion

1. Check-Question: question description (weight: 1.0)

Add an alternative

Option 1

9

Save Cancel (score: 1.0)

10

Option 2 (score: 1.0)

11

Option 2 (score: 1.0)

12

Parte III

Conclusiones

Se ha demostrado que la edición usando plantillas permite definir interfaces de edición que restringen los datos generados tal como lo hace un esquema de documentos o de base de datos. El poder expresivo de la versión 1.0 de Queule permite implementar soluciones para los dos primeros casos de estudios presentados, es decir, permite definir árboles que pueden irse construyendo de manera indefinida gracias al uso de la recursividad. El tercer caso de estudio no es soportado con la versión 1.0 de Queule, pero en una siguiente versión se espera utilizar un esquema basado en el uso de selectores CSS3 como el propuesto en la sección 2.5 para incluir la asociación de instancias no anidadas en una plantilla.

El cuadro 6.1 muestra que Queule es considerablemente más simple en su definición que XTiger, incorporando un total de sólo 15 términos contra los 44 que definen XTiger. La no inclusión de los elementos `xt:bag` y `xt:attribute` responde, en el primer caso, a que Queule no cuenta con un editor base como es el caso de XTiger que posee las herramientas de Amaya y, en el segundo caso, a que editar atributos del árbol DOM es una tarea innecesaria cuando podemos editar simplemente valores y luego guardar estos como atributos mediante el uso de transformaciones.

Cuadro 6.1: Comparación número términos de las especificaciones

Tipo de término	XTiger	Queule
Elementos	9	5
Atributos	27	8
Otros	5	0
Total	44	15

La combinación de los elementos `q:use` y `q:yield` resulta tan poderosa como `xt:use` y `xt:repeat` y con menos atributos. La cardinalidad mínima de elementos anidados, codificada en Xtiger mediante el atributo `xt:repeat@minOccurs`, es expresada en XTiger mediante el atributo `q:yield@delete`. Ingresar elementos que no se pueden borrar a una repetición se deja explícito que la repetición posee un mínimo. Por otra parte, el máximo, expresado en XTiger por el atributo `xt:repeat@maxOcurr` queda, en Queule, determinado por el elemento `q:use@repeat`. Esta manera de estructurar las repeticiones de Queule permite generar documentos definidos de manera recursiva al evitar los problemas en los que cae XTiger con la definición de componentes recursivas. Además, los valores por omisión

de los atributos de Queule han sido escogidos para promover la generación de plantillas más sucintas, por ejemplo, el valor por omisión “1” para las repeticiones se basa en que la mayoría de los terminales tienen cardinalidad 1.

Trabajos futuros

En las siguientes versiones de Queule debe estudiarse la posibilidad de incluir un sistema de tipos extensible, como el definido por XML Schema y que además posea modificadores que permitan definir la forma en que los valores serán ingresados. Se propone utilizar una notación que añada modificadores a los tipos terminales definidos, para que estos puedan ser incluidos en el atributo `q:use@types`, como por ejemplo “string(textfield)” o “string(textarea)”.

Utilizar CSS3 para permitir la asociación entre distintos elementos definidos en la misma plantilla, tal como se comentó en la sección 2.5.

Desarrollar técnicas para generar de manera automática scaffolds basados en Queule a partir de modelos conceptuales o esquemas de documentos o bases de datos.

Generar herramientas para la generación de plantillas desde un documento particular utilizando el análisis y generalización, tal como lo hace Amaya (ver figura 1.2).

Explorar las diversas posibilidades para lograr la persistencia de los datos en las plantillas. No se trata sólo de aplicar transformaciones al documento generado por la edición, además, en algunos usos también se requiere detectar cuándo una parte del documento ha cambiado e inmediatamente enviar los cambios de manera asíncrona y tomar acciones en caso de problemas detectados en el envío.

Bibliografía

[1] Daniel Hernández, repositorio para el código de la implementación del intérprete para el lenguaje Queule.
https://github.com/danielhz/queule

[2] Ruby on Rails, sitio web del proyecto.
http://rubyonrails.org

[3] ActiveScaffold, sitio web del proyecto.
http://activescaffold.com

[4] Django, sitio web del proyecto.
http://www.djangoproject.com

[5] TurboGears, sitio web del proyecto.
http://turbogears.org

[6] CakePHP, sitio web del proyecto.
http://cakephp.org

[7] Plone, sitio web del proyecto.
http://plone.org

[8] Jim Fulton, *Introduction to the Zope Object Database*, Proceedings of the 8th International Python Workshop, Python Software Foundation, 2000
http://www.python.org/workshops/2000-01/proceedings/papers/fulton/zodb3.html

[9] Python, sitio web de la Python Software Foundation.
http://www.python.org

[10] Microformats, sitio web del proyecto.
http://microformats.org

[11] vCard, especificación del microformato.
http://microformats.org/wiki/vcard

[12] Renzo Angles, Claudio Gutiérrez. *Survey of Graph Database Models.* ACM Computing Surveys, Vol. 40, No. 1, Febrero de 2008.

[13] Semantic Web, sitio web del proyecto en la W3C.
http://www.w3.org/standards/semanticweb

[14] Linked Data, sitio web del proyecto.
http://linkeddata.org

[15] RDFa, página del proyecto en la W3C.
http://www.w3.org/standards/techs/rdfa

[16] Istvan Beszteri, Petri Vuorimaa, *An XForms Based Solution for Adaptable Documents Editing*, 2005 ACM Symposium on Applied Computing.

[17] Amaya, sitio web del proyecto.
http://www.w3.org/Amaya

[18] Vincent Quint, Irène Vatton, *Techniques for Authoring Complex XML Documents* Proceedings of the 2004 ACM symposium of Document Engineering, Octubre 28–30 de 2004, Milwaukee, Wisconsin, USA, p. 115–123.

[19] Émilien Kia, Vincent Quint, Irène Vatton, *XTiger Language Specification*, versión 1.0 de 4 de Julio de 2008.
http://www.w3.org/Amaya/Templates/XTiger-spec.html

[20] Francesc Campoy Flores, Vincent Quint, Irène Vatton, *Templates, Microformats and Structured Editing*, Proceedings of the 2006 ACM Symposium on Document Engineering, Octubre 10–13 de 2006, Amsterdam, The Netherlands, p. 188–197.

[21] Fabien Gandon, RDFa2RDFXML, URL de la transformación. http://ns.inria.fr/grddl/rdfa/2007/09/12/RDFa2RDFXML.xsl

[22] John M. Boyer, *XForms 1.1*, sección Editing Hierarchical Bookmarks Using XForms, Technical report, W3C, 2009.
http://www.w3.org/TR/2009/REC-xforms-20091020/#bookmarks-in-x-smiles

[23] L. Feng, E. Chang, T. Dillon. A Semantic Network-Based Design Methodology for XML Documents. ACM Transactions on Information Systems, Volumen 20, Número 4, p. 390-421. Octubre 2002.

[24] Martin Necasky, *Conceptual modeling for XML: A survey* In Proceedings of the DATESO 2006 Annual International Workshop on Databases, Texts, Specifications and Objects (DATESO 2006).

[25] M. Necasky: *Conceptual Modeling for XML: A Survey.* Technical Report No. 2006-3, Dep. of Software Engineering, Faculty of Mathematics and Physics, Charles University, Praga, 2006.

[26] A. Badia. *Conceptual Modeling for Semistructured Data.* Proceedings of the 3rd International Conference on Web Information Systems Engineering Workshops (WISE 2002 Workshops), p. 170-177. Singapur, December 2002.

[27] M. Mani. *EReX: A Conceptual Model for XML.* Proceedings of the Second International XML Database Symposium (XSym 2004), p. 128-142. Toronto, Canadá, Agosto 2004.

[28] G. Dobbie, W. Xiaoying, T.W. Ling, M.L. Lee. *ORA-SS: An Object-Relationship-Attribute Model for Semi-Structured Data.* Technical Report, Department of Computer Science, National University of Singapore. Diciembre de 2000.

[29] B.R. Loscio, A.C. Salgado, L.R. Galvao. *Conceptual Modeling of XML Schemas.* Proceedings of the Fifth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2003), p. 102-105. New Orleans, Louisiana, USA, Noviembre de 2003.

[30] Anders Berglun, *Extensible Stylesheet Language (XSL) Version 1.1*, Technical report, W3C, 2006.
http://www.w3.org/TR/xsl

[31] James Clark, *XSL Transformations (XSLT)*, Technical report, W3C, 1999.
http://www.w3.org/TR/xslt

[32] Haruo Hosoya, Benjamin C. Pierce, *XDuce: A statically typed XML processing language* ACM Transactions on Internet Technology, 2003.

[33] Shinya Kawanaka, Haruo Hosoya *biXid A Bidirectional Transformation Language for XML*, 2008.

[34] Lista de implementaciones de XForms mantenida por el W3C.
http://www.w3.org/MarkUp/Forms/wiki/XForms_Implementations

[35] XML Schema, sitio web del proyecto.
http://www.w3.org/XML/Schema

[36] RELAX NG Specification, OASIS Committee Specification, 3 December 2001. Definitive specification for RELAX NG using the XML syntax.
http://www.relaxng.org/spec-20011203.html

[37] Schematron, sitio del proyecto.
http://www.schematron.com/

[38] YAML, sitio del proyecto.
http://www.yaml.org/

[39] JSON, sitio del proyecto.
http://www.json.org/

[40] Charles Petrie, Robert Caillau, *Interview Robert Caillau on the WWW Proposal: "How it really happened*, IEEE, entrevista del 18 Agosto de 1997.
http://www.computer.org/portal/web/computingnow/ic-cailliau

[41] James Clark, Steve DeRose, *XML Path Language (XPath)*, Technical report, W3C, 1999.
http://www.w3.org/TR/xpath

[42] Tantek Çelik, Elika J. Etemad, Daniel Glazman, Ian Hickson, Peter Linss, John Williams, *Selectors Level 3*, Technical report, W3C, 2009.
http://www.w3.org/TR/css3-selectors

[43] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, *XQuery 1.0: An XML Query Language*. Technical report, W3C, 2001.
http://www.w3.org/TR/xquery/

[44] J. Robie, J. Lapp, D. Schach. *XML Query Language (XQL)*. Proceedoings of the Query Languages workshop, Cambridge, 1998.
http://www.w3.org/TandS/QL/QL98/pp/xql.html

[45] JQuery, sitio web del proyecto.
http://jquery.com/

[46] Prototype, sitio web del proyecto.
http://www.prototypejs.org/