Actualización, Consistencia y Reparación de Información Temporal en XML

Tesista

Marcela Campo (mcampo@dc.uba.ar)

Director

Alejandro Vaisman (avaisman@dc.uba.ar)

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Mayo de 2007

Resumen

En la bibliografía de Bases de Datos existen diferentes modelos para representar, almacenar y consultar información histórica en documentos XML. Sin embargo, ninguno de estos modelos aborda el problema de chequear un documento XML temporal con respecto a un conjunto de condiciones de consistencia temporales. En nuestro trabajo, luego de introducir un modelo abstracto para XML temporal, nos dedicamos a estudiar el problema de chequeo y resolución de inconsistencias temporales. Comenzamos identificando los tipos de inconsistencias temporales que pueden presentarse en el modelo utilizado, y de qué manera podemos resolver cada una de ellas si consideramos que cada una es la única inconsistencia presente en el documento. Luego, pasamos a estudiar un caso más realista, donde varias inconsistencias aparecen concurrentemente en el documento. Para este caso, definimos, en primer lugar, las condiciones bajo las cuales es posible tratarlas como si fueran las únicas en el documento. Para el caso en que esto no es posible, definimos condiciones bajo las cuales resolver las inconsistencias en distintos ordenes resulta en el mismo documento XML. Finalmente, esbozamos una propuesta de un lenguaje de actualizaciones para documentos XML, y presentamos resultados experimentales de chequeo de consistencia. Nuestro enfoque, si bien se encuentra ligado a un determinado modelo, puede ser extendido a cualquier modelo para XML temporal que soporte la noción de tiempo de transacción. Como contribución adicional, presentamos un generador de documentos XML temporales, utilizado en este trabajo para generar los documentos utilizados para la experimentación.

Abstract

Different models have been proposed for representing temporal data, tracking historical information, and recovering a document's state at any given time, in XML documents. However, none of those models address the problem of checking a temporal XML document against temporal constraints. After introducing an abstract model for temporal XML, we discuss the problem of validating and fixing temporal inconsistencies. We first identify the different types of temporal inconsistencies that can be found in a our model, analyzing the possible fixes to each of them when they are isolated. Then, we move on to study a more realistic scenario, in which many inconsistencies of any kind appear concurrently in a document. Conditions under which we can treat the inconsistencies as if they were isolated are defined prior to this analysis. When such conditions are not valid, we state conditions to guarantee that the fixing order does not alter the final document. Finally, we sketch a proposal for an update language for temporal XML documents, and we show experimental results on consistency validation. Our approach, although attached to out model for temporal XML, can be extended to other transaction time-based models. In addition, we present a temporal XML document generator, which was used in

this work to generate experimental data.

Agradecimientos

A mis padres y a mi hermano por el cariño y el esfuerzo que pusieron para allanarme el camino durante toda mi vida.

A mi hermano, por ver más allá de las cosas, y reconocer cuándo lo necesito a mi lado.

A mi novio, por el amor, apoyo y la ayuda durante todos estos años. Por secar mis lágrimas en los momentos difíciles dándome energía para continuar. Por las aventuras que pasamos juntos, y las que vendrán.

A mi cuñi y mi suegra, por darme aliento constante y por el cariño que me brindan a diario.

A mis compañeros y amigos de facultad, por su ayuda y los momentos compartidos. Por las divertidas tardes y noches de tp que nunca vamos a olvidar.

A todos los que están a mi lado, física o espiritualmente, por no olvidarme, aún cuando en el afán por terminar esta etapa, los haya hecho sentir olvidados.

A Alejandro, por la paciencia y ayuda que me brindó durante este trabajo.

Índice general

Capítulo 1

Introducción

El problema de validar un documento XML con respecto a un conjunto de restricciones de integridad luego de un update ha atraído recientemente la atención de la comunidad de base de datos. Muchas técnicas de validación incremental fueron propuestas [KAN/02, PAP/03, BAR/04, BAL/04]. En el escenario de XML temporal, aunque varios modelos existen para representar, consultar y modificar información temporal [AMA/00, DYR/01, CHI/01, GAO/03, WAN/03], poca atención ha sido brindada al problema de validar los restricciones temporales impuestas por estos modelos. En documentos XML Temporales, los updates deben tomar como entrada (y retornar) un documento XML válido, no sólo con respecto al conjunto usual de restricciones, sino también con respecto a los restricciones temporales definidas por el modelo utilizado. Más aún, generalmente estaremos trabajando con documentos que no fueron construidos desde cero utilizando operaciones de update, sino con documentos TXML preexistentes; por lo tanto, necesitaremos un método eficiente para comprobar que un documento verifica un conjunto de restricciones temporales, y si no lo hace, proveer al usuario herramientas suficientes para arreglar (el conjunto de) inconsistencias.
En este trabajo nos abocamos al problema de validar un conjunto de restricciones temporales en documentos XML temporales. El estudio está basado en el modelo introducido en [MEN/04], aunque podría ser extendido a otros modelos de datos para XML temporal. Luego de presentar y discutir el modelo de datos, caracterizamos las inconsistencias temporales en documentos XML temporales. Introducimos entonces el problema de comprobación de consistencia en un documento y la corrección de inconsistencias individuales. Luego, nos movemos a un escenario más realista, donde varias inconsistencias podrían aparecer concurrentemente y estudiamos las condiciones bajo las cuales estas inconsistencias pueden ser consideradas aisladas unas de otras (eso es, como si cada una fuera la única dentro del documento).
Un documento XML temporal (en adelante TXML) está sujeto a operaciones de actualización. Para soportar estas operaciones, introducimos un lenguaje de updates en la línea de [MEN/04], con soporte para tiempo de 'transacción'. Una operación de actualización se aplica a un documento TXML consistente y da

como resultado otro documento TXML consistente. Finalmente, implementaremos los algoritmos de chequeo de consistencia propuestos, y realizamos diversos tipos de experimentos con ellos, cuyos resultados reportamos. Para ello, utilizamos un generador de documentos TXML. Adicionalmente, implementamos dos mejoras al modelo original [MEN/04]: (1) la utilización de valores default para los atributos de carácter temporal, logrando ahorro de espacio de almacenamiento tanto en disco como en memoria; (2) extendemos el modelo para soportar atributos que varíen temporalmente.

Capítulo 2

Estado del Arte

Recientemente se presentaron algunas propuestas para la validación incremental de documentos XML. Kane *et al* [KAN/02] modelaron restricciones XML en forma de reglas, y presentaron un mecanismo de comprobación de restricciones para operaciones de actualización, con el objetivo de asegurar que el resultado de un update deja el documento XML en un estado consistente. Básicamente, esto se logra reescribiendo una query de update en una query de 'safe update'. La validación incremental de documentos XML ha sido también estudiada en [PAP/03, BAR/04, BAL/04]. El problema puede ser enunciado como sigue: dado un documento D, válido con respecto a un DTD X y un operador de update U, es válido U con respecto a D?.
A pesar de que existen varios modelos de datos en el conjunto de bases de datos XML y bases de datos semi-estructuradas, el tópico de validar un documento con respecto a un conjunto de restricciones temporales ha sido extrañamente obviado. Más aún, la mayoría de las propuestas que comentamos luego, definen condiciones de consistencia, pero nada se dice sobre cómo estas restricciones pueden ser aseguradas en un sistema real, y qué hacer si un documento no las verifica. Tampoco se estudió el problema de cómo los operadores de actualización consideran estas restricciones. En lo que sigue utilizaremos indistintamente los términos 'actualización' y 'update'.
Chawathe *et al* [CHA/99] propusieron un modelo para administrar datos históricos semi estructurados. Extendieron el Object Exchange Model (OEM) con la habilidad de representar updates y mantener el detalle de estos por medio de 'deltas', aunque no aplicaron este trabajo sobre XML. En la misma línea, Oliboni *et al* [OLI/01] propusieron un modelo de datos gráfico y un lenguaje de consulta para datos semi-estructurados soportando el tiempo de transacción, agregando un intervalo de validez a los objetos del modelo. Ambos trabajos asumen que, o bien los documentos están construidos de cero, o son consistentes con los restricciones temporales que el modelo impone. En lo que sigue, daremos una mirada rápida a los modelos más relevantes de XML temporal propuestos hasta ahora. Todos ellos carecen de un mecanismo para comprobar los restricciones temporales subyacentes.

En [GRA/01] se provee una buena referencia a trabajos sobre aspectos temporales en la Web. Amagasa *et al* [AMA/00] introdujeron un modelo de datos temporal basado en XPath, pero no un modelo para updates, ni un lenguaje de consulta que explote el modelo temporal. Dyreson [DYR/01] propuso una extensión a XPath con soporte de tiempo de transacción por medio del agregado de varios ejes temporales (de XPath) para especificar direcciones temporales. Chein *et al* [CHI/01] propusieron esquemas de update y versionado para XML, a través de un esquema donde la administración de versiones es realizada manteniendo referencias al subárbol inalterado maximal en la versión anterior. Un enfoque similar fue también seguido por Marian *et al* [MAR/01]. Gao *et al* [GAO/03] introdujeron *TXQuery*, una extensión a *XQuery* que soporta tiempo válido a la vez que mantenía el modelo de datos inalterado. Las consultas se traducen a XQuery, y son evaluadas por un motor de XQuery. Finalmente, Wang *et al* también propusieron soluciones basadas en versionado [WAN/03, WAN/04, WAN/05].
En este trabajo nos basaremos en el modelo de datos introducido inicialmente en [MEN/04]. Este modelo tiene similitud con el trabajo de Buneman *et al* [BUN/02], donde los autores estudian estructuras de datos específicamente apropiadas para mantener información histórica sobre datos científicos, basadas en un esquema de versionado que permite el almacenamiento de toda la información en un único documento.
Finalmente, la parte de este trabajo en la cual se analizan inconsistencias que ocurren simultáneamente en un documento TXML fue publicada en [CAMP/06].

Capítulo 3

Modelos de datos

3.1. Documentos XML Temporales

Antes de plantear el problema central de esta Tesis, introduciremos el modelo formalmente mediante un ejemplo, ilustrado en la Figura 3.1. Esta es una representación abstracta de un documento XML temporal para un fragmento de una compañía que incluye departamentos y empleados. La base de datos también registra sueldos y, probablemente, algunas otras propiedades de los empleados. En este modelo, los nodos que representan empleados no están duplicados a lo largo del tiempo. Por ejemplo, vemos que John y Peter trabajaron para el departamento de ventas en los intervalos $[0, 10]$ y $[0, 20]$ respectivamente, mientras que Susan trabaja para el departamento de Finanzas desde el instante '10'. Cuando un eje no tiene etiqueta temporal, asumimos que su intervalo de validez es $[0, Now]$. La información contenida en la representación abstracta del documento temporal presentado en la Figura 3.1 nos permite recorrer la historia de la empresa usando este único documento. Podemos entonces (a) consultar el estado de la base de datos en un instante dado (técnicamente, un *snapshot* del documento); o (b) realizar consultas temporales como 'empleados que trabajaron para el departamento de ventas ininterrumpidamente desde el año 2000'. Para este tipo de consultas, en [MEN/04] los autores proveyeron esquemas de indexación permitiendo técnicas de evaluación eficientes y un lenguaje de consultas denominado TXPath.

Más formalmente, un documento XML temporal (*TXML*) es un grafo dirigido con etiquetas, donde distinguimos distintos tipos de nodos: (a) un nodo distinguido r (o *raíz*), que no tiene ejes incidentes y tal que cada nodo en el grafo es alcanzable desde r; (b) *Valor o Value nodes*: nodos que representan valores (de texto o numéricos); no tienen ejes salientes, y poseen exactamente un eje incidente desde un nodo atributo o elemento (o desde la raíz); (c) *Atributo o Attribute nodes* etiquetados con el nombre de un atributo, y posiblemente con una anotación 'ID' o 'REF'; (d) *Elemento o Element nodes*: etiquetados con un tag de elemento, y conteniendo links salientes a nodos atributo, valor y otros

elementos. Cada nodo se identifica unívocamente con un entero, el *número de nodo*, y es descripto por una cadena, la *etiqueta del nodo*. Los ejes en el grafo del documento pueden ser de dos tipos: *ejes de contención* o *ejes de referencia*. Un eje de contención $\epsilon_c(n_i, n_j)$ vincula dos nodos n_i *y* n_j de manera que: (a) n_i es r o un elemento, y n_j es un atributo, un valor, u otro elemento; o (b) n_i es un atributo, y n_j es un valor conteniendo el valor del atributo. Los atributos tienen exactamente un eje de contención saliente (hacia el valor del mismo). Un eje de referencia $\epsilon_r(n_i, n_j)$ vincula un atributo n_i de tipo REF, con un elemento n_j. La dimensión temporal se incorpora a los grafos de los documentos etiquetando los ejes con intervalos de tiempo. En lo que sigue, consideraremos que el tiempo es un dominio discreto y linealmente ordenado. Un par ordenado $[a, b]$ de instantes de tiempo, con $a \leq b$, denota el intervalo cerrado que va desde a hasta b. Un conjunto de estos intervalos es llamado *elemento temporal*. Como es usual en bases de datos temporales, el instante de tiempo actual será representado con la palabra distinguida 'Now'. Denotaremos indistintamente el instante de creación del documento como 0 o t_0. Por simplicidad, en este trabajo consideraremos elementos temporales de un único intervalo.
Una *etiqueta temporal* es un intervalo T_{e_c} *o* T_{e_r} que etiqueta un eje de contención e_c o un eje de referencia e_r, respectivamente. El significado de esta etiqueta es que dado un eje e_c entre los nodos n_i *y* n_j, T_{e_c} representa el período de tiempo en el cual el elemento representado por n_i incluyó al elemento representado por n_j. Nuestro modelo soporta el *tiempo de transacción* de la relación de contención. Aunque no lidiamos con el *tiempo válido*, este podría ser abordado de manera análoga. Para un eje de referencia, T_{e_r} representa el tiempo de *transacción* de la referencia. Ejes etiquetados con etiquetas temporales serán llamados *ejes temporales*. El modelo completo soporta otro tipo de nodos, denominados *versionados*. Un *nodo versionado* es un par $(n, NList)$ donde n es un nodo del documento y $NList$ es una lista ordenada de k nodos elemento o k nodos atributo que no sean del tipo ID o REF, para algún $k \geq 2$. Todos los nodos en $NList$ tienen la misma etiqueta, y cada nodo de la lista puede tener a lo sumo un eje saliente (hacia un nodo valor) y tiene exactamente un eje incidente, proveniente de n. Además, las etiquetas temporales de los ejes incidentes a los nodos en $NList$ tienen etiquetas temporales consecutivas, es decir, si $Nlist = [n_1, .., n_k]$ y T_i es la etiqueta temporal del eje incidente a n_i desde n, entonces $(T_i)_f + 1 = (T_{i+1})_i \ \forall i : 1..k-1$. Intuitivamente, un nodo versionado encapsula una secuencia de versiones del mismo nodo. Por ejemplo, en la figura 3.1 vemos que los nodos con etiqueta Sueldo hijos de el nodo Empleado con ID conforman un nodo versionado representando distintos salarios de Mary a lo largo del tiempo.

Notación: En general, si un eje ϵ es etiquetado con una etiqueta temporal T_e usaremos $(T_e)_i$ *y* $(T_e)_f$ para referirnos a los extremos inicial y final del intervalo T_e respectivamente. Decimos que dos etiquetas temporales T_{e_i} y T_{e_j} son consecutivas si $(T_{e_j})_i = (T_{e_i})_f + 1$.

Definición 1 (Lifespan de un nodo)**.**

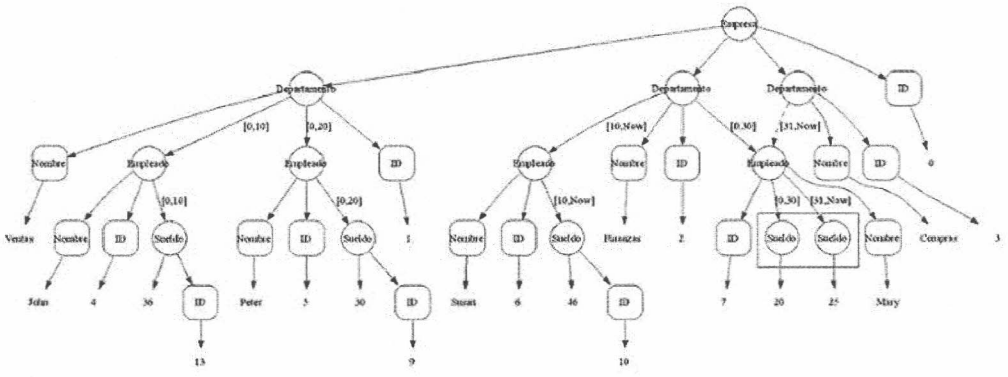


Figura 3.1: Ejemplo de representación abstracta de un documento XML temporal.

El lifespan de un nodo n, denotado l_n es la unión de los elementos temporales de todos los ejes de contención incidentes al nodo. El lifespan de la raíz es el intervalo $[0, Now]$.

Definición 2 (Documento XML temporal).

Un documento XML temporal es un grafo, extendido con etiquetas temporales y nodos versionados, que satisface las siguientes condiciones:

1. *La unión de las etiquetas temporales de los ejes de contención salientes de un nodo está contenida en el lifespan del mismo.*
2. *Las etiquetas temporales de los ejes de contención entrantes a un nodo son consecutivas.*
3. *Para cualquier eje de contención $\epsilon_i(n_i, n_j)$ si n_j es un atributo de tipo ID, la etiqueta temporal de ϵ_i es la misma que el lifespan de n_i (en otras palabras, los atributos ID no son temporales); si n_j es un atributo de tipo REF, de manera tal que existe un eje de referencia $\epsilon_r(n_i, n_j)$ la etiqueta temporal para ϵ_i es el valor de la etiqueta temporal del eje de referencia ϵ_r*
4. *Para cualquier instante de tiempo t, el subgrafo compuesto por todos los ejes de contención ϵ_i tales que $t \in T_{\epsilon_i}$ es un árbol con raíz r. Llamamos a este subgrafo un snapshot del documento en el instante t, denotado D(t)*
5. *Dado un eje de referencia $\epsilon_r(n_i, n_j)$, con etiqueta temporal T_{ϵ_r}, se cumple que $T_{\epsilon_r} \subseteq l_{n_j}$*
6. *Para cualquier eje de contención $\epsilon_c(n_i, n_j, T_{e_c})$, si n_j es un atributo de tipo REF, de manera que existe un eje de referencia $\epsilon_r(n_i, n_j, T_{e_r})$, entonces se satisface $T_{e_r} = T_{e_c}$*

Notemos que la segunda condición en la Definición 2 implica que estaremos trabajando con intervalos en lugar de elementos temporales. Este hecho simplifica la presentación y conlleva a una implementación más eficiente. Nuestras

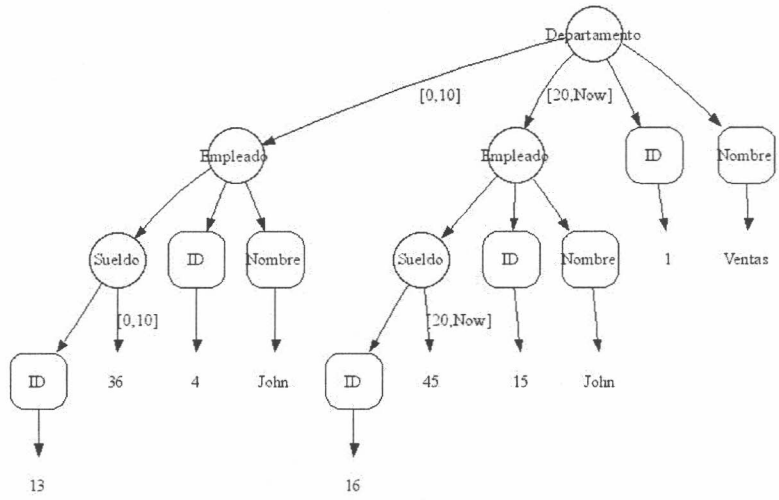


Figura 3.2: Ejemplo de consecutividad de ejes de contención. El nodo que representa al empleado John debió ser agregado nuevamente al documento tras su reincorporación a la empresa puesto que agregar un eje de contención al mismo nodo Empleado existente resultaba en ejes de contención no consecutivos.

definiciones y teoremas pueden ser, de todas maneras, extendidos al caso de elementos temporales, eliminando la limitación antedicha. En [MEN/04] podemos encontrar discusiones sobre este tema, así como una descripción en detalle. Esta condición nos dice a su vez, que en caso de que un elemento deje momentáneamente de ser contenido por cualquier otro en el documento, no puede volver a serlo luego, pues las etiquetas de los ejes de contención incidentes al nodo no serían consecutivas. LA forma de representar esta situación, es crear un nuevo nodo con la misma información en el nuevo intervalo.

Ejemplo 1 (Consecutividad de las etiquetas de los ejes de contención).

Supongamos en el ejemplo de la Figura 3.1, que John se reincorpora a la empresa en el instante '20'. Si simplemente agregáramos un eje desde el Departamento Ventas hacia el nodo que representa al empleado John, entonces este nodo tendría dos ejes de contención incidentes. El primero con intervalo $[0, 10]$ *y el segundo con intervalo* $[20, Now]$. *Estas etiquetas no son consecutivas y por lo tanto no están permitidas en el modelo. Lo que deberíamos hacer en ese caso, es crear un nuevo nodo Empleado que represente a John, con un eje de contención en* $[20, Now]$. *El resultado puede observarse en la Figura 3.2.*

Las condiciones 5 y 6 no serán consideradas en este trabajo, dado que no trataremos con atributos de tipo REF.

Definición 3 (Camino continuo).

Un ***camino continuo*** *con intervalo* T *desde el nodo* n_i *hasta el nodo* n_k *en*

un grafo de un documento temporal es una secuencia $(n_1, ..., n_k, T)$ *de* k *nodos y un intervalo* T *tal que existe una secuencia de ejes de contención de la forma* $\epsilon_1(n_1, n_2, T_1), \epsilon_2(n_2, n_3, T_2), ..., \epsilon_{k-1}(n_{k-1}, n_k, T_{k-1})$*, tal que* $T = \bigcap_{i=1,k} T_i$*. Decimos que hay un* ***camino continuo maximal*** *(mcp) con intervalo* T *desde el nodo* n_1 *al nodo* n_k *si* T *es la unión de un conjunto maximal de intervalos consecutivos* T_i *tal que existe un camino continuo desde* n_1 *hasta* n_k *con intervalo* T_i*.*

Ejemplo 2 (Camino continuo y Camino continuo maximal)**.**

En el documento de la Figura 3.1 podemos ver que existen dos caminos continuos desde el nodo 'Empresa' y el nodo Empleado que representa a 'Mary'. Un camino es el formado por los nodos con los valores para el atributo ID 0,2,7 en el intervalo [0,30]. El segundo camino es el dado por 0,3,7 en [31, Now]. Como los intervalos de ambos caminos son consecutivos, tenemos un camino continuo maximal en el intervalo [0,Now].

3.2. Implementación del modelo temporal

El modelo planteado anteriormente, puede ser mapeado a un documento XML de varias maneras, las cuales son analizadas por Vaisman *et al.* [VAI/03]. Las opciones disponibles son:

- *Top-down sin replicación*, donde la raíz del grafo es la raíz del documento y para cada nodo existe un elemento en el documento. Si los nodos n_k, n_j tienen un eje saliente al nodo n_i entonces en la representación XML, uno de los dos tendrá como hijo físico al elemento equivalente al nodo n_i mientras que el otro tendrá como hijo un elemento nuevo, que tendrá simplemente los atributos 'Time:FROM' y 'Time:TO' correspondientes y un atributo 'IN', cuyo valor será el valor del atributo 'ID' del nodo n_i. La elección de que elemento contiene físicamente al elemento real es arbitraria, nosotros utilizaremos la convención de que el primer padre del nodo (aquel cuyo eje tenga etiqueta temporal menor) será el elegido.

- *Bottom-up sin replicación*, en lugar de tener los elementos ficticios con atributos 'IN' referenciando desde el elemento padre a un elemento hijo, podríamos poner esta referencia dentro del elemento hijo. Esto es, en el ejemplo anterior, el elemento equivalente al nodo n_i tendría físicamente incluido el elemento especial con atributo 'IN' apuntando al elemento que representa al nodo padre que no lo contiene.

- *Representación con replicación de nodos*, esta alternativa evita el uso de nodos con atributo 'IN'. En esta representación el elemento equivalente al nodo n_i es incluido físicamente como hijo de ambos padres.

- *Representación Nodo-Eje*, en esta alternativa se listan los nodos y ejes del grafo, creando elementos 'NODE' y 'EDGE', donde los nodos 'EDGE' tienen atributos '*Origin*' y '*End*' así como un intervalo de validez.

Para este trabajo utilizamos la primer opción. La elección se basa en que la consistencia del documento es más sencilla de comprobar tanto antes como después de la aplicación de una operación de update. El problema que presentan las representaciones con duplicación es que se pierde la identidad del nodo, es decir, al duplicar, perdemos la noción de que las distintas copias son en realidad el mismo nodo. Además tienen un gasto extra de memoria. Con respecto a la representación *Bottom-up*, se hace más difícil navegar el documento desde la raíz hacia los hijos, al igual que en la representación *Nodo-Eje*. La representación *Top-down* facilita la navegación al permitir utilizar modelos estándar como DOM, ya que el documento puede navegarse de la manera tradicional, con el agregado de que al llegar a un elemento con atributo IN, se debe realizar una consulta XPATH para obtener el elemento real al cual apunta. Esta facilidad no está presente en las demás opciones y por lo tanto sería más trabajoso y en consecuencia menos performante reconstruir el documento para responder a una consulta.

Ejemplo 3 (Modelo top-down sin replicación).
En la Figura 3.3 vemos la representación top-down sin replicación del documento ejemplo en 3.1. El nodo especial `SEQUENCE` *indica que los nodos que contiene componen un nodo versionado. Los atributos* `Time:FROM`, *y* `Time:TO`, *indican los extremos desde y hasta del intervalo temporal en el cual el eje de contención es válido. El atributo* `Time:IN` *en*

```
<EMPLEADO ID="8" Time:IN="7" Time:FROM="31" Time:TO="Now" />
```

indica que este nodo hace referencia al nodo con tag `'EMPLEADO'` *con* `ID='7'`, *en el intervalo* `[31, Now]`. *Utilizando el atributo* `Time:IN` *evitamos entonces replicar un mismo nodo varias veces, indicando que el nodo que contiene este atributo debe ser reemplazado al consultar por el nodo con atributo* `ID` *con valor '7'.*

3.3. XML Temporal como un grafo dirigido

Para estudiar las condiciones de consistencia, verificarlas y corregir las inconsistencias detectadas, utilizaremos un modelo distinto al modelo TXML. Este estará estrictamente basado en grafos, y reflejará sólo los conceptos temporales de TXML y no todas las restricciones planteadas por XML. Por ejemplo, omitimos los distintos tipos de nodos (Elementos, Atributos y Valores) y los distintos tipos de ejes (de contención y referencia), tratando a todos ellos de manera similar. Esta simplificación se realizó para facilitar el estudio del problema y se basa en que:

```
<EMPRESA ID="0">
  <DEPARTMENTO ID="1" name="Ventas">
    <EMPLEADO ID="4" Time:FROM="0" Time:TO="10" name="John">
      <SUELDO ID="13" Time:FROM="0" Time:TO="10">
        36
      </SUELDO>
    </EMPLEADO>
    <EMPLEADO ID="5" Time:FROM="0" Time:TO="20" name="Peter">
      <SUELDO ID="9" Time:FROM="0" Time:TO="20">
        30
      </SUELDO>
    </EMPLEADO>
  </DEPARTAMENTO>
  <DEPARTAMENTO ID="2" name="Finanzas">
    <EMPLEADO ID="6" Time:FROM="10" Time:TO="Now" name="Susan">
      <SUELDO ID="10" Time:FROM="10" Time:TO="Now">
        46
      </SUELDO>
    </EMPLEADO>
    <EMPLEADO ID="7" Time:FROM="0" Time:TO="30" name="Mary">
      <SEQUENCE ID="12" Time:FROM="0" Time:TO="Now">
      <SUELDO Time:FROM="0" Time:TO="30">
        20
      </SUELDO>
        <SUELDO Time:FROM="31" Time:TO="Now">
        25
      </SUELDO>
      </SEQUENCE>
    </EMPLEADO>
  </DEPARTAMENTO>
  <DEPARTAMENTO ID="3" name="Compras">
    <EMPLEADO ID="8" Time:IN="7" Time:FROM="31" Time:TO="Now" />
  </DEPARTAMENTO>
</EMPRESA>
```

Figura 3.3: Representación top-down sin replicación del documento de la Figura 3.1

- No tratamos con atributos temporales, por lo tanto no hace falta chequear la consistencia de estos con respecto al eje temporal.
- La única diferencia entre nodos Elemento y Valor es que los últimos tienen un único eje incidente y no poseen ejes salientes. Por lo tanto las restricciones extra no son de carácter temporal y no nos interesan en este trabajo.
- Por simplicidad, prescindimos del análisis de los atributos tipo REF (es decir, de las referencias ID/REF).

En el modelo utilizado para la primera (y mayor) parte de este documento entonces, un documento XML temporal es un grafo
$G =< V, E, V_d, R >$ en donde:

- V son los nodos no versionados del documento.
- V_d son los nodos versionados del documento, y son en realidad pares $(n, NList)$ donde n y los elementos de $NList$ pertenecen a V.

- E son los ejes del documento, definidos como $\epsilon_i(n_i, n_j, T_i)$ donde n_i es el nodo inicial del eje n_j el nodo final y T_i es la etiqueta temporal correspondiente.
- R es un nodo distinguido del documento, la Raíz. Tiene la propiedad de que no posee ningún eje incidente y cada nodo debe ser alcanzado en todo su lifespan desde ella.

En lo que resta del presente documento nos dedicaremos a analizar las condiciones de consistencia planteadas en el modelo temporal y veremos como detectar y corregir inconsistencias. Primero las trataremos de forma aislada, es decir, tomando cada inconsistencia como si fuera la única dentro del documento. Luego pasaremos a un caso más complejo donde varias inconsistencias son detectadas en un mismo documento. En la última parte definiremos un lenguaje de updates para TXML, teniendo en cuenta que estas operaciones deben dejar el documento en un estado consistente luego de ser aplicadas, integrando para ello al lenguaje todo el trabajo realizado en la primer parte con respecto al chequeo de consistencia. Para finalizar, propondremos una extensión al modelo, agregando la noción temporal a los atributos y definiendo entonces el concepto e implementación de Atributo Temporal.

3.4. Resumen

En este capítulo definimos que se entiende por documento XML temporal (o documento TXML), mostrando los distintos tipos de nodos y ejes presentes en el modelo, junto con las condiciones de consistencia que se deben cumplir (Sección 3.1). También presentamos dos implementaciones distintas del modelo que serán utilizadas en el resto del trabajo. La primera, una implementación XML (Sección 3.2), será la utilizada para almacenar los documentos, consultarlos y realizar updates. La segunda, una representación simplificada basada estrictamente en grafos dirigidos (Sección 3.3), será útil para analizar teóricamente la consistencia de los documentos y las distintas maneras de corregir las inconsistencias detectadas, así como también para implementar los algoritmos de chequeo de consistencia de una manera eficiente.

Capítulo 4

Chequeo de consistencia

4.1. Consistencia de los documentos

Las condiciones impuestas por el modelo en la Definición 2, implican que se pueden presentar distintos tipos de inconsistencias en un documento TXML. De ellas, nos focalizaremos en aquellas de carácter temporal, es decir, aquellas que se imponen en las condiciones de consistencia temporales.

La definición 4 presenta los tipos de inconsistencia que podemos encontrar en un documento temporal. En función de las condiciones dadas en la Definición 2.

Definición 4 (Posibles inconsistencias de un documento.).

i. La etiqueta de un eje saliente de un nodo no está incluida en el lifespan del mismo

ii. Las etiquetas temporales de los ejes entrantes a un nodo no son consecutivas

iii. Los ejes entrantes a un nodo versionado no tienen etiquetas temporales consecutivas

iv. Existe un ciclo en un snapshot del documento

v. Dos nodos tienen el mismo valor para el atributo id.

En adelante, nos referiremos a cada una de ellas como inconsistencia de tipo *i*, tipo *ii*...etc

Las inconsistencias de tipo *v* no serán tratadas en este documento, puesto que no son inconsistencias temporales.

El segundo y tercer item de las condiciones, implican a su vez distintos tipos de inconsistencia, a saber

- Dos o más ejes incidentes a un nodo tienen etiquetas temporales con intersección no vacía, es decir, un nodo tiene más de un padre en un intervalo.
- Existen baches entre las etiquetas temporales de los ejes incidentes a un nodo

En este trabajo no analizaremos la corrección de inconsistencias de tipo *iii* dado que es análoga a la corrección de inconsistencias de tipo *ii*.

Definición 5 (Intervalo de una inconsistencia).
Diremos que una inconsistencia determinada se produce en el intervalo $T = [t_i, t_f]$ *si (a) el documento es inconsistente en cada instante de* T *debido a la misma o (b) existe un bache en el lifespan de un nodo en dicho intervalo.*

Ejemplos de los distintos tipos de inconsistencia se pueden apreciar en las Figuras 4.1, 4.2, 4.3.y 4.4.

Ejemplo 4 (Intervalo de una inconsistencia).
En la Figura 4.1 se puede observar una inconsistencia de tipo i, en la cual la etiqueta temporal del eje saliente del nodo node_1 tiene intervalo $[T_1, Now]$ *mientras que el lifespan del mismo es* $[0, T_3]$*, es decir, el intervalo del eje no está incluido en el lifespan del nodo. El intervalo de esta inconsistencia es* $[T_4, Now]$*, puesto que para cada instante de este intervalo el documento es inconsistente.*

En la Figura 4.2 tenemos una inconsistencia de tipo ii en el nodo node_2. Esto se debe a que los ejes incidentes a este nodo tienen intervalos $[0, T_1]$ *y* $[T_5, Now]$*, no consecutivas. El intervalo de esta inconsistencia es* $[T_2, T_4]$*, puesto que hay un bache en el lifespan de node_2 en este intervalo, y se cumple la condición (b) en la definición anterior. Sin embargo podemos ver que la condición (a) no se cumple, puesto cada snapshot del documento es consistente.*

En la Figura 4.3 podemos apreciar una inconsistencia de tipo iii. El nodo versionado compuesto por los nodos node_3 y node_4 es inconsistente, dado que los ejes incidentes a estos nodos tienen intervalos temporales $[0, T_3]$ *y* $[T_6, Now]$ *respectivamente. El intervalo de esta inconsistencia es* $[T_4, T_5]$*, nuevamente debido a la condición (b).*

En la Figura 4.4 se puede observar una inconsistencia de tipo iv, es decir, un ciclo. Este está formado por los nodos node_2 y node_3 en el intervalo $[T_4, T_6]$*. Está claro que las etiquetas temporales de los ejes salientes de ambos nodos se encuentran incluidas en el lifespan de éstos, y los ejes incidentes al nodo node_2 tienen etiquetas consecutivas. No obstante, ninguno de los dos es alcanzable desde la raíz en el intervalo mencionado, y esto se debe a la presencia del ciclo. El intervalo de esta inconsistencia es* $[T_4, T_6]$*, o sea, el intervalo en que se produce el ciclo, dado que se cumple la condición (a) de la definición en todo el intervalo.*

En el caso de inconsistencias de tipo *i*, a lo largo del documento incurriremos en un abuso de notación, refiriéndonos como *intervalo de la inconsistencia* a un intervalo en ocasiones mayor. Esto es, si el intervalo inconsistente según lo

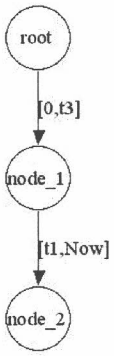


Figura 4.1: Ejemplo de inconsistencia 1. La etiqueta temporal del eje incidente al nodo node_2 no está incluida en el lifespan del nodo 1.

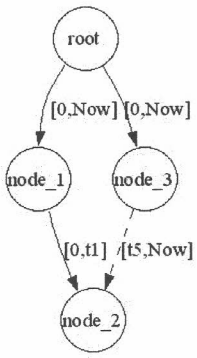


Figura 4.2: Ejemplo de inconsistencia 2. Las etiquetas temporales de los ejes incidentes al nodo node_2 no son consecutivas.

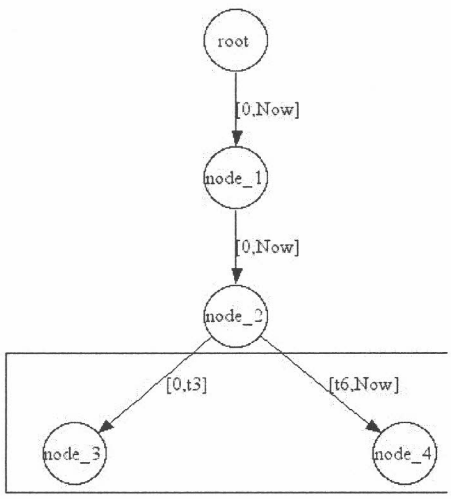


Figura 4.3: Ejemplo de inconsistencia 3. Las etiquetas temporales de los ejes incidentes a los nodos 3 y 4, únicos miembros de un nodo versionado, no son consecutivas.

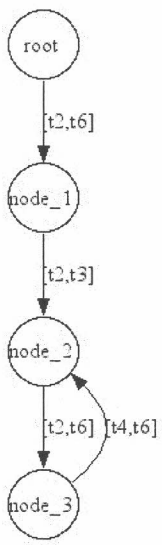


Figura 4.4: Ejemplo de inconsistencia 4. Existe un ciclo en [t4,t6] entre los nodos node_2 y node_3.

definido anteriormente fuese $[T_i, T_f]$ y el lifespan del nodo inconsistente fuese $[L_i, T_i - k]$ entonces incluiremos realmente en el intervalo de la inconsistencia a aquel que se extiende desde $T_i - k + 1$ hasta T_f. Las razones para esto residen en la simplificación de los métodos de corrección de la inconsistencia y serán explicadas en más detalle en 5.3.1.

En la siguiente sección, mostraremos un algoritmo para el chequeo de consistencia de un documento. Más adelante, se describirán las posibles soluciones para cada tipo de inconsistencia que puede presentarse en forma aislada, para concluir analizando el caso en que varias inconsistencias se presentan en un mismo documento.

La notación a utilizar en lo sucesivo, es la siguiente:

Notación: Sea G un grafo, entonces:

i. $\boldsymbol{V(G)}$ denota el conjunto de nodos de G. Se abreviará V en caso de que esté claro a que grafo se hace referencia

ii. $|\boldsymbol{V(G)}|$ o $|\boldsymbol{V}|$ denota el cardinal del conjunto de nodos de G

iii. $\boldsymbol{E(G)}$ o $\boldsymbol{E}$ es el conjunto de ejes de G

iv. $|\boldsymbol{E(G)}|$ o $|\boldsymbol{E}|$ es el cardinal del conjunto de ejes de G

v. $\boldsymbol{deg_{in}(n)}$ se refiere al grado de entrada del nodo n (cantidad de ejes incidentes a él) mientras que $\boldsymbol{deg_{out}(n)}$ denota el grado de salida (cantidad de ejes salientes del nodo)

4.2. Chequeo de consistencia

Un documento TXML válido debe verificar en todo momento las condiciones de la Definición 2. Por lo tanto, es necesario encontrar un mecanismo para poder verificar dichas condiciones. A continuación, presentamos un algoritmo que realiza este chequeo. Nos concentramos en las posibles inconsistencias temporales antes mencionadas (derivadas de las condiciones de consistencia 2), ya que como aclaramos con anterioridad, el aspecto temporal es el que nos interesa en este trabajo.

Los items 1 a 3 y 5 de las condiciones de consistencia presentadas en la Definición 2 son sencillos de verificar, puesto que basta con recorrer cada nodo y chequear condiciones triviales como que los intervalos dentro de una lista sean consecutivos. La única condición no trivial de comprobar es la 4, en la cual se debe verificar que cada snapshot sea un árbol, es decir, que no haya ciclos para un instante t y que todos los nodos sean alcanzables desde la raíz en todo instante t perteneciente a su lifespan.

Para realizar este último chequeo, utilizaremos las propiedades presentadas a continuación.

Como corolario de la Definición 3 (mcp), podemos aseverar que la etiqueta temporal del *mcp* de un nodo calculado desde la raíz ($mcp_{desdelaraiz}(n_i)$) es igual al lifespan del nodo. Más formalmente:

Proposición 1 (Igualdad lifespan-mcp).
Para todo nodo en un documento TXML:

$$T_{mcp_{desdelaraiz}(n_i)} = lifespan(n_i)$$

Demostración.

a) Supongamos que existe un n_i *tal que* $T_{mcp_{desdelaraiz}(n_i)} \neq lifespan(n_i)$
$\Rightarrow \exists t$ *tal que* $t \in lifespan(n_i) \wedge t \notin T_{mcp_{desdelaraiz}(n_i)}$
$\Rightarrow$ $D(t)$ no es un árbol, puesto que en el instante t, no hay un camino desde la raíz hasta el nodo. Por lo tanto en dicho instante, nos encontramos con un bosque en lugar de un árbol, lo cual viola la condición 4 de la Definición 2 (hay un ciclo, o una inconsistencia tipo i, que deja al snapshot disconexo)
$\Rightarrow_{\text{Definición 2}}$ D No es consistente $\Rightarrow_{\text{por hipótesis}}$ *Absurdo*
(puesto que el documento era originalmente consistente)

b) Supongamos que existe un n_i *tal que* $T_{mcp_{desdelaraiz}(n_i)} \neq lifespan(n_i)$
$\Rightarrow \exists t$ *tal que* $t \notin lifespan(n_i) \wedge t \in T_{mcp_{desdelaraiz}(n_i)}$
$\Rightarrow$ que existe un camino entre la raíz y el nodo n_i en el instante t,
pero el nodo n_i no tiene un eje incidente en ese instante ya que $t \notin lifespan(n_i)$
$\Rightarrow_{\text{por definición de mcps}}$ *Absurdo*

□

Proposición 2. *Dado un documento XML temporal en el que ningún nodo tiene más de un padre en un instante, si existe un ciclo en un intervalo, entonces existe algún nodo* n_i *tal que*

$$T_{mcp_{desdelaraiz}(n_i)} \neq lifespan(n_i)$$

Demostración.

Sea T un intervalo temporal en el cual el documento D tiene un ciclo. Sea n_i un nodo en dicho ciclo, se cumple por definición de lifespan que $lifespan(n_i) \supset T$ pero, $T_{mcp_{desdelaraiz}(n_i)} \cap T = \phi$ ya que si no fuera así, existiría un instante t para el cual habría un camino entre la raíz y el nodo; como ningún nodo tiene más de un padre en t (por hipótesis), entonces la raíz pertenecería al ciclo. □

Esta proposición será utilizada en el algoritmo de chequeo de consistencia 4 que mostraremos más adelante, utilizando el hecho de que si $T_{mcp_{desdelaraiz}(n_i)} = lifespan(n_i)$ para todos los nodos, entonces no existe un ciclo. En caso de que no se cumpla, sabemos que el documento es inconsistente. Si consideramos que las únicas inconsistencias posibles son las listadas en 4 entonces, si no existen inconsistencias de tipo *i*, tipo *ii* y tipo *iii*, concluiremos que hay un ciclo en el documento.
A continuación, presentamos los algoritmos necesarios para chequear ésta y las demás condiciones de consistencia. En lo siguiente, supondremos que la estructura de datos requerida para realizar los chequeos o cálculos correspondientes, ya se encuentran alojadas en memoria. El costo de parsear el documento XML para crear la estructura de datos pertinente se analizará en el capítulo de experimentación.

Algoritmo 1 (Cálculo del lifespan de un nodo).

```
Input:   nodo n
Output:  t = [ti,tf] intervalo temporal correspondiente al
         lifespan del nodo, si este es calculable, null sino

(1)TimeInterval lifespan(nodo n){
(2)  listaEtiquetas = []
(3)  t = null
(4)  para cada eje e=(no,n,Te) incidente a n
     begin
(5)     listaEtiquetas.append(Te)
     end
(6)  listaEtiquetas.sort()
(7)  t = listaEtiquetas[1]
(8)  para cada i entre 1 y listaEtiquetas.length()-1
     begin
(9)     si listaEtiquetas[i].tf() + 1 != listaEtiquetas[i + 1].tf()
        begin
(10)       return null
        end
(11)    t = t U listaEtiquetas[i + 1]
     end
(12) return t
 }
```

Teorema 3 (Correctitud del Algoritmo 1).
El Algoritmo 1 termina y calcula el lifespan del nodo apropiadamente.

Demostración.

Es fácil ver que el algoritmo termina. Realiza dos ciclos de $deg_{in}(n)$ iteraciones cada uno y un ordenamiento. El resto de las operaciones son constantes. Resuelve correctamente, ya que ordena las etiquetas de los ejes incidentes según sus instantes iniciales, y luego retorna null si estas no son consecutivas, o la unión de todas si lo son. □

Análisis de complejidad

Las líneas (2), (3) y (7) son de orden constante. El ciclo de la línea (4), realiza $deg_{in}(n)$ iteraciones, ya que recorre una única vez cada uno de los ejes incidentes al nodo. Las operaciones realizadas en este ciclo son de orden constante. El ordenamiento de la línea (6), puede realizarse en $deg_{in}(n) * log(deg_{in}(n))$ si se utiliza un algoritmo estilo quicksort, teniendo en cuenta que los elementos de la lista son las etiquetas de los ejes incidentes a los nodos y entonces su longitud es $deg_{in}(n)$ (el ordenamiento se ejecuta en función de los extremos iniciales de los intervalos temporales). Luego se realiza nuevamente un ciclo de $deg_{in}(n)$ iteraciones recorriendo los elementos de la lista ordenada, en el cual se llevan a cabo cálculos en orden constante. El orden del algoritmo es entonces

$$O(2deg_{in}(n) + deg_{in}(n) * log(deg_{in}(n))) \tag{4.1}$$
$$= O(deg_{in}(n)(log(deg_{in}(n)) + 2)) \tag{4.2}$$
$$\approx O(deg_{in}(n) * log(deg_{in}(n))) \tag{4.3}$$

En el peor caso, si consideramos que todos los ejes del documento inciden a un único nodo, este orden se transforma en $O(|E| * log(|E|)$, mientras en el caso promedio, suponiendo que todos los nodos tienen la misma cantidad de ejes incidentes, es decir, el grado de entrada de cada nodo es $\frac{|E|}{|V|}$, el orden es entonces, $O(\frac{|E|}{|V|} * log(\frac{|E|}{|V|})$
En el mejor caso (aquel en cada nodo tiene un único eje entrante), el algoritmo tiene orden constante, ya que todas las operaciones de listas, al igual que los ciclos, se realizan sobre un único elemento.

Algoritmo 2 (Chequeo de consistencia de los nodos versionados).

```
Input:   Un documento XML temporal g
Output:  true si el documento no contiene inconsistencias de
         tipo iii. false sino.

boolean chequearConsistenciaVersionados(Documento g){
(1)   Para cada nodo versionado n=(v, nList) en Vd de g
      begin
(2)      etiquetas = []
(3)      nombre = nList[1].getNombre()
```

```
(4)      Para cada ni nodo en nList
         begin
(5)          si ni.getNombre != nombre return false
(6)          Si ni tiene mas de un eje saliente return false
(7)          Si ni no tiene exactamente un eje incidente return false
(8)          Sea ei = eje incidente a ni
             begin
(10)            etiquetas.append(Tei)
             end
         end
(11)     Para cada etiqueta e, e' consecutivas en etiquetas
         begin
(12)        si e.final + 1 != e'.inicial
            begin
(13)            return false
            end
(14)    end
      end
(15)  return true
```

Teorema 4 (Correctitud del Algoritmo 2)**.**
El Algoritmo 2 termina y determina la presencia de nodos versionados inconsistentes.

Demostración.

- *Termina*: El ciclo principal se realiza $|V_d|$ veces. Los ciclos de las líneas (4) y (11) se realizan $len(nList)$ veces cada uno, donde $len(nList)$ es la longitud de la lista de nodos versionados presentes en el documento. Ambos ejecutan operaciones constantes suponiendo que el lifespan de los nodos padres de cada nodo versionado ya se halle calculado.
- *Es correcto*: Para cada nodo versionado, se chequean las condiciones de consistencia, a saber:
  - que cada nodo en la lista tenga un único eje incidente (línea (7)),
  - que tenga a lo sumo un eje saliente (línea (6))
  - que todas las etiquetas de los nodos sean iguales (línea (5))
  - las etiquetas temporales de los ejes incidentes a cada uno de los nodos de la lista son consecutivas (líneas (11) a (13))

Que el eje incidente al nodo tiene etiqueta temporal incluida en el lifespan del nodo padre se chequeará junto a la consistencia de los nodos no versionados, puesto que es una propiedad común a todos los nodos del documento. □

Análisis de complejidad

Ambos ciclos internos (línea (4) y (11)) se realizan $len(nList)$ veces, y sólo

incluyen operaciones de orden constante suponiendo que el lifespan de los nodos padres de cada nodo versionado ya se halle calculado.
$\Rightarrow \approx O(2len(nList)) \Rightarrow \sum_{(v,nList)\in V_d} O(2len(nList)) \approx O(2|E|)$ en el peor caso (suponiendo que todos los ejes inciden al mismo nodo, y este es versionado) y, $\approx O(2 * |V_d| \frac{|E|}{|V|})$ en el caso promedio, es decir, cuando cada nodo tiene la misma cantidad de ejes salientes.

Algoritmo 3 (Chequeo de consistencia de los nodos en un documento temporal).

```
Input:   Un documento XML temporal g
Output:  true si el documento no contiene inconsistencias de
         tipo i,ii y iii. false sino.

boolean chequearConsistenciaNodos(documento g){
(1)   Para cada nodo n en g:
      begin
(2)      l = lifespan(n)
(3)      si l es nulo y n no es la raiz return false
(4)      Para cada eje e saliente de n
         begin
(5)          Si Te no está incluido en el lifespan
             begin
(6)              return false
             end
         end
      end
(7)   return chequearConsistenciaVersionados(g)
```

Teorema 5 (Correctitud del Algoritmo 3).
El Algoritmo 3 termina y determina la presencia de inconsistencias de tipo i, tipo ii y tipo iii.

Demostración.

- *Termina*: Recorre a lo sumo una vez cada nodo, y en cada una de las iteraciones realiza operaciones sencillas. Con lo cual realiza $|V|$ iteraciones y termina, o sale antes al encontrar alguna inconsistencia. Además el Algoritmo chequearConsistenciaVersionados termina como vimos en el Algoritmo 2
- *Determina la existencia de inconsistencias de tipo i, tipo ii y tipo iii*:
  - *i.* En la líneas (4) a (6), chequea que no se produzcan inconsistencias de tipo *i*, al verificar que la etiqueta temporal de cada eje saliente se encuentra incluida en el lifespan.
  - *ii.* En la línea (3), verifica que no se produzcan inconsistencias de tipo *ii*, dado que el lifespan no es calculable si los intervalos de los ejes incidentes no son consecutivos (ver Algoritmo 1)

iii. Si hay algún nodo versionado, la línea (7) comprueba su consistencia (ver Algoritmo 2)

□

Análisis de complejidad

El ciclo principal se repite a lo sumo $|V|$ veces. En la líneas (2) y (3) se chequea que el nodo no presente inconsistencias de tipo i calculando el lifespan, cuyo orden es $O(deg_{in}(n) * log(deg_{in}(n)))$. A lo largo de todas las iteraciones esto suma,

$$\sum_n deg_{in}(n) * log(deg_{in}(n))$$

Si consideramos que $deg_{in}(n)$ es en promedio $\frac{|E|}{|V|}$ nos da una cota máxima de

$$\sum_{n \in V} \frac{|E|}{|V|} * log(\frac{|E|}{|V|}) = |E| * log(\frac{|E|}{|V|})$$

Las líneas (4) a (6) forman un ciclo que se repite por cada eje saliente de cada nodo visitado, realizando operaciones a orden constante, esto suma $O(|E|)$

Finalmente, el orden del algoritmo de chequeo de nodos versionados tiene un orden de $O(2 * |V_d| \frac{|E|}{|V|})$ en total, si consideramos que los ejes se encuentran distribuidos equitativamente entre los nodos. Es decir, en el caso promedio, tenemos en total un orden de

$$|E| * log(\frac{|E|}{|V|}) + 2 * |V_d| \frac{|E|}{|V|} =$$

$$|E| * (*log(\frac{|E|}{|V|}) + 2 * \frac{|V_d|}{|V|})$$

Hasta aquí vimos como detectar las inconsistencias de tipo i, tipo ii y tipo iii, las cuales sólo requieren chequear cada nodo por separado. A continuación, presentamos el algoritmo correspondiente a la detección de ciclos. En él, utilizaremos la idea de que el lifespan de un nodo es igual al mcp desde la raíz, o en otras palabras, que se debe poder llegar a un nodo desde la raíz en todo instante perteneciente al lifespan del mismo, y en ningún otro (ver Proposición 1). La idea se implementa aquí recorriendo todo el documento, atravesando un eje siempre y cuando se haya llegado a su nodo origen en toda la extensión del intervalo temporal. Intentaremos en lo posible recorrer una única vez cada nodo, esperando a que todos los ejes incidentes hayan sido transitados en todo su intervalo, para transitar por un eje saliente. Esto no siempre será posible, puesto que si bien los ciclos temporales no son permitidos, pueden existir ciclos

no temporales. Un ejemplo de esta situación puede apreciarse en la Figura 4.5, donde los nodos node_1 y node_2 pertenecen a un ciclo del grafo si no consideramos las etiquetas temporales de los ejes, pero teniéndolas en cuenta no (es decir, no hay ciclos en ningún snapshot). El problema es que para recorrer todos los ejes incidentes a node_1 es necesario recorrer el eje saliente del node_2, pero si para ello esperamos a recorrer todos los ejes incidentes a este último, nos encontramos frente a un deadlock, puesto que uno de los ejes proviene de node_1. Para evitar este problema, transitaremos ejes salientes de nodos cuyos ejes incidentes no han sido aún totalmente transitados, en el caso en que nos quedemos sin nodos para recorrer que sí cumplan esta condición. Obviamente los ejes salientes que podremos transitar deberán cumplir que su intervalo temporal esté incluido dentro del intervalo al que se ha llegado al nodo origen, para evitar recorrer ejes inconsistentes. De esta manera evitamos la situación de deadlock descripta anteriormente, minimizando a su vez la cantidad de veces que pasamos por cada nodo.

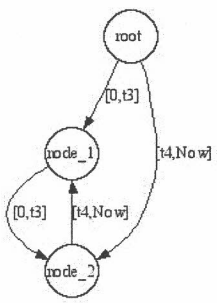


Figura 4.5: Ejemplo de ciclo no temporal. Los nodos node_1 y node_2 forman un ciclo no temporal, dado que existe un ciclo en el grafo subyacente (quitando las etiquetas temporales), teniendo en cuenta las etiquetas temporales de los ejes, no hay ciclo en ningún snapshot

A continuación presentamos el algoritmo que realiza este procedimiento, luego mostramos dos seguimientos del mismo, uno para el caso en el cual el documento presenta un ciclo, y otro para el caso de un documento consistente. Estos ejemplos ayudarán a clarificar el funcionamiento del algoritmo y complementarán a su vez la explicación recién brindada.

Algoritmo 4 (Búsqueda de ciclos en un documento temporal).

```
INPUT: Un documento XML con etiquetas temporales en los ejes g, tal que
     chequearConsistenciaNodos(g) = true
OUTPUT: true si es consistente, false sino

boolean chequearConsistencia(g){

(1)   Cola nodos     = [g.getRoot()]
(2)   Cola nodosWait = []
(3)   Mientras !nodos.empty() hacer
        begin
(4)         n = nodos.first()
(5)         Si !n.isFinalizado()
```

```
            begin
(6)            listaEtiquetas = [Te tal que e es un eje entrante a n
                                 y !e.isTransitado()]
(7)            Para cada eje e saliente de n
                  begin
(8)                  Si !e.isTransitado()
                          begin
(9)                          si Te intersección Te' != vacio para algún
                                              eje e' en listaEtiquetas
(10)                         e.setTransitable(false)
                          end
                     sino
                          begin
(11)                        e.setTransitable(true)
                          end
                  end
            end

(12)      Para cada eje e=(n,nf) saliente de n no transitado hacer:
            begin
(13)          Si ( e.isTransitable() || n.isFinalizado() )
                begin
(14)               e.setTransitado(true)
(15)               nf.setEjesTransitados(nf.getEjesTransitados()++)
(16)               Si todos los ejes incidentes a nf fueron transitados
                             (nf.getEjesTransitados() == nf.getDegIn() )
                     begin
(17)                   nodos.append(nf)
                     end
(18)               Sino
                     begin
(19)                   Si !nf.isVisitado()
                         begin
(20)                       nodosWait.append(nf)
(21)                       nf.setVisitado(true)
                         end
                     end
                end
            end
(22)      Si nodos.empty() and !nodosWait.empty()
            begin
(23)          n = nodosWait.first()
(24)          nodos.append(n)
(25)          n.isVisitado(false)
            end
        end
(26)  Para cada nodo n en g hacer
        begin
(27)      si !n.isFinalizado()
(28)        return false
        end
(29)  return true
}
```

Antes de continuar, detengámonos a analizar como funciona el algoritmo. Diremos que 'transitamos' un eje cuando lo utilizamos para agregar el nodo destino a alguna de las dos colas que se utilizan. La cola `nodos` contiene en general los nodos cuyos ejes incidentes fueron transitados en toda su totalidad. Contendrá un nodo que no cumpla con esta condición en el caso de encontrarnos con un deadlock. Los nodos que tienen al menos un eje incidente transitado, pero que no han sido aún finalizados, se encuentran en la cola `nodosWait`. De esta manera, al encontrarnos frente a un deadlock es sencillo tomar un nodo para poder continuar la validación.
El ciclo principal comienza a recorrer el documento a partir de la raíz. Dentro

de este ciclo, primero se verifica si el nodo que se va a analizar está finalizado. Si lo está, sabemos que podremos transitar todos sus ejes salientes. Sino (nos encontramos con un deadlock y tomamos un nodo no finalizado para continuar la validación), debemos mirar cuales de estos ejes pueden ser transitados. Sólo podrán ser transitados ejes cuya etiqueta temporal no incluya instantes en los cuales no se ha arribado al nodo aún. Marcamos entonces en la línea (11) estos ejes como 'transitables'. En el ciclo de (12) transitamos todos los ejes salientes del nodo permitidos y los marcamos como transitados para no volver a utilizarlos luego. Para cada uno de estos ejes, analizamos su nodo destino. Si detectamos que ya transitamos todos los ejes incidentes a éste, lo marcamos como finalizado y lo agregamos a la cola `nodos`, si no es el caso, lo agregamos a la cola `nodosWait` y lo marcamos como 'visitado' indicando que ya está en la cola y que no debemos volver a agregarlo. Al finalizar este ciclo, nos fijamos si hay más nodos para recorrer. Si hemos agotado los nodos en `nodos` y en `nodosWait`, significa que la validación ha concluido. Debemos recorrer todos los nodos del documento y verificar que haya sido finalizado. Si existe algún nodo sin finalizar, deducimos que hay un ciclo temporal y el documento no es consistente. Si en cambio agotamos los nodos en `nodos` pero aún quedan nodos en `nodosWait`, tomamos uno a uno los nodos en esta cola y vemos si hay algún eje saliente transitable, de manera de romper los ciclos no temporales y continuar con la validación.
A continuación desarrollaremos dos ejemplos, uno con ciclo, y otro con un documento consistente, explicando el paso a paso de la ejecución del algoritmo.

Ejemplo 5 (Ejecución del algoritmo para un documento con un ciclo).
Veremos paso a paso la ejecución del algoritmo para el documento de la Figura 4.4

```
Cola nodos = [ root ]
Cola nodosWait = [ ]
```

Primera iteración:

`nodos` *no está vacía, por lo tanto ingresamos en el ciclo principal.*

`n = root, nodos = [ ]`

`n` *está finalizado por definición, ya que finalizado implica que*

`n.ejesTransitados == n.degIn`

Como la raíz no tiene ejes incidentes, `n.degIn = 0`. *Esto hace que no se analicen los ejes salientes para ver si son transitables.*
Para el ciclo de (12), tenemos para analizar solamente el eje que sale hacia `node_1, nf = node_1`. *Como* `n` *está finalizado, ejecutamos las líneas (14) a (16). Marcamos el eje como transitado, aumentamos la cantidad de ejes tran-*

sitados del nodo destino (`node_1` *) en 1 y, como* `node_1` *queda entonces finalizado* (`nf.ejesTransitados == nf.degIn == 1`) *agregamos* `node_1` *a nodos. Luego saltamos a la línea (22). Como esta condición da falso puesto que* `nodos == [ node_1 ]` *y por lo tanto no es vacía, finaliza la primera iteración, siendo el estado de las estructuras de datos al momento:*

`nodos == [ node_1 ]`
`node_1` *y* `root` *finalizados*
`nodosWait == [ ]`

Segunda Iteración:

`nodos` *no está vacía, por lo tanto ingresamos en el ciclo principal.*

`n = node_1, nodos = [ ]`

`node_1` *está finalizado, por lo tanto no se analizan los ejes salientes para ver cual es transitable. Saltamos a la línea (12), y tenemos solamente para analizar el eje que incide a* `node_2`. *La condición en (13) es verdadera puesto que* `node_1` *está finalizado. Marcamos entonces el eje como transitado, aumentamos la cantidad de ejes transitados de* `node_2` *en 1 y chequeamos para ver si está finalizado. Como* `degIn(node_2)` *es 2 y ejes transitados es 1, el nodo no se encuentra finalizado. Saltamos entonces a la línea (18). Como* `node_2` *no fue visitado, se agrega a* `nodosWait` *y se marca como visitado. Tenemos entonces*

`nodos == [ ]` *y* `nodosWait == [ node_2 ]`,

lo que provoca que la condición en (22) devuelva `true`. *Ejecutando las líneas siguientes, obtenemos* `nodos == [ node_2 ], nodosWait == [ ]` *y* `node_2` *no visitado. El estado de las estructuras de datos al final la segunda iteración es entonces:*

`nodos == [ node_2 ], nodosWait == [ ]`
`node_2` *no visitado*
`node_1, root` *finalizados*

Tercera Iteración:

`nodos` *no está vacía, por lo tanto ingresamos en el ciclo principal.*

`n = node_2, nodos = [ ]`

El nodo node_2 no está finalizado, por lo tanto será necesario analizar los ejes salientes para ver cuales se pueden transitar. `listaEtiquetas=[[`T_4`,` T_6`]]` *puesto que es la etiqueta temporal del eje incidente a* `node_2` *no transitado. el único*

eje saliente tiene etiqueta $[T_2, T_6]$*, y la intersección de este intervalo con los intervalos en la lista de etiquetas no es vacía, por lo tanto, según la condición (9), se debe establecer* `n.setTransitable(false)`.

En el ciclo (12) analizamos el eje, pero en (13) descubrimos que éste no es transitable y no hacemos nada.

Saltamos a la línea (22), donde la condición da falso, ya que ambas colas están ahora vacías. De vuelta en la línea (3) nos encontramos con una cola vacía y salimos del ciclo.

En la línea (26) recorremos el documento buscando nodos no finalizados. Como `node_2` *y* `node_3` *nunca fueron finalizados, el algoritmo retornará* `false` *en la línea (28). El documento es* ***inconsistente.***

Ejemplo 6 (Ejecución del algoritmo para un documento sin ciclos).

Veremos paso a paso la ejecución del algoritmo para el documento de la Figura 4.5

```
Cola nodos = [ root ]
Cola nodosWait = [ ]
```

Primera iteración:

`nodos` *no está vacía, por lo tanto ingresamos en el ciclo principal.*

```
n = root, nodos = [ ]
```

`El nodo n` *está finalizado por definición, ya que finalizado indica que*

```
n.ejesTransitados == n.degIn
```

Como la raíz no tiene ejes incidentes, `n.degIn = 0`. *Esto hace que no se analicen los ejes salientes para ver si son transitables.*
Para el ciclo de (12), tenemos para analizar el eje que sale hacia `node_1, nf` *es entonces* `node_1`. *Como* `n` *está finalizado, ejecutamos las líneas (14) a (16). Marcamos el eje como transitado, aumentamos la cantidad de ejes transitados del nodo destino (*`node_1` *) en 1 y, como* `node_1` *no queda finalizado ni está visitado* `(nf.ejesTransitados <nf.degIn == 2)` *agregamos* `node_1` *a* `nodosWait` *en la línea (20). En este caso tenemos un segundo eje para analizar, el que incide al nodo* `node_2`. *El eje es transitable, por lo tanto se marca como transitado, se aumenta la cantidad de ejes transitados del nodo destino,* `node_2`, *a 1. Como el nodo tiene dos ejes incidentes, no queda finalizado, y como tampoco está visitado se agrega a la cola* `nodosWait`.
No hay mas ejes, saltamos entonces a la línea (22) donde encontramos que `nodos == [ ]` *pero* `nodosWait == [node_1, node_2]`. *Por lo tanto se saca el*

primer elemento de `nodosWait` *y se coloca en* `nodos`.

El estado de las estructuras de datos al finalizar la primer iteración es entonces:

`nodos == [ node_1 ]`
`nodosWait == [ node_2 ]`
`node_2` *no finalizado y visitado*
`node_1` *no finalizado y no visitado*
`root` *finalizado y visitado*

Segunda Iteración:

`nodos` *no está vacía, por lo tanto ingresamos en el ciclo principal.*

`n = node_1, nodos = [ ]`

El nodo `node_1` *no está finalizado, por lo tanto se deben analizar los ejes salientes. El único eje saliente tiene etiqueta* $[T_0, T_3]$ *y el único eje incidente no transitado tiene etiqueta* $[T_4, Now]$. *Las etiquetas tienen intersección no vacía, y se puede marcar el eje como transitable. Ahora pasamos al ciclo en (12). Sólo tendremos un eje para analizar. La condición en (13) es verdadera puesto que el eje fue marcado como transitable. Marcamos entonces el eje como transitado, aumentamos la cantidad de ejes transitados de* `node_2` *(nodo destino del eje) en 1 y chequeamos para ver si está finalizado. Como* `degIn` *de* `node_2` *es 2 y el número de ejes transitados es 2, el nodo se encuentra finalizado y puede ser agregado en nodos (linea (17)). No quedan más ejes para analizar, saltamos entonces a la línea (22). Aquí vemos que* `nodos` *ahora no es vacía* `(nodos==nodosWait==[node_2])` *y termina la iteración. El valor de las estructuras de datos al momento es el siguiente:*

`nodos == [ node_2 ], nodosWait == [ node_2 ]`
`node_1` *no visitado ni finalizado*
`node_2` , `root` *finalizados*
`node_2` *visitado*

Tercera Iteración:

`nodos` *no está vacía, por lo tanto ingresamos en el ciclo principal.*

`n = node_2, nodos = [ ]`

`node_2` *está finalizado, por lo tanto no será necesario analizar los ejes salientes. En el ciclo (12) analizamos el único eje que queda sin transitar, como* `node_2` *está finalizado, lo marcamos como transitado, aumentamos la cantidad de ejes transitados incidentes a* `node_1` *a 2 y como ahora* `node_1` *queda finalizado, lo agregamos a* `nodos`. *Saltamos a la línea (22), donde la condición evalúa a ver-*

dadero, ya que `nodos == [ node_1]`.
El estado de las estructuras de datos al finalizar la iteración es el siguiente:

`nodos == [ node_1 ], nodosWait == [ node_2 ]`
`node_1` , `node_2` , *root finalizados*

Cuarta Iteración:

`nodos` *no está vacía, por lo tanto ingresamos en el ciclo principal.*

`n = node_1`

El nodo está finalizado por lo tanto pasamos a la línea (12). No quedan ejes salientes sin recorrer, por lo tanto pasamos directamente a la línea (22) donde, dado que `nodos` *está vacía, quitamos* `node_2` *de* `nodosWait` *y lo pasamos a* `nodos` *. Termina la cuarta iteración con el siguiente estado de las estructuras de datos:*

`nodos == [ node_2 ], nodosWait == [ ]`
`node_1, node_2, root` *finalizados*

Quinta Iteración:

`nodos` *no está vacía, por lo tanto ingresamos en el ciclo principal.*

`n = node_2`

El nodo está finalizado por lo tanto pasamos a la línea (12). No quedan ejes salientes sin recorrer, por lo tanto pasamos directamente a la línea (22) donde como nodos está vacía al igual que `nodosWait`, *no podemos hacer más nada. Al intentar ingresar nuevamente en el ciclo principal, encontramos que no hay más nodos para analizar y pasamos a la línea (26). En este ciclo final recorremos el documento en busca de un nodo no finalizado. Como todos los nodos fueron finalizados, el algoritmo retorna* `true`. *El documento es entonces* ***consistente.***

Teorema 6 (Correctitud del Algoritmo 4)**.**
El Algoritmo 4 termina y determina correctamente si existen ciclos en el documento.

Demostración.

1. *El algoritmo termina*: En cada iteración principal, visita sólo (algunos o todos) los ejes salientes de un nodo que aún no han sido transitados, y

agrega a la lista los nodos a los que inciden dichos ejes. En algún momento, ya se habrán recorrido todos los ejes posibles (dado que son finitos), y por lo tanto ningún nodo será agregado a la lista, con lo cual se terminará de iterar por los nodos que quedan en ella sin agregar más elementos, llegando en algún momento al final de la misma.

2. *Resuelve*:

   a) Si hay algún ciclo en el documento, el algoritmo lo detecta
   Supongamos que hay un ciclo y el algoritmo no lo detecta (retorna true). Sean $n_1..n_k$ los nodos del ciclo y T su intervalo temporal. Que el algoritmo retorne `true` significa que todos los nodos del documento fueron finalizados, incluyendo $n_1..n_k$. Esto significa que los nodos $n_1..n_k$ fueron visitados en T, ya que para ser finalizados deben tener todos los ejes incidentes transitados en todo el intervalo. Sea n_1 el primero de los nodos del ciclo en ser visitado en T, tiene que haber sido alcanzado desde un nodo n cuyos ejes incidentes en T fueron transitados ya. Cómo n_1 es el primero del ciclo en ser visitado en dicho intervalo, se deduce que n no pertenece al ciclo. Por lo tanto, n_1 tiene un padre fuera del ciclo en T y otro dentro del ciclo. Esto es absurdo puesto que por precondición, el documento no presenta inconsistencias de tipo *ii* o tipo *iii*

   b) Retorna `false` sólo si hay un ciclo
   Supongamos que el algoritmo deja nodos sin visitar y no hay un ciclo. Sea n_1 un nodo sin finalizar. Como no hay inconsistencias de tipo i, todos los ejes incidentes a n_1 tienen su etiqueta temporal dentro del lifespan del nodo origen. Sea $\epsilon_1(n_2, n_1, T)$ un eje no transitado incidente a n_1, entonces el nodo n_2 tampoco pudo haber sido finalizado, y alguno de sus ejes incidentes en T tampoco transitado, ya que sino el eje habría sido transitado. Sea $\epsilon_2(n_3, n_2, T)$ un eje incidente a n_2 no transitado, repitiendo el razonamiento realizado para n_1, concluímos que n_3 tampoco está finalizado. Podemos entonces transitar un eje $\epsilon_3(n_4, n_3, T)$ no transitado anteriormente, cuyo nodo origen no está finalizado. Si continuamos transitando los ejes no transitados anteriormente de esta manera, vamos armando un camino de nodos no finalizados en T, pero como no hay ciclos, no se repiten nodos en el camino. Por lo tanto en algún momento se llega hasta la raíz, pero esta no tiene ejes incidentes y por ende, sus ejes salientes siempre son transitados. Absurdo, no hay nodos sin finalizar o hay un ciclo en el documento.

□

Análisis de complejidad

Cada nodo puede ser visitado más de una vez, dependiendo de la cantidad de ejes incidentes que éste tenga. Las líneas (5) a (11) en el peor caso, recorrerán toda los nodos en la lista *nodosWait* antes de hallar uno con ejes salientes transitables. La lista puede tener como máximo $|V|$ nodos. En la línea (6) se recorren todos los ejes incidentes a un nodo, aportando del orden de $|E|$ operaciones cada vez que recorremos la lista nodosWait en su totalidad. Por otra parte, el ciclo en (7) se ejecuta $n.deg_{out}$ veces para cada nodo en la cola, y realiza del orden de $n.deg_{in}$ operaciones en la línea (9), dando un total de $|E|^2$ al recorrer todos los nodos. Cada vez que nos quedamos sin nodos en la cola *nodos* realizamos en el peor caso entonces $|E|^2 + |E|$ operaciones. En el peor caso, siempre encontraremos un nodo en nodosWait que tenga un eje saliente transitable (sino, el algoritmo termina). Cada vez que encontremos tal nodo, transitaremos al menos un eje. Entonces, la cantidad máxima de veces que se ejecutan las líneas (5) a (11) es de $|E|$ veces (una vez por cada eje no transitado), obteniendo finalmente un orden de $|E|^3 + |E|^2 + |E|$.

- el ciclo de la línea (12) se realiza $deg_{out}(n)$ veces para cada nodo visitado en el ciclo principal. Todas las operaciones son de orden constante aportando entonces un orden total de $deg_{out}(n)$

- las líneas (22) a (25) son de orden constante
  El ciclo principal tiene entonces un orden de $\sum_{n \in |V|} deg_{out}(n) \approx O(|E|)$ teniendo en cuenta que por mas que se pueda pasar por un nodo más de una vez, cada eje será recorrido una única vez como máximo.

- el ciclo final se realiza $|V|$ veces en el peor caso (cuando no hay ciclos en el documento), y las operaciones en él son también de orden constante

Sumando entonces, tenemos para el peor caso:

$$O(|E|^3 + |E|^2 + 2|E| + |V|)$$

Si consideramos el caso en que no se producen ciclos no temporales, entonces, las líneas (5) a (11) nunca se ejecutan, puesto que cuando se vacíe la lista de nodos por primera vez, todos los nodos van a haber sido visitados y por ende todos los nodos en la segunda lista estarán finalizados. Entonces el orden para este caso sería:

$$O(|E| + |V|)$$

4.3. Resumen

En este Capítulo presentamos y ejemplificamos las distintas inconsistencias que se pueden presentar en un documento XML temporal basándonos en las condiciones de consistencia planteadas en la Definición 2 (Seccion 4.1). Definimos el concepto de *intervalo de una inconsistencia* y enunciamos y demostramos propiedades sobre los documentos TXML que fueron utilizados luego para elaborar algoritmos de chequeo de consistencia eficientes (Sección 4.2). Para cada uno de los algoritmos presentados, demostramos su correctitud y su orden de ejecución.
A continuación, veremos en detalle las posibles correcciones a cada tipo de inconsistencia.

Capítulo 5

Resolución de inconsistencias

Hasta aquí hemos estudiado cómo comprobar la consistencia de un documento. Si el documento resulta consistente, podemos entonces aplicar cualquier operación de update que deseemos, siempre que luego de aplicada deje al documento en otro estado consistente. Ahora, qué sucede si el documento no es consistente? Simplemente lo descartamos? Esto podría ser una opción. Otra posibilidad sería simplemente listar todas las inconsistencias encontradas para que un usuario experto decida como corregir cada una de ellas. En los próximos capítulos, veremos algunas soluciones para corregir en forma automática las inconsistencias halladas. Analizaremos las posibles soluciones para cada tipo de inconsistencia considerando que hay una única inconsistencia dentro del documento,luego estudiamos el caso mas realista en el cual existe mas de una inconsistencia simultaneamente en el mismo documento.

En este capítulo analizaremos cada uno de los tipos de inconsistencia listados en la Definición 2 y veremos sus posibles soluciones. En cada caso, trataremos las inconsistencias aisladas, es decir, como única inconsistencia dentro del documento. Nos ocuparemos principalmente de las inconsistencias de tipo i y tipo iv ya que, como veremos más adelante, las otras pueden resolverse con una simple reescritura (esto es, una modificación sintáctica) del documento.

5.1. Definiciones preliminares

Existen varios métodos para resolver cada una de las inconsistencias presentadas en el Capítulo 3. A continuación, analizaremos formalmente las posibles soluciones y veremos de qué manera decidir cual adoptar en caso de que existan varias posibilidades.

Necesitaremos algunas definiciones y ejemplos antes de comenzar con el análisis del problema.

Operaciones entre intervalos

Definición 6 (Comparación de intervalos).
Diremos que un intervalo T_1 es ***posterior*** *(o mayor) a otro intervalo T_2, denotado $T_1 \succ T_2$ siempre que $(T_1)_f > (T_2)_f$. Análogamente diremos que un intervalo T_1 es* ***anterior*** *(o menor) a otro intervalo T_2, denotado $T_1 \prec T_2$ siempre que $(T_1)_f < (T_2)_f$*

Operaciones con ejes

Notación: Sea $T = [t_i, t_f]$ un intervalo temporal, $(T)_i = t_i$, $(T)_f = t_f$

Sea D un documento XML con intervalos temporales en los ejes. Sea $\epsilon(n_i, n_f, T_e)$ un eje del documento y T_e su etiqueta temporal,

Definición 7 (Eliminar un eje).
Eliminar un eje e del documento consiste en quitarlo físicamente para todo tiempo t. (Es equivalente a eliminar un eje de un grafo no temporal)

Ejemplo 7.
En la Figura 5.1, podemos ver el grafo de una porción de un documento antes y después de eliminar del documento el eje (node_2, node_1) según la Definición 7.

Figura 5.1: Ejemplo de eliminación de eje. a) Porción de documento antes de eliminar el eje (node_2, node_1). b) Luego de eliminar el eje

Definición 8 (Eliminar un eje en un instante).
Eliminar e en un instante $t \in T_e$ significa:

$$\begin{cases} \textit{Eliminar el eje (ver Definición 7)} & \textit{si } (T_e)_f = (T_e)_i = t \\ \textit{hacer } (T_e)_f = t - 1 & \textit{si } (T_e)_f = t \ \wedge \ (T_e)_i < t \\ \textit{hacer } (T_e)_i = t + 1 & \textit{si } (T_e)_i = t \ \wedge \ (T_e)_f > t \\ \textit{Duplicar el nodo al que incide el eje a eliminar (Definición 10) en el instante t, y eliminar el eje en t} & \textit{si } (T_e)_i < t < (T_e)_f \end{cases}$$

Ejemplo 8.

En la Figura 5.2 podemos observar el resultado de eliminar el eje (root, node_1) en el instante t_1. Se puede apreciar que el eje no fue físicamente eliminado, sino que cambió su intervalo a $[0, t_0]$ quitando así de su intervalo de validez solamente el instante t_1.

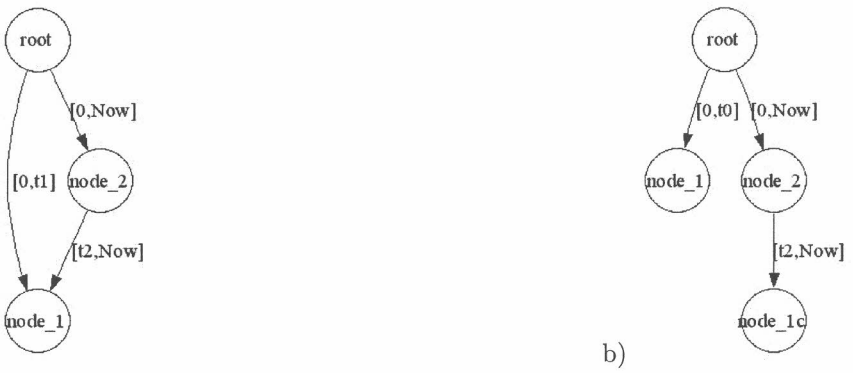


Figura 5.2: Ejemplo de eliminación de eje en un instante. a) Porción de documento antes de eliminar el eje (root, node_1) en el instante t1. b) Luego de eliminar el eje (root, node_1) en el instante t1

Definición 9 (Eliminar un eje en un intervalo)**.**

Eliminar el eje e en un intervalo I significa eliminar el eje e para cada instante $t, t \in I \bigcap T_e$

Ejemplo 9.

En la Figura 5.3 podemos observar un fragmento de documento antes y después de eliminar el eje (n_2, n_1) en el intervalo $[t_5, Now]$. Se puede apreciar que el eje no fue físicamente eliminado, sino que se cambió su intervalo a $[t_2, t_4]$.

Definición 10 (Duplicación de nodos a un instante)**.**

Sea n_i un nodo dentro del documento. Duplicar un nodo a un instante t comprende los siguientes pasos:

Figura 5.3: Ejemplo de eliminación de eje en un intervalo. a) Porción de documento antes de eliminar el eje (node_2, node_1) en el intervalo $[t_5, Now]$. b) Luego de eliminar el eje (node_2, node_1) en el intervalo $[t_5, Now]$

i. Copiar el nodo con toda la información que este contiene, formando el nodo n_{ic}.

ii. Eliminar todos los ejes salientes del nodo original para todo instante posterior a t.

iii. Eliminar todos los ejes salientes del nodo copia para todo instante previo al instante $t + 1$.

iv. Realizar el siguiente procedimiento para eliminar los ejes incidentes duplicados
Sean $\epsilon_1(n_1, n_i, T_1)...\epsilon_k(n_k, n_i, T_k)$ los ejes incidentes a n_i tal que $(T_j)_f \geq t$

- *Copiar los ejes incidentes luego del instante t al nodo n_{ic}*

$$\begin{aligned} &\textit{Crear los ejes } \epsilon'_1(n_1, n_{ic}, T'_1)...\epsilon'_k(n_k, n_{ic}, T'_k) \textit{ donde} \\ &\quad T'_j = [t+1, (T_j)_f] \textit{ si } t \in T_j \\ &\quad T'_j = T_j \textit{ sino} \\ &\textit{para } j = 1..k \end{aligned}$$

- *Eliminar los ejes $\epsilon_1(n_1, n_i, T_1)...\epsilon_k(n_k, n_i, T_k)$ en el intervalo $[t + 1, (T_j)_f]$*

Ejemplo 10. *En el ejemplo de la Figura 5.4 vemos el resultado de duplicar el nodo* `node_1` *al instante* `t10`*. El nodo original reduce su lifespan a* `[0,t9]` *y sólo conserva ejes salientes en este intervalo. El nuevo nodo tiene lifespan* `[t10, Now]` *y sus ejes salientes tienen intervalos incluidos en su lifespan. Para el caso del eje que va de* `node_1` *a* `node_2` *en* `[t10, t10]` *se debió dividir el eje en dos, uno en* `[t0, t9]` *partiendo del nodo* `node_1` *original, y el otro en* `[t10, t10]` *saliente de* `node_1c` .

Figura 5.4: Ejemplo de duplicación del nodo node_1 al instante t_{10}. a) Porción del documento antes de duplicar el nodo node_1. b) Porción del documento luego de duplicar el nodo node_1.

Definición 11 (Expansión de intervalo).
Expandir el intervalo T_e hasta un instante t se define como:

$$\begin{cases} hacer(T_e)_f = t & si\ t > (T_e)_f \\ hacer(T_e)_i = t & si\ t < (T_e)_i \end{cases}$$

Ejemplo 11.
En la Figura 5.5 se puede observar el efecto de expandir el eje entre (root, node_1) hasta el instante t_{10}

Figura 5.5: Ejemplo de expansión de un eje a un instante. a) Porción del documento antes de expandir. b) Porción del documento luego de la expansión.

Definición 12 (Reducción de intervalo).
Reducir el intervalo T_e a un intervalo $T' \subset T_e$ es eliminar el eje e en los intervalos $[(T_e)_i, T'_i - 1], [T'_f + 1, (T_e)_f]$. Reducir a derecha el intervalo T_e a un instante $t < (T_e)_f$ es eliminar el eje en el intervalo $[t + 1, (T_e)_f]$. Reducir a

izquierda el intervalo T_e a un instante $t > (T_e)_i$ es eliminar el eje en el intervalo $[(T_e)_i, t-1]$. En general, hablaremos de Reducción a derecha simplemente como Reducción, siendo claro por el contexto a que definición estamos haciendo referencia.

Ahora estamos en condiciones de analizar las posibles soluciones a cada tipo de inconsistencia. Por simplicidad, comenzaremos el análisis con las inconsistencias tipo *ii*, puesto que es el tipo más sencillo de resolver al tratarse de una inconsistencia sintáctica. Además, dado que esta inconsistencia puede generarse al intentar resolver las más complejas, es útil antes de analizar estas, contar con los conceptos necesarios para su solución.
En la mayoría de los casos, se tendrá más de una posibilidad para la corrección de una inconsistencia. Analizaremos cada una de ellas y enunciaremos un criterio de selección entre los posibles métodos de corrección, concluyendo con la presentación de los algoritmos que los implementan.

5.2. Resolución de inconsistencias de tipo *ii*

Este tipo de inconsistencia se produce cuando los ejes incidentes a un nodo no tienen etiquetas temporales consecutivas. Este tipo de inconsistencias tiene dos manifestaciones posibles:

- Solapamiento, cuando los intervalos de las etiquetas temporales de dos ejes incidentes a un nodo dado tienen intersección no vacía. Un ejemplo de este caso puede apreciarse en la Figura 5.6, donde los dos ejes incidentes al nodo node_2 se solapan en el intervalo $[t_2, t_3]$.
- Baches, cuando hay un intervalo temporal en el cual un nodo no tiene padre, pero sí lo tiene antes y después de este intervalo. Un ejemplo de este caso puede apreciarse en la Figura 5.7 donde el nodo node_2 no tiene eje incidente a él en el intervalo temporal $[t_4, t_5]$.

Para resolver los solapamientos, basta con tomar uno de los ejes cuyo intervalo temporal se solapa con otro, y eliminarlo en el intervalo de la inconsistencia. Asumiendo que la inconsistencia se encuentra aislada, no importa que eje se elija para la eliminación.

En cuanto a la resolución de los baches, existen varias soluciones posibles:

i. Descartar los ejes entrantes a partir del bache. Es decir, eliminar físicamente todos los ejes cuyas etiquetas temporales comiencen luego del bache.

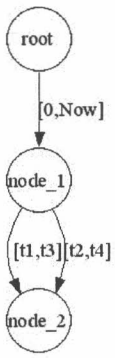


Figura 5.6: Ejemplo de solapamiento

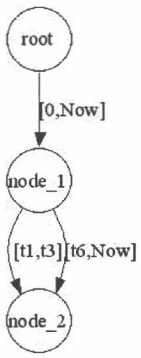


Figura 5.7: Ejemplo de bache temporal

ii. Tratar la inconsistencia como una inconsistencia sintáctica y duplicar el nodo. O sea, transformar el nodo en dos, uno en el cual las etiquetas de los ejes incidentes y salientes están incluidas en el intervalo anterior al bache, y otro con los ejes con etiquetas temporales posteriores al bache.

iii. Expandir los intervalos temporales de los ejes de manera de cubrir el bache, ya sea la última etiqueta antes del bache, o la primera posterior al mismo.

La primera y la última opción pueden provocar nuevas inconsistencias de tipo *i*. La primera, puesto que el nodo inconsistente puede tener ejes salientes en el intervalo correspondiente a los ejes incidentes descartados, mientras que en la última, se puede exceder el lifespan del nodo del que sale el eje cuyo intervalo es expandido.
La segunda opción, además de no provocar nuevas inconsistencias, surge de considerar a una desde el punto de vista sintáctico. La duplicación genera un nuevo documento semánticamente equivalente pero sintácticamente consistente.

A continuación presentamos los algoritmos para resolver estas inconsistencias según la alternativa *ii* (duplicación de nodos).

Algoritmo 5 (Solución de inconsistencias de tipo *ii* en un documento)**.**

```
Input:  Documento D con una inconsistencia de tipo ii o iii
Output: Documento D consistente

resolverNoConsecutivos(D){
(1)  listaNodos = [D.getNodos()]
(2)  Por cada nodo n en listaNodos
     begin
(3)     listaIntervalos = []
(4)     Por cada eje e incidente a n
        begin
(5)        listaEjes.append(e)
        end
(6)     listaEjes.sort()
```

```
(7)     i=0
(8)     pilaBaches=[]
(9)     Mientras i<len(listaEjes)
        begin
(10)       si T(listaEjes[i]).final + 1 !=  T(listaEjes[i+1]).inicial
           begin
(11)          si T(listaEjes[i]).final + 1 <  T(listaEjes[i+1]).inicial
              begin
(12)              pilaBaches.push(T(listaEjes[i]).final )
              end
(13)          sino
              begin
(14)              Eliminar eje listaEjes[i+1] en el intervalo
                      [T(listaEjes[i+1]).inicial,T(listaEjes[i]).final]
              end
           end
        end
(15)    Mientras !pilaBaches.empty()
        begin
(16)       D = duplicarNodo(D,n,pilaBaches.pop())
        end
    end
(17) Return D
}
```

Teorema 7 (Correctitud del Algoritmo 5)**.**
El Algoritmo 5 termina y corrige los baches presentes en un documento.

Demostración.

- *Termina*: El ciclo en la línea (2) realiza exactamente $|V|$ iteraciones, en cada una de las cuales, a su vez, realiza dos ciclos de $deg_{in}(n)$ iteraciones y un ordenamiento. En el segundo ciclo se lleva a cabo un simple chequeo, que de ser verdadero, hace que se agregue un elemento a la pila definida en la línea (8) o se elimine un eje en un intervalo. En el ciclo de la línea (15) se vacía la pila, llamando al algoritmo de duplicación (cuya correctitud es demostrada en el Teorema 8) por cada elemento de la misma, es decir, a lo sumo $deg_{in}(n)$ veces.
- *Resuelve*: El algoritmo pasa por cada nodo, buscando baches o solapamientos temporales entre las etiquetas temporales de los ejes incidentes a cada uno de ellos. Si se encuentra un solapamiento, se elimina uno de los ejes en el intervalo en que este se produce (línea (14)). En caso de encontrar baches en la línea (11), se agrega el último instante antes de que se produzca el bache a la pila. Esto producirá que se llame en la línea (16) al algoritmo de duplicación para ese instante sobre el nodo en cuestión, generando otro nodo para los instantes posteriores. Se utiliza una pila para duplicar desde el instante mayor hacia atrás en caso de haber más de una inconsistencia sobre el mismo nodo, de manera de no trasladar las inconsistencias restantes al nodo copia, ya que este no será analizado nunca.

□

Análisis de complejidad

La línea (1) tiene orden constante. El ciclo de la línea (2) se ejecuta $|V|$ veces. En cada una de ellas, se llevan a cabo

- Operaciones constantes en las líneas (3), (7) y (8)
- Dos ciclos de $deg_{in}(n)$ iteraciones con operaciones de orden constante, en las líneas (4)-(5) y (9)-(14)
- Un ordenamiento, con un orden de $O(deg_{in}(n)log(deg_{in}(n)))$
- Otro ciclo de $deg_{in}(n)$ iteraciones con una llamada al algoritmo de duplicación, cuyo orden es

$$O(duplicarNodo) = O(deg_{in}(n) + deg_{out}(n))$$

Por lo tanto tenemos

$$\sum_n((2deg_{in}(n) + (deg_{in}(n)log(deg_{in}(n))) + (deg_{in}(n) * (deg_{in}(n) + deg_{out}(n))$$
$$= 2|E| + |E|log(\frac{|E|}{|V|}) + 2\frac{|E|^2}{|V|}$$
$$\approx O(|E|^2)$$

Si tenemos en cuenta que hay una única inconsistencia en el documento, entonces el ciclo en (15) se realiza una única vez, y el orden por lo tanto se transforma en $2|E| + |E|log(\frac{|E|}{|V|}) + 2\frac{|E|}{|V|} \approx O(|E|log(\frac{|E|}{|V|}))$

Algoritmo 6 (Duplicación de nodos).

```
Input: Documento D
       Nodo n que se desea duplicar
       Instante t al que se desea duplicar el nodo
Output: Documento D con el nodo duplicado

duplicarNodo(D,n,t){

(1)  nc = copia de n (atributos)
(2)  Para cada eje e=(n,nf,T) saliente de n
     begin
(3)    Si t<Ti
       begin
(4)        agregar el eje (nc,nf,T) a D
(5)        eliminar el eje e de D
       end
```

```
(6)    Sino
       begin
(7)        si t pertenece a T
           begin
(8)           agregar el eje (nc,nf,[t+1,tf]) a D
(9)           eliminar e en el intervalo [t+1,tf]
           end
       end
    end
(10) Para cada eje e=(ni,n,T)  incidente a n
    begin
(11)   Si t<Ti
       begin
(12)      borrar e
(13)      agregar el eje (ni,nc,T)
       end
(14)   sino
       begin
(15)      Si t pertenece a T
          begin
(16)         borrar e en [t+1,tf]
(17)         agregar el eje (ni,nc,[t+1,tf])
          end
       end
    end
(18) retornar D
}
```

Teorema 8 (Correctitud del Algoritmo 6).
El Algoritmo 6 termina y duplica correctamente el nodo indicado.

Demostración.

- *El algoritmo termina*: Los ciclos realizan una cantidad finita de iteraciones: El ciclo en (2) se realiza $deg_{out}(n)$ veces y el ciclo en (10) se realiza $deg_{in}(n)$ veces. Todas las operaciones son simples asignaciones y chequeos de pertenencia.

- *Resuelve*: La línea (1) realiza el paso i de la definición de duplicación (Definición 10). El ciclo en (2) realiza los pasos *ii* y *iii* de esta definición, eliminando todos los ejes salientes del nodo original más allá del instante t y agregando estos ejes al nodo copia, en el intervalo eliminado. El ciclo en (10) lleva a cabo el paso *iv* de la duplicación.

□

Análisis de complejidad

Hay dos ciclos en las líneas (2) y (10). Las operaciones realizadas en ellos

tienen orden constante, y se realizan $deg_{out}(n)$ y $deg_{in}(n)$ veces respectivamente. En el peor caso, el orden es entonces de

$$\begin{aligned} & O(deg_{out}(n) + deg_{in}(n)) \\ \approx & O(|E|) \end{aligned}$$

Mientras que en el caso promedio es $O(\frac{|E|}{|V|})$

5.3. Corrección de inconsistencias de tipo i

Como vimos anteriormente, este tipo de inconsistencias aparece cuando un eje tiene una etiqueta temporal que no está totalmente incluída en el lifespan del nodo origen. Para simplificar la explicación, daremos a continuación algunas definiciones.

Denominaremos ***nodo inconsistente*** al nodo del que parte el eje cuya etiqueta temporal no se encuentra incluída en el lifespan del nodo origen y ***eje inconsistente*** al eje en cuestión.

Para resolver este tipo de inconsistencias, denominadas inconsistencias de tipo i tenemos dos alternativas de interés:

1. Expandir el lifespan del nodo inconsistente de manera que abarque la totalidad del intervalo de la inconsistencia
2. Reducir el intervalo del eje inconsistente quitando del mismo el intervalo de la inconsistencia

A continuación analizaremos en profundidad cada una de las posibilidades, dando los algoritmos necesarios para llevarlas a cabo así como sus ventajas y desventajas.

5.3.1. Resolución por expansión

Definición 13 (Ultimo y primer padre).

Sea n_j un nodo padre de n_i. Sea el eje $\epsilon_1(n_j, n_i, T_{e_1})$ denominaremos a n_j ***último padre*** *de n_i si $(T_j)_f \geq (T_{e_k})_f \ \forall$ eje e_k incidente a n_i; **último padre en el intervalo temporal T** si el intervalo temporal del eje $\epsilon_j(n_j, n_i, T_{e_1})$ es el mayor a todos los intervalos temporales de los ejes incidentes a n_i con $(T_j)_i \leq (T)_f$. Y por último, **último padre en T absoluto** si es el último padre en T y además es el último padre. Análogamente, denominaremos a n_j el* ***primer padre*** *de n_i si $(T_j)_f \leq (T_{e_k})_f \ \forall$ eje e_k incidente a n_i.*

Definición 14 (Ultimo eje incidente).

Sea n_j un nodo padre de n_i. Sea el eje $\epsilon_1(n_j, n_i, T_{e_1})$ denominaremos a ϵ_1 ***último eje incidente a*** *n_i si n_j es el último padre de n_i.*

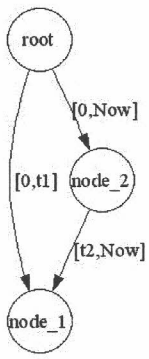


Figura 5.8: Ejemplo de último padre. node_2 es el último padre de node_1. *root* es el último padre de node_1 en el intervalo $[0, t_1]$. node_2 es el último padre absoluto en $[t_2, Now]$ de node_1.

Ejemplo 12.

Ejemplo de estas definiciones pueden verse en la Figura 5.8. En ella node_2 es el último padre de node_1 ya que $(T_{\epsilon_1(n_2,n_1)})_i$ *es el mayor extremo inicial de intervalo entre todos los intervalos de los ejes incidentes a node_1 tal como indica la definición anterior.*
El nodo root es el último padre de node_1 en el intervalo $[0, t_1]$ *ya que no hay un eje incidente a node_1 cuyo intervalo temporal sea posterior al del eje incidente desde root y cuyo extremo inferior sea anterior a el extremo mayor del intervalo* $[0, t_1]$.
Por último, node_2 es el último padre en $[t_2, t_5]$ *absoluto, dado que además de ser el último padre en* $[t_2, t_5]$*, puesto que no existe un eje incidente a node_1 con intervalo posterior al del proveniente de node_2 y y cuyo extremo inferior sea anterior a el extremo superior del intervalo mencionado, éste es el último padre de node_1.*

Una opción para resolver este tipo de inconsistencia es agrandar el lifespan del nodo inconsistente. Para esto debemos tomar el eje en el extremo correcto del lifespan (último eje incidente o primer eje incidente, según corresponda) y modificar su intervalo de manera que abarque la totalidad del intervalo del eje inconsistente. Si es T el intervalo de la inconsistencia y n el nodo inconsistente, y $T \succ lifespan(n)$ entonces tomamos el último eje incidente. Si por el contrario $T \prec lifespan(n)$ entonces utilizamos el primer eje incidente al nodo inconsistente. Veamos un ejemplo:

Ejemplo 13 (Corrección por expansión)**.**

En la Figura 5.9 vemos que el nodo node_1 tiene una inconsistencia de tipo i ya que el eje saliente hacia el nodo node_3, tiene intervalo $[t_6, Now]$ *y el lifespan de node_1 es* $[0, t_{10}]$*. El nodo node_1 es entonces el nodo inconsistente, el eje (node_1,node_3) es el eje inconsistente y el intervalo de la inconsistencia es*

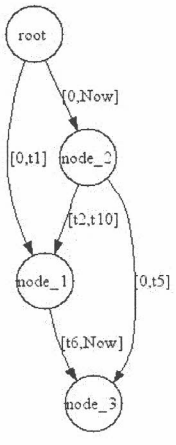


Figura 5.9: Ejemplo de inconsistencia tipo i

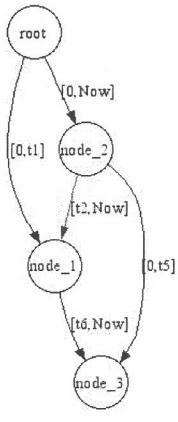


Figura 5.10: Solución por expansión para el ejemplo de la Figura 5.9

$[t_{11}, Now]$. *Para resolver por expansión esta inconsistencia, deberíamos tomar el último eje incidente a node_1, (node_2,node_1), y modificar su etiqueta temporal a* $[t_2, Now]$ *como se muestra en la Figura 5.10.*

Ejemplo 14 (Corrección por expansión)**.**

En el documento de la Figura 5.11 el nodo inconsistente es también node_1, que tiene un eje saliente a node_3 en $[0, t_1]$. *Se debe tomar en este caso el primer eje incidente a node_1, (root, node_1) con intervalo* $[t_2, Now]$. *Para resolver la inconsistencia mediante expansión, se debe entonces expandir la etiqueta temporal de este eje a* $[0, Now]$, *de manera que abarque el intervalo antes inconsistente, tal como se muestra en la Figura 5.12.*

En los ejemplos anteriores sólo fue necesario expandir el lifespan del nodo inconsistente, sin embargo, podría suceder que al expandir la etiqueta temporal del eje indicado, éste quede a su vez en estado inconsistente. Esto pasaría si el intervalo modificado no quedara totalmente incluido dentro del lifespan del nodo origen del eje modificado. Deberíamos entonces resolver esta nueva inconsistencia sin otra posibilidad para ellos que expandir el lifespan del nuevo nodo inconsistente, dado que de otra manera, desharíamos lo ya corregido.

En el caso general tendríamos que expandir todos los ejes de un camino (formado en sentido inverso), buscando siempre el último o primer eje incidente al nuevo nodo inconsistente, dependiendo del extremo en que se de la inconsistencia, siendo el nuevo nodo inconsistente el último o primer padre del nodo que se acaba de corregir. El camino terminaría al llegar a un nodo tal que al expandir su lifespan no se produzca otra inconsistencia de tipo i, es decir, que la etiqueta temporal del eje modificado quede incluída dentro del lifespan de su nodo origen.

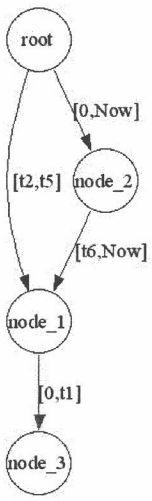


Figura 5.11: Ejemplo de inconsistencia tipo i

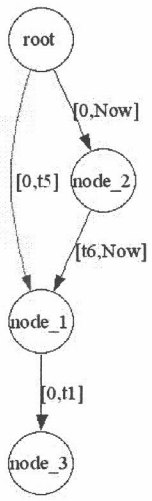


Figura 5.12: Solución por expansión para el ejemplo de la Figura 5.11

A continuación formalizaremos estos conceptos, llamaremos a este camino *camino de últimos padres* o *camino de primeros padres* dependiente del caso, o genéricamente, *camino de expansión*. También mostraremos que estos caminos de expansión estarán siempre formados por primeros padres, o por últimos padres, nunca por una combinación de ambos. Luego daremos los algoritmos necesarios para calcular el camino de expansión y finalmente los algoritmos que resuelven la inconsistencia.

Definición 15 (Camino de expansión)**.**
Un ***camino de últimos (primeros) padres*** *entre dos nodos n_i, n_j es un camino no necesariamente temporal, en el que cada nodo es el último (primer) padre del siguiente del camino. En general, nos referiremos a los caminos de últimos o de primeros padres genéricamente como* ***caminos de expansión****. Estos caminos se recorren en sentido inverso para expandir las etiquetas temporales de los ejes que lo forman.*

Ejemplo 15.
En la Figura 5.13, el camino de últimos padres para el nodo node_3 es root, node_2, node_1, mientras que el camino de primeros padres para el mismo nodo, señalado en la Figura 5.14, es root, node_2. Ambos serán denominados caminos de expansión.

Proposición 9. *Un camino de expansión sólo contiene últimos padres o primeros padres.*

Demostración.

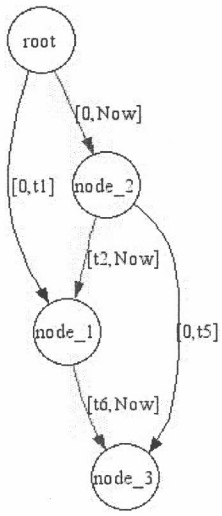


Figura 5.13: Ejemplo de camino de últimos padres

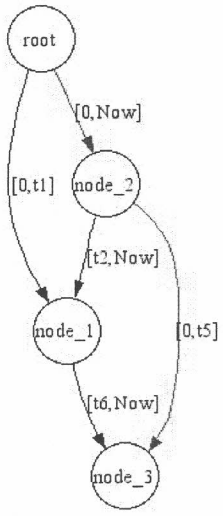


Figura 5.14: Ejemplo de camino de primeros padres

Sea D un documento con una sola inconsistencia. Sea esta inconsistencia de tipo i. Llamemos I al nodo inconsistente, $\epsilon_I(n_I, n_d, T_I)$ el eje inconsistente, con $T \cap T_I \neq \emptyset$ el intervalo en el que se produce la inconsistencia, siendo $T_f = (T_I)_f$. Sea $C = n_1, n_2 ... n_k, n_I$ el camino de expansión para la inconsistencia. Supongamos que existen $\epsilon_i(n_{i+1}, n_i, T_i), \epsilon_{i+1}(n_{i+2}, n_{i+1}, T_{i+1})$ en C tal que ϵ_i es el último eje incidente a n_i y ϵ_{i+1} es el primer eje incidente a n_{i+1} (o sea, el camino alterna últimos y primeros padres).
Si comenzamos a expandir los ejes desde el nodo inconsistente subiendo por el camino, cuando llegamos a n_i nos encontramos con que debemos expandir el último eje incidente, ϵ_i. El intervalo temporal de este eje será expandido desde el instante $(T_i)_f$ hasta (T_f). Como la inconsistencia inicial estaba aislada dentro del documento, sabemos que $lifespan(n_{i+1}) \supseteq T_i$ y por lo tanto al expandir el eje saliente hacia n_i quedará a lo sumo inconsistente en $T - T_i = [(T_i)_f + 1, T_f]$. Este intervalo es claramente posterior a $lifespan(n_{i+1})$ y por lo tanto debemos continuar expandiendo el último eje incidente al nodo. Esto es un absurdo, puesto que el siguiente nodo al recorrer el camino de expansión en sentido inverso, es n_{i+2} el cual supusimos que era el primer padre de n_{i+1}.
En consecuencia, deducimos que el camino no alterna primeros y últimos padres. □

Para expandir los ejes de un camino, debemos prestar particular atención a que no se formen ciclos. Para esto definimos el concepto de *Instante de máxima expandibilidad de caminos* (*IMEC*). Este valor nos indicará hasta que instante se pueden expandir las etiquetas temporales de los ejes de un camino, sin provocar ciclos. De esta manera, si el instante hasta el cuál debemos expandir para salvar

la inconsistencia es menor al *IMEC* (o mayor si se trata de un camino de primeros padres) , se podrá expandir sin provocar ciclos; caso contrario, la expansión no podrá llevarse a cabo.

Definición 16 (Instante de máxima expansibilidad de caminos (*IMEC*)).
Sea $C = (n_1, n_2..., n_f)$ *un camino de expansión entre dos nodos* n_1, n_f, *donde cada eje es de la forma* $\epsilon_i(n_i, n_j, T_i)$ *con* T_i *la etiqueta temporal del i-ésimo eje en el camino.*
Sea $mcp_(n_f, n_k) = [(C_{k1}, T_{k1})...(C_{kn}, T_{kn})]$ *la lista de caminos continuos maximales desde* n_f *hasta* n_k $\forall k \in 1...f-1$, *con* T_{ki} *el intervalo temporal asociado al i-ésimo camino continuo maximal entre* n_f *y* n_k. *El instante de máxima expansibilidad se define entonces como*

$$IMEC = \begin{cases} \text{máximo instante } t \text{ tal que} \\ t \geq min((T_i)_f)\ \forall i : 1..f-1\ \wedge \\ [min((T_i)_f), t] \cap T_{kj} = \phi\ \forall j \in 1..n, \\ \forall k \in 1..f-1 & \text{si el camino de expansión es de últimos padres.} \\ \\ \text{mínimo instante } t \text{ tal que} \\ t \leq max((T_i)_i) \forall i : 1..f-1\ \wedge \\ [t, max((T_i)_i)] \cap T_{kj} = \phi \quad \forall j \in 1..n, \\ \forall k \in 1..f-1 & \text{si el camino de expansión es de primeros padres.} \end{cases}$$

Ejemplo 16 (IMEC).
Un ejemplo de esta definición puede verse en la Figura 5.15, donde el camino de expansión node_2→node_4→node_5 es de últimos padres. El nodo I es inconsistente y tiene un camino en el intervalo de la inconsistencia hasta el nodo node_2. El instante de máxima expansibilidad es $t = 24$, *puesto que es mayor a los intervalos del camino de expansión node_2→node_4→node_5 y menor al intervalo del camino I →node_2. Si se expandiera hasta* $t = 25$, *se generaría el ciclo* $I \rightarrow node\_6 \rightarrow node\_2 \rightarrow node\_4 \rightarrow node\_5 \rightarrow I$.

Teorema 10. *El IMEC indica el instante hasta el cual se pueden expandir las etiquetas temporales de los ejes de un camino de expansión sin generar ciclos.*

Demostración.

Supongamos que se expanden los intervalos de los ejes del camino hasta *IMEC* y se genera un ciclo. Eso significa que hay un camino para algún instante $t \in [min((T_i)_f), IMEC]$ (o, en el caso de primeros padres $t \in [IMEC, max((T_i)_i)]$) entre

i. n_f y n_i para algún n_i en el camino de últimos (primeros) padres

ii. n_j y n_i para algún n_i, n_j en el camino de últimos (primeros) padres

Supongamos el primer caso. Pero entonces $t \in T_{ij}$ para algún par (C_{ij}, T_{ij}) en

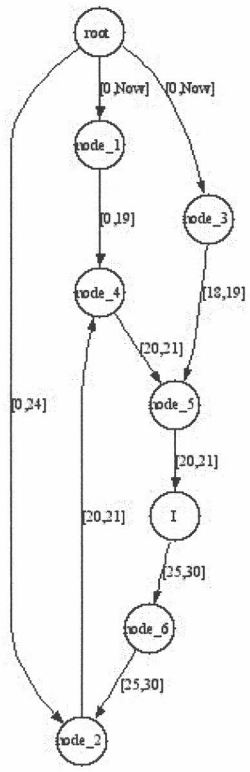


Figura 5.15: Ejemplo máxima expansibilidad

$mcp(n_f, n_i)$ entre n_i y n_f $\Rightarrow_{\text{por definición de } IMEC}$ Absurdo.

Si en cambio se da el segundo caso, entonces antes de la expansión se cumple que $t \in lifespan(n_i) \wedge t \in lifespan(n_j)$ ya que ambos nodos eran consistentes y para que haya un camino entre ellos en un instante, el nodo origen debe contener ese instante en su lifespan, y el nodo destino del camino dirigido tiene un eje incidente en el mismo. Por lo tanto:

- en la porción del camino entre n_i y n_j, el instante t en que se produce el camino $n_j \rightarrow n_i$ ya estaba incluido, y por lo tanto el ciclo ya existía
- $t < min((T_k)_f)$ (o, equivalentemente $t > max((T_k)_i)$ para cada $i < k < j$, con lo cual, de todas maneras la expansión no produce ningún efecto en esta porción del camino $\Rightarrow$ absurdo, puesto que entonces el ciclo ya existía.

□

Teorema 11. *Sea $T = [t_i, t_f]$ el intervalo de la inconsistencia, si $IMEC \geq t_f$ (alternativamente, $IMEC \leq t_i$ si el camino de expansión fuera de primeros*

padres) entonces se puede resolver la inconsistencia expandiendo, sin provocar ciclos.
En otras palabras: el Algoritmo 7 resuelve la inconsistencia bajo la condición anterior.

Demostración.

Sea $n_1 ... n_f$ el camino de expansión tal que n_f es el nodo inconsistente y $lifespan(n_1) \supset T, T$ intervalo de la inconsistencia. Entonces, como $IMEC \geq T_f$ podemos expandir hasta T_f (por definición de *IMEC*) y además expandiendo todos los intervalos del camino hasta el instante T_f claramente la inconsistencia queda corregida. Además, no se generan otras inconsistencias en el proceso:

i. De tipo *i*, puesto que se expanden todos los intervalos en el camino de expansión (que no intercala últimos y primeros padres), hasta llegar a un nodo cuyo lifespan incluye al intervalo de la inconsistencia.

ii. De tipo *ii* y tipo *iii*, porque se expande siempre el intervalo de un eje en el camino de expansión sin dejar baches ni solapamientos.

iv. De tipo *iv*, por teorema 10.

□

A continuación presentamos los algoritmos que implementan estos conceptos. El cálculo explícito de *IMEC* no es necesario ya que queremos saber si podemos expandir y no conocer con exactitud el valor de *IMEC*, sólo se necesita verificar que $IMEC \geq t_f$ ($IMEC \leq t_i$), esto es, corroborar que no haya caminos en $[min((t_i)_f), t_f]$ ($[t_i, max((t_i)_i)]$) entre el nodo inconsistente y los nodos internos del camino, siendo el intervalo de la inconsistencia $T = [t_i, t_f]$. Aprovechamos este hecho dado que es más eficiente que calcular el *IMEC*, ya que si $t_f < IMEC$ ($t_i > IMEC$) entonces el algoritmo terminará antes, ya sea porque encuentre un camino, o porque llegue a t_f. Si $t_f \geq IMEC$ ($t_i \leq IMEC$) entonces es lo mismo realizar la verificación descripta o calcular el *IMEC*.

Por cuestiones de claridad, tanto en el siguiente algoritmo cómo en lo que resta del documento, se tomará en cuenta sólo el caso de que la inconsistencia se dé en un intervalo posterior al lifespan del nodo inconsistente, siendo entonces el camino de expansión un camino de últimos padres.

Algoritmo 7 (Corrección de inconsistencias tipo *i* mediante la expansión de intervalos).

```
input:  un documento D con una única inconsistencia, de tipo i,
           que pueda resolverse mediante la expansión
        nodo I, el nodo inconsistente
        T intervalo temporal de la inconsistencia con Tf <= IMEC
output: D consistente

Document expandirCamino(D,I,T){
(1) nf = I
```

```
(2) Mientras lifespan(nf) no incluya a T:
    begin
(3)    Sea e=(ni,nf,t) el último padre de nf
       begin
(4)        hacer tf = Tf
(5)        nf = ni
       end
    end
(6) return D
}
```

Este algoritmo es específico para el caso en que el camino de expansión sea de últimos padres. Si fuera de primeros padres, se debe buscar el primer padre en la línea (3) y hacer hacer $ti = Ti$ en la línea (4). La estrategia a utilizar puede decidirse antes de entrar al ciclo, sabiendo en que extremo se encuentra la inconsistencia del nodo I, ya que los caminos de expansión no alternan primeros y últimos padres.

Teorema 12 (Correctitud del Algoritmo 7)**.**
El Algoritmo 7 termina y realiza la expansión del camino correspondiente de manera correcta.

Demostración.

- *Termina*: En cada paso alcanza un nodo del camino de últimos padres, hasta llegar a un nodo cuyo lifespan incluye al intervalo de la inconsistencia. En el peor caso, llegará hasta la raíz, dado que al considerar la inconsistencia aislada sabemos que el documento no tiene ciclos.
- *Resuelve*: El algoritmo comienza por el nodo inconsistente, y recorre el camino de últimos padres en sentido inverso, expandiendo en cada paso el intervalo temporal de uno de los ejes del camino, hasta el instante final de la inconsistencia. Si $IMEC \geq (T)_f$ entonces el algoritmo resuelve la inconsistencia según el teorema 11

□

Análisis de complejidad

Como en el cálculo de un camino de últimos padres, no es posible pasar dos veces por el mismo nodo si el documento no contiene ciclos, el algoritmo recorre a lo sumo todos los nodos hasta llegar a la raíz. Esto suma una una complejidad $O(|E|)$, puesto que en cada nodo debe calcular el último padre recorriendo todos los ejes incidentes.

Algoritmo 8 (Búsqueda de caminos que impiden expandir)**.**

```
input:  D documento con una inconsistencia de tipo i
        (única inconsistencia del documento),
         T intervalo donde se da la inconsistencia
         I nodo inconsistente
```

```
ouput: TRUE si hay un camino
       FALSE sino

boolean generaCiclosExpandir(D,I,T){
(1) nf = I; camino = [];inicial=Ti
(2) Mientras lifespan(nf) no incluya a T
    begin
(3)   e = lastEdgeIn(nf) //en T para t<Tf
(4)   si inicial > (Te)i  inicial = (Te)i
(5)   camino.append(e.nodeFrom())
(6)   nf=ni
    end
(7) Para cada nodo ni en camino
    begin
(8)     si hayCamino(ni, D, [inicial,Tf])
        begin
(9)        return TRUE
        end
    end
(10)return FALSE
}
```

Teorema 13 (Correctitud del Algoritmo 8)**.**
El Algoritmo 8 termina y responde correctamente si se generan ciclos o no al expandir.

Demostración.

- *Termina*: El algoritmo va recorriendo el camino de últimos padres en sentido inverso, es decir, desde el nodo inconsistente en dirección hacia la raíz, hasta hallar el nodo cuyo lifespan incluye a T. Esto siempre ocurre dado que al no haber ciclos, en última instancia se alcanza la raíz, cuyo lifespan es, por definición, $[0, Now]$. Luego recorre esos mismos nodos, buscando caminos entre el nodo inconsistente y los nodos del camino. Como el ciclo es finito y hayCamino (Algoritmo 9) termina, el algoritmo termina.
- *Resuelve*: el algoritmo retorna verdadero si hay un camino continuo entre el nodo inconsistente y algún nodo en el camino de expansión en el intervalo de la inconsistencia T. Si existe tal camino, entonces $IMEC < T_f$ y la inconsistencia no puede resolverse expandiendo. No es necesario verificar la existencia de caminos entre dos nodos internos en el camino de expansión, dado que de existir, la expansión no afectaría el intervalo en el cual éste es válido (ver caso *ii* de la demostración del teorema 10)

□

Análisis de complejidad

El ciclo en la línea (2) se realiza a lo sumo una vez por cada nodo, la línea (3) tiene complejidad $O(deg_{in}(n))$ con lo cual el ciclo tiene un orden total de $O(|E|)$.
El ciclo en la línea (7) se realiza la misma cantidad de veces que el anterior, un

máximo de una vez por nodo del documento. El Algoritmo hayCamino (Algoritmo 9) tiene un orden de $O(|E|)$ en el peor caso (bajo la suposición de que no hay ciclos no temporales), con lo cual tenemos un orden total del ciclo de $O(|E| * |V|)$. El orden total del algoritmo es entonces $O(|V| * |E| + |E|)$.
Una mejora al orden del algoritmo podría obtenerse desarrollando otra versión de hayCamino que reciba una estructura de hash con los nodos a chequear, y que en cada paso chequee si efectivamente el nodo por el que se pasa está en la estructura y con que intervalo. Esto resultaría en un orden en el peor caso de $O(|E|)$ solamente.

Para verificar la no existencia de caminos que generen ciclos, utilizaremos el hecho de que la inconsistencia es la única dentro del documento. Esto nos permitirá valernos de la siguiente propiedad, para verificar la existencia de ciclos en tiempo polinomial.

Proposición 14. *Dado un nodo n_j consistente en el documento y otro nodo n inconsistente. Si existe una única inconsistencia en el documento en el intervalo T y ésta es de tipo i o tipo iv, entonces hay un camino desde n hasta n_j en $T' \subseteq T$ sí y sólo sí el nodo $mcp_{desdelaraiz}(n_j) \cap T' = \emptyset$*

Demostración.

Supongamos que hay un camino a n_j desde el nodo inconsistente en T' y que también es alcanzable desde la ráiz. Como no hay inconsistencias de tipo *ii* y tipo *iii*, no puede haber más de un camino a n_j en ningún instante $t \in T'$ (puesto que sólo se puede llegar al nodo a través de un único eje). Dado que hay un único camino, el camino que lo alcanza desde la raíz debe pasar por el nodo inconsistente en el intervalo T'. Esto significa que $mcp_{desdelaraiz}(n) \cap T \supseteq T'$ lo cual es absurdo dado que:

1. Si el nodo tiene una inconsistencia de tipo i en T, no tiene ejes incidentes en ese intervalo y por ende no puede ser nunca alcanzable desde la raíz en el mismo
2. Si el nodo tiene una inconsistencia de tipo *iv* en T, ningún nodo perteneciente al ciclo es alcanzable en el intervalo del mismo en virtud de la proposición 2

□

Algoritmo 9 (Para verificar la existencia de caminos).

```
input:nd nodo destino,
      D documento,
      T intervalo en el que se produce una inconsistencia de tipo i o iv
        ( única inconsistencia del documento )
output:false si existen un camino desde la raíz a nf en todo el intervalo T
            o nf no tiene un eje incidente en T,
       true sino

boolean hayCamino(nd,D,T){
```

```
(0)   cps = []
(1)   Cola nodos     = [g.getRoot()]
(2)   Cola nodosWait = []
(3)   Mientras !nodos.empty() hacer
        begin
(4)         n = nodos.first()
(5)         Si !n.isFinalizado()
              begin
(6)              listaEtiquetas = [Te intersección T, tal que e es un eje entrante a n
                                   y !e.isTransitado() y Te intersección T es no vacía]
(7)              Para cada eje e saliente de n
                    begin
(8)                    Si !e.isTransitado()
                           begin
(9)                           si Te intersección Te' != vacio para algún Te'
                                          en listaEtiquetas
(10)                          e.setTransitable(false)
                           end
                       sino
                           begin
                             si Te interseccion T != vacío
                             begin
(11)                          e.setTransitable(true)
                             end
                           end
                    end
              end

(12)        Para cada eje e=(n,nf,Te) saliente de n no transitado hacer:
              begin
(13)            Si ( e.isTransitable() || n.isFinalizado() )
                  begin
(14)                 e.setTransitado(true)
(15)                 nf.setEjesTransitados(nf.getEjesTransitados()++)
(16)                 Si todos los ejes incidentes a nf en T fueron transitados (
                             nf.getEjesTransitados() == nf.getDegIn( T ) )
                       begin
(17)                     nodos.append(nf)
                       end
(18)                 Sino
                       begin
(19)                     Si !nf.isVisitado()
                           begin
(20)                         nodosWait.append(nf)
(21)                         nf.setVisitado(true)
                           end
                       end
(22)                 Si nf == nd
                     begin
(23)                   cps.add(Te.interseccion(T))
                     end
                end
              end
(24)        Si nodos.empty() and !nodosWait.empty()
              begin
(25)            n = nodosWait.first()
(26)            nodos.append(n)
(27)            n.isVisitado(false)
              end
        end
(28)  cps.sort()
(29)  unir los intervalos en cps
(30)  Para cada intervalo I en cps
        begin
(31)      si I.interseccion(T) == T
(32)        return false
        end
(33)  return true
```

}

Teorema 15 (Correctitud del Algoritmo 9).
El Algoritmo 9 termina e indica si existe un camino entre los nodos indicados.

Demostración.

El Algoritmo es casi idéntico al de chequeo de consistencia para inconsistencias de tipo *iv*. Se recorre el documento en una forma similar a BFS minimizando las veces que se pasa por un nodo dado. Las diferencias radican en que sólo nos importa el intervalo de la inconsistencia y en que el ciclo final se realiza sobre los intervalos de cps.

- *Termina*: Como dijimos es similar al algoritmo dado en el capítulo anterior. Cada eje se recorre una única vez y en algún momento no se agregan más nodos a las listas, con lo cual se irán consumiendo los mismos en sucesivas iteraciones hasta finalizar.
- *Resuelve*: Es decir, retorna *true* si hay un camino que impida la expansión y *false* en caso contrario.

  El algoritmo recorre el documento desde la raíz y cada vez que llega al nodo destino guarda el intervalo en el cual llega en una lista (el llamado a add lo arregla uniéndolo a otro preexistente en la lista si es posible), luego agrega el nodo en la cola para seguir adelante (o a la cola de espera según corresponda), logrando un recorrido del documento en BFS. En el ciclo en (30) revisa los intervalos obtenidos, y si el intervalo T se encuentra totalmente incluido en alguno de ellos, significa que fue alcanzado en su totalidad desde la raíz. Por lo tanto, en virtud de la Proposición 14 no hay camino desde un nodo inconsistente y el algoritmo retorna false. Si ninguno de los intervalos cumple esta condición, entonces el nodo es alcanzable desde el nodo inconsistente en el intervalo, en virtud de la Proposición 14.

□

Análisis de complejidad

Podemos distinguir dos casos en este problema, dependiendo si hay o no ciclos no temporales en el documento. En el caso de que no haya ciclos no temporales, las líneas (5) a (11) nunca se ejecutan. Se recorre entonces a lo sumo una vez cada nodo. Cada iteración de (12) realiza operaciones constantes salvo en la línea (16), donde debe calcular la cantidad de ejes incidentes al nodo en

un intervalo dado. Esto puede realizarse una única vez por cada nodo, con un orden total de $|E|$. Luego, se realiza un ordenamiento en la línea (28) con orden $deg_{in}(n_d)log(deg_{in}(n_d))$ y en la línea (29) la unión tiene un orden de $deg_{in}(n_d)$ lo mismo que el ciclo en la línea (30). Con lo cual, en el mejor caso el algoritmo es $O(|E| + \frac{|E|}{|V|} * log(\frac{|E|}{|V|}))$ considerando que $deg_{in}(n_d) \approx \frac{|E|}{|V|}$ en promedio.
En el caso en que haya ciclos no temporales, las primeras líneas deberán ejecutarse. En este caso de todas maneras transitaremos a lo sumo una vez cada eje dado que un eje se marca como transitable en la línea (10) sólo en caso que sea transitable en todo el intervalo de su etiqueta temporal. Teniendo esto en cuenta, sabemos que cada eje incidente a un nodo se puede transitar una única vez y por lo tanto sólo se ejecutaran las primeras líneas $deg_{in}(n)$ veces para cada nodo n. Esto significa que cada nodo puede aportar a lo sumo $deg_{in}(n)(deg_{in}(n) + deg_{out}(n)deg_{in}(n))$ que es la cantidad de veces que llegamos al nodo multiplicado el orden de la línea (6) más el orden del ciclo en la línea (9). Sumando para todos los nodos, y considerando que $deg_{in}(n) \leq |E|$ *y* $deg_{out}(n) \leq |E|$ tenemos una cota de $|E|^3 + |E|^2$ para las primeras líneas. La cota es poco ajustada, pero suficiente para ver que el algoritmo es polinomial aún en el peor caso. El orden total en este caso es entonces de $|E|^3 + |E|^2$, puesto que el resto del algoritmo no tiene un aporte significativo.

5.3.2. Resolución por reducción

Hasta aquí vimos cómo resolver inconsistencias de tipo *i* expandiendo los intervalos temporales de los ejes necesarios. Ahora veremos otra solución, en la que tomamos el intervalo temporal del eje inconsistente y lo modificamos de manera que quede incluido en el lifespan del nodo origen. Podría ser necesario eliminar este eje, si su intervalo temporal tuviese intersección vacía con el lifespan del nodo inconsistente. Al reducir el intervalo (o eliminar el eje), se pueden producir nuevas inconsistencias de tipo *i* en el nodo destino del eje modificado si éste tiene ejes salientes en el intervalo eliminado, y de tipo *ii* (bache) si el intervalo eliminado no se encontraba en un extremo del lifespan.

Ejemplo 17.

En la Figura 5.16 podemos como se corrige una inconsistencia de tipo i reduciendo el intervalo del eje (node_1, node_2) para que quede totalmente incluido en el lifespan del nodo node_1.

En la Figura 5.18 podemos como se corrige una inconsistencia de tipo i eliminando el eje (node_1, node_3) puesto que el intervalo de su etiqueta temporal está totalmente fuera del lifespan del nodo origen, node_1.

En la Figura 5.20 podemos como se corrige una inconsistencia de tipo i eliminando el eje (node_2, node_4) puesto que el intervalo de su etiqueta temporal está totalmente fuera del lifespan del nodo origen, node_2. Este proceso deja a su vez un bache en el lifespan del nodo node_4. Esta nueva inconsistencia deberá también ser corregida por los algoritmos proporcionados.

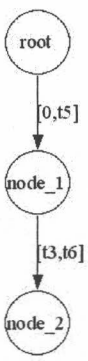


Figura 5.16: Ejemplo de inconsistencia de tipo i

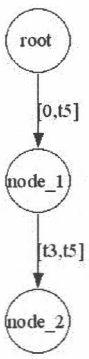


Figura 5.17: Ejemplo de corrección de inconsistencia de tipo i, reduciendo un intervalo

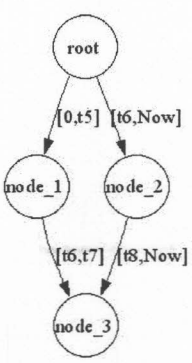


Figura 5.18: Ejemplo de inconsistencia de tipo i

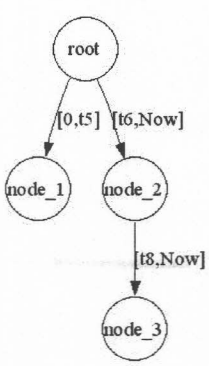


Figura 5.19: Ejemplo de corrección de inconsistencia de tipo i, eliminando un eje

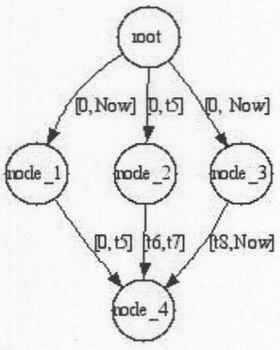


Figura 5.20: Ejemplo de inconsistencia de tipo i

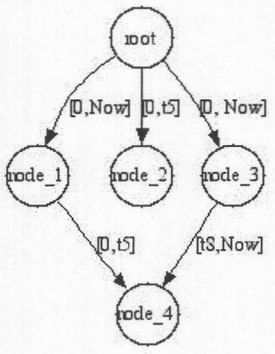


Figura 5.21: Ejemplo de corrección de inconsistencia de tipo i, eliminando un eje

A continuación presentamos el algoritmo que resuelve la inconsistencia aplicando estos conceptos, analizamos su correctitud y complejidad.

Algoritmo 10. Solución de inconsistencias mediante la eliminación de ejes

```
Input: Documento D con una inconsistencia de tipo i
       nodo np en el cual se da la inconsistencia
       intervalo T en el que se produce la inconsistencia

reducirCaminos(D, np, T){
(1)   cps = {(np,[T])}
(2)   listaNodos = [np]
(3)   Para cada nodo np en listaNodos
      begin
(4)       marcar np como no visitado
(5)       Para cada eje ei=(n, nf, Ti) saliente de n
          begin
(6)           Para cada intervalo temporal T' en cps.get(n)
              begin
(7)             si T' interseccion Ti != vacío
                begin
(8)               Tactual = T' interseccion Ti
(9)               cpsNf = cps.get(nf)
(10)              si cpsNf == null
                  begin
(11)                cpsNf = []
                  end
(12)              cpsNf.addInterval(Tactual)
(13)              si (Ti)i < (Tactual)i
                  begin
(14)                si (Ti)f > (Tactual)f
                    begin
(15)                  crearEje(n,nf, [(Tactual)f + 1, (Ti)f])
(16)                  Ti = [(Ti)i, (Tactual)i - 1]
                    end
                    else
                    begin
(17)                  Ti = [(Tactual)f  + 1, (Ti)f ]
                    end
                  end
(18)              sino
                  begin
(19)                si (Ti)f > (Tactual)f
```

```
                    begin
(20)                  Ti = [(Tactual)f + 1, (Ti)f ]
                    end
(21)                sino
                    begin
(22)                  eliminar ei
                    end
                  end
(23)              si nf no fue visitado
                  begin
(24)                nodos.append(nf)
(25)                marcar nf como visitado
                  end
                end
              end
            end
      end
(26)  resolverBaches(D)
}
```

Teorema 16 (Correctitud del Algoritmo 10).
El Algoritmo 10 termina y resuelve la inconsistencia de tipo i encontrada.

Demostración.

- *Termina*: En cada paso se recorre un eje distinto agregando su nodo destino a la lista de nodos. Si bien puede recorrer más de una vez el mismo eje, no lo recorrerá más de una vez en el mismo intervalo, por lo tanto la cantidad de iteraciones en las que se agregan nodos son finitas, luego sólo se avanza en la lista sin poder transitar los ejes, dado que sus intervalos ya fueron modificados. Finalmente entonces se llega al final de la lista y el algoritmo termina

- *Resuelve*: El algoritmo deja al documento sin inconsistencias

  a. tipo *i*: Inicialmente elimina el eje inconsistente en el intervalo en el que se produce la inconsistencia, esto puede producir otras inconsistencias, puesto que el nodo destino puede tener ejes salientes en ese intervalo. El algoritmo entonces en el ciclo de la línea (3), más particularmente en la línea (3), agrega el nodo destino a la lista de nodos a evaluar, con lo que en las siguiente iteración, este es revisado y sus ejes salientes eliminados en T. Siguiendo el procedimiento, resulta entonces que todo eje en T, o bien no es alcanzable desde el nodo inconsistente en dicho intervalo, o bien es eliminado en el mismo.

  b. tipo *ii* y tipo *iii*: Si bien al eliminar un eje en un intervalo, pueden producirse baches en el lifespan del nodo al que éste incide, estas son resueltas luego en última línea, con la llamada al algoritmo 5 en la última línea.
  Este procedimiento no se realiza junto con la eliminación puesto que podría resultar ineficiente, dado que sucesivas eliminaciones sobre ejes incidentes a un mismo nodo provocarían más de un bache (si no

se encontraran en un extremo del lifespan) resultando en varias duplicaciones de nodos, aunque los intervalos de los ejes eliminados fuesen consecutivos y pudiera resolverse con una única duplicación al final. Un ejemplo de este caso puede observarse en la Figura 5.22, donde si eliminamos primero el eje $\epsilon(n_6, n_5)$ y duplicamos, y luego eliminamos el eje $\epsilon(n_2, n_5)$, también debemos duplicar, y luego eliminaríamos el eje $\epsilon(n_3, n_5)$, quedando eliminado uno de los nodos resultantes de la segunda duplicación, siendo el resultado el mismo que el de eliminar todos los ejes, y luego duplicar una única vez.

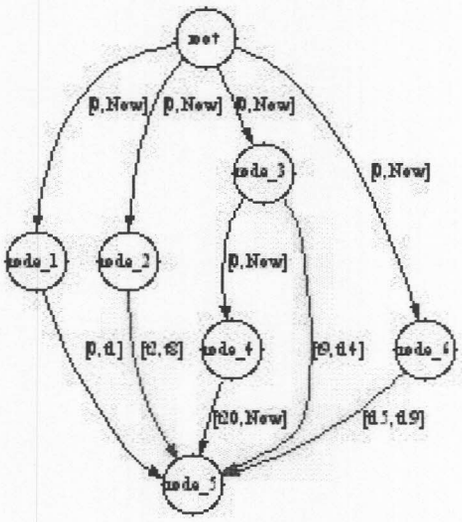


Figura 5.22: Duplicación al final vs. duplicación con cada eliminación

c. tipo *iv*: No pueden producirse ciclos, dado que se están eliminando ejes, no agregando.

□

Análisis de complejidad

El algoritmo no es polinomial, puede recorrerse más de una vez cada eje y cada nodo, dependiendo del documento. Se puede mejorar el orden de manera que sea polinomial en el caso de documentos sin ciclos temporales agregando un mecanismo con lista de espera como en el Algoritmo 4 y calculando como primer paso el $mcp_{desdelaraiz}(n)$ para cada nodo, de manera de poder distinguir si un nodo puede finalizarse o no, dado que un eje dado puede no ser alcanzable desde el nodo inconsistente y por ende no debería tomarse en cuenta. Así, en

el caso en que no haya ciclos no temporales en el intervalo de la inconsistencia, el algoritmo resultaría polinomial ya que lograría recorrer una única vez cada nodo. Sin embargo en otro caso no lo es.
Analizaremos el mejor caso, aquel en el cual el documento es un árbol, resultando en un orden polinomial para el algoritmo presentado. En este caso, la primer línea tiene orden constante. El ciclo principal se ejecuta a lo sumo $|V|$ veces. Por cada nodo visitado, se realiza un ciclo de $deg_{out}(n)$ iteraciones donde todas las operaciones tienen orden constante (incluyendo las operaciones de las líneas (6) y (12) puesto que cada nodo tiene un sólo eje incidente). Por lo tanto tenemos en total:

$$\sum_{n \in V} deg_{out}(n) \approx O(|E|)$$

Luego, en la última línea, se ejecuta el Algoritmo 5 para resolver las posibles inconsistencias de tipo *ii* y tipo *iii* provocadas al eliminar ejes. Este algoritmo como vimos tiene orden $O(|E|^2)$ por lo tanto sumamos

$$O(|E| + |E|^2) \approx O(|E|^2) \text{ en el mejor caso}$$

Este cálculo puede hacerse extensivo también al caso en que el documento es un árbol sólo en el intervalo de la inconsistencia.

5.4. Corrección de inconsistencias de tipo *iv* (ciclos)

Al igual que sucede con las inconsistencias de tipo *i*, existen diversas posibilidades para corregir las inconsistencias de tipo *iv*:

1. Eliminar los ejes del ciclo en el intervalo temporal en que se produce el mismo. Esta opción tiene a su vez dos posibilidades:
   a) Eliminar todos los subgrafos en el intervalo de la inconsistencia con raíz en cada uno de los nodos del ciclo
   b) Expandir los intervalos de las etiquetas temporales de los ejes en el camino de expansión de cada nodo perteneciente al ciclo de manera de abarcar el intervalo en que se produce el mismo.
2. Eliminar uno de los ejes del ciclo. Al eliminar el eje del ciclo se forma una inconsistencia de tipo *i* y debemos entonces expandir las etiquetas temporales de los ejes en el camino de expansión del nodo al que incidía el eje eliminado para obtener un documento consistente. Debemos tener especial cuidado de elegir para la eliminación un eje cuyo nodo destino tenga un eje incidente fuera del ciclo que pueda expandirse para resolver la nueva inconsistencia generada.

En la Figura 5.23 podemos apreciar un ciclo entre los nodos *node_4* → *node_2* → *node_3* → *node_4* en el intervalo $[t_2, t_5]$. A continuación, veremos como se resolvería utilizando las diferentes opciones que acabamos de mencionar.

Ejemplo 18. *En la Figura 5.24 podemos ver el resultado de eliminar todos los subgrafos a partir de algún nodo del ciclo para el intervalo de la inconsistencia, del documento de la Figura 5.23, los nodos node_5 y node_3 fueron eliminados puesto que no tienen ejes incidentes fuera del intervalo del ciclo. (solución 1a)*

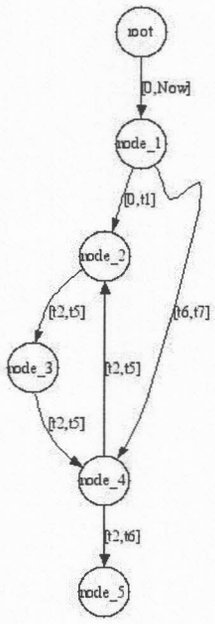


Figura 5.23: Ejemplo de ciclo en un documento

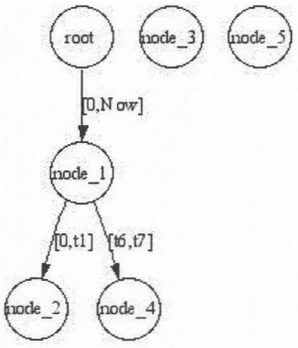


Figura 5.24: Corrección de ciclos mediante la eliminación

Ejemplo 19. *En la Figura 5.25 podemos ver el resultado de expandir los caminos de expansión para cada nodo del ciclo y eliminar todos los ejes que forman este ciclo, para el documento de la Figura 5.23. El nodo node_3 fue eliminado puesto que no tiene ejes incidentes fuera del intervalo del ciclo que pudiesen ser expandidos. (solución 1b)*

Ejemplo 20. *En la Figura 5.26 podemos ver el resultado de eliminar un eje del ciclo (en este caso el eje del nodo node_4 al nodo node_2) en el intervalo de la inconsistencia, para el documento la Figura 5.23. En este caso, ningún nodo fue eliminado del documento y sólo fue necesario eliminar el eje node_4 a node_2 y expandir el eje node_1 a node_2. (solución 2)*

La opción 1a) puede acarrear una pérdida importante de la información, y la 1b), agrega demasiada información (no necesariamente correcta) ya que ex-

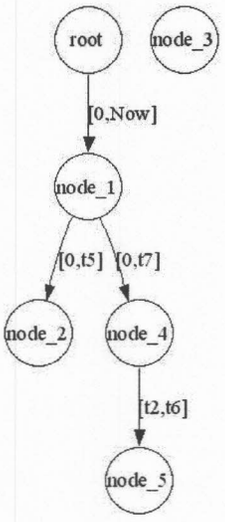


Figura 5.25: Corrección de ciclos mediante la expansión para todos los nodos del ciclo

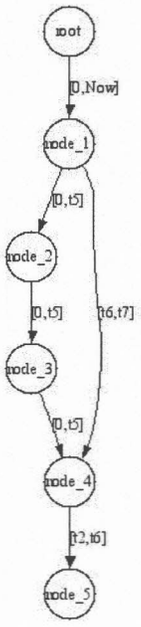


Figura 5.26: Corrección de ciclos mediante la eliminación de un eje del mismo y la modificación del camino de expansión para el nodo destino del eje eliminado.

pande para todos los nodos del ciclo. Si suponemos que el ciclo se produce por un error de tipeo o alguna equivocación semejante, no es adecuado suponer que todos los ejes fueron mal agregados; es más probable que haya uno solo que no debiera estar. La opción 1b) entonces quedará descartada para el resto del trabajo. Más adelante veremos como seleccionar entre las opciones 1a) y 2.

Si miramos atentamente, ambos métodos pueden explicarse en dos pasos. El primer paso es reducir el intervalo de un eje perteneciente al ciclo en el intervalo del mismo, y el segundo, resolver la inconsistencia de tipo tipo i generada al reducir. Para el segundo paso puede utilizarse cualquiera de los métodos propuestos para la resolución de inconsistencias de tipo tipo i, expansión o reducción. Si elegimos la reducción, terminaremos eliminando todos los ejes del ciclo en el intervalo en que este se produce, así como todos los caminos de reducción que comienzan en cada uno de los nodos pertenecientes al ciclo. Si en cambio elegimos de expansión, lo que haremos será expandir todos los intervalos de los ejes pertenecientes al camino de expansión del nodo destino del eje cuyo intervalo se redujo.

A continuación presentamos en detalle las posibles soluciones, al finalizar, veremos de que manera podemos decidir cual de todas ellas aplicar.

5.4.1. Corrección de ciclos por eliminación

Teorema 17 (Eliminación de ciclos)**.**

Sea D un documento temporal sin otra inconsistencia más que un ciclo C en el intervalo T. Llamemos S_t a la componente conexa del snapshot de D en t para cada $t \in T$ en la que se encuentran los nodos del ciclo C. Sea $I = \bigcup_{t \in T} V(S_t)$ la unión de los conjuntos de nodos de estas componentes. Sea D' el resultado de eliminar de D cada eje en S en el intervalo T, siendo S =Subgrafo de D inducido por I. Entonces D' es un documento XML temporal sin inconsistencias.

nota: la eliminación es completa, es decir, resuelve las inconsistencias que ella misma puede generar

Demostración.

i. D' no tiene ciclos (inconsistencias tipo *iv*).
D' no puede tener ciclos, dado que C era el único ciclo existente, y recortar intervalos o quitar ejes y nodos no puede producir nuevos ciclos.
$\Rightarrow$ D' no tiene inconsistencias de tipo *iv*.

ii. D' no tiene inconsistencias de tipo *i*.
Supongamos que D' tiene entonces una inconsistencia de tipo *i*. Entonces por definición de inconsistencia de tipo *i* (inconsistencias 4), existe un nodo n_i en el documento, con un eje saliente cuyo intervalo temporal no pertenece al lifespan del nodo. Como esta inconsistencia no estaba en D inicialmente, entonces se introdujo al eliminar el ciclo. Dado que el único tipo de modificaciones realizadas fue la eliminación de ejes dentro del intervalo de la inconsistencia, se debió eliminar un eje incidente a n_i cuya etiqueta temporal tenía intersección no vacía con T. Pero entonces, el eje ahora inconsistente también pertenecía a la componente conexa del snapshot en t (para algún $t \in T$) y debía haber sido eliminado.
$\Rightarrow$ D' no tiene inconsistencias de tipo *i*.

iii. D' no tiene inconsistencias de tipo *ii o* tipo *iii*.
El procedimiento para eliminar ejes en un intervalo, contempla la creación de baches temporales (Definición 9), corrigiéndolos mediante la duplicación para mantener la consistencia (ver las definiciones al principio del capítulo). De esta manera, no pueden generarse nuevos baches entre las etiquetas temporales de los ejes incidentes a un nodo. Cómo D inicialmente no tenía este tipo de inconsistencia,
$\Rightarrow$ D' no contiene inconsistencias de tipo *ii* o tipo *iii*

$\Rightarrow$ D' es un documento TXML. □

Algoritmo 11. Algoritmo de corrección de ciclos mediante la eliminación.

```
Input: Documento D,
       nodo n en el ciclo,
       intervalo T en que se produce el ciclo
Output: Documento D' sin inconsistencias
Resuelve la inconsistencia, eliminando todos los ejes del ciclo

eliminarCiclo(D,n,T){
(1) eje e=(ni,n,Te) incidente a n en T
(2) eliminar Te en T
(3) reducirCaminos(D, n, T)
(4) return D
}
```

La línea (4) resuelve las inconsistencias de tipo *i*, tipo *ii* y tipo *iii* que hayan sido generadas por la eliminación del eje en la línea (3).

Teorema 18 (Correctitud del Algoritmo 11)**.**
El Algoritmo 11 termina y retorna un documento consistente.

Demostración.

- *El algoritmo termina*
  Como ya vimos antes el Algoritmo reducirCaminos (Algoritmo 10) termina. Las demás operaciones son de orden constante.
- *El documento resultante no presenta inconsistencias* El ciclo es eliminado al eliminar el eje en T en la línea (3). A su vez se genera una inconsistencia de tipo *i* que se resuelve con la llamada al Algoritmo reducirCaminos en la línea (4). Cómo ya vimos antes, este algoritmo deja al documento en un estado consistente, por lo tanto el Algoritmo 11 deja al documento también en un estado consistente.

□

Análisis de complejidad

El algoritmo tiene el mismo orden que el Algoritmo 10 ya que todas las operaciones son de orden constante excepto la llamada al Algoritmo 10.

5.4.2. Corrección de ciclos por eliminación de un único eje

Algoritmo 12. Corrección de ciclos mediante la eliminación de un único eje

```
Input:    Un documento D con un ciclo como única inconsistencia,
          tal que al menos un nodo del ciclo tiene un eje
          incidente en un intervalo distinto de T.
          T = [ti,tf] el intervalo en el que se produce el ciclo
          n = uno de los nodos pertenecientes al ciclo con un eje
          incidente en un intervalo distinto de T.

Output:   D' consistente si es posible. El documento sin cambiar
          sino

eliminarCicloPorExpansion(D,T,n) {
(1) D'  = D
(2) Eliminar el eje incidente a n en T (o sea, en el ciclo)
(3) si !generaCiclosExpandir(D',T,n)
    begin
(4)     D = expandirCamino(D',n,T)
    end
(5) Retornar D
}
```

Teorema 19 (Correctitud del Algoritmo 12).
El Algoritmo 12 termina y retorna el documento consistente o sin ninguna modificación.

Demostración.

- *El algoritmo termina*: Las operaciones de las líneas (1) y (2) no pueden presentar problemas. Las líneas (3) y (4), se resuelven con los algoritmos 8 y 7, los cuales está demostrado, realizan un número finito de pasos.

b. *Resuelve*: Si se puede resolver la inconsistencia expandiendo (línea 3), entonces el documento resultante es consistente:

   i. no tiene inconsistencias de tipo *i*. Se produce una sola, al eliminar el eje, dado que el nodo n tiene ejes salientes en el intervalo T. Esta inconsistencia es resuelta en los siguientes pasos, y el algoritmo para resolverla asegura que siendo la única inconsistencia, retorna un documento consistente.

   ii. no tiene inconsistencias de tipo *ii* y tipo *iii* ya que se resuelven en las líneas (3) - (4) al expandir, puesto que se cubre el intervalo del eje eliminado expandiendo en forma consistente el intervalo de otro eje incidente al mismo nodo.

iii. tipo *iv*: al eliminar ejes no se provocan ciclos, la condición de la línea (3) da verdadera si el algoritmo para expandir no provoca ciclos. Si no es posible resolverla, se retorna el documento sin modificar.

□

Análisis de complejidad

Todas las operaciones son de orden constante salvo las líneas (3) y (4). El Algoritmo expandirCamino (7) tiene un orden de $O(|E|)$, mientras que generaCiclosExpandir (8) tiene $O(|V| * |E| + |E|)$. En total suman entonces $O(|V| * |E| + 2|E|)$

5.5. Evaluación de soluciones alternativas

En las secciones anteriores hemos visto que pueden existir más de una posible solución para determinados tipos de inconsistencia. En estos casos necesitaremos un criterio suficientemente objetivo para seleccionar la solución adecuada en cada escenario. Si bien a simple vista pareciera mejor eliminar información inconsistente reduciendo los intervalos temporales de los ejes, que agregar información que no sabemos si es cierta expandiendo estos intervalos, también es importante pensar que pasaría si la solución por reducción implicara por ejemplo, 10 cambios, mientras que expandiendo sólo se necesitara un cambio. Si suponemos que los errores son accidentales, uno podría pensar que la mejor solución sería aquella que requiera menor cantidad de pasos. Además, es posible que una reducción elimine totalmente un nodo del documento, por lo que no siempre eliminar información es necesariamente lo más correcto. Por otro lado, podemos razonar acerca de qué significa realmente agregar o eliminar información. No parece ser más correcto decir que el nodo A es padre de un nodo B en el intervalo $[0, 5]$ que decir que la relación se cumple en $[0, 3]$, si la realidad es que lo es en $[0, 4]$. En ambos casos la información parece ser igual de incorrecta: en el primero porque se asegura que una relación se cumple en un intervalo mayor que el real, mientras que en la segunda, se asegura que la relación se cumple en un intervalo menor que el verdadero y no más allá. Entonces también resulta intuitivo pensar que la mejor solución es aquella que necesite menor cantidad de cambios para llevarse a cabo, dado que es la que modifica menor cantidad de ejes. Bajo esta suposición, definimos entonces la métrica a utilizar de la siguiente manera:

Definición 17 (Métrica de comparación de soluciones)**.**
Dada una inconsistencia I, cuya resolución tiene alternativas $m_1 ... m_f$, la métrica a utilizar para decidir la alternativa de resolución a aplicar es la cantidad de cambios que produce cada posible método de resolución, donde un cambio puede ser:

- *la expansión de un intervalo*
- *la reducción de un intervalo*
- *la eliminación de un eje en un intervalo*
- *la duplicación de un nodo*
- *la eliminación total de un eje*

En este trabajo daremos a todos los cambios el mismo peso, es decir, todos suman 1 punto al resultado. Llamaremos $\kappa(m, I, D)$ *a la cantidad de cambios requerida por el método m, para la inconsistencia I en el documento D.*
Utilizaremos por comodidad, las abreviaturas $\kappa_r(I)$ *y* $\kappa_e(I)$ *para referirnos a la cantidad de cambios necesarias para resolver la inconsistencia I por reducción y expansión respectivamente, siempre que el contexto deje en claro el documento al que aplican.*

Definición 18 (Heurística de selección de método)**.**
Según la definición anterior, el método de corrección a aplicar para cada inconsistencia se define como:
Dados $m_1 ... m_k$ *las posibles soluciones para la inconsistencia I en el documento D,* $sol(I, D) = m_j$ *tal que* $\kappa(m_k, I, D) = \min_i \kappa(m_i, I, D)$ $i : 1..k$

5.6. Resumen

En este Capítulo vimos como corregir de manera eficiente las inconsistencias presentadas en la Definición 4, bajo la condición de que exista una única inconsistencia dentro del documento. Definimos conceptos importantes para la corrección de inconsistencias (*camino de reducción*, *camino de expansión* e *instante de máxima expansibilidad*) y presentamos diversos métodos de corrección para cada uno de los tipos de inconsistencia (*duplicación de nodos*, *reducción de intervalos* y *expansión de intervalos*). Además definimos una heurística basada en la cantidad de cambios necesarios para llevar a cabo la corrección, para seleccionar apropiadamente el método a utilizar ante la presencia de más de una posibilidad.
En el Capítulo siguiente, veremos que pasa cuando se presenta más de una inconsistencia en el documento.

Capítulo 6

Inconsistencias combinadas

Hasta aquí hemos visto cada inconsistencia de manera aislada. Ahora veremos que sucede en el caso más real, en el cual se presenta más de una inconsistencia en un documento. En primer lugar veremos cuál es el área del documento que afecta cada inconsistencia y, en función de ésta, analizaremos cuándo una inconsistencia se puede considerar aislada. En este caso podremos aplicar los conceptos vistos en el capítulo anterior. Finalmente presentaremos soluciones para los casos en que las inconsistencias concurrentes no puedan resolverse en forma aislada.

6.1. Areas de influencia

Definición 19 (Nodo Afectado)**.**

Denominaremos nodos afectados por una inconsistencia a aquellos nodos que se vean alterados de alguna manera por la resolución de la inconsistencia. Las alteraciones que puede sufrir un nodo son las siguientes

1. *Eliminación de un eje incidente o saliente*
2. *Reducción del intervalo de un eje incidente o saliente*
3. *Expansión del intervalo de un eje incidente o saliente*

Definición 20 (Area de influencia)**.**

Llamaremos ***área de influencia*** *de la inconsistencia I, y la denotaremos $A_{inf}(I)$ al conjunto formado por los nodos afectados por las posibles soluciones I. Llamaremos* ***área de expansión*** *de la inconsistencia I, y la denotaremos $A_e(I)$, al conjunto de nodos pertenecientes a el/los camino/s de expansión que forman el área de influencia. De la misma manera, llamaremos* ***área de reducción*** *de la inconsistencia I, y la denotaremos $A_r(I)$ al conjunto de nodos pertenecientes a los caminos de reducción que forman el área de influencia.*

Definición 21 (Camino de reducción).
Dada una inconsistencia de tipo i o tipo iv sobre un nodo I en el intervalo T, llamaremos ***camino de reducción*** *a cada camino continuo desde I a otro nodo del documento, en un intervalo T' tal que* $T' \cap T \neq \phi$

A continuación, definiremos con exactitud el área de influencia de cada tipo de inconsistencia, presentando a la vez un algoritmo que calcula cada una de ellas.

Definición 22 (Area de influencia de una inconsistencia de tipo *i*).
Sea T el intervalo en el que se produce la inconsistencia, llamaremos ***área de influencia de una inconsistencia de tipo i*** *al conjunto de nodos formado por los nodos pertenecientes al camino de expansión y los nodos pertenecientes a todos los caminos de reducción.*

Ejemplo 21. *Consideremos la Figura 6.1. El nodo node_3 presenta una inconsistencia de tipo i, ya que el intervalo del eje saliente hacia el nodo node_4 tiene asociado un intervalo no incluido en el lifespan del nodo node_3. Las posibles soluciones a estas inconsistencias son, como vimos con anterioridad, reducir o expandir. Hay un único camino de reducción, node_3 → node_4 → node_6 y el camino de expansión es node_1 → node_2 → node_3. Si tenemos en cuenta la reducción, se ven afectados todos los nodos que pertenecen a algún camino de reducción, es decir, los nodos node_3, node_4 y node_6. Si elegimos resolver mediante la expansión, se verán afectados todos los nodos pertenecientes al camino de expansión, es decir, node_3, node_2 y node_1, ya que el lifespan de node_1. Entonces, los nodos que pueden llegar a sufrir modificaciones son node_1,node_2,node_4,node_6 y node_3.*

A continuación presentamos un algoritmo para calcular el área de influencia de esta inconsistencia.

Algoritmo 13. (Cálculo del área de influencia de una inconsistencia tipo *i*)

```
Input: g con una única inconsistencia de tipo i
       n el nodo inconsistente
       T intervalo de la inconsistencia
Output: Lista de nodos afectados

NodeList getAreaInfluenciaI(Documento g, Nodo n, TimeStamp T){
  (1)listaNodos = getCaminoUltimosPadres(g,n,T)
  (2)listaNodos.addAll(getNodosAlcanzables(g,n,T)
  (3)return listaNodos
}
```

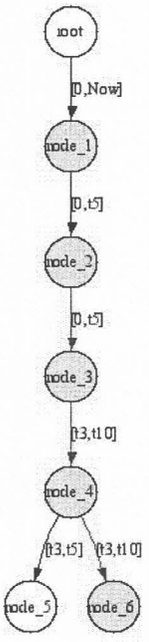


Figura 6.1: Ejemplos de áreas de influencia de las inconsistencias de tipo i

El área de influencia de una inconsistencia de tipo i está formada, como dijimos en la Definición 22 por todos los nodos pertenecientes al camino de expansión (calculado en la línea (1)) y los nodos pertenecientes a todos los caminos de reducción (calculados en la línea (2)). El método *addAll* en (2) agrega todos los elementos retornados por el llamado a $getNodosAlcanzables$ (que computa los caminos de reducción) a la lista resultante del llamado a $getCaminoUltimosPadres$ (que computa los caminos de expansión) en la línea (1).

Análisis de complejidad

El primer algoritmo tiene orden $O(|E|)$ en el peor caso, dado que para cada nodo del documento, se recorren una única vez todos sus ejes incidentes para hallar el último padre.
Para el segundo algoritmo (línea (2)) es orden también es $O(|E|)$ si suponemos que los documentos no tienen ciclos no temporales. Si nos olvidamos de esta suposición, el algoritmo se transforma en un algoritmo exponencial, como ya discutimos anteriormente con el caso del algoritmo reducirCaminos (10).

El siguiente algoritmo calcula el camino de expansión para la inconsisten-

cia. Difiere del algoritmo presentado en el capítulo anterior en que aquí no se modifica el camino, sólo se obtienen los nodos pertenecientes a él, para formar el área de expansión de la inconsistencia y poder analizar las interferencias. El algoritmo provee una manera de lidiar con posibles ciclos presentes en el documento, al retornar la lista vacía en la línea (8) si sucede que se pasa dos veces por el mismo nodo indicando de esta manera que no hay camino de expansión posible.

Algoritmo 14. (Cálculo del camino de expansión de un nodo)

```
Input: Documento g
       Nodo n para el cual se quiere el camino de últimos padres
       T Intervalo temporal para el cual se quiere el camino de
                 últimos padres
Output: Lista con los nodos pertenecientes al camino de últimos padres de n

NodeList getCaminoUltimosPadres(Documento g, Nodo n, TimeStamp T){
(1)   nf = n
(2)   camino = [n]
(3)   Mientras lifespan(nf) no incluya a T
      begin
(4)       e = getEjeLimitrofe(nf,T)
(5)       camino.append(e.nodeFrom())
(6)       nf = e.nodeFrom()
(7)       si nf fue visitado
          begin
(8)          return []
          end
(9)       sino
          begin
(10)         marcar nf como visitado
          end
       end
(11)   return camino
}
```

El Algoritmo 14 construye el camino de últimos padres en T para el nodo n. Esto se logra comenzando por el nodo n, buscando su último eje incidente en T y agregando el nodo origen de este eje (n_i) al camino. Luego se repite el procedimiento para n_i, concluyendo cuando el lifespan del nodo al que se llega incluye totalmente al intervalo T, según la condición del ciclo en la línea (3). Si en una iteración se llega a un nodo por segunda vez (lo cual se puede saber porque se marcan los nodos cada vez, en la línea (10)) sabemos que nos encontramos con un ciclo temporal dentro del camino de últimos padres y por lo tanto éste no puede ser calculado, por lo cual se retorna una lista vacía en la línea (8).

El Algoritmo 15 obtiene el último padre al nodo n_f en el intervalo T.

Algoritmo 15. (Búsqueda del eje limítrofe)

```
Input: Nodo n para el cual se quiere encontrar el eje incidente
              fronterizo al intervalo de la inconsistencia
       Intervalo T=[ti,tf] de inconsistencia
Output: Eje e, último padre de n en T

Edge getEjeLimitrofe(nf,T){
```

```
(1)   Edge[] edges = nf.inEdges();
(2)   Edge e =  edges[0]
(3)   Para cada eje ea en edges
      begin
(4)      si (T(e))f < T((ea))f < Ti
         begin
(5)        e = ea
         end
      end
(6)   return e
}
```

Para esto recorre todos los ejes incidentes al nodo n_f (ciclo de la línea (3)) quedándose con aquel cuyo intervalo temporal tiene mayor instante final, siendo este instante menor al del intervalo T. Si quisiéramos buscar el primer padre en T del nodo, deberíamos cambiar la condición en la línea (4) por $Tf < T((ea))i < (T(e))i$.

Análisis de complejidad

Vemos que el algoritmo tiene un orden de $O(|E|)$ en el peor caso. Vale la pena aclarar que si estamos en la presencia de un bache, se retornará siempre el último eje incidente, pero también podría cambiarse el algoritmo para que retorne el primer eje incidente en T. Analizar todas las posibles combinaciones de primeros y últimos padres a lo largo del un camino en este caso, sería potencialmente exponencial (si hay muchos baches) y no tiene demasiada importancia, sobre todo si consideramos, como veremos más adelante cuando tratemos inconsistencias combinadas, que los baches es conveniente resolverlos en primera instancia.

El Algoritmo 16 calcula los nodos alcanzables desde un nodo n_p en el intervalo T. Estos nodos serán los que se verán afectados si optamos por reducir intervalos para resolver una inconsistencia de tipo i. El algoritmo recorre el documento a partir del nodo n_p, calculando los caminos continuos a todos los nodos.

Algoritmo 16. (Cálculo de los caminos de reducción)

```
Input: Documento g
       Nodo n para el cual se quiere el conjunto de nodos
       alcanzables
       T Intervalo temporal para el cual se quiere el conjunto
                 de nodos alcanzables
Output: Nodos alcanzables desde el nodo n en el intervalo T.

NodeList getNodosAlcanzables(Documento g, Nodo np, TimeStamp T){
(1)   Cola nodos     = [np]
(2)   Hash cp = { (np, [ T ]) }
(3)   np.setVisitado(true)
(4)   Mientras !nodos.empty() hacer
      begin
(5)      n = nodos.first()
(6)      n.setVisitado(false)
(7)      Tcp = cp.get(n)
(8)      Para cada eje e=(n,nf,Te) saliente de n
```

```
        begin
(9)       Para cada intervalo Ti en Tcp
          begin
(10)        Si Ti interseccion Te != vacía
            begin
(11)          Si (!e.transitado(Ti interseccion Te))
              begin
(12)            e.addTransitado(Ti interseccion Te)
(13)            cp.put(nf, cp.get(nf).add(Ti interseccion Te))
(14)            si !nf.isVisitado()
                begin
(15)              nodos.append(nf)
(16)              nf.setVisitado(true)
                end
              end
            end
          end
        end
      end
(17)  return cp.keys();
}
```

La cola `nodos` guarda los nodos que alcanzamos desde el nodo inicial pero que aún no hemos traspasado. La estructura de hash c_p mantiene para cada nodo visitado, la lista de intervalos en los cuales éste ha sido alcanzado desde el nodo inicial. En cada iteración, se toma un nodo de la lista en la que se guardan los nodos visitados, y se analizan sus ejes salientes. Al analizar el nodo n_i veremos para cada uno de sus ejes salientes, cuales tienen etiqueta temporal con intersección no vacía con los intervalos en los cuales se ha llegado a n_i. Si el eje $\epsilon_f(n_i, n_f, T_f)$ cumple con esta condición, se agregan a la lista dada por $c_p(n_f)$, los intervalos dados por $T_f \bigcap T_j$ para cada intervalo T_j en $c_p(n_i)$, indicando que se alcanzó el nodo n_f desde el nodo inicial en cada uno de estos intervalos. Finalmente, las claves del hash indican que nodos son alcanzables desde n_p en algún instante perteneciente a T. El algoritmo calcula también precisamente en que instantes estos nodos son alcanzables desde el nodo.

Análisis de complejidad

El algoritmo, como ya dijimos anteriormente, es exponencial en el peor caso. Se puede modificar de manera de pasar una única vez por cada eje en caso de que no existan ciclos no temporales y que la consistencia sea única dentro del documento, con un mecanismo de colas de espera al igual que el algoritmo reducirCaminos (10).

Definición 23 (Area de influencia de una inconsistencia de tipo *ii*)**.**
El área de influencia de una inconsistencia de tipo ii o tipo iii está formada únicamente por el nodo inconsistente si se trata de un bache, mientras que se compone del nodo inconsistente y los nodos origen de los ejes inconsistentes en el caso de un solapamiento.

Como vimos en el capítulo anterior la solución a la presencia de baches es duplicar el nodo, por lo tanto sólo se verán afectados algunos de los ejes con extremo en el mismo y se agregará un nodo al documento, viéndose afectado sólo el nodo inconsistente. En cambio, en el caso de los solapamientos, el área de influencia incluirá también los nodos origen de los ejes inconsistentes, puesto que debe modificarse la etiqueta temporal de alguno de ellos para eliminar el solapamiento. Ejemplos de estas áreas pueden verse en la Figura 6.2

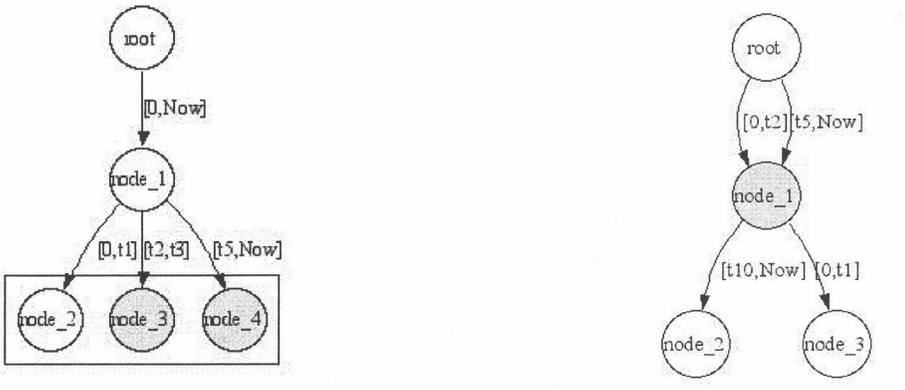


Figura 6.2: Ejemplos de áreas de influencia de las inconsistencias de tipo *iii* y *ii*

Definición 24 (Area de influencia de una inconsistencia de tipo *iv*).
El área de influencia de un ciclo está formada por

1. *todos los nodos en cada camino de reducción.*
2. *todos los nodos pertenecientes al camino de expansión (con etiqueta temporal anterior al intervalo del ciclo) para cada nodo del ciclo.*

El item 1 es producto de solución que involucra eliminar todos los ejes del ciclo, dado que de esta manera los nodos pertenecientes a él no incluirán en su lifespan el intervalo del mismo, y por lo tanto, todos los caminos con origen en esos nodos en dicho intervalo también deben ser eliminados.

Como ya vimos, otra solución sería eliminar un eje y modificar las etiquetas temporales del camino de expansión de ese nodo, con lo cual, todos los nodos en estos caminos (recordemos que hay un camino por cada nodo del ciclo que tenga un eje incidente fuera de éste y por ende pueda ser expandido su lifespan) son candidatos a ser modificados y por ende son parte del área de influencia de este tipo de inconsistencia.

Ejemplos de estas áreas pueden verse en la Figura 6.1.

Ejemplo 22 (Area de influencia de una inconsistencia tipo *iv*)**.**

En la Figura 6.3 se puede apreciar en color los ejes que forman el ciclo. Vemos entonces que si optamos por eliminarlo por completo, tanto los nodos del ciclo como el nodo node_7 se verán afectados dado que este último es alcanzado desde el nodo node_3 en el intervalo del ciclo. De hecho, todos los nodos salvo el node_2 serán eliminados físicamente.

Otra de las opciones consiste en eliminar sólo uno de los ejes del ciclo. Para esto necesitamos que el nodo al que el eje incide tenga al menos un eje incidente cuyo nodo origen no pertenezca al ciclo. Esto lo cumple sólo el node_2. Es decir, podemos eliminar el eje $\epsilon(n_5, n_2, T_{52})$ y expandir los nodos intervalos del camino de últimos padres del nodo node_2, con lo cual el nodo node_1 también se ve afectado.

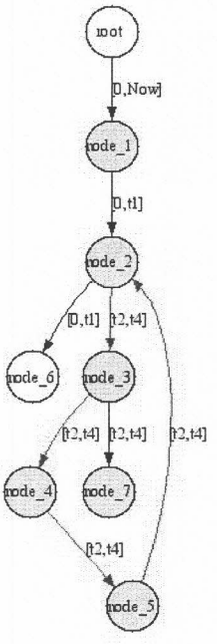


Figura 6.3: Ejemplos de áreas de influencia de las inconsistencias de tipo *iv*

A continuación presentamos los algoritmos correspondientes al cálculo de esta área de influencia.

Algoritmo 17. (Cálculo del área de influencia de una inconsistencia tipo *iv*)

```
Input: Documento  g
       NodeList listaNodos, los nodos que conforman el ciclo
       TimeStamp T, el intervalo en el cual se produce el ciclo
```

```
Output: Lista de nodos afectados por la inconsistencia.
NodeList getAreaInfluenciaIV(Documento g, NodeList listaNodos,
                                                    TimeStamp T){
(1)   res = []
(2)   Para cada nodo n en listaNodos
(3)     res.addAll(getCaminoUltimosPadres(g,n,T))
(4)     res.addAll(getNodosAlcanzables(g,n,T))
(5)   return res
}
```

El Algoritmo 17 calcula el área de influencia de una inconsistencia de tipo *iv*. Según la Definición 24, esta área se compone por los nodos pertenecientes al camino de expansión y de todos los caminos de reducción para cada nodo perteneciente al ciclo. El algoritmo recorre entonces los nodos del ciclo (ciclo de la línea (2)) y para cada uno de ellos calcula el camino de últimos padres (linea(3)) y los nodos pertenecientes a los caminos de reducción (línea (4)). El método addAll agrega todos los nodos retornados por $getCaminoUltimosPadres$ y $getNodosAlcanzables$ al resultado.

Análisis de complejidad

Como ya vimos anteriormente en el análisis del Algoritmo 13 al calcular el área de influencia de una inconsistencia de tipo *i*, el algoritmo de la línea 3 tiene orden $O(|E|)$ mientras que el de la línea 4 también pero sólo si el documento no contiene ciclos no temporales, sino sería exponencial. Suponiendo que no hay ciclos no temporales, el orden es entonces (peor caso)
$O(|V||E|) \leq O(|E|^2)$

6.2. Interferencia entre las inconsistencias

Como dijimos en la Sección anterior, hasta ahora hemos estudiado las inconsistencias aisladas en un documento. En un caso real, es probable que haya más de una inconsistencia al mismo tiempo. Esto puede provocar que al resolver una inconsistencia modifiquemos las posibles maneras de resolver las otras, es decir, interfieran entre sí. En este capítulo abordaremos este problema, pero para esto debemos definir primero qué entendemos por *interferencia entre inconsistencias*

Definición 25 (Interferencia entre inconsistencias).
Diremos que dos inconsistencias I_1, I_2 interfieren si sus áreas de influencia tienen intersección no vacía, es decir, si se cumple:

$$A_{inf}(I_1) \cap A_{inf}(I_2) \neq \emptyset$$

Como consecuencia de esta definición, podemos decir que una inconsistencia se encuentra *aislada* si no interfiere con ninguna otra en el documento. En este caso, podremos resolver cada una utilizando los algoritmos y conceptos

brindados en el capítulo anterior, sin preocuparnos por el orden utilizado al resolverlas.
Dedicaremos el resto del capítulo a analizar qué sucede cuando las inconsistencias en efecto interfieren. Nos interesará buscar, a bajo costo, conjuntos de inconsistencias que no interfieran entre ellas, o que de hacerlo, podamos resolverlas en cualquier orden sin que varíe el resultado. Para ello clasificaremos las interferencias en *relevantes* e *irrelevantes*, siendo *irrelevantes* aquellas para las cuales el orden de corrección altera el resultado final. Daremos entonces algunas propiedades bajo las cuales una interferencia es irrelevante.
Para comenzar, limitaremos el área donde efectivamente se produce la interferencia, reduciendo entonces el área de conflicto sobre la que se realizará el análisis. Llamaremos a esta área *Area de Interferencia.*

Definición 26 (Area de Interferencia)**.**
Llamaremos Area de Interferencia entre las inconsistencias al conjunto de nodos que se encuentran en la intersección de las áreas de influencia.

En el caso en que trabajemos con la interferencia entre dos inconsistencias, I_1 e I_2 denotaremos al área de interferencia como $A_i(I_1, I_2)$.

Si bien la intersección entre las áreas de influencia determina si dos o más inconsistencias interfieren, esta interferencia no siempre resulta ser problemática a la hora de resolverlas. En algunos casos, se puede proceder a resolver las mismas de la misma manera en que lo haríamos si no interfirieran en absoluto. En virtud de esto, podemos clasificar las interferencias según sus consecuencias para la corrección de las inconsistencias involucradas. Tenemos entonces las siguientes categorías:

Definición 27 (Clasificación de interferencias según sus consecuencias)**.**

- *Irrelevantes: un conjunto de k inconsistencias interfieren de manera irrelevante sus áreas de influencia tienen intersección no vacía y el documento resultante es el mismo sin importar en cuál de los k! ordenamientos posibles se resuelvan.*
- *Relevantes: un conjunto de inconsistencias interfieren de manera relevante si interfieren, y el documento resultante depende del orden que se aplique para resolver cada una de ellas.*

Ejemplo 23 (Interferencia Irrelevante)**.**
En algunos casos, si bien las inconsistencias interfieren (la intersección de las áreas de influencia no es vacía), esta interferencia no acarrea consecuencias. Por ejemplo, dos inconsistencias de tipo ii, si se trata de baches en ambos casos, interfieren de esta manera. Podemos apreciar este caso en la Figura 6.4 en la que se muestran dos inconsistencias de tipo ii sobre el mismo nodo. La solución a ambas sería duplicar el nodo inconsistente. Es fácil ver que el resultado será el mismo no importa que inconsistencia resolvamos primero.

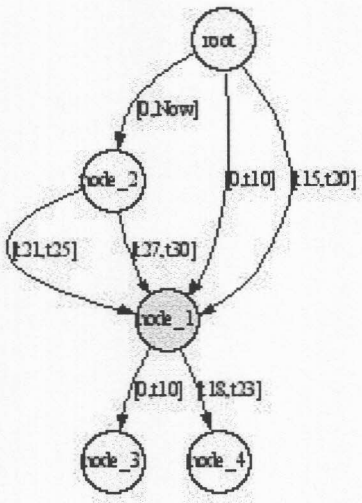


Figura 6.4: Interferencia irrelevante entre dos inconsistencias de tipo *ii*

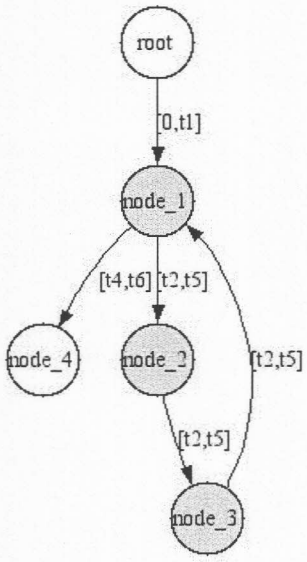


Figura 6.5: Interferencia Relevante entre una inconsistencia de tipo *i* y otra de tipo *iv*

Ejemplo 24 (Interferencia relevante)**.**

En este caso, una inconsistencia puede limitar o modificar las posibilidades de solución de otra, por ejemplo en la Figura 6.5 tenemos un ciclo interfiriendo con una inconsistencia de tipo i. El problema aqui es que el ciclo involucra nodos en el camino de últimos padres del nodo con la inconsistencia de tipo i, con lo cual, esta última no puede resolverse expandiendo. También puede darse el caso de dos inconsistencias de tipo i interfiriendo entre sí, una en el camino de expansión de la otra, con lo que la detección de posibles ciclos al expandir se ve afectada. En ambos casos, es necesario tener en cuenta la naturaleza de ambas inconsistencias para poder resolverlas correctamente.

6.3. Interferencias irrelevantes

No es fácil determinar cuando dos inconsistencias interfieren de manera irrelevante. Sin embargo, sabemos que si así ocurre, entonces podemos aplicar los métodos de corrección de inconsistencias vistos en el capítulo anterior.[1] Por es-

[1] En el caso del Algoritmo hayCamino, sólo podremos utilizar la versión no exponencial en el caso en que el nodo inicial del camino de expansión en cuestión no se encuentre en el área de interferencia de las inconsistencias. Caso contrario deberá utilizarse la versión exponencial de dicho algoritmo, puesto que no aplica la Proposición 14.

ta razón, dedicaremos el resto del capítulo a presentar y demostrar propiedades que hacen que la interferencia entre inconsistencias sea irrelevante.

A continuación, veremos algunos lemas que nos serán de utilidad para demostrar las propiedades deseadas.

Proposición 20. *Sean dos inconsistencias I_1, I_2. Si ambas se resuelven por reducción, entonces el resultado final es el mismo sin importar el orden de corrección*

Demostración.

Si no fuera así, entonces pasaría alguna de las siguientes cosas:

1. Al resolver las inconsistencias en el orden $I_2 I_1$ no se modifica un eje que sí se ve afectado si resolvemos en el orden $I_1 I_2$
2. Resolviendo las inconsistencias en el orden $I_2 I_1$ un eje queda con un intervalo distinto que si se resuelven el el orden inverso $(I_1 I_2)$
3. Resolviendo las inconsistencias en el orden $I_2 I_1$ el documento resultante no contiene un eje que sí está presente en el documento resultante de resolver las inconsistencias en el orden $I_1 I_2$

1. Supongamos que hay un eje ϵ_i que resulta modificado al resolver las inconsistencias en el orden $I_1 I_2$ pero no al resolverlas en orden inverso. Este eje debe pertenecer al menos a una de las áreas de reducción ($A_r(I_2)$ y $A_r(I_1)$) para ser afectado al corregir ambas inconsistencias reduciendo intervalos. Supongamos en primer lugar que el eje ϵ_i se encuentra en $A_r(I_1)$. Sabemos que al resolver en el orden $I_2 I_1$ no se ve afectado, es decir, luego de resolver I_2 reduciendo intervalos, el eje ϵ_i ya no se encuentra en $A_r(I_1)$. Supongamos que $C_r = (n_1 n_2 ... n_k, T')$ es un camino de reducción de I_1 que pasa por ϵ_i en $T' \subseteq T(I_1)$. Sea $\epsilon_i = (n_i, n_{i+1}, T_i)$ $(T_i \supseteq T')$. Al menos un eje del camino tiene que haber sido eliminado en T' para que C_r haya dejado de ser un camino de reducción para I_1. En particular, algún eje anterior a ϵ_i en el camino debe haber sido eliminado en T' (ya que ϵ_i no pudo haberlo sido, porque supusimos que no fue modificado). Si $\epsilon_j(n_j, n_{j+1}, T_j)$ $(T_j \supseteq T')$ es el eje modificado, con $j < i$ entonces para que C_r se vea afectado, tiene que haberse eliminado este eje en T'. Pero n_{j+1} pertenece a C_r, por lo tanto tiene un eje saliente en T' también dentro de C_r y será modificado al resolver I_2 por reducción de intervalos. Si continuamos con el mismo razonamiento, podemos ver que todos los ejes que conforman C_r a partir del eje ϵ_j se verán afectados. Entre ellos ϵ_i. Esto resulta en un absurdo porque partimos de la suposición que ϵ_i no se vería afectado.
Supongamos entonces que ϵ_i se encuentra en $A_r(I_2)$. Resolviendo $I_2 I_1$ no se afecta el eje, pero sin embargo está en el área de reducción de la primer inconsistencia a resolver. Como la reducción no agrega ejes ni expande sus intervalos, obtenemos un absurdo, puesto que el eje será modificado al reducir para resolver la primera inconsistencia y llegará así hasta el final.
Por último, si se encuentra en ambas entonces al resolver la primera, se modifica

y luego o bien queda fuera del área de reducción de la segunda o bien se vuelve a reducir. De cualquiera de las dos maneras el eje definitivamente se ve afectado. De los tres casos deducimos que 1 no se cumple.

2. Un eje queda con intervalo distinto. Si una sola de las inconsistencias lo tiene en su área de reducción, entonces no se verá modificado por la otra de ninguna manera, por lo tanto no puede suceder que quede con intervalos distintos al cambiar el orden. Si en cambio se encuentra en las dos áreas de reducción entonces:

Sea $T = [t_i, T(I_1) + t]$.

Si resolvemos en el orden $I_1 I_2$ entonces,

$T = [t_i, T(I_1) - 1]$ y luego
$T = [t_i, min(T(I_1) - 1], T(I_2) - 1)$

Si en cambio resolvemos en el orden $I_2 I_1$
$T = [t_i, T(I_2) - 1]$ y luego
$T = [t_i, min(T(I_2) - 1], T(I_1) - 1)$

Claramente esto sucede para todos los ejes que se encuentran dentro del área de reducción, por lo tanto los intervalos son iguales sin importar el orden de corrección.

3. Es la inversa del caso 1.
De 1., 2. y 3. deducimos que el orden de las reducciones no afecta el resultado. □

Proposición 21. *Sean dos inconsistencias I_1, I_2, con $\kappa_e(I_1)$ y $\kappa_e(I_2)$ (Definición 17) respectivamente. Sea $\kappa\prime_e(I_2)$ la cantidad de cambios necesaria para resolver I_2 luego de expandir para resolver I_1, entonces $\kappa\prime_e(I_2) \leq \kappa_e(I_2)$.*

Demostración.

Expandir un intervalo es hacer el eje válido por un tiempo mayor. No se eliminan ni agregan ejes en este proceso. Por lo tanto, la única modificación que puede sufrir el camino de últimos padres de la inconsistencia I_2 si resolvemos primero I_1 por expansión es que se agranden los intervalos de sus ejes, ya que tampoco sucede que los nodos cambien la relación de últimos padres al expandir. Esto bien puede dejar $\kappa_e(I_2)$ intacto (no se expande ningún intervalo lo suficiente para cubrir el intervalo de la inconsistencia I_2) o puede acortarse el camino, con la consiguiente reducción de $\kappa_e(I_2)$, si $T(I_1) \geq T(I_2)$ y los caminos de últimos

padres comparten ejes. En conclusión, nunca se aumenta la cantidad de cambios para expandir de la segunda inconsistencia. □

Proposición 22. *Sean dos inconsistencias* I_1, I_2.

$$A_e(I_1) \cap A_r(I_2) = A_e(I_2) \cap A_r(I_1) = \emptyset$$

implica que al resolver una de las inconsistencias por expansión, no se generan nuevos posibles ciclos para las soluciones por expansión de la otra.

Demostración.

Si expandir para resolver ambas inconsistencias es una opción, significa que en el documento original no se producen ciclos al expandir ninguna de las dos por separado (sin resolver la otra).
Supongamos que al expandir I_1 se imposibilita la corrección de I_2 y que la razón de esto es que se forma un ciclo al expandir. Esto quiere decir que se formó un camino C entre el nodo N_2 (nodo inconsistente de I_2 y un nodo n_i en el camino de expansión de I_2 en un intervalo $T \subseteq T_2$, al expandir para resolver I_1, de manera que al expandir para resolver I_2 se formaría un camino desde n_i hasta N_2 en el mismo intervalo dando como resultado un ciclo en T.
El camino se formó entre N_2 y n_i, sea $C = N_2, n_1, n_2, ..., n_{i-1}, n_i$, al menos parte de los nodos de este camino se encontraban en el camino de expansión de I_1 supongamos que el camino de expansión era $n_1, n_2, ..., n_{i-1}$. Entonces existía un camino de N_2 a n_1 en T con $T \bigcap T_2 \neq \emptyset$ porque sino no se formaría ciclo. Esto quiere decir que n_1 estaba en el área de reducción de I_2 y $A_r(I_2) \bigcap A_e(I_1) \neq \emptyset$ lo cual es un absurdo puesto que supusimos que estas áreas no contenían nodos en común.

□

Interferencias entre inconsistencias de tipo *i* o tipo *iv*

A continuación veremos algunas propiedades en las cuales dos inconsistencias de tipo *i* o dos inconsistencias de tipo *iv* interfieren de manera irrelevante. Las propiedades enunciadas y demostradas valen para ambos tipos de inconsistencia dado que hay una estrecha relación entre ambas. Esto es porque para eliminar un ciclo en primer instancia eliminamos un eje dando lugar de esta manera, a una inconsistencia de tipo *i*.

Proposición 23. *Sean* I_1, I_2 *dos inconsistencias de tipo i (o dos inconsistencias de tipo iv) y* T_1, T_2 *los intervalos en que se producen, respectivamente, interfieren de manera irrelevante si se cumple alguna de las siguientes propiedades.*

a. La cantidad de cambios necesaria para resolver cada una de ellas por reducción, es menor que la cantidad necesaria para resolverlas por expansión y además no se intersecan el área de expansión de una con el área de reducción de la otra. Es decir,

i. $\kappa_r(I_1) < \kappa_e(I_1) \wedge$

ii. $\kappa_r(I_2) < \kappa_e(I_2) \wedge$

iii. $A_e(I_1) \cap A_r(I_2) = A_e(I_2) \cap A_r(I_1) = \emptyset$

b. La cantidad de cambios necesaria para resolver cada una de las inconsistencias por expansión es menor que la cantidad de cambios para resolver cada una por reducción y además no se intersecan el área de expansión de una con el área de reducción de la otra. Es decir,

i. $\kappa_r(I_1) > \kappa_e(I_1) \wedge$

ii. $\kappa_r(I_2) > \kappa_e(I_2) \wedge$

iii. $A_e(I_1) \cap A_r(I_2) = A_e(I_2) \cap A_r(I_1) = \emptyset$

c. Se cumple

i. $A_i(I_1, I_2) = A_r(I_1) \cap A_r(I_2) \wedge$

ii. $T_1 \cap T_2 = \emptyset \wedge$

iii. $A_i(I_1, I_2) \cap A_e(I_1) = A_i(I_1, I_2) \cap A_e(I_2) = \emptyset$

d. Se cumple

i. $A_i(I_1, I_2) = A_e(I_1) \cap A_r(I_2) \wedge$

ii. $T_1 \prec T_2 \ \wedge \ T_1 \cap T_2 = \emptyset$

iii. $A_i(I_1, I_2) \cap A_e(I_2) = A_i(I_1, I_2) \cap A_r(I_1) = \emptyset$

(o $T_1 \succ T_2$ si el camino de expansión es de primeros padres)

Demostración.

De no ser irrelevantes, al resolver la primer inconsistencia influiríamos en la

restante, al menos para uno de los posibles órdenes de corrección. Supongamos entonces que se cumple la hipótesis de la proposición para las inconsistencias I_1 e I_2 y veamos que sucede si intentamos resolver en los dos posibles órdenes.

a. Orden 1: I_1, I_2. Para resolver I_1 optaríamos por reducir intervalos, dado que es la opción con menor cantidad de cambios y según la sección **??**, sería entonces la mejor opción. $A_e(I_2)$ no se ve afectada ya que por el punto *iii*, sus nodos no se encuentran en el área de interferencia. Por lo tanto $\kappa_e(I_2)$ permanece constante.
Supongamos que $\kappa_r(I_2)$ aumenta, entonces se agregó un eje al área de reducción de I_2. Sin embargo, para resolver la primer inconsistencia se redujeron intervalos, y esta operación nunca crea ejes nuevos, por lo tanto es absurdo y $\kappa_r(I_2)$ no aumenta al resolver la primer inconsistencia con reducción y la segunda inconsistencia seguirá teniendo como mejor opción de solución el método por reducción.

   Orden 2: I_2, I_1. Como no supusimos nada sobre I_1 e I_2 para definir el orden, y el razonamiento anterior tampoco supone propiedades distintas para ellas, entonces la demostración es la misma que para el caso anterior.

   Siendo que en ambos casos la solución es la reducción y haciendo uso de la proposición 20 concluimos que interfieren de manera irrelevante.

b. De no interferir de manera irrelevante, al resolver la primer inconsistencia influiríamos en la restante, al menos para uno de los posibles órdenes de corrección. Supongamos entonces que se cumple la hipótesis de la proposición para las inconsistencias I_1 e I_2 y veamos que sucede si intentamos resolver en los dos posibles órdenes.

   Orden 1: I_1, I_2. Al resolver I_1 por expansión, pueden disminuir la cantidad de cambios necesaria para expandir I_2, pero nunca aumentar (ver 21). Además, el $A_r(I_2)$ no se achica, puesto que no se quitan ejes ni se achican intervalos temporales, sólo se expanden intervalos, y la única manera de que esta área sea afectada es que se agregasen nodos producto de la expansión. Por lo tanto, ya que $\kappa_e(I_2)' \leq \kappa_e(I_2) \wedge \kappa_r(I_2)' \geq \kappa_r(I_2)$, y sumado a que la proposición cumple con 21 (no se forman ciclos), el método seleccionado para resolver la segunda inconsistencia no varía.
   Como además, no impusimos ninguna restricción para seleccionar cual de las dos inconsistencias se resuelve primero, concluimos que interfieren de manera irrelevante.

c. Supongamos que el orden es I_1, I_2.

   - Si se expande para resolver I_1, no se modifican las posibles soluciones a I_2, dado que $A_e(I_1)$ no se halla en el área de interferencia.

- Si se reduce para resolver I_1, como los intervalos son disjuntos, no cambia el área de reducción de I_2, sólo pueden eliminarse ejes o reducirse intervalos temporales (en este último caso, puede duplicarse un nodo, pero como la intersección de intervalos es vacía, sólo uno de los nodos resultantes quedará en el área y por lo tanto $\kappa_r(I_2)$ permanece constante). Tampoco cambia el área de expansión, ya que estos nodos no se encuentran en el área de interferencia. Por lo tanto, dado que las áreas de influencia no varían y por 22 no hay posibilidad de formar nuevos ciclos al expandir, la solución de la segunda inconsistencia no cambia.

Supongamos que el orden es I_2, I_1.

Como ambas inconsistencias son del mismo tipo y no se supuso nada que las distinguiera, vale la demostración anterior.
Por lo tanto interfieren de manera irrelevante.

d. Supongamos que el orden es I_1, I_2.
Reduciendo para resolver I_1

No afecta las posibles soluciones de I_2 puesto que $A_r(I_1) \cap A_i(I_1, I_2) = \emptyset$
Si era posible resolver I_2 por expansión, aún podemos hacerlo (no se forman ciclos) puesto que sólo se redujeron ejes y por lo tanto no se formaron nuevos caminos en el documento.

Expandiendo para resolver I_1

Como $T_1 \prec T_2$ no se modifica el área de reducción de I_2. De hecho, sólo el último nodo del camino de últimos padres puede estar en el área de reducción de I_2 ya que los nodos cuyo lifespan se expande no contienen el intervalo T_2 y se expande sólo hasta cubrir el intervalo $T(I_1)$.

Por lo tanto no afecta las soluciones de I_2.

Supongamos que el orden es I_2, I_1.

Si se expande para resolver I_2 no afecta las soluciones para I_1 puesto que los nodos de $A_e(I_2)$ no se encuentran en el área de interferencia. Además, no hay nuevos posibles ciclos, ya que si fuera así, significa que se creó un camino entre $N_1 y N_2$ al expandir I_2. Este camino tiene que haberse formado en un intervalo $T \subset T_2$ pero al expandir para resolver I_1 sólo se modifica el intervalo $T(I_1)$, si tenemos en cuenta que $T_1 \prec T_2 \Rightarrow T_1 \cap T_2 = \emptyset$ entonces no se pueden tener ciclos porque ambos caminos tienen intervalo distinto.

Si en cambio se reduce, se modifican nodos del área de expansión de I_1, pero como $T_1 \prec T_2$ no se llega a modificar el intervalo de la inconsistencia I_1, puesto que si un nodo en un camino de últimos padres de I_1 inclu-

ye en su lifespan parte del intervalo de I_2 cumpliéndose la relación entre los intervalos, incluye íntegramente al intervalo T_1 y se trata entonces del último nodo en el camino de últimos padres.

Por lo tanto no afecta las soluciones de I_1.

□

Interferencias entre inconsistencias de tipo *ii*

Proposición 24. *Dadas dos inconsistencias de tipo ii en un documento, estas interfieren de manera irrelevante a menos que se trate de dos solapamientos con un eje en común y con intervalos solapados.*

Demostración.

Se tienen dos inconsistencias I_1 e I_2 sobre el mismo nodo n_1.

- Supongamos en primer lugar, que ambas son baches en el lifespan del nodo n_1. Sea

  $lifespan(n_1) = [T_1, T_2] \bigcup [T_3, T_4] \bigcup [T_5, T_6]$

  donde $T_2 < T_3 - 1 \ \wedge \ T_4 < T_5 - 1$.

  Al resolver I_1, se creará un nuevo nodo, n_{1c}. El lifespan de n_1 será $[T_1, T_2]$ y el de n_{1c}, $[T_3, T_4] \bigcup [T_5, T_6]$. Está claro que al resolver I_2 sólo se verá afectado el nuevo nodo. Algo similar sucede al resolver primero I_2, quedando:

  $lifespan(n_i) = [T_1, T_2] \bigcup [T_3, T_4]$ y
  $lifespan(n_i) = [T_5, T_6]$

  siendo también independiente la corrección de la segunda inconsistencia. Está claro que el orden de corrección no afecta el resultado final, siendo entonces irrelevantes.

- Ahora veamos el caso en que ambas son solapamientos de intervalos sin ejes en común. Sean los ejes

  $\epsilon_1(n_i, n_1, T_1), \epsilon_2(n_j, n_1, T_2), \epsilon_3(n_k, n_1, T_3), \epsilon_4(n_l, n_1, T_4)$ donde
  $T_1 \bigcap T_2 \neq \emptyset \ \wedge \ T_3 \bigcap T_4 \neq \emptyset \ \wedge \ T_1 \bigcap T_2 \bigcap T_3 \bigcap T_4 = \phi$ y además
  $\epsilon_1 \neq \epsilon_2 \neq \epsilon_3 \neq \epsilon_4$

Al resolver la primera inconsistencia, se reducirá uno de los dos ejes indistintamente en el intervalo de la intersección. Claramente este hecho no afecta a la segunda inconsistencia, puesto que los ejes candidatos a ser modificados son otros. Cómo no se supuso nada que diferencie ambas inconsistencias, el orden de corrección es indistinto.

- La inconsistencia se produce con sólo 3 ejes, pero los intervalos no son solapados.

  $\epsilon_1(n_i, n_1, T_1), \epsilon_2(n_j, n_1, T_2), \epsilon_3(n_k, n_1, T_3)$
  $T_1 \bigcap T_2 \neq \emptyset \; y \; T_2 \bigcap T_3 \neq \emptyset \; y \; T_3 \bigcap T_1 = \emptyset$

  Supongamos sin pérdida de la generalidad que $T_1 \prec T_2$. Claramente, si para resolver la primer inconsistencia reducimos T_1, esto no afectará la solución de la segunda. Si en cambio reducimos T_2, esto se hará por la izquierda del intervalo (es decir, se correrá el inicio del intervalo hasta $(T_1)_f + 1$. Como $T_1 \cap T_3 = \emptyset$ entonces la segunda inconsistencia no se ve alterada y por ende la solución no varía. En conclusión, ambas inconsistencias interfieren de manera irrelevante.

- Finalmente, veremos que pasa si se combina un bache con una superposición. Claramente estas inconsistencias no pueden ocurrir en el mismo intervalo (no puede haber un bache si existe un eje incidente en el mismo intervalo, en particular, en este caso habría dos). Si se resuelve primero el bache, el solapamiento quedará en uno de los dos nodos resultantes (original y duplicado) y se resolverá de la misma manera que si el bache nunca hubiera existido. Si en cambio se resuelve primero el solapamiento, claramente el bache no se verá total ni parcialmente cubierto, ni el nodo se habrá dividido en ese intervalo (dado que los intervalos son distintos, además de que la duplicación no es solución a un solapamiento), y la división del nodo se hará de la misma manera que si el solapamiento nunca hubiese existido. Por lo tanto son irrelevantes.

□

Interferencias entre inconsistencias de tipo *i* y tipo *ii*

Si bien las inconsistencias de tipo *ii* pueden ser consideradas un mero error sintáctico y por ende ser corregidas en primer lugar sin importar el hecho de que interfieran con otras, veremos algunas propiedades que nos aseguran que la interferencia es irrelevante en el sentido estricto.

Proposición 25. *Una inconsistencia I_1 de tipo i y otra I_2 de tipo ii interfieren de manera irrelevante si:*

a. la inconsistencia I_2 se encuentra en el área de reducción de la inconsistencia I_1 y se cumple alguna de:

1. no es un solapamiento

2. $T_2 \bigcap T_1 = \emptyset$

b. la inconsistencia I_2 es un bache, se encuentra en el área de expansión de la inconsistencia I_1 ($A_i(I_1, I_2) = A_{inf}(I_2)$) y $T_1 \cap T_2 = \emptyset$.

Demostración.

a.
- Caso 1: I_2 es un bache.
  - $(T_1 \cap T_2 \neq \emptyset \wedge T_1 \neq T_2) \vee (T_2)_f + 1 = (T_1)_i$ Los intervalos son solapados no iguales, o consecutivos

    Sea n_2 el nodo inconsistente de I_2. Al duplicar para resolver I_2 obtendríamos dos nodos, n_2 y n_{2c}.

    $(lifespan(n_i))_{new} = [(lifespan(n_i))_i, (T_2)_i - 1]$ y
    $(lifespan(n_i))_{new} = [max((T_2)_f + 1, (T_1)_i), (lifespan(n_i))_f]$

    Si luego reducimos para resolver I_1 tenemos que reducir entonces el nuevo nodo n_{2c} siendo su nuevo lifespan:

    $$[(T_1)_f, (lifespan(n_i))_f]$$

    el cual podría ser un intervalo nulo si:

    $$(T_1)_f = (lifespan(n_i))_f$$

    Si resolvemos en el orden inverso, reduciendo I_1 dejaría al nodo n_2 con un lifespan de:

    $$\underbrace{[(lifespan(n_i))_i, (T_2)_i - 1]}_{a} \cup \underbrace{[(T_1)_f + 1, (lifespan(n_i))_f]}_{b}$$

Al duplicar para resolver I_2 inconsistencia quedan dos nodos entonces, cada uno con el lifespan igual a el miembro **a** unión y el otro con **b**. Podemos observar que el resultado obtenido es el mismo en ambos órdenes.

- $T_1 \cap T_2 = \emptyset \wedge (T_2)_f + 1 < (T_1)_i$ Los intervalos son no solapados y no consecutivos.
  Reduciendo primero para resolver I_1

  $$(lifespan(n_i))_{new} = \underbrace{[(lifespan(n_i))_i, (T_2)_i - 1]}_{a} \cup \underbrace{[(T_2)_f + 1, (T_1)_i - 1]}_{b}$$

  y
  $lifespan(n_i) = [(T_1)_f + 1, (lifespan(n_i))_f]$

  siendo que es necesario duplicar. Luego, se debe duplicar el nodo original para resolver la inconsistencia I_2 dando origen a un nuevo nodo que tendrá como lifespan el intervalo **b** mientras que el nodo original conservará el intervalo **a**.
  Si resolvemos en el orden inverso, la duplicación inicial produce dos nodos con intervalos

  $[(lifespan(n_i))_i, (T_2)_i - 1]$ y
  $[(T_2)_f + 1, (lifespan(n_i))_f]$

  Luego al reducir, se toma el segundo nodo, obteniendo dos nuevos nodos con intervalos:

  $[(T_2)_f + 1, (T_1)_i - 1]$ y
  $[(T_1)_f + 1, (lifespan(n_i))_f]$

  Evidentemente, ambas soluciones son equivalentes.

En ambos casos, si se expande para resolver I_1 no se ve afectada I_2 puesto que $A_e(I_1)$ no se encuentra en el área de interferencia.

- Caso 2: I_2 es un solapamiento.
  - si los intervalos son disjuntos, no importa, porque se reducen intervalos distintos en cada caso (y permite elegir unívocamente el último padre para corregir la otra inconsistencia).
  - solapados, no puede ser por hipótesis.

b. I_2 en el área de expansión de I_1 con intervalos disjuntos.

- I_1, I_2: si se expande no afecta a I_2 ya que no se modifica su intervalo debido a que el intervalo de I_1 se encuentra en un extremo del

lifespan del nodo inconsistente de I_2 (n_2) y por ende no se expande hacia el intervalo del bache. Es decir,

$lifespan(n_i) = [(lifespan(n_i))_i, (T_2)_i - 1] \cup [(T_2)_f + 1, (lifespan(n_i))_f]$
expandiendo
$(lifespan(n_i))_{new} = [(lifespan(n_i))_i, (T_2)_i - 1] \cup [(T_2)_f + 1, (T_1)_f]$

Al duplicar para resolver I_2 se obtienen dos nodos, cada uno con lifespan igual a uno de los intervalos de la unión.

- Si se resuelve primero I_2 se obtienen dos nodos

$lifespan(n_i) = [(lifespan(n_i))_i, (T_2)_f - 1]$
$lifespan(n_i) = [(T_2)_f + 1, (lifespan(n_i))_f]$

Luego, al expandir, se modifica sólo n_{2c} quedando

$lifespan(n_i) = [(T_2)_f + 1, (T_1)_f]$

Como puede observarse, ambos resultados son equivalentes. Si se reduce para resolver I_1, no se altera el nodo inconsistente de I_2 porque el área de reducción de I_1 no se encuentra en el área de interferencia.

□

Interferencias entre inconsistencias de tipo *i* y tipo *iv*

En una sección anterior vimos interferencias entre inconsistencias de tipo *i* y tipo *iv* indistintamente (cualquier combinación). Ahora veremos otra propiedad que se refiere sólo al caso en que tenemos una inconsistencia de tipo *i* y otra de tipo *iv*.

Proposición 26. *Una inconsistencia I_1 de tipo i y otra inconsistencia I_2 de tipo iv interfieren de manera irrelevante si:*
Nodo de I_2 en el área de expansión de I_1, $T_1 \succ T_2$ y $T_{ejedenodo} \succ T_2$

Demostración.

Nodo de I_2 en el área de expansión de I_1 Los intervalos son disjuntos y $T_1 \succ T_2$
Si se resuelve primero la inconsistencia I_1. Sea N_2 el nodo en el área de interferencia. Sean $n_1, ...n_j$ los nodos en el camino de expansión para la inconsistencia

I_1 y $\epsilon_1, ...\epsilon_j$ los ejes respectivos

Expandiendo para resolver la inconsistencia I_1

$(T(\epsilon_i))_{new} = [(T(\epsilon_i))_i, (T(\epsilon_1))_f]$ para cada $i = 1, ...j$

Por lo tanto, para el nodo inconsistente resulta que:

$(lifespan(n_i))_{new} = [(lifespan(n_i))_i, (T_1)_f]$

No hay otros nodos del área de influencia de I_2 en el área de interferencia por lo que sólo interesan los cambios a N_2 El hecho de que $T_1 \succ T_2$ garantiza que la existencia del ciclo no impida calcular el camino de expansión (si el ciclo se encontrara en el camino de expansión, claramente el orden alteraría el resultado, puesto que la expansión de I_1 no es posible sin antes resolver el ciclo).

Si luego se reduce I_2, como $T_2 \prec T_1$

$(lifespan(n_i))_{new} = [(lifespan(n_i))_i, (T_2)_i - 1]$
$(lifespan(n_i))_c = [(lifespan(n_i))_f + 1, (T_1)_f]$

Resolviendo primero I_2 por reducción

$lifespan(n_i) = [(lifespan(n_i))_i, (T_2)_i - 1]$
$lifespan(n_i)_c = [(lifespan(n_i))_f + 1, (lifespan(n_i))_f]$

Al expandir I_1 como $T_2 \prec T_1$

$(lifespan(n_i))_c = [(T_2)_f + 1, (T_1)_f]$

Puede verse que ambos órdenes son equivalentes.
Si se reduce I_1 no pasa nada puesto que el área de reducción de I_1 no se halla en el área de interferencia y por lo tanto no se modifican los caminos de expansión. Los ciclos también tienen varias soluciones, la demostración abarca el caso en que sólo se elimina, sin expandir ejes. Si se utiliza el método de eliminar un solo eje y expandir para cubrir el bache, entonces, si el eje eliminado no es el incidente a N_2 el nodo no cambia en ningún momento a causa de la corrección de la inconsistencia y es fácil ver que sin importar el orden de corrección, el nodo quedará: $(lifespan(n_i))_{new} = [(T_2)_i, (T_1)_f]$

Si en cambio el eje eliminado es el incidente al nodo, entonces el último eje incidente a él con intervalo menor al del ciclo será expandido. Este eje no está en el camino de últimos padres de I_2 y por lo tanto no afecta la solución de la segunda inconsistencia, de la misma manera, se puede ver entonces que el resultado final

será igual al anterior, aunque los ejes incidentes a N_2 hayan cambiado.

Como el orden de corrección no influye en el resultado final, concluimos que la interferencia es irrelevante.

□

Interferencias entre inconsistencias de tipo *ii* y tipo *iv*

Proposición 27. *Una inconsistencia de tipo ii y otra de tipo iv que interfieren de manera irrelevante siempre que la inconsistencia de tipo ii se trate de un bache y no se encuentre en un camino de expansión de la inconsistencia tipo iv*

Demostración.

Caso 1) La inconsistencia de tipo *ii* está en un nodo del ciclo: No influye puesto que el ciclo obviamente se produce en otro intervalo, y el bache nunca se cubre, con lo cual no se alteran los nodos de un camino de expansión
Caso 2) En un camino de reducción: se reduce un intervalo distinto, y la inconsistencia tipo *ii* no produce cambios en los intervalos, por lo tanto no influye el orden de corrección en el resultado final.

□

Proposición 28. *Una inconsistencia de tipo ii y otra de tipo iv que interfieren de manera irrelevante siempre que la inconsistencia de tipo ii se trate de un solapamiento y*

a. Se cumple la proposición 25 para el caso de solapamientos.

b. La inconsistencia de tipo ii está en el ciclo pero los intervalos en que se producen ambas inconsistencias son distintos

6.4. Interferencias relevantes

Hasta aquí hemos analizado los casos en que las inconsistencias se encuentran aisladas o interfieren de manera irrelevante, es decir, se pueden resolver las inconsistencias en cualquier orden sin alterar el resultado final. Para el caso en que el resultado varía según el orden en que resolvemos las inconsistencias, podríamos comparar las posibles soluciones y ver cual produce finalmente la menor cantidad de cambios, teniendo en cuenta que al resolver una podemos estar cambiando las posibilidades de resolver la segunda. En este trabajo sólo daremos una heurística para resolver el caso general.

En principio, optaremos por resolver primero las inconsistencias de tipo *ii* y tipo *iii* que interfieran de forma relevante con otras. Esto es razonable si consideramos que estas inconsistencias son de carácter sintáctico y pueden ser resueltas sin modificar la información presente en el documento. Queda entonces definir como proceder ante la interferencia relevante de inconsistencias tipo *i* y tipo *iv*. Existen varias opciones que se pueden tener en cuenta

- resolver primero la que implique menor cantidad de cambios. La motivación de esta solución estaría en que parecería más adecuado pensar que si son menos los cambios, se está más cerca de la solución real al problema, al menos de una forma golosa de corrección.
- intentar que no interfieran, esto es, resolverlas de manera de que sean irrelevantes dentro de lo posible (por ejemplo, expandiendo ambas o reduciendo ambas). El problema con esta opción es que no favorece las soluciones más naturales en cuanto a la cantidad de cambios netos a realizar.

Entre estas opciones, la primera produce la menor cantidad de cambios en la mayoría de los casos.

6.5. Resumen

En este Capítulo abordamos el problema de tener más de una inconsistencia dentro de un documento XML temporal. Comenzamos definiendo el área de influencia para cada tipo de inconsistencia y presentando algoritmos para calcularlas (Sección 6.1). En la Sección 6.2 Definimos el concepto de *Interferencia entre inconsistencias* y diferenciamos dos categorías: *irrelevantes* y *relevantes*. Presentamos y demostramos propiedades bajo las cuales se puede asegurar que dos inconsistencias interfieren de manera irrelevante (Sección 6.3) y concluimos presentando una heurística para resolver los casos de interferencias relevantes (Sección 6.4).

Capítulo 7

Actualización de documentos TXML

Hasta ahora nos concentramos en el caso en que un documento TXML es preexistente y su consistencia no está garantizada. Analizamos estas inconsistencias, cómo encontrarlas en un documento, y cómo corregirlas. Ahora definiremos un lenguaje de updates, que nos permitirá construir un documento desde cero, sin generar inconsistencias y realizar actualizaciones sobre documentos TXML consistentes garantizando que el resultado será también consistente. Para esto, cada operación de update realizada deberá garantizar que el documento resultante se encontrará en un estado consistente, utilizando con este objetivo los conceptos vistos en los capítulos anteriores.

En esta sección, presentamos una sintaxis para updates en documentos TXML y analizamos la manera de chequear si un update dado conservará la consistencia del documento. Este chequeo será incremental (es decir, se intenta no revalidar todo el documento, sino sólo aquellas partes afectadas por la operación de update).

Definición 28 (Nodo Actual).
Llamaremos nodo actual a todo nodo tal que su lifespan es de la forma $[t, Now]$ *para cualquier instante* t.

En todos los casos, sólo se permitirán updates a nodos actuales que no violen las condiciones de consistencia de la Definición 2.

Podemos clasificar las operaciones de actualización en:

- Actualización de nodos. Aquí incluimos la inserción de un nuevo nodo o la modificación del valor de un nodo, cómo también la inserción de atributos, o su modificación, en el caso de tratarse de un atributo temporal.

- Actualización de ejes. Aquí incluimos el cambio de padre de un nodo, el borrado de un nodo a un instante dado. Se consideran updates de ejes ya que son estos los que se ven modificados durante la operación.

A continuación, propondremos una sintaxis para el lenguaje de updates y analizaremos en más detalle la semántica de cada operación. También expondremos los chequeos de consistencia necesarios para garantizar la consistencia del documento actualizado mediante estas operaciones.

7.1. Actualizaciones de nodos

Definimos las siguientes operaciones de actualización de nodos: inserción de elementos, inserción de atributos (temporales y no temporales), update de elementos (de sus valores de texto) y update de atributos temporales. En la siguientes secciones, veremos en detalle la sintaxis de cada una de ellas, así como los requisitos que se deben cumplir para garantizar la consistencia posterior de un documento.

7.1.1. Inserción de nodos

Permitiremos insertar un nodo con intervalo temporal $[t, Now]$ siempre que esta operación deje al documento en un estado consistente. El instante t puede ser especificado o no. En caso de no ser especificado, tomará el valor por defecto (tiempo actual).

```
for XPATH_SELECTION_EXPRESSION
INSERT NEWNODE
[NAME name]
[VALUE value]
[AT instant]
[POSITION position]
```

Para cada nodo resultante de la expresión XPATH_SELECTION_EXPRESSION de búsqueda, se realizará el insert de un nuevo nodo hijo con etiqueta dada por `NAME` ***name***, con un valor dado por `VALUE` ***value***, obligatorio si se trata de un 'Value Node'. El instante a partir del cual se inserta será el dado por `AT` ***instant***, siendo el valor de la fecha actual si la opción es omitida. También se puede indicar el índice en el cual se desea agregar el nodo indicando un valor para `'POSITION'`, así, si se indica `'POSITION 2'` se insertará el nodo como segundo hijo para cada nodo seleccionado por la expresión XPATH de búsqueda.

Ejemplo 25 (Inserción de un nodo).
Veamos el ejemplo de la Figura 7.1. Si realizamos la siguiente actualización sobre ese documento entonces dará como resultado la Figura 7.2.

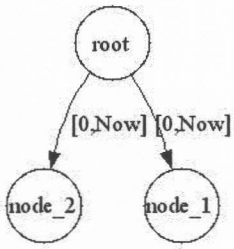


Figura 7.1: Documento original

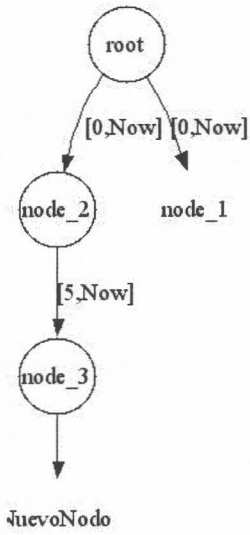


Figura 7.2: Resultado de la inserción del nodo node_3, con valor 'NuevoNodo' al instante 5

```
for //node_2
INSERT NEWNODE
NAME = 'node_3'
VALUE = 'NuevoNodo'
AT = 5
```

Para que la inserción se pueda efectuar, debe dejar el documento en un estado consistente. Se debe corroborar entonces, que el intervalo para el cual se inserta el nodo, esté incluido en el lifespan del nodo padre. Esto es, si revisamos el Ejemplo 25 donde insertamos un nodo en el intervalo $[5, Now]$, entonces para cada nodo n que cumpla la condición de búsqueda, se debe verificar que $[5, Now] \in lifepan(n)$. Las condiciones de consistencia 2 y 3 en la Definición 2 no necesitan verificarse, ya que el nodo es nuevo, y por lo tanto tiene un único eje incidente, con lo que no puede suceder que haya baches en el lifespan o que sea un nodo versionado. Tampoco pueden existir ciclos, debido a que el nuevo nodo no tiene aún ejes salientes.

7.1.2. Inserción de atributos

En la Sección 8.1.1 explicaremos cómo nuestra implementación soporta atributos temporales. Por lo tanto, necesitaremos una sintaxis para actualizar atributos. Siempre permitiremos insertar un atributo, especificando su nombre y valor. Si es temporal, puede indicarse el instante a partir del cual será válido

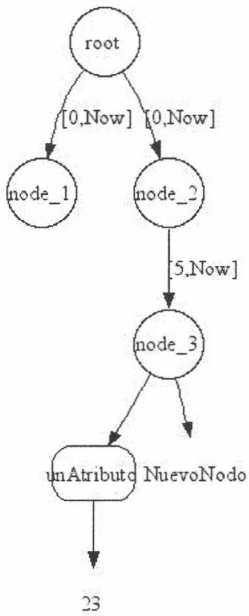


Figura 7.3: Documento de la Figura 7.2 luego de la inserción del atributo unAtributo con valor '23'

incluyendo la opción `AT`. En ese caso, se insertará el atributo con intervalo temporal $[t, Now]$. El valor por defecto de `instant` es el instante actual.

```
for XPATH_SELECTION_EXPRESSION
INSERT NEWATTRIBUTE
NAME attName
VALUE value
[AT instant]
```

Para cada nodo resultante de la expresión XPATH de búsqueda, se realizará el insert de un nuevo atributo con el nombre y valor especificados. Si se trata de un atributo temporal, el instante a partir del cual se inserta será el dado por AT *instant*.

Ejemplo 26 (Inserción de un atributo)**.**

Veamos el ejemplo de la Figura 7.2. Si realizamos la siguiente actualización sobre ese documento entonces dará como resultado la Figura 7.3.

```
for //node_3
INSERT NEWATTRIBUTE
NAME unAtributo
VALUE '23'
```

Para garantizar la consistencia posterior del documento, en términos temporales, sólo deberíamos chequear que el nodo al que pertenece el atributo incluya en su lifespan al intervalo $[instant, Now]$ y que de haber otro atributo con el mismo nombre, este es temporal y su lifespan es [t, Now] con $t < instant$.

7.1.3. Actualización de nodo

Siempre permitiremos modificar el valor de un nodo a partir del instante t, que de no ser especificado, toma el valor por defecto (instante actual). Se debe especificar el o los nodos a actualizar y el valor. Son opcionales el instante a partir del cual toma efecto la modificación, siendo por defecto el instante actual.

```
for XPATH_SELECTION_EXPRESSION
SET VALUE = newValue
[AT instant]
```

Para cada nodo resultante de la expresión XPATH de búsqueda, se realizará el update de su valor, con un valor dado por `VALUE` *`value`*. El instante a partir del cual se realiza la modificación será el dado por `AT` *`instant`*, siendo el valor de la fecha actual si la sentencia es omitida. La actualización se realizará en el marco de un nodo versionado, es decir, no se perderá el valor previo, sino que sólo se modificará su atributo `Time:TO`.

Para el chequeo de consistencia al realizar esta operación, debe verificarse que el instante especificado se encuentre en el lifespan del nodo padre. Además de realizar dicho chequeo, se debe constatar que el último nodo presente en la secuencia sea un nodo actual (su lifespan abarque la fecha actual).

Ejemplo 27 (Update de nodo).

Veamos el ejemplo de la Figura 7.4. Si realizamos la siguiente actualización sobre el documento en a) entonces dará como resultado el documento en b).

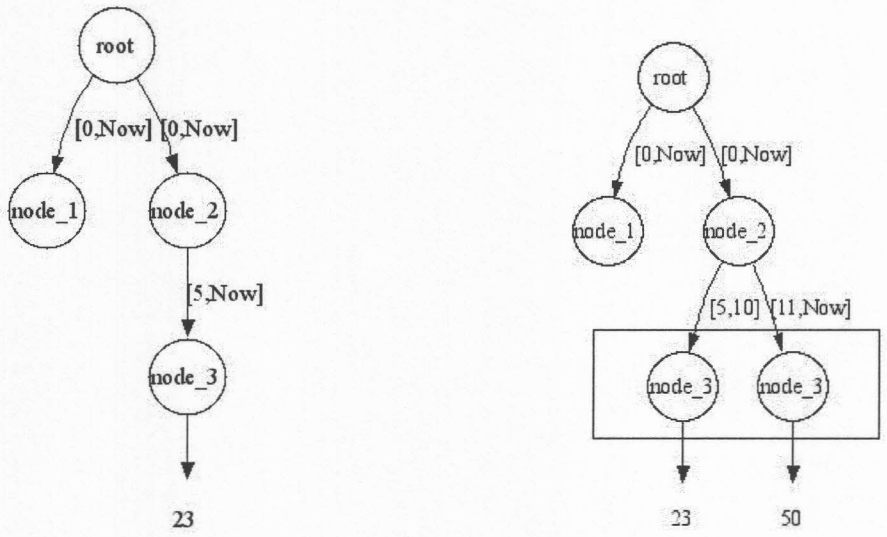
```
for //node_3
SET VALUE = 50
AT 11
```

7.2. Actualizaciones de ejes

7.2.1. Modificación de padre

Puede modificarse también el nodo padre de un nodo, indicando el nuevo padre, y a partir de que instante este cambio se realiza.

```
for XPATH_SELECTION_EXPRESSION
SET PARENT XPATH_SELECTION_EXPRESSION2
[AT instant]
```



a) b)

Figura 7.4: Ejemplo de modificación de nodo. a) Porción de documento antes de modificar el nodo node_3. b) Luego de modificar el nodo node_3 con el valor '50'. El nodo se convierte en un nodo versionado.

El instante especificado por `AT instant` indica el momento a partir del cual toma efecto el cambio y debe pertenecer al intervalo de la etiqueta temporal del último eje incidente a cada uno de los nodos retornados por la expresión de búsqueda especificada en `XPATH_SELECTION_EXPRESSION`. La expresión de búsqueda `XPATH_SELECTION_EXPRESSION2` debe retornar un único nodo que será el nuevo padre de cada uno de los nodos retornados por la primera expresión. El intervalo $[t, Now]$, donde t indica el instante especificado en la opción `AT` o el instante actual si no se incluye esta opción, debe estar incluido en el lifespan del nuevo nodo padre de manera que no viole la condición 1 de consistencia en la Definición 2. Como resultado de esta operación, se agregará un nuevo eje desde el nodo indicado como padre en la segunda expresión de búsqueda, hasta cada uno de los nodos seleccionados por la primera expresión, con etiqueta temporal $[t, Now]$, siendo t el instante actual, o el especificado mediante la opción `AT`.

Ejemplo 28 (Update de padre).

Veamos el ejemplo de la Figura 7.3. Si realizamos la siguiente actualización sobre ese documento entonces dará como resultado la Figura 7.5.

```
for in //node_3
SET PARENT //node_1
AT 10
```

Hemos visto que existen diversas chequeos de consistencia que deben llevarse a cabo para esta operación. Si t es el instante especificado por la opción `AT` (o,

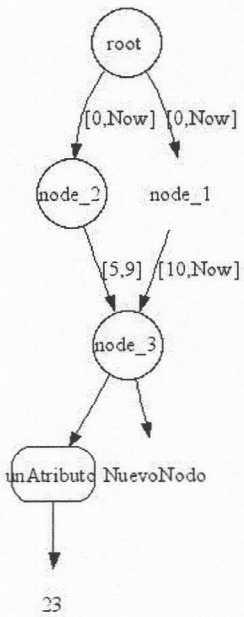


Figura 7.5: Documento de la Figura 7.3 luego de la modificación del padre del nodo node_3

si esta opción es omitida, el instante actual), se debe verificar que

- El lifespan del nuevo padre incluya el intervalo $[t, Now]$.
- Sólo se permitirán modificar nodos actuales (Definición 28) y el instante t debe pertenecer a la etiqueta temporal del último eje incidente cada uno de los nodos cuyo padre cambiará.
- Corroborar que no se formen ciclos. Para esto es necesario chequear que no exista un camino desde cada nodo a modificar y el nuevo padre en el intervalo $[t, Now]$

Con los chequeos anteriores, y dado que el documento era consistente antes de realizar la modificación, garantizamos que se cumplirán las condiciones de consistencia de la Definición 2 luego de realizar la operación.

7.2.2. Borrado de un nodo

Puede borrarse un nodo a partir de un instante t, que puede ser especificado por la opción `INSTANT`, siendo el valor default el instante actual. Se actualizará la etiqueta temporal del último eje incidente poniendo como instante final el especificado (o el default). Si el nodo tiene descendientes en $[t, Now]$, estos deberán ser eliminados a partir del instante t también.

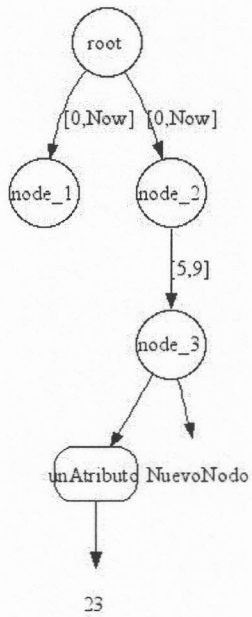


Figura 7.6: Documento de la Figura 7.3 luego del borrado de node_3 al instante 10

```
DELETE XPATH_SELECTION_EXPRESSION
```

Para cada nodo resultante de la expresión XPATH_SELECTION_EXPRESSION de búsqueda, se reducirá su lifespan hasta el instante `t`, eliminando o reduciendo los ejes necesarios. Para que pueda realizarse el borrado, los nodos descendientes que deban ser eliminados deben tener al nodo principal en el camino de últimos padres.

Ejemplo 29 (Borrado de un nodo).
Veamos el ejemplo de la Figura 7.3. Si realizamos la siguiente actualización sobre ese documento entonces dará como resultado la Figura 7.6.

```
DELETE //node_3
```

Ejemplo 30 (Borrado recursivo de un nodo).
Veamos el ejemplo de la Figura 7.5. Suponiendo que el instante actual es $t = 11$, si realizamos la siguiente actualización sobre ese documento entonces dará como resultado la Figura 7.7. Se puede ver que fue necesario modificar tanto el eje root → node_2 como el eje node_2 → node_3.

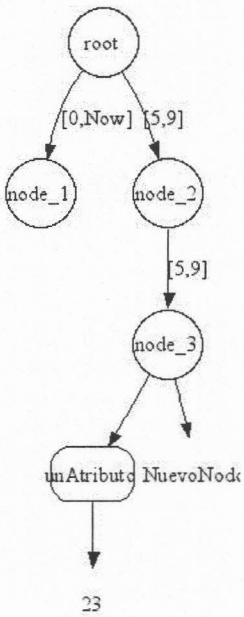


Figura 7.7: Documento de la Figura 7.5 luego del borrado de node_2 al instante 10

```
DELETE //node_2
```

Aquí no debe realizarse chequeo alguno, lo único que puede fallar, es que el nodo en cuestión no incluya en su lifespan al instaste especificado, en cuyo caso la operación no podrá realizarse. Habrá que realizar el borrado recursivamente, si el nodo objetivo tiene ejes salientes en el intervalo $[t, Now]$, siendo t el instante en el cual se realiza la operación de update. Un ejemplo de esto puede apreciarse en el Ejemplo 30. Se debe tener especial cuidado al momento de borrar recursivamente, puesto que si el nodo objetivo no se encuentra dentro del camino de últimos padres de algún nodo descendiente, este deberá duplicarse ya que se generará un bache en su lifespan. Por ejemplo, si en el Ejemplo 30 hubiésemos borrado en el instante

Los chequeos que deben realizarse son entonces (siendo t el instante en que se realiza la operación):

- Ver que efectivamente el intervalo $[t, Now]$ está incluido en el nodo objetivo y
- Verificar que el nodo objetivo esté en el camino de últimos padres para todo nodo descendiente en el intervalo $[t, Now]$.

7.3. Resumen

En este Capítulo definimos la sintaxis para un lenguaje de updates sobre documentos XML temporales. Definimos operaciones de update sobre nodos (inserción de nodos y atributos, update de nodos) en la Sección 7.1 y sobre ejes (modificación del padre de un nodo y borrado de un nodo) en la Sección 7.2. Para cada una de las operaciones definidas, analizamos los chequeos de consistencia necesarios para garantizar la consistencia del documento resultante luego de aplicar la operación, tomando como premisa que el documento es inicialmente consistente.

Capítulo 8

Implementación y experimentación

Durante este trabajo se realizaron implementaciones tanto de los algoritmos de chequeo de consistencia (Algoritmos 2, 3 y 4) como del lenguaje de updates cuya sintaxis fue especificada en el Capítulo anterior. En esta parte del documento mostraremos la implementación realizada de los algoritmos de chequeo de consistencia. Comenzaremos comparando distintas estructuras de datos para la implementación de estos algoritmos, analizando la eficiencia de los mismos utilizando cada una de las estructuras. Finalmente, estudiaremos la eficiencia de la implementación realizada en función de variables como el tamaño de los documentos y la proporción de punteros presentes en estos. Para estas pruebas, desarrollamos un generador automático de documentos TXML.
Comenzaremos presentando algunas mejoras al modelo TXML descripto en la Sección 3.1, en el cual se basa el trabajo. Las mejoras que realizamos fueron las siguientes:

- Introducción de atributos temporales.
- Utilización de valores default para ahorrar espacio.

A continuación desarrollamos cada una de ellas. Luego presentaremos la implementación realizada de los algoritmos de chequeo de consistencia y analizaremos su performance.

8.1. Mejoras al modelo TXML

8.1.1. Atributos temporales

El modelo original presentado en la Sección 3.1 no soporta atributos temporales. El valor de un atributo no cambia con el tiempo, y si lo hace, se pierde el valor anterior. Dado que XML da la posibilidad de diferenciar entre atributos

y elementos, extendemos aquí el modelo de modo de introducir el aspecto temporal a los atributos. Para esto, definiremos una sintaxis para especificar dicho aspecto y también para realizar consultas mediante TXPATH.

Sintaxis de atributos temporales

Los atributos temporales se representarán como elementos dentro de un elemento distinguido, utilizando para ello un nuevo elemento distinguido <ATTRIBUTES>.

Ejemplo 31 (Sintaxis de atributos temporales).
Supongamos el elemento <PERSONA> *, que puede tener como atributos el apellido y el nombre de una persona. Si bien estos atributos podrían considerarse inmutables, es cierto que también el valor del atributo apellido puede cambiar en determinadas situaciones, por ejemplo, con el casamiento. Entonces, una persona de nombre 'Juana Perez' puede pasar a llamarse 'Juana Perez de Gomez' en el instante* t_1.
Esto sería descripto entonces de la siguiente manera. Inicialmente

```
...
<PERSONA nombre="Juana">
  <ATTRIBUTES>
    <APELLIDO Time:FROM="0" Time:TO="Now">Perez</APELLIDO>
  </ATTRIBUTES>
</PERSONA>
...
```

Luego en el instante t_1

```
...
<PERSONA nombre="Juana">
  <ATTRIBUTES>
    <APELLIDO Time:FROM="0" Time:TO="t1-1">Perez</APELLIDO>
    <APELLIDO Time:FROM="t1" Time:TO="Now">Perez de Gomez</APELLIDO>
  </ATTRIBUTES>
</PERSONA>
...
```

Consulta de atributos temporales mediante TXPATH

Si queremos consultar estos atributos, podríamos realizar la consulta utilizando el hecho de que hay un elemento <ATTRIBUTES> que contiene los atributos temporales y que los mismos son a su vez, elementos. Por ejemplo, la consulta para recuperar el apellido de Juana en el instante t1 sería:

```
//PERSONA[@nombre="Juana"]/ATTRIBUTES[name="APELLIDO" and @Time:FROM < t1]
```

Claro está que esta sintaxis no es transparente y es dependiente de la implementación. Una mejora es entonces brindar otra sintaxis e interpretarla, de manera de trabajar con el concepto de atributo temporal independientemente de su implementación. Adoptamos entonces la misma notación para referirnos a atributos temporales que aquellos no temporales, agregando la posibilidad de referirnos a la dimensión temporal de los mismos mediante la notación de punto.

Ejemplo 32 (Consulta de atributos temporales).
Para consultar el apellido de Juana en un instante anterior a t_1, utilizaremos la siguiente expresión:

```
//PERSONA[@nombre="Juana" @apellido.Time:FROM < t1]
```

Utilizamos entonces `@apellido` *para referirnos al atributo al igual que para los atributos no temporales y agregamos* `@apellido.Time:FROM` *y* `@apellido.Time:TO` *para referirnos a los extremos del intervalo temporal del atributo.*

Consistencia y Update de atributos temporales

Es evidente que al agregar la noción temporal a los atributos, también debemos comprobar que los atributos de un documento sean consistentes con respecto a ésta. Se debe garantizar que no haya dos atributos con el mismo nombre cuyos intervalos se solapen, para un nodo dado.
Si bien los únicos casos analizados en este trabajo son el de inserción y modificación de un atributo temporal, el caso de eliminación se trata de manera análoga a la eliminación de nodos.

8.1.2. Valores Default

Si queremos mantener todas las versiones de un documento dentro del mismo, surge el problema de consultarlo en memoria y almacenarlo de manera eficiente. Para reducir el tamaño de los documentos proponemos utilizar valores default para los atributos temporales, de manera que no sea necesario guardar toda la información, sino que pueda deducirse del contexto.

Valores default para nodos no versionados

En este caso podemos observar dos situaciones. La primera, el nodo raíz. Este tiene por definición, un lifespan de $[0, Now]$ dado que no tiene ejes incidentes que le aporten datos temporales.
Para los demás nodos no versionados entonces, podemos asumir que si un eje incidente no posee etiqueta temporal, entonces la misma tiene un intervalo temporal que abarca todo el lifespan del nodo padre, si este es único.

Ejemplo 33 (Valores default para nodos no versionados).
La Figura 33 se muestra el mismo fragmento de un documento. En el primer caso utilizando valores default, y a continuación con todos los atributos explicitados.

Valores default para nodos versionados

Los nodos versionados tienen la propiedad que sus elementos deben tener un solo eje incidente y que las etiquetas temporales de los mismos deben ser consecutivas. Por lo tanto, bastaría con definir sólo uno de los extremos de los

```
<EMPRESA>
  <DEPARTAMENTO ID='1' nombre='Ventas'>
    <EMPLEADO ID='2' nombre='Juan Perez'/>
  </DEPARTAMENTO>
  <DEPARTAMENTO ID='3' nombre='Finanzas'>
    <EMPLEADO ID='4' nombre='María Gimenez' Time:FROM='5' />
    <EMPLEADO ID='5' nombre='Julián Suarez' Time:TO='10' />
  </DEPARTAMENTO>
</EMPRESA>

<EMPRESA Time:FROM='0' Time:TO='NOW'>
  <DEPARTAMENTO ID='1' nombre='Ventas' Time:FROM='0' Time:TO='NOW' >
    <EMPLEADO ID='2' nombre='Juan Perez' Time:FROM='0' Time:TO='NOW'/>
  </DEPARTAMENTO>
  <DEPARTAMENTO ID='3' nombre='Finanzas' Time:FROM='0' Time:TO='NOW' >
    <EMPLEADO ID='4' nombre='María Gimenez' Time:FROM='5' Time:TO='NOW' />
    <EMPLEADO ID='5' nombre='Julián Suarez' Time:FROM='0' Time:TO='10' />
  </DEPARTAMENTO>
</EMPRESA>
```

Figura 8.1: Ejemplo de la utilización de valores default para atributos temporales.

intervalos de los ejes internos para tener la información completa. Además, el instante inicial y final del primer y último nodo respectivamente, pueden tomar como valor default, los extremos del lifespan del padre. Entonces tenemos:
Sea un nodo versionado $(n, listanodos)$ donde $listanodos = [n_1, ..., n_k]$
Sea $T_i = T(n_i)$ y $T = T(n)$ entonces

$$
\begin{aligned}
(T_1)_i &= (T)_i && \\
(T_i)_i &= (T_{i-1})_f + 1 && si\ i : 2..k \\
(T_i)_f &= (T_{i+1})_i - 1 && si\ i : 1..k-1 \\
(T_k)_f &= (T)_f &&
\end{aligned}
$$

Ejemplo 34 (Valores default para nodos versionados).
La Figura 34 se muestra el mismo fragmento de un documento. En el primer caso utilizando valores default tanto para nodos no versionados como para los nodos versionados, y a continuación con todos los atributos explicitados.

8.1.3. Implementación de valores default

Aplicando valores default intentamos ahorrar espacio de almacenamiento para los documentos TXML. A continuación analizaremos el impacto de esta mejora.

Objetivo

El objetivo de este experimento es comprobar que se obtiene una mejora sustancial en el tamaño de los documentos utilizando valores default para los atributos `'Time:FROM'` y `'Time:TO'`.

```
<EMPRESA>
  <DEPARTAMENTO ID='1' nombre='Ventas'>
    <EMPLEADO ID='2' nombre='Juan Perez'>
     <SEQUENCE ID='6'>
     <SALARIO ID='7' >2.500</SALARIO>
     </SEQUENCE>
    </EMPLEADO>
  </DEPARTAMENTO>
  <DEPARTAMENTO ID='3' nombre='Finanzas'>
    <EMPLEADO ID='4' nombre='María Gimenez' Time:FROM='5' >
      <SEQUENCE ID='8'>
     <SALARIO ID='9' Time:TO='10' >2.500</SALARIO>
     <SALARIO ID='10' >2.800</SALARIO>
     </SEQUENCE>
    </EMPLEADO>
    <EMPLEADO ID='5' nombre='Julián Suarez' Time:TO='10' >
     <SEQUENCE ID='11'>
     <SALARIO ID='12' >2.500</SALARIO>
     </SEQUENCE>
    </EMPLEADO>
  </DEPARTAMENTO>
</EMPRESA>

<EMPRESA Time:FROM='0' Time:TO='NOW'>
  <DEPARTAMENTO ID='1' nombre='Ventas' Time:FROM='0' Time:TO='NOW' >
    <EMPLEADO ID='2' nombre='Juan Perez' Time:FROM='0' Time:TO='NOW' >
      <SEQUENCE ID='6' Time:FROM='0' Time:TO='NOW' >
     <SALARIO ID='7' Time:FROM='0' Time:TO='NOW' >2.500</SALARIO>
     </SEQUENCE>
    </EMPLEADO>
  </DEPARTAMENTO>
  <DEPARTAMENTO ID='3' nombre='Finanzas' Time:FROM='0' Time:TO='NOW' >
    <EMPLEADO ID='4' nombre='María Gimenez' Time:FROM='5' Time:TO='NOW' >
      <SEQUENCE ID='8' Time:FROM='5' Time:TO='NOW'>
     <SALARIO ID='9' Time:FROM='5' Time:TO='10'>2.500</SALARIO>
     <SALARIO ID='10' Time:FROM='11' Time:TO='NOW'>2.800</SALARIO>
     </SEQUENCE>
    </EMPLEADO>
    <EMPLEADO ID='5' nombre='Julián Suarez' Time:FROM='0' Time:TO='10' >
     <SEQUENCE ID='11' Time:FROM='0' Time:TO='10'>
     <SALARIO ID='12' Time:FROM='0' Time:TO='10 >2.500</SALARIO>
     </SEQUENCE>
    </EMPLEADO>
  </DEPARTAMENTO>
</EMPRESA>
```

Figura 8.2: Ejemplo de la utilización de valores default para atributos temporales.

Procedimiento

Para realizar este experimento se utilizaron documentos de distinto tamaño. Estos documentos se obtuvieron a partir de un documento pequeño armado a mano, copiándolo reiteradas veces a continuación de sí mismo hasta lograr el tamaño deseado. El documento original fue producido sin valores default, y luego se le aplicó un algoritmo que extrajo los valores redundantes según las secciones anteriores.

Resultados

A continuación se presentan los resultados de aplicar valores default a cada uno de los documentos (Tabla 8.1). La tabla presentada muestra el tamaño original de cada archivo, junto al tamaño logrado luego de eliminar la información temporal redundante.

Tamaño sin Defaults	Tamaño con Defaults
272KB	169KB
519KB	415KB
5.127KB	4.087
6.538KB	4.389KB

Cuadro 8.1: Tamaño en KB de los archivos con y sin valores default.

Análisis de resultados

Según se puede observar en la Tabla 8.1, la reducción en tamaño obtenida utilizando valores default es significativa. El impacto de utilizar esta mejora es mayor a medida que trabajamos con archivos más grandes, variando desde unos pocos KB para archivos de alrededor de 300KB hasta valores de uno o varios MB para archivos más de 5MB. Resulta por lo tanto una mejora simple y considerable al modelo propuesto.

8.2. Algoritmos de chequeo de consistencia

En esta sección expondremos la implementación de los algoritmos de chequeo de consistencia 3 y 4. Discutiremos primero las distintas estructuras de datos que se consideraron. Una vez seleccionada una estructura, evaluaremos distintas alternativas para crearla analizando luego la eficiencia de los algoritmos de chequeo de consistencia implementados.

8.2.1. Estructura de Datos

Se analizaron dos estructuras de datos alternativas como soporte para los algoritmos de chequeo de consistencia. La primera considerada fue DOM [1] (Document Object Model), dado que es una estructura estándar y existen numerosas implementaciones para crear el documento a partir del archivo XML y navegarlo. Sin embargo, esta estructura resulta ser muy poco eficiente en el uso de memoria, ya que guarda gran cantidad de información totalmente innecesaria para el problema de chequeo de consistencia y además resulta poco práctica

[1] DOM es una interfaz independiente de la plataforma o el lenguaje, que permite acceder y modificar dinámicamente el contenido, estructura y estilo de un documento.
http : //www.w3.org/DOM/

para la detección de ciclos, puesto que la navegación dentro de la estructura es poco flexible. Se decidió analizar entonces una segunda opción, parseando el documento XML utilizando un parser SAX[2](Simple API for XML) para generar una estructura alternativa a la brindada por DOM que se acerca más a los requerimientos del problema particular. La estructura propuesta es un grafo en el cual los nodos puntero (nodos con el atributo `Time:IN`) no son incluidos, y se incluye en su lugar un eje desde el nodo padre del puntero hacia el nodo apuntado (ver Figura 8.3). Esto disminuye la complejidad del problema.

Ejemplo 35. *Estructura de datos*

Veremos como ejemplo para este modelo, el diagrama de objetos correspondiente al documento de la Figura 8.5.

Por simplicidad, se omitió representar la instancia de la clase `Graph` *correspondiente. Esta colaboraría con todos las instancias de* `Node` *existentes. Están representadas las instancias de* `Node` *y* `Edge` *así como las relaciones entre ellas. La instancia correspondiente al empleado con* `IN`*3 no se encuentra representada como un nodo, puesto que en este modelo, los nodos puntero se representan como un eje más. El eje adicional es en este caso el eje* `Eje1`*, con origen en la empresa con* `ID` *4 y destino en el empleado con* `ID` *3. Como puede observarse, todas las instancias de* `Node` *conocen a sus ejes incidentes y salientes, y las instancias de* `Edge` *conocen a sus nodos origen y destino. Además, tal como se especifica en la Figura 8.4, estos objetos tienen otras propiedades. En principio cada nodo tiene un edgeInterval que denota a la unión de todos los intervalos temporales de los ejes incidentes al nodo. También tiene un* `mcp` *que contiene el intervalo resultante de la unión de todos los intervalos de los* $mcp_{desdelaraiz}(n)$ *al nodo. Si el documento es consistente, edgeInterval será igual a mcp para cada nodo del documento, y ambos representarán al lifespan. Para realizar los chequeos de consistencia se utilizan algunas propiedades no expuestas, como* `labels`*, en donde se van guardando los intervalos de los caminos que llevan al nodo. También hay algunas propiedades específicas para realizar los chequeos, como* `visited`*, que indica si se ha pasado ya por el nodo o no. La propiedad* `versioned` *indica si se trata de un nodo versionado, para poder realizar los chequeos correspondientes.*

Para armar esta estructura utilizamos como dijimos anteriormente un parser SAX. Este parser detonará un evento por cada principio y fin de un elemento. De esta manera podremos ir creando los nodos y los ejes necesarios para armar el grafo. Sin embargo, la estructura presentada no posee punteros, sino que se permite a un nodo tener varios padres, por lo tanto no deberán crearse nodos para los punteros presentes en un documento sino solamente un eje entre el nodo padre del puntero en el documento XML y el nodo apuntado según el atributo 'Time:IN'. Aquí surge entonces un problema, si parseamos un puntero que apunta a un nodo aún no leído y por lo tanto no creado (esto pasa siempre

[2]SAX es una API orientada a eventos, la cual reporta eventos de parseo directamente a una aplicación a través de callbacks en lugar de crear un árbol interno representando al documento. Permite construir eficientemente estructuras de datos propias sin tener que primero almacenar todo el documento en memoria. *http : //www.saxproject.org*

```
class Graph{
  Vector nodos;
  Node raiz;
}

class Node{
  Vector in_edges;
  Vector out_edges;
  TempElement edgeInterval;
  TempElement mcp;
  String id;
  String name;
  Vector labels;
  Vector mcps;
  boolean versioned = false;
  boolean visited = false;
}

class Edge{
  TempElement label;
  Node node_in;
  Node node_out;
  boolean visited;
}
```

Figura 8.3: Estructura de datos utilizada para la implementación de los algoritmos de chequeo de consistencia

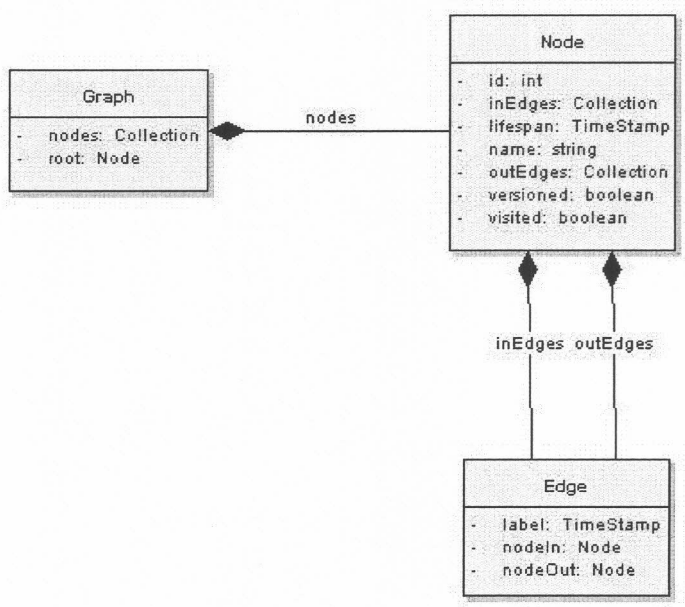


Figura 8.4: Diagrama de clases correspondiente a las estructuras 8.3.

```
<Empresa ID='1' Time:FROM='0' Time:TO='Now'>
  <Empleado ID='2' Time:FROM='0' Time:TO='Now'/>
  <Empleado ID='3' Time:FROM='0' Time:TO='5'/>
</Empresa>
<Empresa ID='4' Time:FROM='0' Time:TO='Now'>
  <Empleado ID='5' Time:IN='3' Time:FROM='0' Time:TO='Now'/>
</Empresa>
```

Figura 8.5: Porción de documento utilizado para ejemplificar el modelo propuesto.

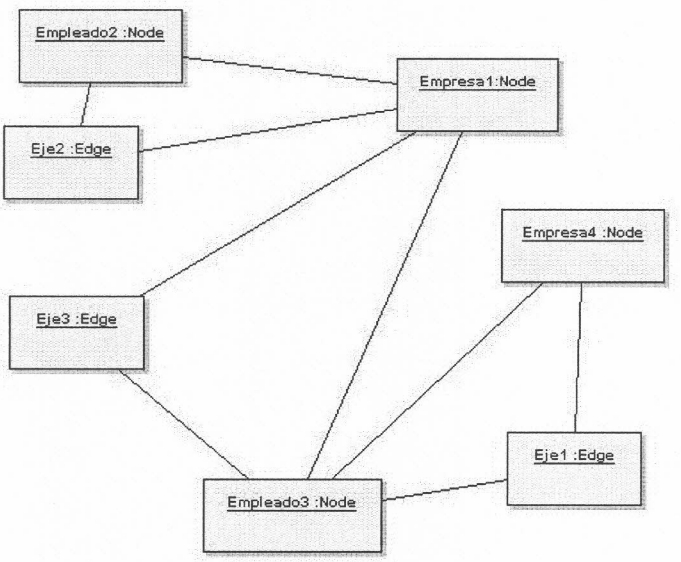


Figura 8.6: Diagrama de objetos representando el documento de la Figura 8.5

que un puntero se ubique físicamente antes del nodo apuntado en el archivo) no podremos crear el eje. Debemos entonces contemplar este caso y definir alguna manera de acordarnos de que ese puntero fue parseado para poder crear el eje correspondiente una vez que el nodo apuntado sea leído. Existen varias posibles soluciones:

1. Realizar tres pasadas con el parser: Esta opción consiste en realizar tres pasadas por el documento. La primera para identificar que nodos son apuntados por otros, la segunda para crear dichos nodos y la tercera para levantar el resto del documento, pudiendo entonces crear los ejes correspondientes a los punteros sin inconvenientes.

2. Realizar dos pasadas con el parser: Es una optimización sobre la primera, en la cual en la primer pasada se identifican los punteros y se irán creando aquellos nodos que al ser leídos ya se haya identificado un puntero al mismo (o sea, los nodos que se encuentran físicamente después de un puntero al

mismo en el documento). En la segunda pasada entonces, se arma el grafo, con la seguridad de que todo nodo que tenga un puntero al mismo, o bien habrá sido creado en la primer pasada, o bien se encontrará físicamente antes que cualquiera de sus punteros y no existirá problema alguno para crear los ejes correspondientes.

3. Efectuar una única pasada: En esta opción se irán guardando en una estructura de hash los nodos creados de manera de hacerlos accesibles a partir de su atributo `'ID'`. Si leemos un puntero a un nodo aún no creado, incluiremos en la table de hash una entrada con el valor de `'Time:IN'` como clave y crearemos un nodo guardando una referencia al mismo como valor en esta entrada. Si leemos un nodo común, la entrada tendrá como clave el valor del atributo `'ID'` del nodo y como valor el nodo creado. De esta manera, si leemos un puntero para el cual el nodo apuntado ya fue creado, lo sabremos buscando por el atributo `'Time:IN'` en el hash y podremos crear el eje fácilmente. Si parseamos un nodo para el cual ya habíamos leído un puntero, también lo sabremos fácilmente buscando por el atributo `'ID'` en el hash, obteniendo una referencia al puntero y pudiendo de esta manera reemplazar en el grafo el nodo creado anteriormente a partir del puntero por el nuevo nodo creado a partir del nodo original (los ejes entrantes y salientes del nodo puntero son modificados para incluir al nuevo nodo en el extremo respectivo, y son agregados al nuevo nodo en la lista de ejes). Por último, si leemos un segundo puntero para un nodo aún no creado, utilizaremos el nodo creado a partir del primer puntero en lugar de crear uno nuevo, facilitando la tarea de reemplazar las referencias más adelante.

4. Solución híbrida: se realizan dos pasadas sobre el documento. Es una mezcla entre las dos últimas opciones. En la primer pasada se identifican los nodos que serán apuntados (mirando cada valor de un atributo `'Time:IN'`). En la segunda, se utilizará la técnica de la estructura de hash, pero sólo se guardarán en éste los nodos o punteros para los IDs de nodos que efectivamente tengan un puntero y para los cuales será necesario efectuar un reemplazo en algún momento.

Entre estas opciones, se eligió utilizar la de una pasada, puesto que si bien la última opción es más eficiente en el uso de memoria, para documentos grandes podría ser mas costoso parsear dos veces que guardar una tabla de hash con todos los nodos, dado que el costo en memoria de guardar los IDs y las referencias a los nodos es bajo.

Una vez armada la estructura de datos con la cuál validaremos los documentos, podemos proceder a ejecutar los algoritmos de validación implementados. Aquí podemos considerar que hay situaciones fácilmente identificables bajo las cuales no es necesario realizar todas las validaciones. En particular podemos, bajo ciertas condiciones, no efectuar el chequeo de ciclos. Por ejemplo, si un documento no posee punteros, no es necesario verificar que no existan ciclos, y podemos evitarnos recorrer el documento en su búsqueda. En este trabajo

preferimos implementar los algoritmos tal cual fueron descriptos en los capítulos anteriores. Sin embargo, es posible realizar todas las validaciones en un sólo recorrido por el documento. Bajo estas circunstancias, las mejoras producidas por no ejecutar el algoritmo para verificar la presencia de ciclos no serían tales, puesto que se puede implementar de manera que al no existir punteros se recorra cada nodo una única vez.
La ejecución del algoritmo de chequeo de ciclos puede ser evitada bajo ciertas condiciones, a saber:

1. No existen punteros. Esta condición implica directamente que no pueden existir ciclos, puesto que cada nodo puede tener únicamente un padre, y estamos entonces en presencia de un árbol.

2. Todo puntero apunta a un hijo del nodo que lo contiene (es decir, a un nodo hermano). De esta manera no pueden existir ciclos, puesto que no hay un camino de ninguna longitud desde el hijo de un nodo hacia su padre y no permitimos la existencia de un puntero a otro nodo puntero.

3. Todo puntero apunta a un nodo que no tiene punteros como descendientes. Así no existirá un camino desde el nodo apuntado al nodo que contiene al puntero, y por lo tanto, no habrá ciclos.

Ciertamente buscamos condiciones fáciles de detectar, de manera que el evitar la ejecución del algoritmo resulte efectivamente en una mejora de eficiencia. Asegurando cualquiera de las anteriores, es posible sólo verificar las condiciones de consistencia 1, 2 y 3 de la Definición 2.
Definimos entonces dos estrategias de validación.

- Completa (algoritmo descripto anteriormente)

- Estrategia sin punteros (No es necesario el chequeo de ciclos, sólo se recorre el grafo verificando las condiciones sobre el lifespan, etiquetas de ejes salientes y nodos versionados)

A continuación presentamos la comparación entre los tiempos necesarios para validar documentos utilizando DOM y utilizando la estructura propuesta en este trabajo.

8.2.2. Comparación entre estructuras de datos

Objetivo

El objetivo de este experimento es comparar la eficiencia de los algoritmos de validación utilizando las estructuras de datos discutidas.

Procedimiento

Para realizar este experimento se construyeron archivos de distintos tamaños y características y se midió el tiempo y la memoria utilizada por cada una de las

```
<?xml version="1.0" encoding="UTF-8"?>
<!--es incorrecto el nodo p1-->
<Time:ROOT ID="1" xmlns:Time="http://www.cs.toronto.edu/db/time">
<franchise ID="f1"  Time:FROM="0" Time:TO="Now" >
  <team ID="t1" Time:FROM = "0" Time:TO="Now">
    <player ID="p1" Time:FROM = "1995/01/01" Time:TO="1997/12/31">
      <SEQUENCE ID="s1" Time:FROM="1999/01/01" Time:TO="2004/12/31">
        <goals ID="g1" Time:FROM="1995/01/01" Time:TO="1995/12/31"/>
        <goals ID="g2" Time:FROM="1996/01/01" Time:TO="1996/12/3 1"/>
       </SEQUENCE>
    </player>
    <player ID="p4" Time:FROM = "2001/01/01" Time:TO="Now"/>
    <player ID="p2" Time:FROM = "1999/01/01" Time:TO="Now" >
      <SEQUENCE ID="s2" Time:FROM="1999/01/01" Time:TO="2004/12/31">
        <goals ID="g3" Time:FROM="1999/01/01" Time:TO="2001/12/31"/>
        <goals ID="g5" Time:FROM="2002/01/01" Time:TO="2002/12/31"/>
        <goals ID="g6" Time:FROM="2003/01/01" Time:TO="2003/12/31"/>
        <goals ID="g7" Time:FROM="2004/01/01" Time:TO="2004/12/31"/>
      </SEQUENCE>
    </player>
  </team>
  <team ID="t2" Time:FROM = "0" Time:TO="Now">
    <player ID="p5" Time:FROM = "1998/01/01" Time:TO="2000/12/31"/>
    <player ID="p3" Time:FROM = "1998/01/01" Time:TO="Now" >
      <goals ID="g4" Time:FROM="1998/01/01" Time:TO="2001/12/31"/>
    </player>
  </team>
</franchise>
</Time:ROOT>
```

Figura 8.7: Ejemplo de documento utilizado para generar archivos de distinto tamaño concatenando.

estructuras durante la ejecución de los algoritmos de validación. Los archivos se construyeron concatenando un archivo consigo mismo (por ejemplo, el archivo de la Figura 8.7) tantas veces cómo fuera necesario para obtener los tamaños deseados.
Se utilizaron dos tipos de archivos,

- Sin punteros : no contienen nodos con atributo `'Time:IN'`
- Con punteros : contienen aproximadamente un 10 por ciento de nodos con atributos `'Time:IN'`

Se comparó la memoria utilizada por cada una de las estructuras así como el tiempo de ejecución de los algoritmos. Se tuvo en cuenta tanto el tamaño de los archivos, como la presencia de punteros, lo que, según discutimos antes, resultaría en una menor eficiencia en los algoritmos implementados utilizando el modelo DOM.

Se utilizó la versión 1.4 de JVM con memoria máxima de 256 MB.

Resultados

La siguiente tabla muestra los tiempos necesarios para realizar la validación con ambas estructuras. En ella nos referimos con OOM a una excepción de falta de memoria, por las siglas del inglés *Out of memory.*

*con memoria máxima en la JVM de 350MB

Tipo archivo	Tamaño	DOM	SAX
Sin punteros	5MB	9	3
Con Punteros	5MB	7	3
Sin punteros	15MB	29	9
Con punteros	15MB	140	8
Sin punteros	25MB	OOM	14
Con punteros	25MB	OOM	14

Cuadro 8.2: Tiempos en segundos de ejecución para el algoritmo de validación

Tipo archivo	Tamaño	DOM	SAX
Sin punteros	5MB	28.4	23
Con Punteros	5MB	29	23
Sin punteros	15MB	86	69.7
Con punteros	15MB	87,7	69
Sin punteros	25MB	143.2	115.8
Con punteros	25MB	147	116

Cuadro 8.3: Memoria requerida por el documento en las distintas representaciones

Análisis de resultados

Se puede ver que, cómo adelantamos, DOM tiene una gran limitación en el tamaño de archivos que se pueden procesar. Los archivos de 25MB no pudieron ser validados con 256MB de memoria para la JVM.
Los archivos de 15MB pudieron ser validados mucho mas eficientemente aumentando la memoria máxima de la JVM, con 300MB el archivo sin punteros fue procesado en 22 segundos mientras el archivo con punteros tardó 27 segundos con 350MB (con memoria menor seguía ocurriendo la OOM). Mientras que los archivos de 25MB pudieron ser procesados aumentando la memoria máxima de la máquina virtual a 400MB, tardando el archivo sin punteros 780 segundos, y el con punteros el tiempo fue mayor a 50 minutos. En ambos casos la memoria física disponible no era suficiente para mantener las estructuras en memoria, y he allí los tiempos logrados.
La memoria utilizada al validar utilizando DOM, no proviene en su mayoría de la estructura DOM en sí, sino del hecho de que esta no representa un documento TXML, y por lo tanto es necesario crear estructuras adicionales para poder procesar los documentos, cómo por ejemplo, listas de punteros por nodo para localizar rápidamente los padres de un nodo dado, sin tener que recorrer cada vez todo el árbol DOM.
La implementación con SAX resulta más eficiente tanto en el uso de memoria como en el tiempo de procesamiento, para todos los tamaños de archivo, tanto con punteros como sin ellos. Será por lo tanto la opción que utilizaremos para realizar los demás experimentos expuestos en este documento.

Tipo archivo	Tamaño	DOM	SAX
Sin punteros	5MB	51.3	41.6
Con Punteros	5MB	52.6	41.6
Sin punteros	15MB	154.7	125.5
Con punteros	15MB	157.8	124.4
Sin punteros	25MB	254	208
Con punteros	25MB	333*	208

Cuadro 8.4: Memoria máxima requerida para armar la estructura de datos en memoria, en MB.

Tipo archivo	Tamaño	DOM	SAX
Sin punteros	5MB	120	35
Con Punteros	5MB	145	35
Sin punteros	15MB	260	100
Con punteros	15MB	260	100
Sin punteros	25MB	OOM	182
Con punteros	25MB	OOM	180

Cuadro 8.5: Memoria máxima requerida por el algoritmo de validación para cada estructura de datos, en MB

8.3. Chequeo de consistencia de documentos

Según los resultados del experimento expuesto en la sección anterior, decidimos entonces implementar los algoritmos de chequeo de consistencia utilizando la estructura de datos presentada en la Figura 8.3. Como mencionamos anteriormente, los algoritmos fueron implementados separadamente tal como se presentaron en este documento, para poder comparar con propiedad el comportamiento de los mismos frente a los distintos tipos de inconsistencia. La implementación se realizó también en Java, y se utilizó, al igual que en el punto anterior, la implementación XERCES del parser SAX.
A continuación analizaremos la performance de los algoritmos de validación en distintas situaciones. Comenzaremos analizando qué sucede con documentos inconsistentes, variando los siguientes parámetros.

- *Concentración de punteros.* Mide la cantidad relativa de punteros dentro del documento. Que un documento tenga punteros indica que hay nodos que tienen más de un padre, y por lo tanto ciertos cálculos como el del lifespan, o el chequeo de ciclos tomarán más tiempo.

- *Distribución de los punteros.* Indica la ubicación de los punteros. Tomaremos tres opciones, en los primeros niveles del documento (antes de llegar al nivel medio), en los últimos niveles del documento (luego de la mitad del documento) y uniforme (distribuidos equitativamente en todo el documento). Esta es una variable más para analizar el comportamiento de los algoritmos cuando un nodo tiene más de un padre. Analizaremos cómo

varían los tiempos de ejecución según si tuvimos que atravesar o no los punteros al validar.

- *Forma del documento.* Analizaremos dos casos. Primero el caso de documentos con forma regular, en el cual los los niveles se van 'ensanchando' a medida que vamos hacia las hojas, y un caso patológico, en el cual hay una rama que crece desbalanceando el documento.

Para obtener documentos variando las distintas características descriptas anteriormente, así como el tamaño de los mismos, se construyó un generador de documentos. Describiremos este generador a continuación.

8.3.1. Generador de documentos TXML

El generador desarrollado construye documentos TXML consistentes según las condiciones de consistencia de la Definición 2.

Puede generar documentos con datos aleatorios teniendo en cuenta los siguientes parámetros:

- Altura. Define la altura (cantidad de niveles) máxima del documento.
- Ancho. Indica la cantidad máxima de nodos en un nivel.
- Porcentaje de punteros. Indica el porcentaje de punteros máximo dentro del documento.
- Posición de punteros. Indica la posición de los punteros dentro del documento (primeros niveles, niveles centrales, niveles inferiores).
- Cantidad máxima de hijos. Define la cantidad máxima de hijos que puede tener cada nodo.
- Cantidad mínima de hijos. Define la cantidad mínima de hijos que puede tener cada nodo.

Con estos parámetros, se generará un documento que tendrá siempre la cantidad de niveles indicada como máximo. Se irá incrementando la cantidad de nodos en cada nivel de manera controlada, de forma que los documentos resultantes siempre son de forma regular. De esta manera, cada nivel tiene una cantidad de nodos aproximada prefijada por el alto y el ancho. La cantidad de hijos de cada nodo será un número aleatorio entre las cantidades mínimas y máximas de hijos indicada, teniendo especial cuidado en no agregar más nodos a un nivel, ni que esta cantidad quede muy por debajo que lo que corresponde según la cantidad prefijada.
Se generará exactamente el porcentaje de punteros máximo indicado, distribuidos uniformemente dentro de los niveles definidos según el parámetro que indica la posición de los punteros. En algunos casos algunos punteros creados serán luego desechados para mantener la consistencia del documento, con lo cual lo que

asegura el generador es que el documento contiene una proporción de punteros aproximada a la indicada. Garantizar el porcentaje exacto es complicado porque los intervalos temporales y la cantidad de hijos de cada nodo, así como el nodo apuntado son totalmente aleatorios. Para determinar a qué nodo hace referencia un puntero, se comienza buscando un nodo al azar dentro del documento y verificando que se pueda apuntar al mismo. Si no se puede, se comienzan a recorrer los nodos dentro del nivel donde se encuentra el nodo analizado y hacia los niveles inferiores, hasta encontrar un nodo al que sí se pueda apuntar sin generar inconsistencias. Puede ocurrir que el lifespan del nodo puntero no le permita apuntar a ningún nodo dentro de los niveles requeridos sin violar las condiciones de consistencia. En este caso, de no poder modificarse el lifespan del puntero, este deberá descartarse.
Cada nodo generado tiene inicialmente un lifespan incluido aleatoriamente dentro del lifespan del nodo padre. Es decir, los instantes 'Time:FROM' y 'Time:TO' son seleccionados aleatoriamente dentro del intervalo dado por el lifespan del nodo padre.

Ejemplo 36 (Documento generado)**.**
El documento de la Figura 8.8 fue construido por el generador. Los parámetros utilizados fueron,

- *Altura: 10*
- *Ancho: 20*
- *Proporción de punteros 0.4*
- *Posición de los punteros: todo el documento*
- *Cantidad máxima de hijos por nodo: 10*
- *Cantidad mínima de hijos por nodo: 0*

De esta manera, se pudieron generar documentos de hasta 1MB de tamaño en un tiempo no mayor a un par de horas. Documentos de mayor tamaño requerían mucho tiempo de procesamiento para garantizar la consistencia al añadir los punteros y por lo tanto se eligió generarlos copiando un documento de menor tamaño reiteradas veces luego de sí mismo.
Todos los documentos creados con el generador tienen forma regular. Para los documentos de formato irregular se implementó un algoritmo en el cual la estructura del documento está previamente definida (según el tipo de documentos que buscamos analizar), manteniendo la aleatoriedad en los intervalos temporales, pudiendo indicar la cantidad de niveles que tendrán las ramas más largas del documento.

Ejemplo 37. *Documento irregular*
El documento de la Figura 8.9 fue construido por el algoritmo previamente mencionado. La cantidad de niveles indicada fue de 4.

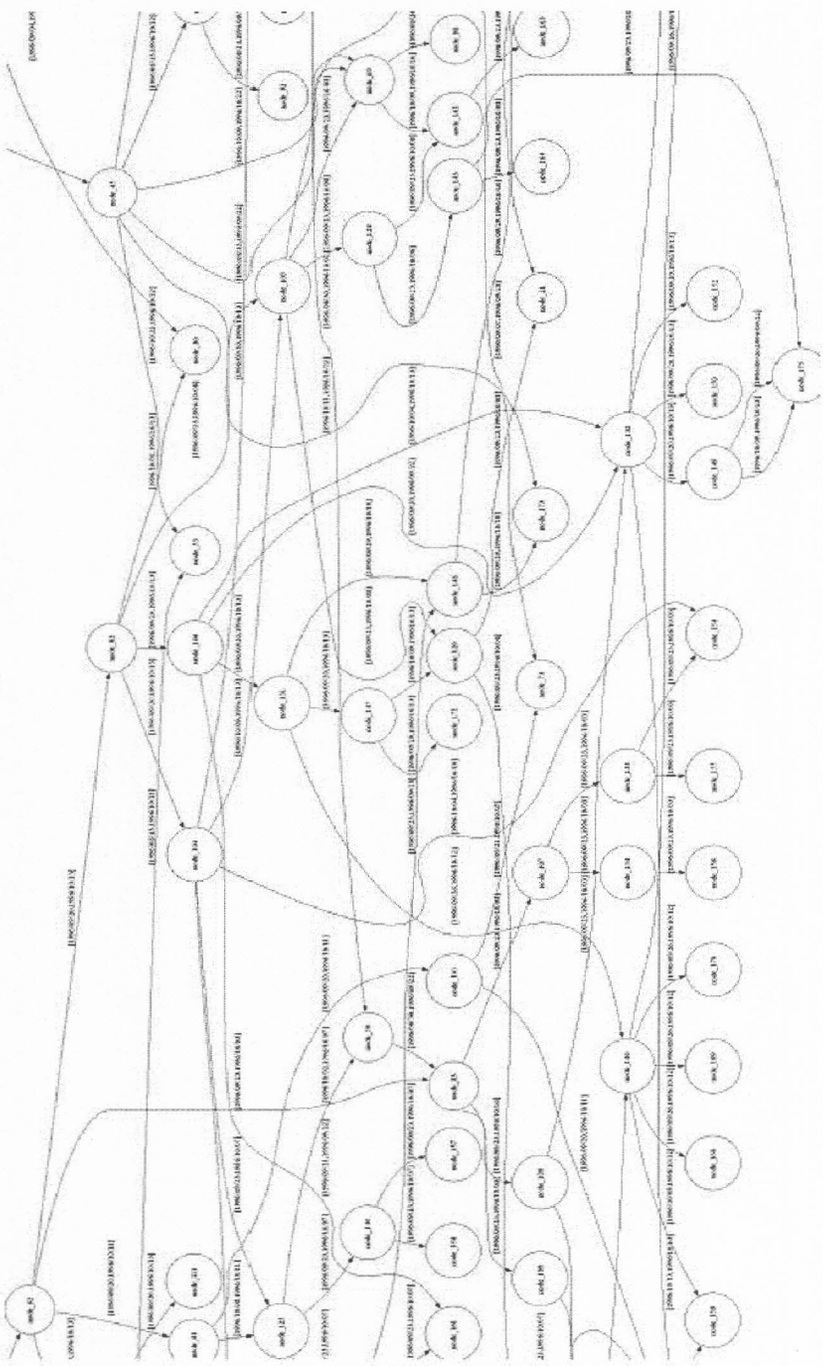


Figura 8.8: Porción de documento construido por el generador.

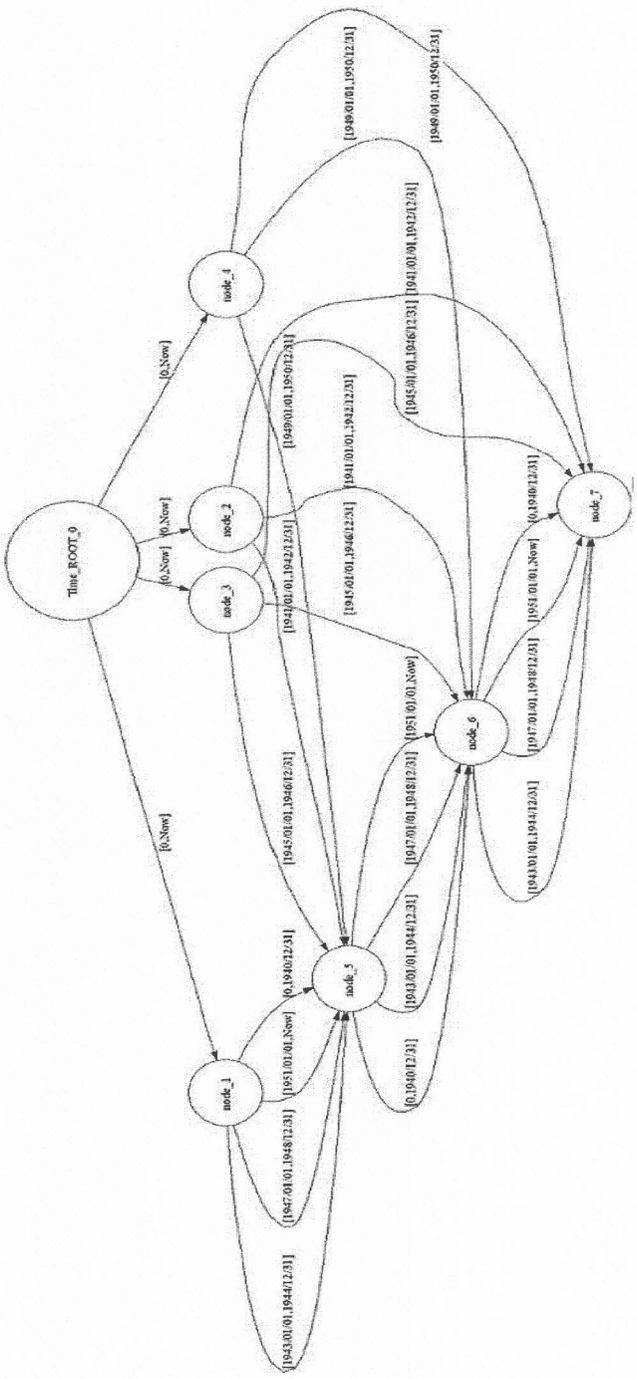


Figura 8.9: Documento construido por el generador para archivos irregulares.

8.3.2. Chequeo de consistencia en documentos regulares

Objetivo

La intención de este experimento es analizar la performance del algoritmo de chequeo de consistencia para documentos regulares, con una única inconsistencia. Se observará cómo varían los tiempos de ejecución de los algoritmos según el tipo de inconsistencia, su ubicación dentro del documento y la proporción de punteros presentes en el mismo.

Procedimiento

Se tomaron archivos de distintos tamaños, entre 12KB y 21048KB, todos con las siguientes características:

- Alta concentración de punteros ($\approx$ 40 por ciento)
- Distribución uniforme de punteros dentro del documento
- Forma regular

Los archivos con tamaño menor a 1MB fueron generados de manera aleatoria con un generador desarrollado durante este trabajo. Mientras que los de tamaño mayor, fueron obtenidos concatenando los archivos antes generados uno al lado del otro, debido al gran tiempo requerido para generar archivos grandes, con punteros y que preservaran las condiciones de consistencia de la Definición 2. Para cada tipo de inconsistencia, se crearon tres sets de documentos, partiendo del set anteriormente descripto y agregando una inconsistencia en distintas posiciones del documento. La posición de la inconsistencia se varió entre:

- Primeros niveles (o posición alta)
- Niveles centrales (o posición media)
- Últimos niveles (o posición baja)

Luego se variaron las características de los archivos para ver en que manera los diferentes parámetros afectan los resultados. En primer lugar se cambió la proporción de punteros disminuyéndola a un 20 por ciento. En segundo lugar se cambió también la distribución de los punteros para ambas proporciones. Se crearon archivos con punteros en:

- Niveles superiores: Los punteros sólo se encuentran en la mitad superior del documento
- Niveles inferiores: Los punteros sólo se encuentran en la mitad inferior del documento

Resultados

En las Figuras 8.10 y 8.11 se pueden apreciar los resultados obtenidos para cada tipo de inconsistencia y cada variante de los parámetros anteriormente descriptos.
En las tablas allí presentadas pueden observarse, para cada tipo de inconsistencia, y para posible posición de la misma (alta, media, o baja) el tiempo en milisegundos necesario para validar archivos con las distintas concentraciones y distribuciones de punteros, y los distintos tamaños, medidos en KB, de archivos.

Análisis de resultados

Las Figuras 8.12 y 8.13 ilustran los resultados para documentos con alta proporción de punteros distribuidos uniformemente con una inconsistencia tipo *i* o tipo *ii* respectivamente. Podemos ver en ellas que el costo de la validación aumenta a medida que la inconsistencia baja en los niveles del documento. Se puede observar que para inconsistencias cerca de la raíz, el algoritmo de chequeo termina rápidamente, esto sucede puesto que se recorre el documento con BFS desde la raíz y por lo tanto se alcanza rápidamente el nodo inconsistente, momento en el cual el algoritmo termina, ya que que el chequeo de inconsistencias tipo *i* y tipo *ii* se realiza nodo a nodo.
En cambio, el algoritmo va disminuyendo su eficiencia a medida que la inconsistencia avanza en los niveles del documento, ya que deben recorrerse muchos más nodos antes de llegar al nodo inconsistente.
Según podemos observar en las Figuras 8.12, 8.15 y 8.16 la posición de los punteros no afecta notoriamente la eficiencia del algoritmo. Si bien de los algoritmos presentados supondríamos que para alta concentración de punteros el tiempo de validación sería mayor debido por un lado al cálculo del lifespan de cada nodo y por otro a la posibilidad de existencia de ciclos no temporales, en la práctica la implementación varió levemente, pasándose el cálculo del lifespan al momento del armado del grafo, con lo cual este efecto no puede verse en los tiempos de validación. Lo mismo pasa para los demás tipos de inconsistencia.

Podemos ver en la Figura 8.14, que con baja proporción de punteros, la relación según la posición de la inconsistencia se mantiene, y los tiempos de validación no son muy diferentes en comparación con el set de archivos de alta cantidad de punteros, ilustrado en 8.12. Si bien sería de esperarse que los tiempos se redujeran sustancialmente, debemos aclarar en este momento, que la implementación varió levemente de la planteada en el Capítulo referente al chequeo de consistencia y el cálculo del lifespan se realiza a medida que se va levantando el documento a memoria, con lo cual por ejemplo, el algoritmo que chequea la existencia de inconsistencias de tipo *ii* se ve simplificado puesto que el cálculo del lifespan tiene en realidad $O(1)$ y el orden total del algoritmo es entonces $O(|E|)$. Esto hace que la proporción de punteros no influya realmente en el orden del algoritmo durante el chequeo, ya que esa complejidad fue trasladada al momento de creción de la estructura de datos.

		Alta cantidad de punteros																							
		Punteros uniformemente distribuidos								Punteros en la zona superior								Punteros en la zona inferior							
		13	242	603	1718	5182	10449	15749	21049	23	261	514	1495	6017	10609	15210	21344	27	105	592	1654	4987	10049	15139	20230
Tipo i	Alta	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Central	0	0	15	32	78	125	172	218	0	0	0	0	31	63	78	125	0	0	15	15	31	78	141	188
	Baja	0	0	16	31	109	235	359	484	0	0	0	31	125	203	297	453	0	16	0	31	109	343	312	422
Tipo ii	Alta	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	32	94	109	172
	Central	0	0	15	31	62	125	188	250	0	0	0	16	47	78	110	172	0	0	0	31	94	188	297	391
	Baja	0	0	15	31	109	203	329	422	0	0	15	31	125	203	297	406	0	0	15	15	15	47	63	93
Tipo iv	Alta	0	0	16	0	1296	3219	5141	7094	0	31	63	16	813	1610	2391	3437	0	0	0	0	641	1641	2610	3656
	Central	0	31	78	594	1937	3922	5828	7765	0	31	31	31	844	1625	2421	3547	0	0	16	0	687	1641	2625	3656
	Baja	16	218	62	359	1640	3609	5562	7484	0	47	62	250	1078	1859	2672	3718	0	31	94	328	1000	2047	3000	4063

Figura 8.10: Tiempos de validación en milisegundos para archivos con alta proporción de punteros.

		Baja cantidad de punteros																							
		Punteros uniformemente distribuidos								Punteros en la zona superior								Punteros en la zona inferior							
		25	223	595	1614	6495	11456	16417	21378	26	52	581	1583	6368	11230	16091	20953	23	261	580	1620	6514	11487	16459	21432
Tipo i	Alta	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Central	0	0	0	16	62	109	141	188	0	0	0	16	62	110	156	219	0	0	15	16	62	109	156	203
	Baja	0	0	16	31	125	219	312	406	0	0	0	32	109	203	296	390	0	0	16	31	125	235	329	375
Tipo ii	Alta	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Central	0	0	0	16	63	110	141	203	0	0	16	15	63	125	172	235	0	0	0	0	47	94	141	188
	Baja	0	0	0	31	125	203	313	407	0	0	16	16	125	203	297	375	0	16	0	31	109	219	297	390
Tipo iv	Alta	0	0	0	16	625	1203	1828	2438	0	0	0	15	437	859	1312	1672	0	0	0	0	625	1218	1828	2547
	Central	16	16	16	46	672	1250	1844	2485	0	0	16	78	500	953	1313	1734	0	0	15	63	687	1297	1906	2500
	Baja	0	32	63	188	844	1485	2062	2656	0	0	31	125	531	968	1391	1765	0	31	32	203	828	1437	2016	2641

Figura 8.11: Tiempos de validación en milisegundos para archivos con baja proporción de punteros.

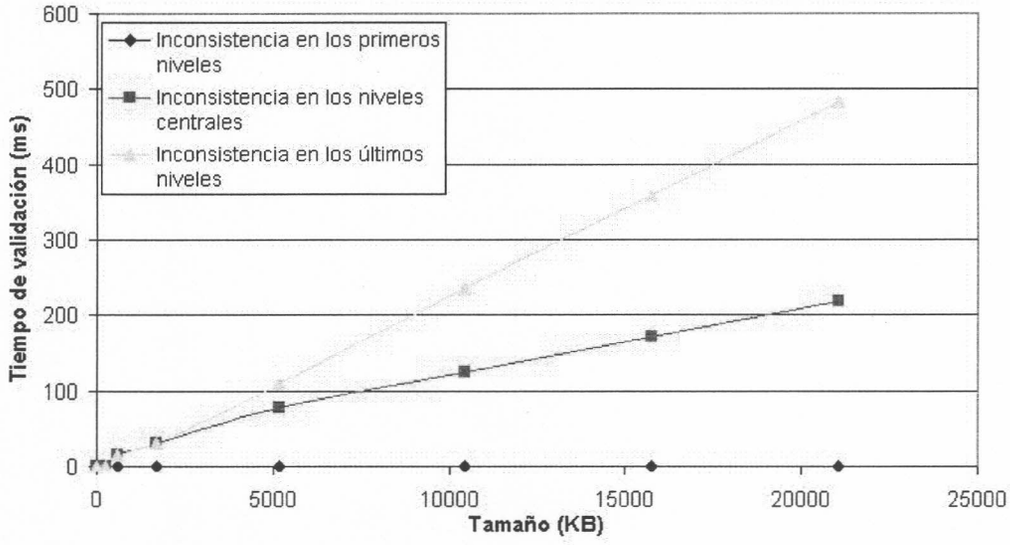


Figura 8.12: Validación para distintas ubicaciones de la inconsistencia insertada. Inconsistencia de tipo i en un documento con alto porcentaje de punteros distribuidos uniformemente

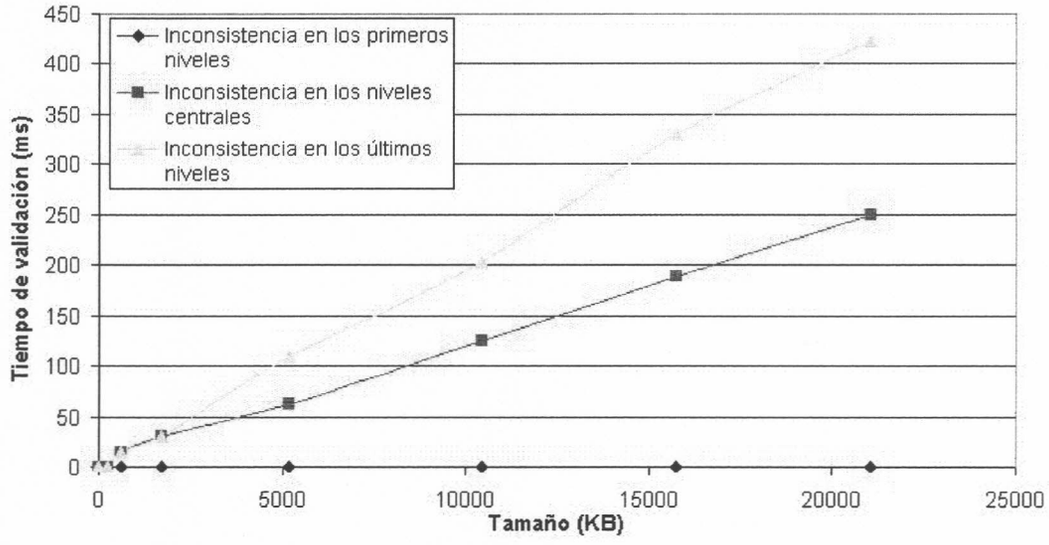


Figura 8.13: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo ii en un documento con alto porcentaje de punteros distribuidos uniformemente

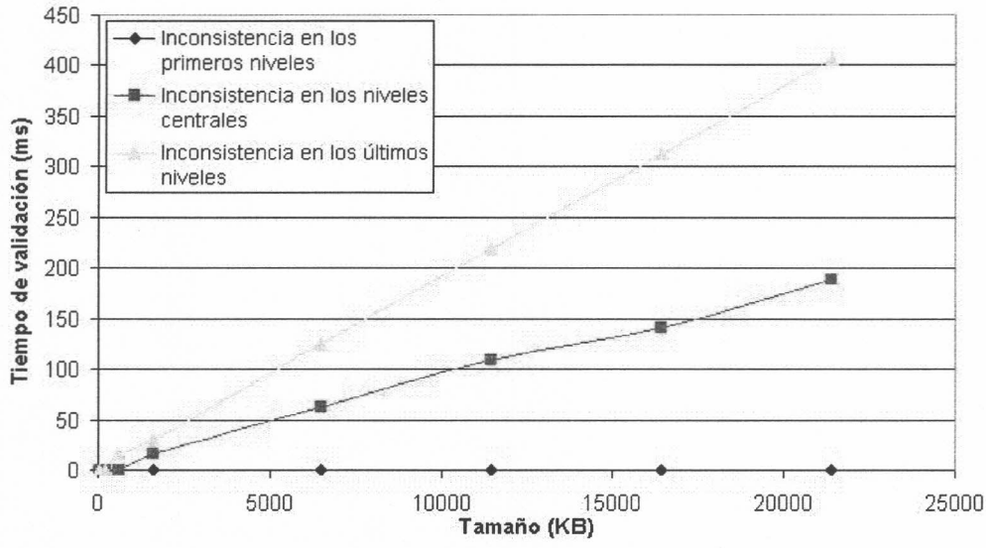


Figura 8.14: Validación para distintas ubicaciones de la inconsistencia insertada. Inconsistencia de tipo i en un documento con bajo porcentaje de punteros distribuidos uniformemente

A diferencia de las inconsistencias tipo i y tipo ii, para las inconsistencias de tipo iv en cambio, si bien importa la ubicación de la misma, ésta no es la única variable en la detección de la inconsistencia, sino que también importa hasta donde se puede continuar recorriendo el documento antes de encontrarse con un deadlock. Si la inconsistencia se halla en un nivel superior, pero recién detectamos el deadlock luego de recorrer la mayor parte del documento, puede tardarse más que si la inconsistencia se encuentra en un nivel inferior pero se detecta el problema inmediatamente después de visitar el nodo. Este hecho puede apreciarse en las Figuras 8.25 a 8.29 donde los tiempos de validación para todas las posiciones de las inconsistencias son similares. Al igual que para las inconsistencias de tipo tipo i y tipo ii, el cálculo del lifespan no se encuentra dentro del chequeo de consistencia con lo cual la cantidad de punteros no influye de manera apreciable. Sin embargo, a mayor cantidad de punteros, mayor probabilidad de existencia de ciclos no temporales, pudiendo aumentar el tiempo de validación.

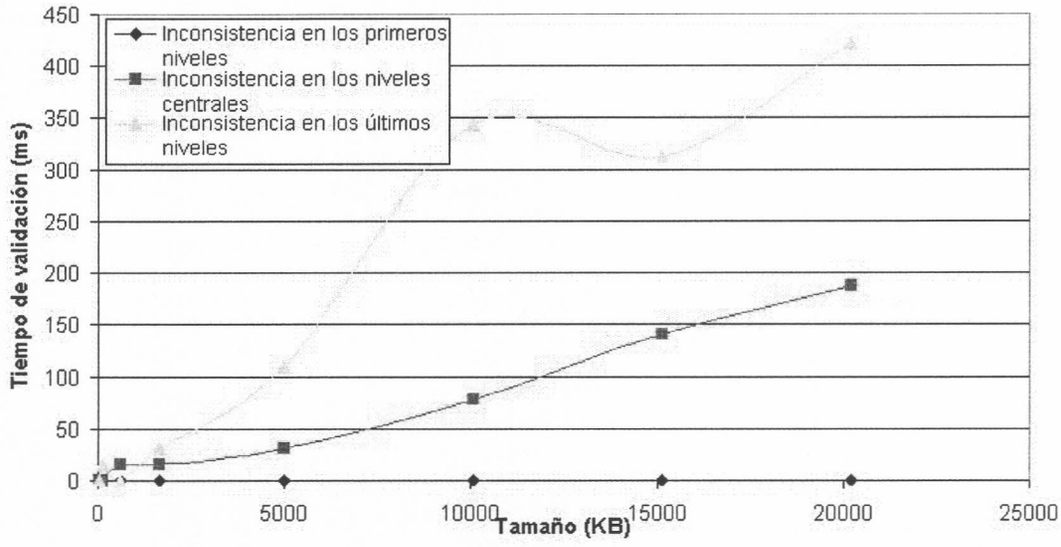


Figura 8.15: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo i en un documento con alto porcentaje de punteros ubicados en los niveles inferiores

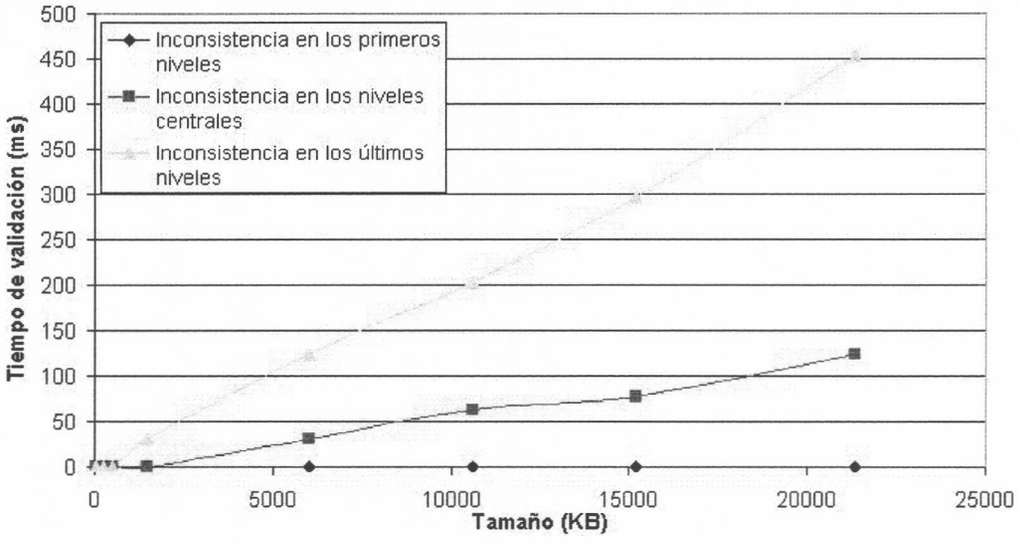


Figura 8.16: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo i en un documento con alto porcentaje de punteros ubicados en los niveles superiores

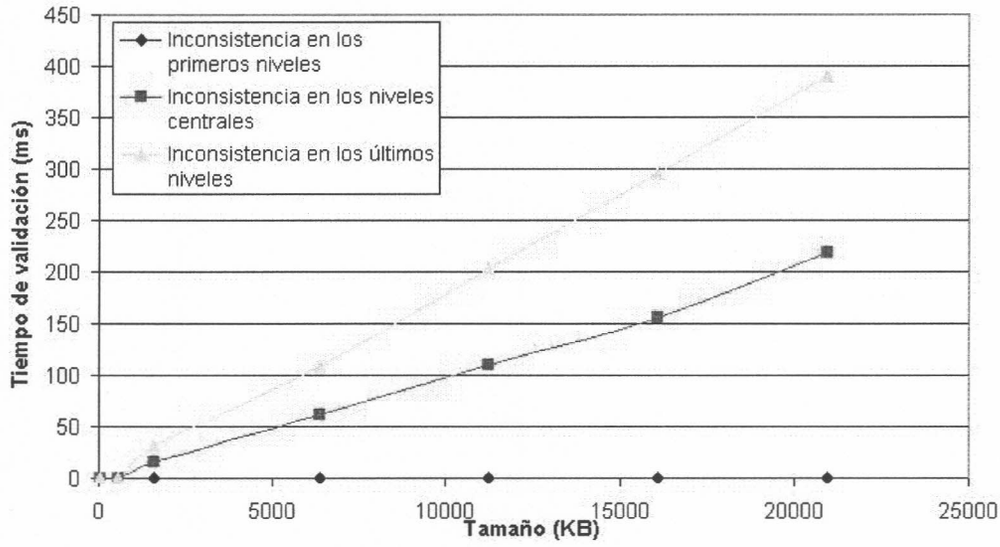


Figura 8.17: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo i en un documento con bajo porcentaje de punteros distribuidos en los primeros niveles del documento

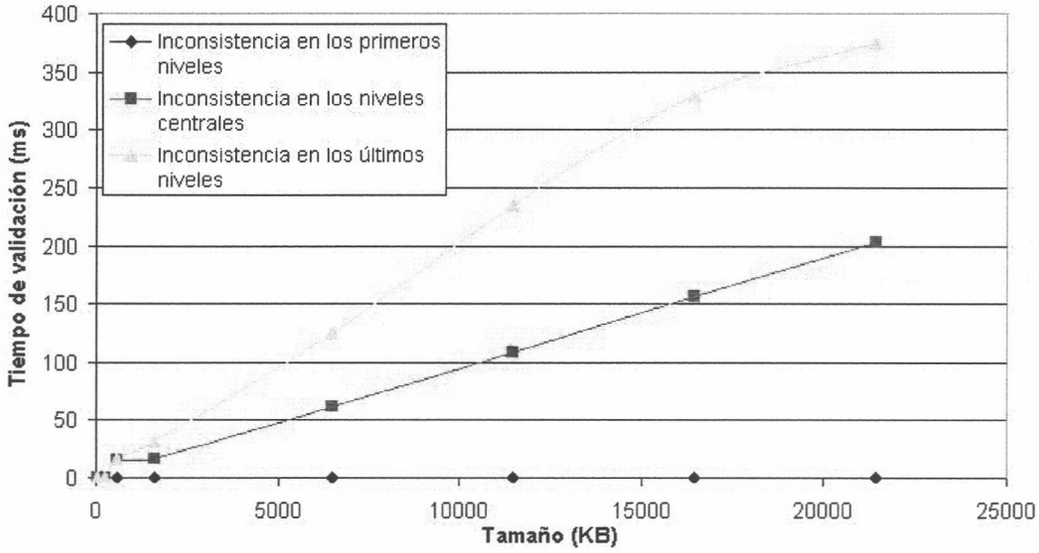


Figura 8.18: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo i en un documento con bajo porcentaje de punteros distribuidos en los últimos niveles del documento

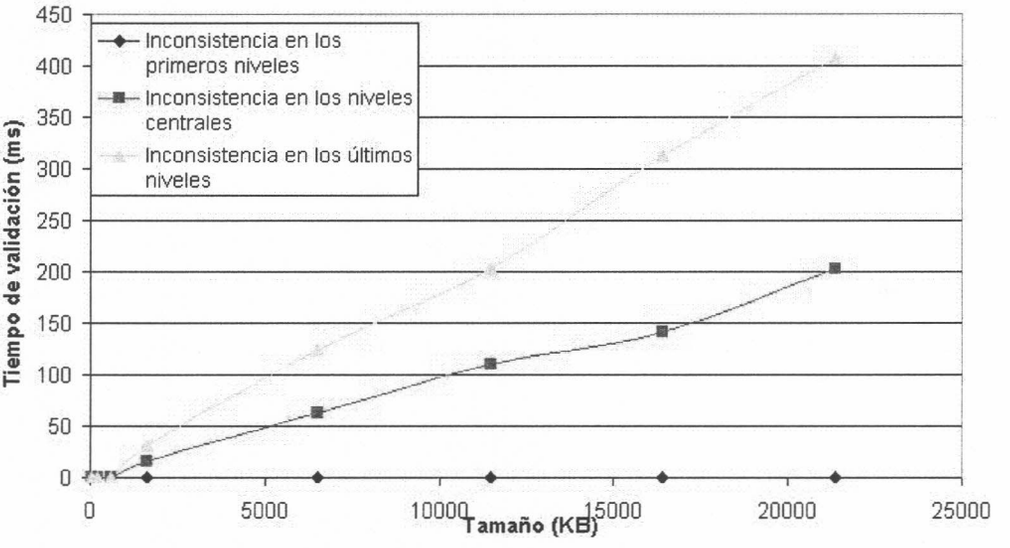


Figura 8.19: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *ii* en un documento con bajo porcentaje de punteros distribuidos uniformemente

8.3.3. Chequeo de consistencia en documentos consistentes regulares

Objetivo

El objetivo de este experimento es observar la eficiencia de los algoritmos de chequeo de consistencia cuando se aplican a documentos consistentes. Este es considerado el peor caso de los algoritmos puesto que todo el documento debe ser recorrido. Se compararán entonces los resultados con los de los experimentos anteriores

Procedimiento

Se utilizaron los documentos consistentes generados para el primer experimento.

Resultados

A continuación presentamos los resultados obtenidos para la validación de documentos consistentes según el tamaño de los mismos ,la ubicación y proporción de punteros presentes en cada documento.

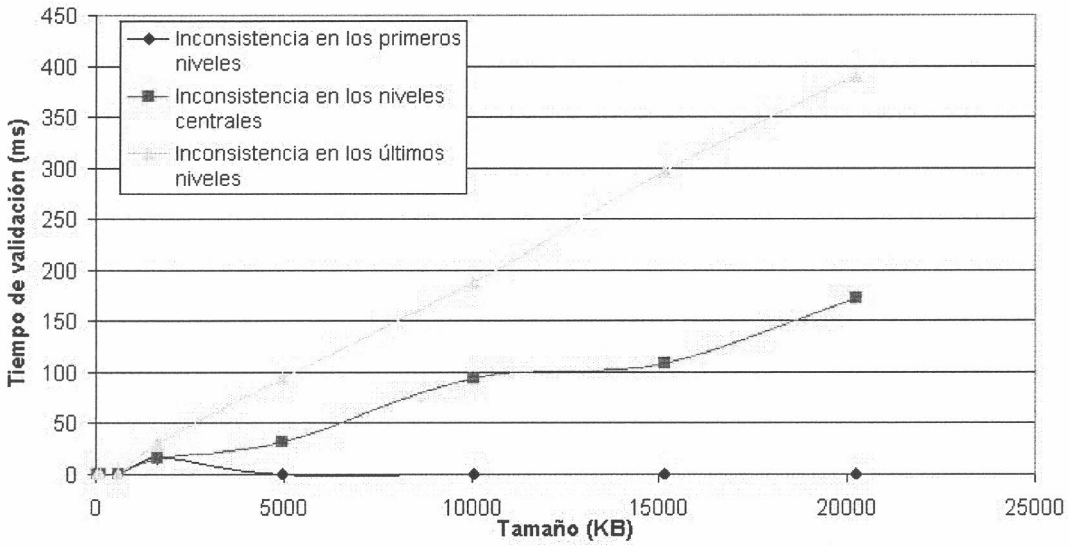


Figura 8.20: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *ii* en un documento con alto porcentaje de punteros distribuidos en los últimos niveles del documento

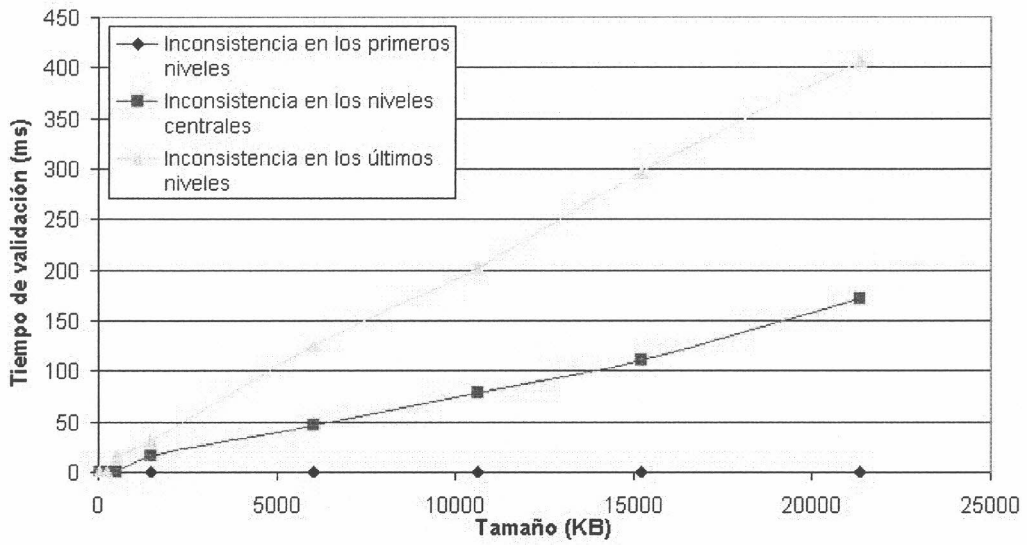


Figura 8.21: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *ii* en un documento con alto porcentaje de punteros distribuidos en los primeros niveles del documento

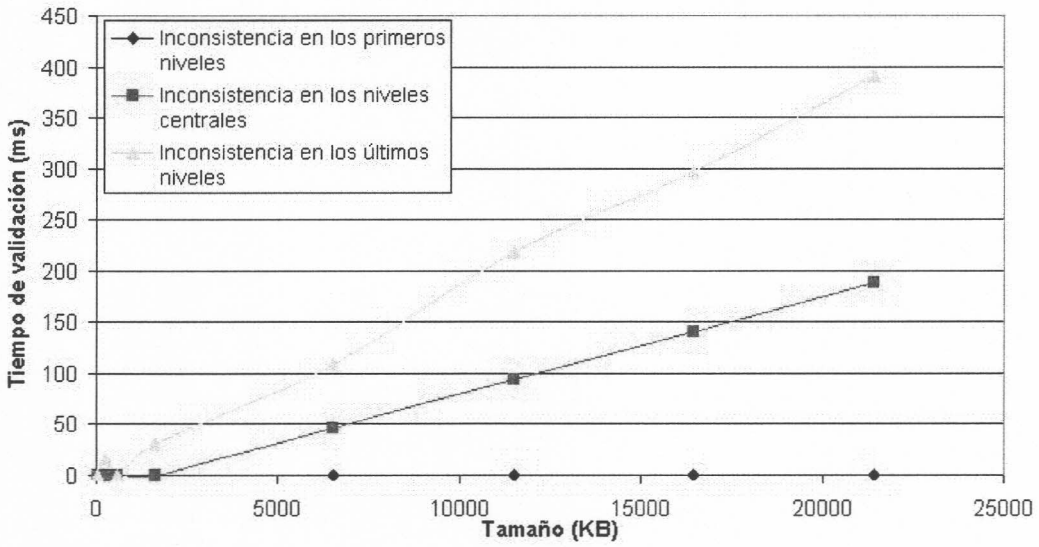


Figura 8.22: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *ii* en un documento con bajo porcentaje de punteros distribuidos en los últimos niveles del documento

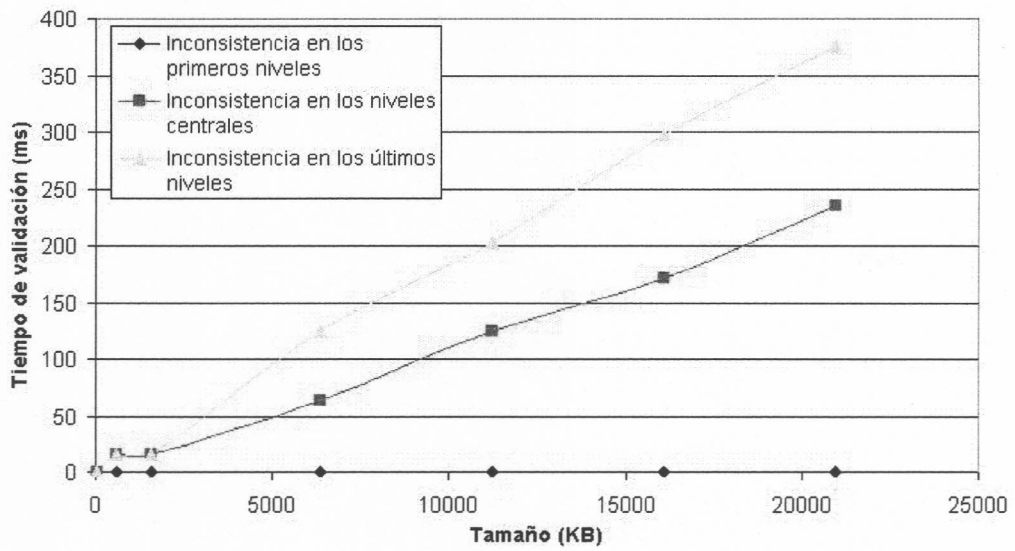


Figura 8.23: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *ii* en un documento con bajo porcentaje de punteros distribuidos en los primeros niveles del documento

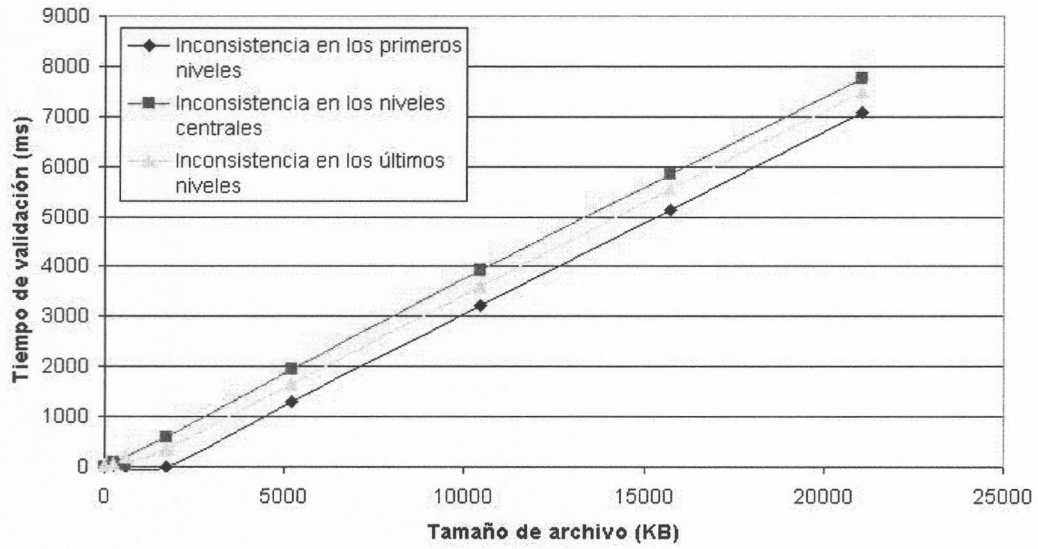


Figura 8.24: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *iv* en un documento con alto porcentaje de punteros distribuidos uniformemente

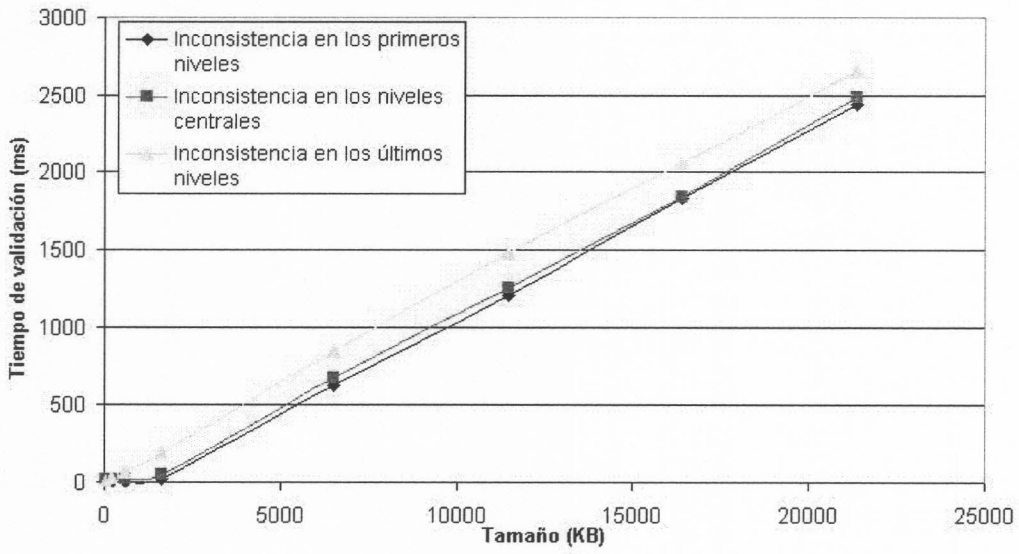


Figura 8.25: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *iv* en un documento con bajo porcentaje de punteros distribuidos uniformemente

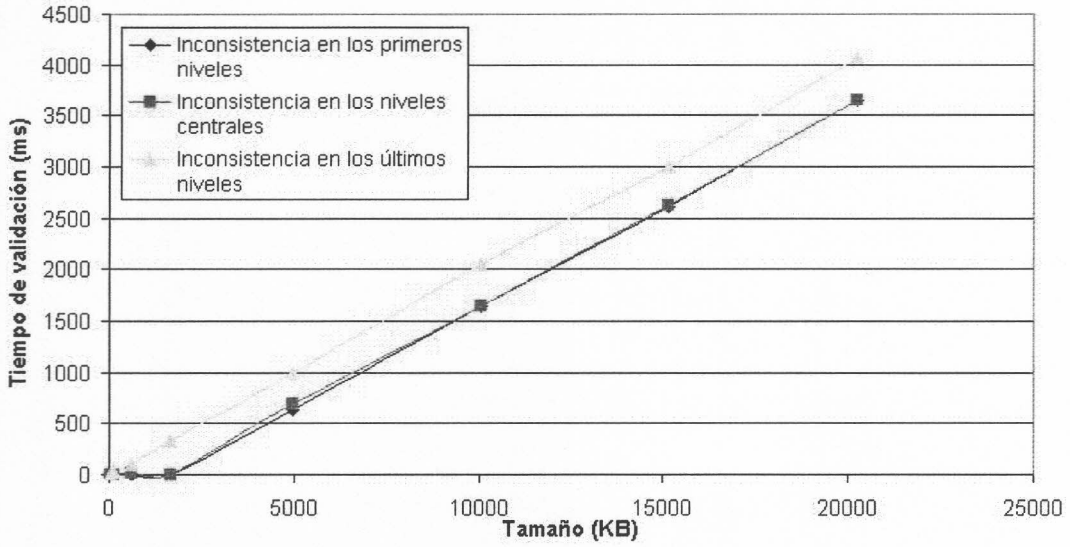


Figura 8.26: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *iv* en un documento con alto porcentaje de punteros distribuidos en los últimos niveles del documento

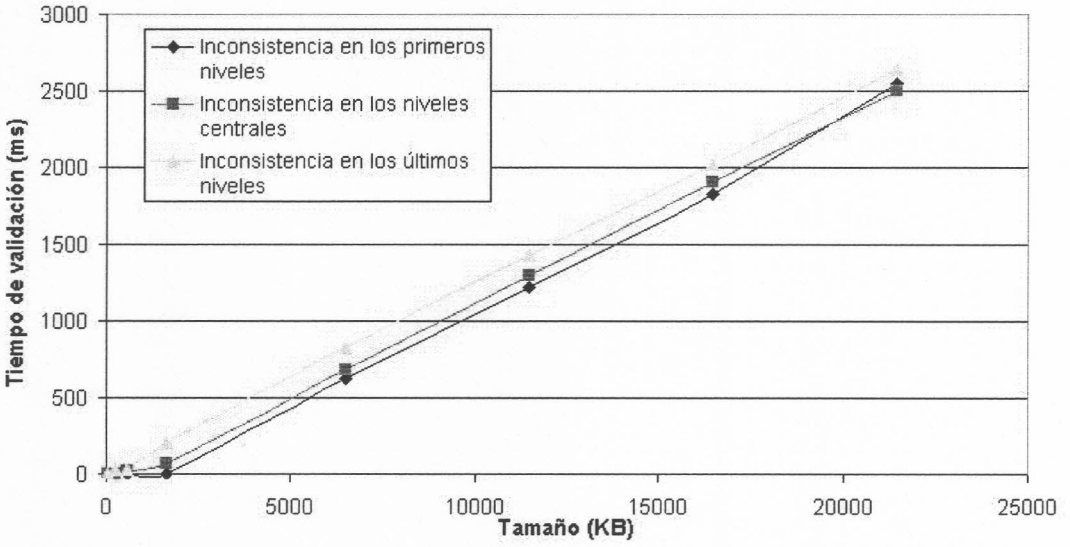


Figura 8.27: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *iv* en un documento con bajo porcentaje de punteros distribuidos en los últimos niveles del documento

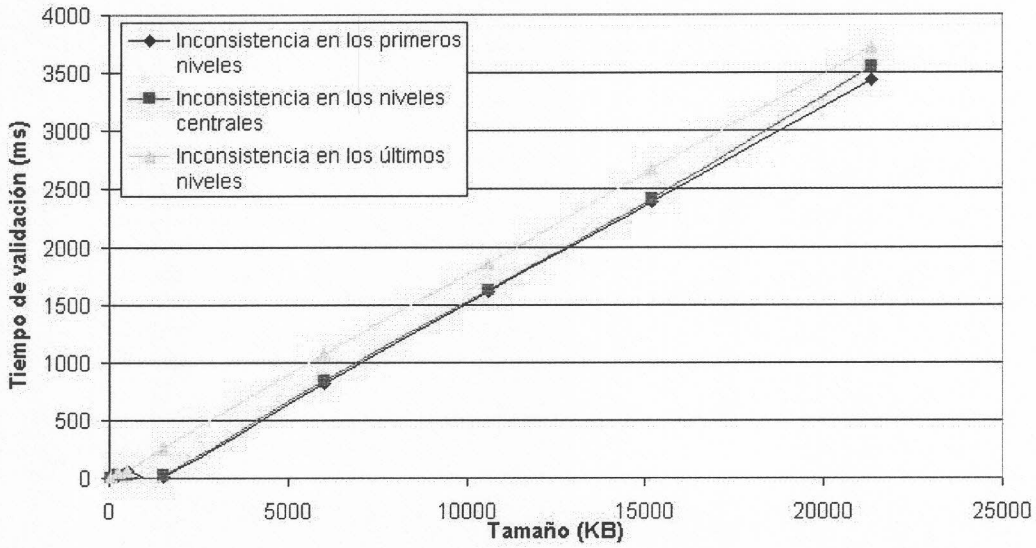


Figura 8.28: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *iv* en un documento con alto porcentaje de punteros distribuidos en los primeros niveles del documento

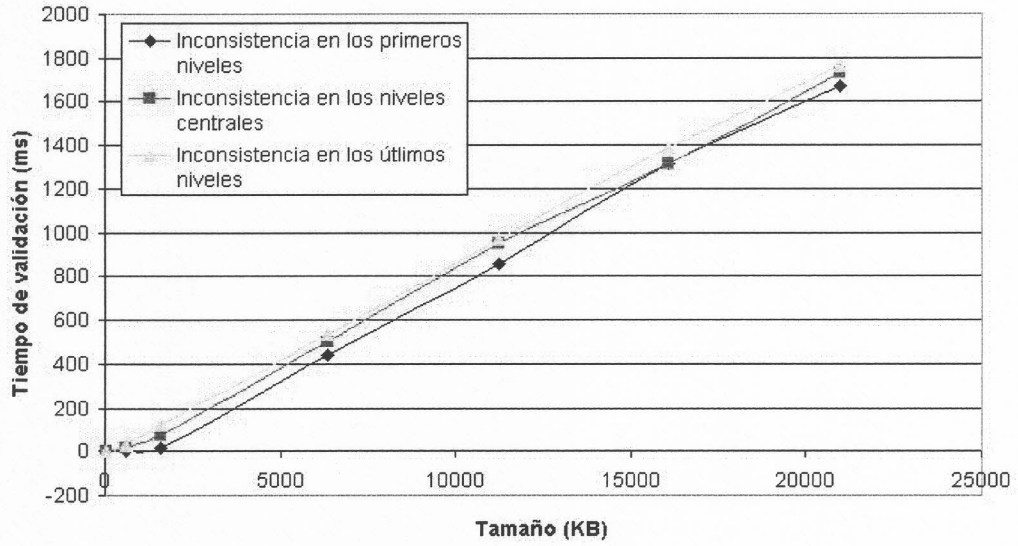


Figura 8.29: Comparación de curvas para distintas ubicaciones de la inconsistencia. Inconsistencias de tipo *iv* en un documento con bajo porcentaje de punteros distribuidos en los primeros niveles del documento

	Tam (KB)	t (ms)
Punteros distribuidos en los niveles inferiores	26	0
	104	47
	591	109
	1653	344
	4986	1187
	10048	2219
	15138	3328
	20229	4407

Figura 8.30: Tiempos de validación en milisegundos para documentos con distintas inconsistencias

	Tam (KB)	t (ms)
Punteros distribuídos en los niveles superiores	22	0
	260	63
	513	78
	1494	297
	6016	1282
	10608	2203
	15209	3203
	21343	4390

Figura 8.31: Tiempos de validación en milisegundos para documentos con distintas inconsistencias

	Tam (KB)	t (ms)
Distribución uniforme de punteros	12	0
	241	94
	602	375
	1717	672
	5181	2172
	10448	4297
	15748	6359
	21048	8516

Figura 8.32: Tiempos de validación en milisegundos para documentos con distintas inconsistencias

	Tam (KB)	t (ms)
Punteros distribuidos en los niveles inferiores	22	0
	260	31
	579	47
	1619	219
	6513	953
	11486	1687
	16458	2422
	21431	3156

Figura 8.33: Tiempos de validación en milisegundos para documentos con distintas inconsistencias

	Tam (KB)	t (ms)
Punteros distribuídos en los niveles superiores	25	0
	51	0
	580	47
	1582	188
	6367	750
	11229	1297
	16090	1875
	20952	2375

Figura 8.34: Tiempos de validación en milisegundos para documentos con distintas inconsistencias

	Tam (KB)	t (ms)
Distribución uniforme de punteros	24	16
	222	31
	594	78
	1613	234
	6494	1046
	11455	1812
	16416	2578
	21377	3313

Figura 8.35: Tiempos de validación en milisegundos para documentos con distintas inconsistencias

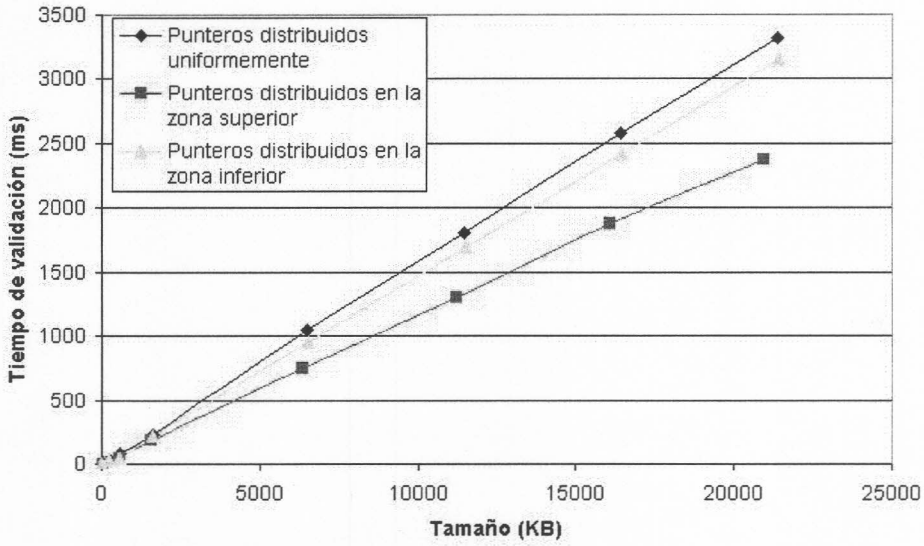


Figura 8.36: Tiempos de validación en milisegundos para documentos consistentes con baja proporción de punteros

Análisis de resultados

Comparando con el chequeo para las inconsistencias de tipo *iv* con alta proporción de punteros en el documento (Figuras 8.24, 8.26 y 8.28) vemos que los tiempos de validación para cada caso son muy similares. Esto se debe que para validar un documento consistente se ejecutan todos los algoritmos de chequeo y en cada caso se chequean todos los nodos, mientras que en el caso de documentos inconsistentes los algoritmos cortan al pasar por el nodo inconsistente (inconsistencias tipo *i* y tipo *ii*) o bien, como puede darse en el caso de las inconsistencias de tipo *iv*, al encontrar un deadlock en la verificación.
De todas maneras, los tiempos máximos registrados en este caso, con documentos de tamaño mayor a 20MB son menores a 9000ms, esto es 9 segundos. Teniendo en cuenta que es poco probable que se presenten documentos de tanto tamaño y que el hardware en el que se realizaron las pruebas es estándar el tiempo de validación en el peor caso con documentos regulares resulta por demás alentador.

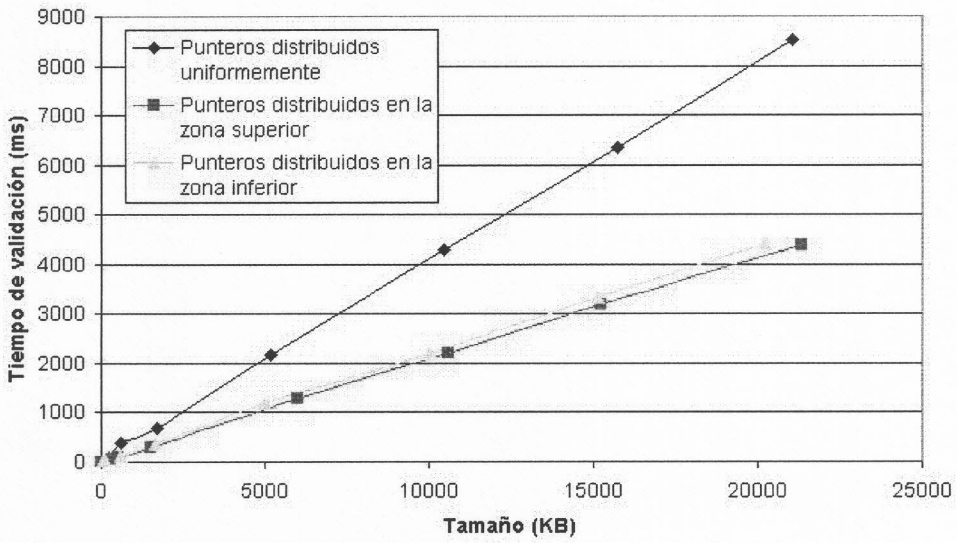


Figura 8.37: Tiempos de validación en milisegundos para documentos consistentes con alta proporción de punteros

8.3.4. Chequeo de consistencia en documentos no regulares

Objetivo

Analizar la performance de los algoritmos de chequeo de consistencia en casos patológicos (documentos con forma no regular), tanto para documentos inconsistentes como consistentes.

Procedimiento

Se generó un set de archivos de distintos tamaños a partir de un documento pequeño con las características deseadas, pegando el archivo repetidas veces a continuación de si mismo. El documento utilizado como modelo presenta las siguientes características:

- 4 niveles de profundidad en una única rama
- cada nodo de la rama que crece en profundidad, tiene 6 ejes incidentes a él, mitad provenientes del nodo padre dentro de la rama, y la otra mitad de nodos que se agregan como hijos de la raíz con el único objetivo de aumentar la cantidad de ejes incidentes a cada nodo de la rama.

Para insertar inconsistencias en los archivos, se generó una copia del archivo modelo, y se insertó la inconsistencia en el nodo deseado. Luego se generaron los archivos pegando el archivo modelo la cantidad de veces deseada, e intercalando el archivo inconsistente en el lugar apropiado.

Resultados

En las Figuras 8.38 y 8.39 pueden apreciarse los resultados de aplicar el chequeo de consistencia sobre documentos con las características antes mencionadas.

Análisis de resultados

Se puede ver que el chequeo de inconsistencias tipo *i* y tipo *ii* es casi instantáneo para archivos con un tamaño de hasta 2,5MB. Al igual que en el caso de grafos regulares, esto se debe principalmente al hecho de que la complejidad del cálculo del lifespan fue extraída de los algoritmos de chequeo. En el caso de inconsistencias de tipo *iv*, la tardanza se debe principalmente al hecho de que se requiere recorrer todo el documento en el peor caso. Además, podemos ver que los tiempos en esta última fueron más elevados que en para los documentos con formato regular y alta cantidad de punteros 8.24. Es probable que el motivo de este aumento en los tiempos de validación se debieran a que la inconsistencia tipo *iv* fue ubicada siempre en el último nivel y por ende el documento fue recorrido casi íntegramente para su validación. Igualmente se puede apreciar que

Tam(KB)	Tipo i	Tipo ii	Tipo iv	Consistentes
10	0	0	16	0
75	0	0	62	94
147	0	0	141	203
294	0	0	266	328
590	15	0	547	641
1182	16	15	1109	1266
2302	15	16	2125	2469

Figura 8.38: Tiempos de validación en milisegundos para documentos irregulares con distintas inconsistencias

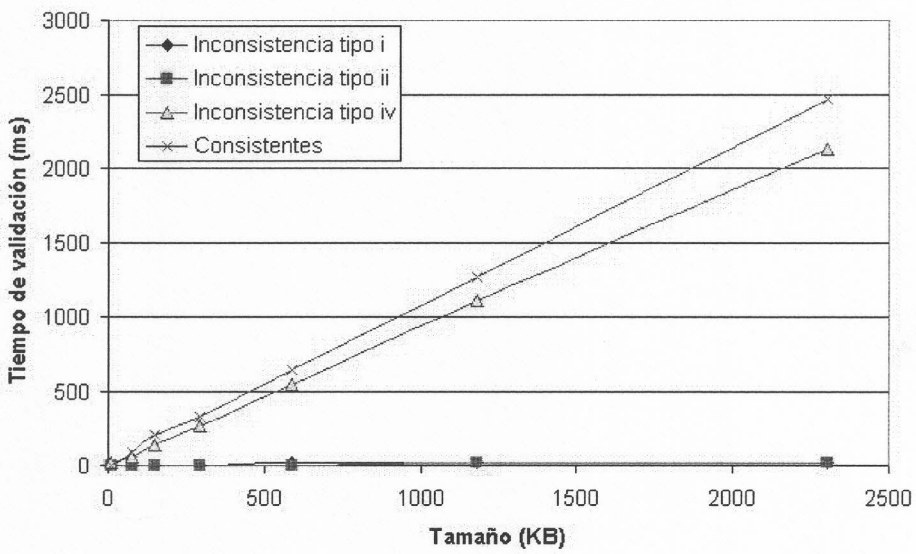


Figura 8.39: Tiempos de validación en milisegundos para documentos con formato irregular

el tiempo de validación es levemente menor al de los documentos consistentes, para los cuales el documento es recorrido completamente siempre.

8.4. Resumen

En este Capítulo implementamos dos mejoras al modelo en la Sección 8.1. La primera, la introducción de atributos temporales dentro del modelo original, permite modificar atributos manteniendo los valores históricos. La segunda, la introducción de valores default para los atributos de carácter temporal (`Time:FROM` y `Time:TO`) de manera de ahorrar espacio de almacenamiento dentro del documento. Se pueden apreciar los resultados de esta última mejora en la Sección 8.1.3. Luego, presentamos la implementación realizada para los algoritmos de chequeo de consistencia. Comenzamos con la comparación de distintas estructuras de datos en la Sección 8.2.1, mostrando los resultados de las distintas alternativas en la Sección 8.2.2. En la Sección 8.3 nos dedicamos a analizar la performance de los algoritmos de chequeo de consistencia implementados. En la Sección 8.3.1 presentamos el generador de documentos desarrollado para generar los datos de prueba. Finalmente, mostramos los resultados de los distintos experimentos realizados para medir la eficiencia de estos algoritmos bajo distintas circunstancias en las Secciones 8.3.2, 8.3.3 y 8.3.4.

Capítulo 9

Conclusiones y trabajo futuro

9.1. Conclusiones

Hemos estudiado el problema de validar un conjunto de restricciones temporales en un documento XML temporal, basado en el modelo de datos presentado en [MEN/04]. Propusimos métodos para chequear la presencia de inconsistencias en un documento y corregirlas. Dividimos el proceso de chequear y resolver con el objetivo de brindar al usuario la posibilidad de desechar el documento antes de intentar el proceso de corrección. Estudiamos las inconsistencias en forma individual y combinadas, y expusimos un conjunto de condiciones que hacen irrelevante la interferencia entre ellas (esto es, pueden tratarse y corregirse independientemente unas de otras). Estas condiciones pueden ser incorporadas en algoritmos para realizar de manera eficiente el proceso de corrección. Este trabajo puede ser un buen punto de partida para estudiar y razonar sobre restricciones con fechas indeterminadas en la línea introducida en [DYR/01, GRA/01].
Propusimos un lenguaje de Updates y expusimos las maneras de garantizar que las operaciones den como resultado un documento consistente. Incorporamos al modelo atributos temporales y valores default enriqueciendo el mismo y haciéndolo más eficiente en el aprovechamiento del espacio.Desarrollamos también en el marco de este trabajo un generador aleatorio de datos XML temporal que es capaz de generar documentos consistentes de hasta un 1MB en un tiempo razonable y acercamos una discusión cuantitativa a diferentes opciones de representación del documento en el contexto de inconsistencias combinadas.
Mucho trabajo queda por hacer en este campo, a continuación reseñamos algunos temas abiertos que surgieron a partir de la presente Tesis.

9.2. Trabajo futuro

9.2.1. Chequeo de consistencia

A lo largo de este trabajo presentamos algoritmos eficientes para el chequeo de consistencia en documentos TXML. Además brindamos métodos para resolver las mismas tanto en el caso de que se encuentren aisladas como en el caso en que las mismas interfieran entre sí. Quedan pendientes analizar y mejorar diferentes aspectos del problema. Un item importante es brindar un algoritmo que encuentre todas las inconsistencias dentro del documento de manera eficiente. Esto es trivial en el caso de inconsistencias de tipo i, tipo ii y tipo iii pero se complica para el caso de los ciclos. Si existe más de un ciclo dentro del documento puede resultar difícil aislar cada uno para proceder a resolverlos.

9.2.2. Corrección de inconsistencias

En este trabajo estudiamos algoritmos para resolver las inconsistencias una vez halladas. Cuando seleccionamos el modelo a utilizar lo hicimos en base a consideraciones de eficiencia tanto en el uso de procesador como de memoria. Sin embargo, al corregir inconsistencias debemos luego almacenar el documento resultante, con lo cual se debería considerar también este punto para lograr una estrategia eficiente. Hay diferentes opciones, que discutimos a continuación.

- Transformar a DOM antes de grabar: Para esto deberíamos levantar toda la información presente en el documento y no sólo los atributos necesarios para validar como hemos hecho en este trabajo. Se deberían comparar nuevamente las alternativas de modelo para ver si continúa siendo redituable construir una estructura alternativa a DOM. De todas maneras la estructura utilizada en el presente trabajo resultó ser por demás beneficiosa a la hora de analizar el problema y todos los resultados teóricos son aplicables a cualquier modelo (por ejemplo, las inconsistencias interfieren bajo las mismas condiciones)
- Guardar los cambios realizados y realizarlos directamente sobre el archivo.
- Traducir los algoritmos a la representación DOM equivalente: esta alternativa es la contrapartida de la primera.
- Aprovechar estructuras de indexación para realizar la validación.

9.2.3. Mejoras al modelo

Varias mejoras pueden realizarse en el modelo de TXML. A continuación brindamos una síntesis de las más importantes.

- Incluir atributos tipo REF: estos atributos no fueron considerados en el primer análisis de este modelo por simplicidad.

- Soporte para tiempo de validez y tiempo de transacción: En la implementación actual del lenguaje de updates sólo se permite modificar nodos actuales, es decir, solo se soporta tiempo de transacción. Se podría extender la implementación para permitir la modificación de cualquier nodo.

Bibliografía

[AMA/00] T. Amagasa, M. Yoshikawa, and S. Uemura. A temporal data model for XML documents. In Proceedings of DEXA Conference, pages 334 344, 2000.

[BAL/04] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of xml documents. ACM Transactions on Database Systems, 29(4):710 751, 2004.

[BAR/04] D. Barbosa, A.O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient incremental validation of XML documents. In ICDE, pages 671 682, 2004.

[BUN/02] P. Buneman, S. Khanna, K Tajima, and W. Tan. Archiving scientific data. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pages 1 12, Madison, USA, 2002.

[CAMP/06] Marcela Campo, Alejandro A. Vaisman: Consistency of Temporal XML Documents. In XSym 2006: 31-45, Seoul, Korea

[CHA/99] S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. In Theory and Practice of Object Systems, Vol 5(3), pages 143 162, 1999.

[CHI/01] S. Chien, V. Tsotras, and C. Zaniolo. Efficient management of multiversion documents by object referencing. In Proceedings of the 27th International Conference on Very Large Data Bases, pages 291 300, Rome, Italy, 2001.

[DOM] http://www.w3.org/DOM/

[DYR/01] C.E. Dyreson. Observing transaction-time semantics with TTXPath. In Proceedings of WISE 2001, pages 193 202, 2001.

[DYR/98] C. Dyreson and R. Snodgrass. Supporting valid-time indeterminacy. ACM Transactions on Database Systems, 23(1):1 57, 1998.

[GAO/03] C. Gao and R. Snodgrass. Syntax, semantics and query evaluation in the XQuery temporal XML query language. Time Center Technical Report TR 72, 2003.

[GRA/01] F. Grandi and F. Mandreoli. Effective representation and efficient management of indeterminate dates. In TIME'01, pages 164 169, 2001.

[GRA/04] F. Grandi. Introducing an annotated bibliography on temporal and evolution aspects in the world wide web. SIGMOD Record 33(2), pages 4 86, 2004.

[KAN/02] B. Kane, H. Su, and E. Rundensteiner. Consistently updating XML documents using incremental constraint check queries. In WIDM, pages 1 8, 2002.

[MAR/01] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In Proceedings of the 27th VLDB Conference, pages 581 590, Rome, Italy, 2001.

[MEN/04] A.O. Mendelzon, F. Rizzolo, and A. Vaisman. Indexing temporal XML documents. In Proceedings of the 30th International Conference on Very Large Databases, pages 216 227, Toronto, Canada, 2004.

[OLI/01] B Oliboni, E. Quintarelli, and L. Tanca. Temporal aspects of semistructured data. Proceedings of the Eight International Symposium of Temporal Representation and Reasoning, pages 119 127, 2001.

[PAP/03] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In ICDT, pages 47 63, 2003.

[RIZ/06] F. Rizzolo and A. Vaisman. Temporal XML documents: Model, index and implementation. In Submitted, 2006.

[SAX] http://www.saxproject.org

[WAN/03] F. Wang and C. Zaniolo. Temporal queries in xml document archives and web warehouses. In Proceedings of the 10th International Symposium on Temporal Representation and Reasoning (TIME'03), pages 47 55, Cairns, Australia, 2003.

[WAN/04] F. Wang and C. Zaniolo. XBiT: An XML-based bitemporal data model. In Proceedings of the 23rd International Conference on Conceptual Modeling, pages 810 824, Shanghai, China, 2004.

[WAN/05] F. Wang, X. Zhou, and C. Zaniolo. Efficient xml-based techniques for archiving, querying and publishing the histories of relational databases. In Time Center Technical Report, 2005.

[VAI/03] A. Vaisman, A. Mendelzon, E. Molinari, P. Tome. Temporal XML: Data Model, Query Language and Implementation. In GASCON 2003 Toronto, Canada