aNa: API for NCL Authoring

Joel A. F. dos Santos
Laboratório MídiaCom,
Instituto de Computação,
Universidade Federal
Fluminense
Niterói, RJ, Brazil
joel@midiacom.uff.br

Julia V. da Silva
Laboratório MídiaCom,
Instituto de Computação,
Universidade Federal
Fluminense
Niterói, RJ, Brazil
julia@midiacom.uff.br

Renan R. Vasconcelos
Programa de Engenharia de
Sistemas e Computação,
COPPE, Universidade Federal
do Rio de Janeiro
Rio de Janeiro, RJ, Brazil
renanrv@cos.ufrj.br

Wagner Schau
Programa de Engenharia de
Sistemas e Computação,
COPPE, Universidade Federal
do Rio de Janeiro
Rio de Janeiro, RJ, Brazil
schau@cos.ufrj.br

Cláudia Werner
Programa de Engenharia de
Sistemas e Computação,
COPPE, Universidade Federal
do Rio de Janeiro
Rio de Janeiro, RJ, Brazil
werner@cos.ufrj.br

Débora C. M. Saade
Laboratório MídiaCom,
Instituto de Computação,
Universidade Federal
Fluminense
Niterói, RJ, Brazil
debora@midiacom.uff.br

ABSTRACT

There are several available NCL (Nested Context Language) authoring and formatting tools using their own metamodel to represent the code they are working on. This paper presents an API that implements a metamodel specifically created to represent NCL documents. This API helps the creation of tools to manipulate NCL documents and brings some benefits to code reuse to the Digital TV Systems development context. The API here presented is called aNa, an acronym for API for NCL Authoring. aNa is available for free download and open for contributions. aNa has already been used for the development of some NCL authoring and analysis tools.

1. INTRODUCTION

Nested Context Language (NCL) is the standard declarative language of the Brazilian Digital TV system [1] and ITU standard for IPTV services [8]. The growth in the use of NCL for the creation of interactive content shall increase the need for tools to help interactive application creation. Those tools can be authoring tools, analysis tools and even presentation tools.

Usually, each tool that has been created to manipulate NCL code implements its own metamodel to represent the code it is working on. If one does not create a model to represent the document, it is common to use available XML parsing tools, like DOM (Document Object Model) [4] or SAX (Simple API for XML)[1]. Although those parsers produce a good result and help the tool developer to manipulate NCL code, their purpose is generic, allowing the parsing of any XML document.

This paper presents a metamodel specifically created to represent NCL documents in a model-oriented environment and help the creation of tools to manipulate NCL code. This metamodel is implemented in Java by an API called aNa, an acronym for API for NCL Authoring. aNa is available for free download and is open for contributions. aNa has already been used for the development of some NCL authoring and analysis tools.

The remaining of this paper is structured as follows. Section 2 summarizes available tools to help NCL authoring. Section 3 presents aNa structure. Section 4 discusses the API implementation and its main characteristics that facilitate the creation of NCL tools, like code reuse. Section 5 presents some tools that already use aNa and Section 6 concludes the paper and presents future work.

2. NCL AUTHORING TOOLS

Currently there are some available tools for helping the authoring of NCL documents. Those tools present different capabilities, focusing on different user profiles. The following paragraphs present those NCL authoring tools.

Composer 3 [9] provides a single authoring environment suitable for different user profiles, from home users to content producers. Composer 3 proposes the basis for building an integrated environment that can adapt itself to various profiles and support non-functional requirements.

NCL-Eclipse [2] is an authoring tool, available as an Eclipse plugin, which helps the author creating an NCL document. It presents some facilities to help coding as: code completing, code highlighting and errors and warnings highlighting.

NCL-Inspector [7] is a tool based on other tools for code quality critique, which supports the authoring of NCL applications. It supports the author in terms of code quality.

XTemplate 3.0 [6] defines a language and tools for the authoring of NCL documents using composite templates. Composite templates define generic structures of nodes and links that can be reused in different document compositions, facilitating the authoring of interactive applications in Dig-

[1]http://www.saxproject.org/

WebMedia '12, October 15-18, 2012, São Paulo, SP, Brazil

ital TV systems.

Berimbau iTV Author [3] is a graphical authoring tool for digital TV applications aimed at media professionals who have no programming knowledge. The tool provides a simple and intuitive interface and a media repository to be used while creating the application.

Notice that each one of those tools uses its own metamodel, in an informal manner, to represent NCL documents. If there were an NCL reusable API that implemented an unified metamodel available to model and represent NCL documents, the programming effort to develop NCL tools would be much smaller. This paper proposes such API.

3. ANA METAMODEL

aNa was created to represent an NCL document as a model. Its structure is optimized so the author of NCL tools that manipulate XML code does not need to worry about the language representation, but with the model itself. Every NCL element is represented as a class, which will be called element class. An element class contains the same attributes of the NCL element it represents. Every element class in aNa inherits from the basic type *NCLElement*.

The element classes follow the same hierarchy of the NCL elements, therefore, an element class will be associated to all element classes that represent its child elements. The cardinality of those associations is defined following the NCL language specification. By representing parent-child relations as associations, aNa makes it possible to navigate from an element to its children and the opposite as well. Figure 1 presents the related representation in aNa of the element <*ncl*>, represented by class NCLDoc, which has attributes *id*, *title* and *xmlns* and the elements *head* and *body* as children.

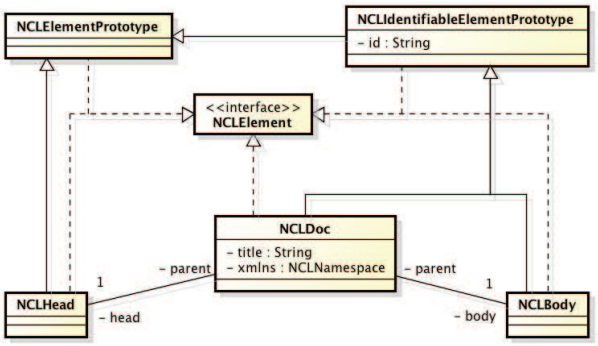

Figura 1: NCLDoc class representation

NCL also defines attributes that refer to other elements. This reference is usually done by defining the attribute value equal to the referred element id (more common) or another attribute of the referred element. In order to improve navigation over element classes, aNa represents those references as associations between them. Listing 1 presents an example of reference between elements and Figure 2 presents how those elements are represented in aNa.

Listing 1: Element reference example

```
1 <regionBase>
2     <region id="reg1"/>
3 </regionBase>
4 <descriptorBase>
5     <descriptor id="desc1" region="reg1"/>
6 </descriptorBase>
```

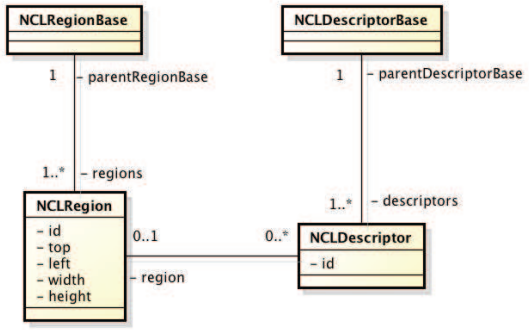

Figura 2: Element reference representation

Listing 1 presents an NCL document part. The example defines a <*regionBase*> element that has a <*region*> element as child. The <*region*> element *id* attribute is a String that represents its identification. The example also defines an element <*descriptorBase*>, that has a <*descriptor*> element as child. The <*descriptor*> defines two attributes, *id* and *region*. The *id* attribute represents its identification and the *region* attribute is a String that represents the identification of the <*region*> used by that <*descriptor*>. In Figure 2, it is possible to observe that the *region* attribute of the <*descriptor*> is represented as an association between the *NCLDescriptor* and *NCLRegion* classes.

Some element attributes may have a value from a specific value set, like the *xmlns* attribute. In those cases, aNa defines the attribute type as an Enumeration with all the possible values for that attribute (see *xmlns* in Figure 1). Sometimes an attribute value can have more than one type. For example, consider the elements presented in Listing 2.

Listing 2: Attribute value examples

```
1 <region id="reg1" top="10" left="10"/>
2 <region id="reg3" top="10.5%" left="10%"/>
```

Notice that the element <*region*> has attributes that may be an integer without a percent sign (%) or a integer or a double with a percent sign. NCL also defines other elements whose attributes can be numbers or strings, as the *max* attribute of a connector condition, where it can be a positive integer or the string "unbounded"; elements whose attributes can be a value (like a string, a number, etc) or another element representing a parameter, as the *delay* attribute of a connector action, where it can be a double or a reference to a connector parameter element.

In aNa, those attributes are defined with type *Object*. So, they can receive any of the possible values the attribute demands. When receiving a new value, the API tests if its type is correct. If not, aNa raises an error indicating the values the attribute may receive. In those cases, if the value received is an string, aNa tries to parse the value to the format used by NCL.

Figure 3 presents a fragment of the API structure, representing relations among elements of the document body. In order to simplify the diagram, the figure presents only class names and associations between them.

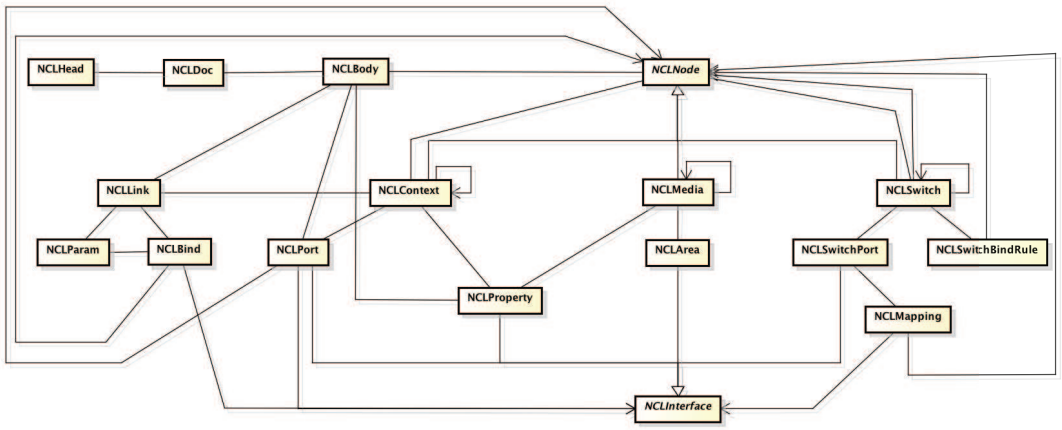

Figura 3: aNa document body structure

4. ANA IMPLEMENTATION

aNa is implemented in Java, providing portability in different platforms. It presents methods to get and to change the value of an element class attribute, to navigate through the NCL document, to create Java objects from an NCL document and to write an NCL document from those objects.

aNa provides facilities to increase reusability in the development of Digital TV applications. NCL itself already allows code reuse, since the same elements can be used in different contexts, without the need to redefine them. Because of the Java implementation, the benefits of object oriented design, such as allowing reuse by class inheritance or object composition [12], help increasing the application development productivity in comparison to direct NCL programming.

The document parsing is done using DOM [4]. aNa walks through the DOM tree gathering information about the NCL elements and creating Java objects that represent them. During that object creation, aNa already creates references between objects. For example, if aNa finds the value "reg1" in the attribute *region* of a <*descriptor*> element, it will search for a <*region*> element in the region base with that *id* in order to create this reference and it will raise an error if no <*region*> with that *id* is found.

The search for a referred element is done only in the attribute scope. That is, if an attribute indicates the *id* of an element inside the same context, aNa will search for that element only inside the context. For example, suppose a <*port*> element that defines *component* and *interface* attributes. aNa will search for an NCL node with the *id* defined in the *component* attribute inside the port parent context. Once that element is found, aNa will search for an interface with the *id* defined in the *interface* attribute inside that node.

During parsing, aNa gathers from the DOM representation of an NCL element only the information that makes sense to it, that is, the attributes and child elements defined in the language specification. Also, when reading an NCL document, DOM already verifies if the XML document is well written, that is, all the XML tags are opened and closed correctly. So, after the document parsing, aNa will have a consistent document representation. If a wrong definition is found in the NCL document, aNa will raise errors. Listing 3 presents an error example. Errors always present the whole path from the root document element to the element where the error occurred. It also shows a message that informs the error found to the author.

Listing 3: Parsing error example

```
1 Error parsing Head > ConnectorBase >
      CausalConnector(onKeySelectionStop) >
      SimpleCondition
2 Could not find a param in connector with name: tecla
```

The opposite way, that is, create an NCL document from the aNa representation is done by getting, from each Java object, its XML representation. The method that implements it returns a String with the XML element representation. It is worth to highlight that the code returned is indented, making the document reading easier.

Once aNa is developed to be used by tools that manipulate NCL code, it is able to notify the tool that uses it about a modification in an element class. This notification can help the tool maintaining a consistent document representation. For example, suppose a tool that is built to graphically show all document regions. Every time a modification occurs in the position of any region, the tool is notified, so it is able to apply the necessary changes in the graphical position of the modified region. The API has a *ModificationNotifier* which may have one or more *ModificationListeners*. The notifier will send notifications when the value of an attribute is set and a child element is added or removed. As this feature may not be necessary for all kinds of tools, the tool is not obliged to implement the *ModificationListener*.

Also, in order to maintain a consistent document representation, once element references are represented as associations, when removing an element from its parent, aNa verifies if such element is used in a reference. If so, aNa indicates to the author the elements that make reference to it and disable its removal until the references are removed.

Another characteristic of aNa is that it is implemented using parameterized classes, which is done using the Generics Java language feature[2]. Using that feature, aNa element classes extensions are simpler, requiring less coding effort.

[2]more information available in http://docs.oracle.com/javase/tutorial/java/generics

5. ANA USE CASES

As mentioned before, aNa was developed to help tools modeling and manipulating NCL code. One use case is a graphical editor that was built for helping users creating NCL connectors [10]. The editor uses aNa to read an NCL document and extract the <*connectorBase*> element. This element represents a base containing all connectors that may be used in <*link*> elements. Once the editor has the object, created by aNa, representing the connector base, it can use aNa methods to get necessary information from the element and to graphically show the connector to the author. The editor also allows the author to perform creations, removals and modifications on connector child elements. All these modifications are performed through aNa methods. Moreover, the editor allows the author to create a new connector base document. The final NCL document code is also created by aNa.

Another tool that uses aNa is called NEXT (NCL Editor supporting XTemplate) [11]. It is a graphical authoring tool developed to facilitate the creation of digital TV applications using the NCL language and supporting the use of composite templates. Its architecture is based on a core that is able to communicate with plugins and is completely independent from them. NEXT plugins are allowed to manipulate the document and, when a plugin makes any modification on it, NEXT notifies other plugins about the modification. In order to obtain knowledge about the change, NEXT uses aNa's feature that notifies about changes occurred in the document. Moreover, NEXT uses aNa objects to create a tree model representing the nesting structure of the NCL document. aNa also enables NEXT to open and to edit any standard NCL document.

aNa is also used as the basis for the development of an NCL document validation tool. That tool, called aNaa - API for NCL Authoring and Analysis, extends aNa by adding methods that allow the validation of the NCL document being authored [5]. It uses aNa for reading an NCL document and gathering information about document elements. After that, aNaa creates different representations of the NCL document, which allow the tool to investigate some document properties and verify its temporal consistency.

6. CONCLUSION

In general, when developing tools to manipulate NCL code, developers have to implement their own metamodel to represent the code to be manipulated. Sometimes no metamodel is created at all. Since those tools need to read an NCL document, it is common to use available XML parsing tools, like DOM or SAX. Although those parsers produce a good result and help the tool developer to manipulate NCL code, their metamodel is generic for any XML document.

This paper presented aNa, an API that provides a metamodel created specifically for representing NCL code and helping the creation of tools that manipulate NCL documents. aNa considers NCL attribute types and verifies if the document follows NCL syntactic and reference rules as defined in the Brazilian standard [1]. Besides API specificities, with a common core that represents an NCL document, aNa makes it possible to exchange object-oriented data among different tools without the need to generate XML code.

Currently, aNa is being used as a basis for the creation of authoring and validation tools. Its code is available for free download[3] and the tool is also open for contributions. We intend to continuously improve aNa, based on feedback from the NCL developers community.

In the current version of the API, error messages are presented in English. A future work is the creation of aNa error messages in different languages. Another future work is to improve the API to receive NCL live editing commands and to produce the necessary modifications in the document.

7. ACKNOWLEDGEMENTS

This work was partially supported by CNPq, FAPERJ and CAPES.

8. REFERÊNCIAS

[1] ABNT. Digital terrestrial television - Data coding and transmission specification for digital broadcasting - Part 2: Ginga-NCL for fixed and mobile receivers - XML application language for application coding, 2011.

[2] R. G. A. Azevedo, M. M. Teixeira, and C. S. S. Neto. NCL Eclipse: Ambiente Integrado para o Desenvolvimento de Aplicações para TV Digital Interativa em Nested Context Language. In *Salão de ferramentas - SBRC*, 2009.

[3] Batuque TV digital. Berimbau iTV Author. http://www.batuque.tv/, 2011.

[4] W. W. W. Consortium. Document Object Model (DOM) Level 2 Core Specification, 2000. W3C Recommendation DOM-Level-2-Core-20001113.

[5] J. A. F. dos Santos. Multimedia and hypermedia document validation and verification using a model-driven approach. Master's thesis, Universidade Federal Fluminense, 2012.

[6] J. A. F. dos Santos and D. C. Muchaluat-Saade. XTemplate 3.0: spatio-temporal semantics and structure reuse for hypermedia compositions. *Multimedia Tools and Applications*, 2011.

[7] G. S. C. Honorato and S. D. J. Barbosa. NCL-Inspector: Towards Improving NCL Code. In *ACM SAC*, pages 1946–1947, 2010.

[8] ITU. Nested Context Language (NCL) and Ginga-NCL for IPTV services. http://www.itu.int/rec/T-REC-H.761-200904-P, 2009. ITU-T Recommendation H.761.

[9] B. Lima, R. Azevedo, M. Moreno, and L. Soares. Composer 3: Ambiente de autoria extensível, adaptável e multiplataforma. In *WebMedia – Workshop de TV Digital Interativa (WTVDI)*, 2010.

[10] J. V. Silva and D. C. Muchaluat-Saade. Editor Gráfico de Conectores Hipermídia para Linguagem NCL 3.0. In *WebMedia*, 2011.

[11] J. V. Silva and D. C. Muchaluat-Saade. NEXT - Editor Gráfico para Autoria de Documentos NCL com Suporte a Templates de Composição. In *WebMedia*, 2012.

[12] S. Srinivasan and J. Vergo. Object Oriented Reuse: Experience in Developing a Framework for Speech Recognition Applications. pages 322–330, 1998.

[3] https://github.com/joeldossantos/aNa