Trabajo Final de Grado

Título: ASLX - Analizador Semántico para lenguajes de programación basados en XML

Alumnos:
Gabriela Yanel Buffa y Franco Gastón Pellegrini

Director: Marcelo Arroyo
Co-director: Francisco Bavera

Licenciatura en Ciencias de la Computación
Departamento de Computación
Facultad de Ciencias Exactas Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

ASLX:Un analizador semántico para lenguajes de programación basados en XML

1. Introducción

Este trabajo tiene la finalidad de desarrollar una solución para validar semánticamente lenguajes de programación o similares basados en XML.

XML (Extensible Markup Language) es un lenguaje usado para estructurar información en un documento o en general en cualquier fichero que contenga texto, por ejemplo ficheros de configuración de un programa.
Los ficheros XML son ficheros de texto. Los símbolos “mayor que” y “menor que” se usan para delimitar las marcas que dan la estructura al documento. Cada marca tiene un nombre; veamos un ejemplo:

La marca <figura>, puede tener uno o más atributos:
<figura fichero=”foto1.jpg” tipo=”jpeg”>

En este caso tiene dos atributos, “fichero” y “tipo”.

Los atributos toman valores que tienen que estar entre comillas o entre apostrofes, ninguna marca se puede dejar abierta, o sea, por cada marca, *<figura>* por ejemplo, deberá existir una marca *</figura>* correspondiente que indica donde termina el contenido de la marca.
Que un documento esté «bien formado» solamente se refiere a su *estructura sintáctica básica*, es decir, que se componga de elementos, atributos y comentarios como XML especifica que se escriban.

Ahora bien, cada aplicación de XML, es decir, cada lenguaje definido con esta tecnología, necesitará especificar cuál es exactamente la relación que debe verificarse entre los distintos elementos presentes en el documento. Algunas formas de verificar que un archivo XML está bien formado, es mediante:

1. **Document Type Definition** o **DTD:** Define los tipos de elementos, atributos y entidades permitidas, y puede expresar algunas limitaciones para combinarlos. Los documentos XMLque se ajustan a su DTD son denominados válidos.

2. **XML Schemas.** Un Schema es similar a un DTD, define qué elementos puede contener un documento XML, cómo están organizados, qué atributos y de qué tipo pueden tener sus elementos.

Entre ambos tipos de verificación, hay varios campos que no se cubren, y que son muy útiles para verificar lenguajes de programación basados en XML:

1. Relación semántica entre contenido de atributos y tags.
2. Importación de variables (y tipo) de otro archivo.
3. Verificación semántica de formatos de archivos en variables.
4. Declaraciones de Ámbitos y verificación del alcance de variables.
5. Verificaciones condicionales a el contenido de un atributo.

Algunas de estas pueden realizarse de forma muy básica mediante expresiones regulares en XML Schemas, pero el problema es que se limita a un atributo en particular y **no** a la relación entre varios.
Surge así, la necesidad de una herramienta para verificar lenguajes de programación basados en XML o similares en el aspecto semántico. En síntesis, desarrollar una aplicación que pueda extender la verificación de validez que ofrece XML con los puntos mencionados y más.

Para aplicar nuestra herramienta a un problema real, se decidió utilizar la misma para validar semánticamente código fuente del lenguaje declarativo NCL[1].
Si bien crear aplicaciones NCL no es de gran dificultad para personas en el ámbito de la programación, requiere de un aprendizaje previo el cual se dificulta debido a que por ejemplo el lenguaje solo posee un tipo de datos básicos ("String") el cual es usado para referencia a cualquier tipo de entidad básica o elemento multimedia (video, imagen, etc), y las herramientas de interpretación de Ginga-NCL **no ofrecen** las facilidades básicas de un compilador moderno para realizar informes pertinentes al usuario de sus errores.

De esta forma y por lo mencionado anteriormente, de acuerdo a nuestros conocimientoss, *"ASLX", es la primer herramienta semántica para el lenguaje NCL y una de las pocas herramientas semánticas para lenguajes basados en XML o similares.*

1.1 Objetivos de desarrollo:

Teniendo en cuenta que:

- Se hace muy complejo encontrar errores en un programa si el compilador no ayuda.
- Encontrar y corregir errores observando salidas de errores confusas es muy poco productivo.
- Que no se llegan a cubrir todos los aspectos de verificación para poder corroborar si un archivo XML esta bien formado.

Se plantean como objetivos principales, crear un validador semántico que detecte en el menor tiempo posible los diferentes errores semánticos y/o sintácticos que puede tener un lenguaje de programación o similar basado en XML.

La herramienta pueda mostrar un informe detallado de los errores, de esta forma dicha herramienta podrá ser de gran ayuda para muchos desarrolladores en el momento de corregir sus aplicaciones, ya que al dar un análisis detallado podrán ir directamente al punto de error y solucionarlo. Otra funcionalidad que debe tener el programa, es la de ser utilizado como librería (para que por ejemplo un IDE pueda aprovechar la velocidad y las ventajas de esta solución).

Por otra parte, analizando que Ginga-NCL no verifica semántica, se utilizara dicho lenguaje como caso de estudio y prueba, para de esta forma verificar conceptos en el ámbito semántico (así también como en el sintáctico) que Ginga-NCL no verifica por si solo. De esta forma y de acuerdo a nuestros conocimientos es la primera herramienta semántica para el lenguaje mencionado con anterioridad.

1 **Gomes Soares, Luis Fernando.** "Programando em NCL": Desenvolvimento de aplicações para middleware Ginga, TV digital e Web. 2009.

2. Lenguaje de Scripting

2.1 Introducción

Un lenguaje Scripting es un tipo de lenguaje de programación que es generalmente interpretado (opuesto a compilado). Un script puede verse como un programa que puede acompañar un documento HTML o estar contenido en su interior.
Los scripts permanecen en su forma original (su código fuente en forma de texto) y son interpretados comando por comando cada vez que se ejecutan.

Características de los lenguajes Scripting:

1. Los scripts suelen escribirse más fácilmente, pero con un costo sobre su ejecución.
2. Suelen implementarse con intérpretes en lugar de compiladores.
3. Tienen fuerte comunicación con componentes escritos en otros lenguajes.
4. Los scripts suelen ser almacenados como texto sin formato.
5. Los códigos suelen ser más pequeños que el equivalente en un lenguaje de programación compilado.

En el desarrollo de esta herramienta, el lenguaje de Script para crear reglas semánticas se diseñó con el objetivo de poder brindar soluciones a problemas más amplios y diferentes que limitarse a verificar semántica de un solo lenguaje (por ejemplo Ginga-NCL). Mediante este mecanismo, se puede adaptar nuestro programa a verificar semántica sobre otros lenguajes de programación basados en XML.[2]

Un script comienza con un tag llamado *"semanticParser"*, el cual tiene referencia al esquema de script. También se debe especificar mediante el atributo "fileFormat" una lista (separada con comas) de extensiones que este script soporta al validar.
Luego, dentro del tag *"rules"* se especifican todas las reglas. Para este ejemplo se quiere definir un conjunto de reglas que sólo se deben aplicar a tags etiquetados con el nombre "media". Con este objetivo, se declaró en el atributo "name" en el tag "tag", el valor "media".

Las reglas que se quieren verificar sobre "media" son dos:
1. El tag "media" es un ***tipo*** de "objeto" o "variable" que se puede instanciar, por lo que requiere de un atributo donde llevar un i**dentificador** para futuras referencias a este.
2. El tag "media" posee un atributo cuyo contenido hace referencia a un "objeto" o "variable" de ***tipo*** "descriptor".

Para lograr estos objetivos, se realizó lo siguiente:

1. Se creó una regla mediante el tag "idProperties" en cuyo atributo "attributeName" se especificó el nombre del atributo dentro de "media" que va a hacer referencia, y atributo "attributeName".
2. Se creó una regla mediante el tag "reference" en cuyo atributo "idAttribute" se especificó el nombre del atributo dentro de "media" que va a contener el identificador. En el valor del atributo "isReferenceTo" se definió que el identificador en "idAttribute" hace referencia a tags con el nombre "descriptor".

2 Para mas detalle sobre los lenguajes de scripting y su uso en esta herramienta, referirse al apéndice I.

3. Diseño

3.1 Introducción NCL-XML

La finalidad de esta herramienta es intentar informar la mayor cantidad de errores semánticos ignorados por lenguajes de programación o similares, basados en XML, como por ejemplo Ginga-NCL. Muchos de estos errores podrían solucionarse al tener un buen diseño del esquema XML, pero por el momento no pueden ser verificados dada la limitación de los esquemas XML o DTD, de ahí el porqué del desarrollo e importancia de esta herramienta.
Al basar nuestros ejemplos y aplicación del programa en validar semántica del lenguaje Ginga-NCL, decidimos incluir ciertos chequeos sintácticos de XML (como la verificación de texto mediante expresiones regulares) ya que son necesarios para informar ciertos posibles errores que NCL ignora y son de importancia.

El intérprete que utiliza Ginga-NCL tiene muchos defectos a la hora de facilitarle el trabajo al programador, informado de errores triviales como por ejemplo:

1. Advertir sobre la falta de un archivo al cual se hace referencia
2. La utilización de objetos no declarados.
3. Relación inválida entre el contenido de objetos.
4. Chequeo de “tipos” en referencias a objetos declarados.
5. Etc..

“Mediante esta herramienta, se intentará brindar al usuario de todos estos posibles problemas que pueda tener en su código fuente”.

3.2 Diseño general

El proceso de desarrollo está centrado en una arquitectura definida con reglas desarrolladas en un script basado en XML. Primeramente se desarrolló un esquema XML ***“ScriptParserSchema.xsd”***, en el cual se definió un lenguaje para crear scripts con reglas para la semántica a verificar.

Para realizar un programa que ofrezca soluciones genéricas, se decidió utilizar un lenguaje de scripting, para así poder adaptar el mismo a las especificaciones del programador. El sistema de reglas mediante script, permite que este proyecto no solo pueda ser utilizado para verificar semántica de Ginga-NCL, sino que también de cualquier otro lenguaje basado en XML.

Dada la complejidad del mismo, se decidió realizar un diseño de manera incremental. Nuestro programa toma el script desarrollado y un código basado en xml como parámetro. Sigue las reglas definidas en el script y trata de validar el código XML informando los errores o advertencias pertinentes.

Posteriormente, se estudiaron las fallas especificas de Ginga-NCL para conocer las reglas y/o herramientas que se necesitaban incorporar.[3]

3 **Gomes Soares, Luis Fernando.** “Programando em NCL”: Desenvolvimento de aplicações para middleware Ginga, TV digital e Web. 2009.
Associacao Brasileira de Normas Técnicas. *“ABNT NBR 15606-2”*: Televisão digitalterrestre – Codificação de dados e especificações de transmissão para radiodifusão digital.

En el caso de Ginga-NCL se diseñó un script con todas las reglas semánticas del mismo (***"semanticCheck.xml"***), para que al brindarle uno o varios archivos con código NCL, el programa pueda verificarlos y crear un informe.
En este se definieron a través de expresiones regulares y reglas (aplicadas a "Tags") las condiciones que debe cumplir cualquier programa NCL para ser un aplicación válida y libre de errores en el ámbito semántico. Se diseñaron ademas ciertas verificaciones sintácticas debido a los defectos de implementación de Ginga-NCL (ej: expresiones regulares para verificar contenido).

El programa se dividió dos secciones: **Parser** y **Validador**. Si no hay errores de sintaxis, el Parser interpreta las reglas definidas en el script y genera como resultado un Validador.
Este Validador es utilizado para verificar las reglas sobre uno o varios archivos con formato especificado en el script.
El Parser recorre cada nodo XML del script, interpreta la regla a verificar y a que tags se debe aplicar, y lo almacena en un Validador. Este verifica que cada vez que aparezca un Tag (que tenga en su lista) en el archivo a verificar, cumpla con todas las reglas definidas para dicho Tag. Con esta estrategia, el diseño quedo lo mas simple posible, al no estar especializado a verificar un lenguaje en particular.

El conjunto básico de reglas que reconoce Parser son:

1. Existencia de un atributo con determinado nombre.
2. Correcto contenido de un atributo o correcta relación de contenido entre varios atributos.
3. Verificar existencia de archivos locales o externos.
4. Creación de objetos/variables referenciables.
5. Chequeo de tipos (para las referencias a objetos/variables creados).
6. Incluir código desde otros archivos.
7. Verificar contenido de texto de un nodo XML.
8. Validación condicional de ciertas reglas.

El programa puede ser utilizado directamente mediante una terminal o también puede ser importado como una librería.
Por ultimo, se realizaron diferentes casos de testing para cada uno de las expresiones regulares presentes en Ginga-NCL para de esta forma garantizar que cada entidad funciona de forma correcta.
El script general se verificó con la ejecución de cada uno de los ejemplos presentes en el libro Estándar NCL, para de esta forma garantizar que la herramienta implementada cumple de forma eficiente con el Estándar y que se puede correr cualquier tipo de programa desarrollado en el lenguaje NCL, independientemente del tamaño o complejidad del mismo.

3.3 Estructura y Recorrido de un archivo XML

Dado que XML es un lenguaje utilizado ampliamente en el desarrollo de la World Wide Web, existen ya herramientas y estándares de programación para leer documentos XML. Las herramientas o programas que leen el lenguaje XML y comprueban si el documento es válido sintácticamente, se denominan analizadores o "parsers".
El diseño para parsear los archivos XML se basa en la utilización de las API "SAX" y "DOM". Dichas API, son dos herramientas que sirven para analizar el lenguaje XML y definir la estructura de un documento, aunque existen muchas otras.

3.3.1 DOM

Document Object Model (DOM) es un modelo de objetos estandarizado para documentos HTML y XML. DOM es un conjunto de interfaces para describir una estructura abstracta para un documento XML, el mismo define la estructura lógica de los documentos y el modo en que se accede y manipula un documento. Con DOM los programadores pueden construir documentos, navegar por su estructura, y añadir, modificar o eliminar elementos y contenido.

Sin embargo, DOM no especifica que los documentos deban ser desarrollados como un árbol, ni tampoco especifica cómo deben implementarse las relaciones entre objetos. El DOM es un modelo lógico que puede desarrollarse de la manera que sea más conveniente, por eso se debe hablar de un modelo de estructura en general, y no de estructura en forma de árbol, en particular.

3.3.2 SAX

El Simple API for XML (**SAX**) es una interfaz simple para aplicaciones XML. Por lo general, se usa SAX cuando la información almacenada en los documentos XML, es decir, los datos, han sido generados por máquina o son legible por máquina. En este caso, SAX es la forma más directa de API para que los programas tengan acceso a esa información. Los datos generados y legibles por máquina incluyen algunos elementos como los siguientes:

SAX permite crear rápidamente una herramienta u operador de clase que puede crear instancias de los modelos de objetos basados en el almacenamiento de datos de los documentos.

3.3.3 Utilización de DOM y SAX en el diseño

Dado que **SAX** es más completo para almacenar información, y **DOM** es más simple de utilizar ya que no hay que implementar un árbol como estructura de dato, se decidió combinar ambos.
Mediante **SAX** modificamos los nodos parseados del archivo XML y les añadimos localización (para conocer en qué línea de código se encuentra el nodo).

Utilizando **DOM** creamos la estructura de datos tipo árbol para poder recorrer los archivos de forma sencilla sin tener que implementar nada.
También se utilizó **SAX** para validar archivos XML dado un **Schema XML**[1]**,** útil para verificar que el script y los archivos que se van a verificar tengan una base estable. [2]

1 http://www.w3.org/XML/Schema
2 Para mas información dirigirse al código fuente “XML_Utils.java” y a los métodos “tryValidateWithXMLSchema”

4. Implementación

4.1 Introducción

Como lenguaje de programación se eligió Java 7 solo por dos importante característica:

1. Desarrollar software en una plataforma y ejecutarlo en prácticamente cualquier otra plataforma.
2. Simple para producir cantidad y calidad de código (productividad).
3. **SAX** y **DOM** son incluidos en la librería estándar.

Para detalles técnicos de implementación, referirse a la documentación técnica del código (Javadoc o el mismo código fuente). En esta sección solo se discutirán decisiones y detalles generales del código fuente.

4.2 Detalles de implementación

Todo el proyecto fue creado bajo el IDE **NetBeans**[4], el cual facilita el archivo "build.xml" para poder compilar todo el código sin la necesidad de tener instalado dicho IDE.
Mediante la utilización de **Apache Ant**[5] podemos ejecutar el archivo "build.xml" y automáticamente generará la documentación Javadoc y un archivo ejecutable con extensión JAR.

Dado que el diseño es muy simple porque se utiliza DOM y SAX para parsear los scripts y archivos a validar, solo se comenta en esta sección detalles generales de las clases más importantes del programa. Para detalles, referirse al informe técnico o al código fuente.

4.3.1 Parser (ScriptParser.java)

Se recorre recursivamente todos los nodos XML válidos del archivo con las reglas definidas (script) mediante el algoritmo descrito en la sección de Diseño general del programa. Cada vez que el nombre del nodo coincide con el nombre de una palabra reservada del script, se actúa acorde al significado de la misma creando reglas, ER-Ex (patterns) o contenedores de reglas asociados a un tag (TagValidator.java).
Para facilitar la implementación y escritura de script, todos los caracteres blancos de las ER-Ex serán ignorados. Si se necesita de ellos, utilizar los caracteres de escapes adecuados.

Para resolver las macros de las ER-Ex, el parser intenta hacer la traducción de la macro solo una vez, y almacena los resultados. Ya que una macro siempre se reduce a una misma ER, es eficiente solo calcular esta reducción solo una vez. Esto permite un manejo más eficiente de la memoria y velocidad de parsing.
También se añadió un límite para el cálculo de las macros, ya que resolver recursivamente una macro mal definida puede dejar al programa en un estado de calculo infinito. Una vez obtenidas las ER y las Reglas, este devuelve un Validador

4 http://netbeans.org/
5 http://ant.apache.org/

4.3.2 Contenedor de Reglas (TagValidator.java)

Representa un conjunto de reglas solamente aplicables a un tag en particular. Se almacenan las reglas creadas mediante el script en listas, para su fácil recuperación.

4.3.3 Validador (Validator.java)

Se recorre recursivamente todos los nodos XML válidos del archivo que se está verificando mediante el algoritmo descrito en la sección *"Diseño general del programa"*, y si el nombre de algunos de los nodos esta en la lista de contenedores de reglas (TagValidator), se aplican todas las reglas que este contenga para validar dicho nodo/tag

Capítulo 5: Testing

5.1 Introducción

Las pruebas realizadas para comprobar el correcto funcionamiento del programa, fueron de Caja Negra. Se utilizó JUnit para diseñar un conjunto de casos de prueba y se comprobó a medida que se implementaba, que las salidas del programa sean exactamente las esperadas. El caso de prueba más completo es el de la demostración del script con las reglas de validación semántica para el lenguaje Ginga-NCL el cual utiliza la mayoría de las funcionalidades del programa, en un caso real.
Todas las pruebas están descritas junto al código fuente y los informes técnicos.

5.2 Optimizaciones

Los primeros test que se realizaron fueron de optimización. La evaluación fue dedicada especialmente al tiempo de respuesta de las estructuras de datos principales y la administración de las ER-Ex. En un comienzo de implementaron 5 versiones del programa, las cuales se evaluaron con 4 escenarios diferentes.

Las 5 versiones tenían las siguientes características:

Versión	***Característica***
1	Sin optimizaciones
2	Las ER iguales se resuelven solo 1 vez
3	Las Macros en ER-EX se resuelven solo 1 vez
4	Las Macros y ER se almacenan en HashMap en lugar de SortedMap(TreeMap)
5	HashMap en toda estructura de dato que requiera busquedas en Validador y Parser

Cada versión tiene incluidas las optimizaciones de la versión anterior.

Los parámetros evaluados fueron:

1. **Tiempo promedio de compilación del script (en ms):** Se ejecutaron los escenarios 10 veces, y se calculó el tiempo promedio de compilación.
2. **ER creadas:** Cantidad de ER (Java Patterns) creados.
3. **ER reutilizadas:** Cantidad de ER (Java Patterns) no creadas ya que su solución ya se conoce.
4. **Macros Analizadas:** Cantidad de Macros computadas para resolver las ER-Ex.
5. **Macros Reutilizadas:** Cantidad de Macros no computadas ya que su solución ya se conoce.

También se gráfico la cantidad de memoria RAM ocupada para ejecutar todos los escenarios en forma consecutiva.
Todas las medidas de performance se obtuvieron en una Notebook con un procesador Intel® Core™ i7-2630QM CPU @ 2.00GHz × 8 y 4Gb DDR3 de memoria ram, bajo Ubuntu 10.04 64bits (kernel Linux 3.2.0-25-generic).

Los resultados fueron:

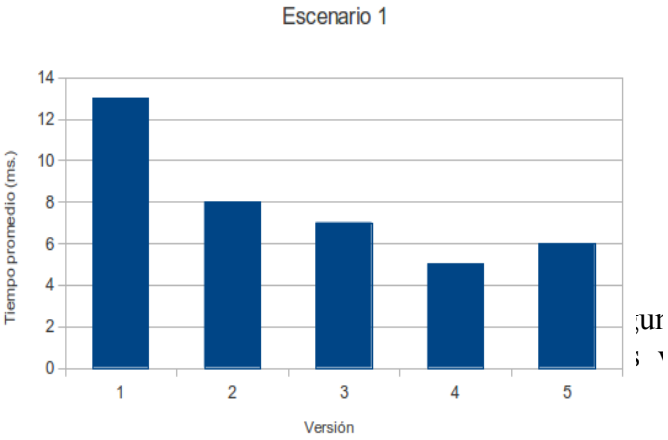
Versión 1

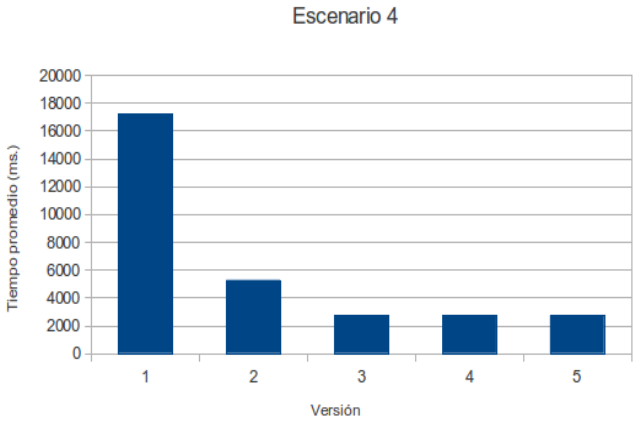
Escenario	Tiempo promedio (ms)	ER creadas	ER reutilizadas	Macros analizadas	Macros reutilizadas
1	13	4	0	7	0
2	938	5000	0	0	0
3	2477	5000	0	764178	0
4	17202	20	0	2334634	0

Se realizaron 3 versiones mas con la misma cantidad de escenarios, las cuales serán omitidas en este informe.[6]
A continuación se muestra en detalle la ultima versión y las optimizaciones con respecto a la primera.

Versión 5

Escenario	Tiempo promedio (ms)	ER creadas	ER reutilizadas	Macros analizadas	Macros reutilizadas
1	6	4	0	4	3
2	700	5000	0	0	0
3	166	100	4900	3	5049
4	2717	10	10	104	2





5.3 JUnit y Escenarios de prueba del lenguaje de Script

Para una correcta implementación de manera incremental, se optó por ir implementando el programa validando el mismo mediante el framework **JUnit**[7].
JUnit devolverá que método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

6 Para mas detalle referirse al informe completo de la herramienta desarrollada “ASLX”.
7 http://www.junit.org/

JUnit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

5.4 "Casos de Testing

5.4.3 Errores sintácticos en el Esquema XML de Ginga-NCL

Para la etapa de testing, se extrajeron del libro oficial[8], un total de 33 esquemas XML de Ginga-NCL. A continuación se da un detalle de aquellos esquemas que presentaron errores, la pagina donde puede encontrarse cada uno de ellos en el libro oficial, un detalle de los errores y las posibles soluciones que podrían implementarse, teniendo en cuenta la herramienta desarrollada *"**ASLX**"* para de esta forma ser aplicaciones validas.

"CausalConnector.xsd"

El esquema se encuentra en la pagina numero 48 del libro oficial.

Código donde se encuentra el error (linea 40):

```
<complexType name="nclType">
    <complexContent>
        <restriction base="structure:nclPrototype">
            <sequence>
                <element ref="structure:head" minOccurs="0" maxOccurs="1"/>
                <element ref="structure:body" minOccurs="0" maxOccurs="0"/>
            </sequence>
        </restriction>
    </complexContent>
</complexType>
```

Tipo de error:

```
cos-particle-restrict.2: Forbidden particle restriction: 'choice:all,sequence,elt'.
```

Solución:
Definir que el máximo de ocurrencias *(maxOccurs)* de la estructura body sea 1 en lugar de 0.

5.4.4 Errores sintácticos en ejemplos oficiales de Ginga-NCL

Se extrajeron todos los ejemplos de código del manual oficial mencionado con anterioridad para poder usarlos como prueba. De esta forma se obtuvieron un total de 20 ejemplos a los cuales primeramente se les corrigieron errores sintácticos.
De esta manera, puede observarse que varios ejemplos que se consideraban correctos en realidad **no** lo son. En caso de **no** existir ***"ASLX"***, la herramienta creada, dichos ejemplos serían considerados válidos.

8 **Gomes Soares, Luis Fernando.** *"Programando em NCL": Desenvolvimento de aplicações para middleware Ginga, TV digital e Web*. 2009.

Mas allá de que la herramienta desarrollada permite detectar errores sintácticos para lograr aplicaciones totalmente validas, como el objetivo principal es el ámbito semántico, solo se mostrara en este informe un ejemplo de como muestra "ASLX" un detalle de los errores sintácticos encontrados.[9]

Ejemplo 3.37

Es la primera aplicación con efectos de animaciones y transiciones. Dicho ejemplo puede verse en la página numero 85 del libro oficial.

Código donde se encuentra el error (linea 80)**:**

```
<link id="lOff" xconnector="conEx#onKeySelectionStopSetStart">
        <media id="enForm" src="../media/enForm.htm" type="text/html"
descriptor="formDesc"/>
</switch>
```

Tipo de error:

```
The element type "link" must be terminated by the matching end-tag "</link>".
```

Solución:
Agregar la sintaxis correspondiente para indicar el fin del elemento <link>.

```
<link id="lOff" xconnector="conEx#onKeySelectionStopSetStart">
    <media id="enForm" src="../media/enForm.htm" type="text/html"
    descriptor="formDesc"/>
</link>
```

5.4.5 Errores semánticos en ejemplos oficiales de Ginga-NCL

Se detectaron los siguientes errores presentes en los ejemplos de Ginga-NCL. Dichos errores fueron encontrados por el *validador semántico* creado. Como puede observarse, varios ejemplos que no tenían errores en el ámbito sintáctico, si los tienen en el aspecto semántico. En caso de no existir la herramienta creada, dichos ejemplos serían considerados válidos cuando en realidad no lo son.

Ejemplo 1

Este ejemplo es la versión inicial del documento NCL de O Primeiro João. Dicho ejemplo puede verse en la página nro 51 del Libro oficial.

Código donde se encuentra el error (linea 36)**:**

```
file:Ejemplo1.ncl: line:36: The symbol "conEx#onBeginStartDelay" is not declared in this scope.
file:Ejemplo1.ncl: line:45: The symbol "conEx#onBeginStart" is not declared in this scope.
file:Ejemplo1.ncl: line:49: The symbol "conEx#onBeginStart" is not declared in this scope.
file:Ejemplo1.ncl: line:53: The symbol "conEx#onEndStop" is not declared in this scope.
```

Solución:

9 Para mas detalle referirse al informe completo de la herramienta desarrollada "ASLX".

Definir un nuevo archivo causalConnBase.ncl, el cual contiene cada uno de los conectores necesarios para la compilación de los ejemplos presentes en Ginga-NCL. En este caso específicamente se definen en dicho archivo, los *connectores* "*onBeginStartDelay*", "*onBeginStart*" y "*onEndStop*".

A continuación se mostrara una tabla con los demás ejemplos presentes en el libro oficial de GINGA_NCL que poseen errores de tipo semántico.

Ejemplo	**Descripción**	**Linea del error**	**Tipo de error**	**Solución**
Ejemplo 3.15 (Pagina 56)	Aplicación con sincronismo por interacción	41-50-54-58-63-67-77	Elements are not declared in this scope.	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 3.19 (Pagina 63)	Aplicación con Rehúso	43-47-57-64-73-77-81	Elements are not declared in this scope.	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 3.22 (Pagina 67)	Aplicación con canal interactivo	47-51-62-69-78	Elements are not declared in this scope.	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 3.27 (Pagina 73)	Aplicación con adaptación de contenido.	60-64-75-82-91-95-99	Elements are not declared in this scope.	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 3.32 (Pagina79)	Aplicación con control de anuncios interactivos.	53-60-66-70-98-100-103-114-121-130-134-138	Elements are not declared in this scope. Invalid value in attribute ***"role"***	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes. Dar un valor válido al atributo "***interface"*** y al atributo "***role"***.
Ejemplo3.37 (pagina 85)	Aplicación con efectos de animaciones y transiciones	60-67-77-80-85-90-120-127-136-140	Elements are not declared in this scope. Invalid value in attribute ***"interface"***	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 3.45 (Pagina 95)	Aplicación para el menú con las teclas de navegación	72-79-89-117-122-133-139-167-174-185-194-198-207-213	Elements are not declared in this scope.	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 3.52 (Pagina 105)	Aplicación con objeto NCLua	60-74-81-91-120-122-125-136-142-156-175-185-199-208-212-221-229	Invalid value in attribute ***"end"*** y ***"role".*** Elements are not declared in this scope.	Asignar un valor válido al atributo *"end", "role".* Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 7.1 (Pagina 144)	Ejemplo donde se muestra la interactividad de un "boton" para un tiempo fijado	18	The symbol ***"dTVtelaInteira"*** is not declared in this scope <even forward>.	Escribir el descriptor de forma correcta.
Ejemplo 10.12 (Pagina 199)	Aplicación que muestra un menú cuando el usuario presiona el botón rojo	22-30	Invalid value in attribute ***"delay"*** y ***"key".***	Asignar un valor válido al atributo "***delay"*** y al atributo ***"key".***

	del mando a distancia			
Ejemplo 10.14 (Pagina 202)	Conector "onKeySelectionStartStop" que ilustra la composición de las acciones.	22-30	Invalid value in attribute ***"delay"*** y ***"key".***	Asignar un valor válido al atributo "***delay"*** y al atributo ***"key".***
Ejemplo 12.4 (Pagina 234)	Definición de los metadatos en diferentes partes de un documento NCL		Elements are not declared in this scope.	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 14.4 (Pagina 259)	Objeto de codificación hipermedia NCLAdvert.	28-35	Elements are not declared in this scope.	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 14.4 (Pagina 268)	Aplicación con múltiples dispositivos de visualización	51-60-66-72-76-80-84	Elements are not declared in this scope.	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 15.3 (Pagina 272)	Documento arranque de propaganda.	28-37	Elements are not declared in this scope.	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.
Ejemplo 15.6 (Pagina 274)	Iniciación y control al objeto NCLAdvert		Elements are not declared in this scope.	Definir en el archivo ***causalConnBase.ncl*** los conectores correspondientes.

6. Bibliografía

1. **Gomes Soares, Luis Fernando.** "Programando em NCL": Desenvolvimento de aplicações para middleware Ginga, TV digital e Web. 2009.

2. **Associacao Brasileira de Normas Técnicas.** "*ABNT NBR 15606-2"*: Televisão digitalterrestre – Codificação de dados e especificações de transmissão para radiodifusão digital.

Parte 2: Ginga-NCL para receptores fixos e móveis – Linguagem de aplicação XML para codificação de aplicações. 2007.

3. **Gomes Soares, Luis Fernando and Ferreira Rodrigues, Rogerio.** "Nested Context Language 3.0 , NCL Digital TV Profiles". 2006.

4. **Soares, Luiz Fernando Fernando Gomes.** "Construindo Promgramas Audiovisuais Interativos Utilzando a NCL 3.0 e a la Ferramentea Composer". 2007.

5. **Zambrano, Arturo.** Introducción a la TV Digital Interactiva y Ginga.ar - Lifia -Universidad Nacional de La Plata.

6. **Balaguer Federico & Isasmendi, Leonardo.** "Desarrollo de Aplicaciones para Televisión Digital". Rio2011 - Universidad Nacional de Río Cuarto. 2011.