J Braz Comput Soc (2013) 19:235–256
DOI 10.1007/s13173-013-0110-1

ORIGINAL PAPER

Incremental validation of NCL hypermedia documents

Rodrigo Costa Mesquita Santos · José Rios Cerqueira Neto · Carlos de Salles Soares Neto · Mário Meireles Teixeira

Received: 22 August 2012 / Accepted: 23 April 2013 / Published online: 18 May 2013

Abstract This paper proposes an incremental, structural and contextual validation method for Nested Context Language (NCL) documents. As part of the proposed method, we define a declarative metalanguage to ensure low coupling between NCL structure and the validator code, which simplifies the validation of new language profiles. Requirements such as incremental processing and multilingual messages are also covered by this work. We present an implementation of this method using component architecture as a proof of concept and also conduct a performance evaluation to compare the traditional and incremental validation approaches.

Keywords NCL · Incremental validation · Code validation · Ginga · ISDB-Tb · iDTV

This is an extended version of the paper "Método de Validação Estrutural e Contextual de Documentos NCL", presented as one of the Best Papers of Webmedia 2011 (Brazilian Symposium on Web and Hypermedia Systems).

R. C. M. Santos · J. R. C. Neto
Post Graduate Program in Computer Science,
Federal University of Maranhão, Av. dos Portugueses,
s/n, Campus do Bacanga, CEP 65080-040 São Luís, MA, Brazil
e-mail: rodrim.c@laws.deinf.ufma.br

J. R. C. Neto
e-mail: rios@laws.deinf.ufma.br

C. de Salles Soares Neto · M. M. Teixeira (✉)
Department of Informatics, Federal University of Maranhão,
Av. dos Portugueses, s/n, Campus do Bacanga,
CEP 65080-040 São Luís, MA, Brazil
e-mail: mario@deinf.ufma.br

C. de Salles Soares Neto
e-mail: csalles@deinf.ufma.br

1 Introduction

Digital TV applications are those in which an application is sent by broadcast together with extra media contents to viewers, enriching the user experience. With digital TV, soap operas, reality shows, auditory programs and TV news allow the viewers to access extra content through interactive multimedia applications.

The authoring of these applications is basically done based on two main paradigms: imperative and declarative. In the former, it is necessary an algorithmic description of the entire presentation and of the relationships among media, what is commonly made with use of general-purpose languages, such as Java. In the latter, domain-specific languages (DSL) are frequently used, whose main concern is the final author's description. When there is a DSL that matches the domain of a given problem, its use, in general, implies in reduction of the authoring time when compared to general-purpose languages [1], because one single DSL sentence can have an equivalent meaning to several imperative ones. Some declarative languages commonly used to hypermedia authoring are XHTML [2], SMIL [3] and NCL [4], all of them being XML-based.

When one use a development environment for designing and implementing a hypermedia application, in general, the effort spent during the authoring is reduced compared with non-specialized authoring tools. This occurs because authoring tools implement a set of features that guides authors throughout development process. Thus, an Integrated Development Environment (IDE) is an environment that can provide both graphical abstractions for users who feel more comfortable with visual authoring, and useful features for textual authoring. In [5], a study is done on visual and textual authoring, explaining the advantages and disadvantages of each of these two approaches. A functionality provided by

both graphical and textual IDEs is code validation. Validation can be understood as a routine that checks whether a given instance of source-code has any syntactic or semantic errors regarding the grammatical rules of the chosen programming language. At authoring time, this validation may help programmers to identify programming mistakes more quickly, which may result in a reduction of the time needed to develop the application.

Regardless of the approach chosen, the authoring of hypermedia documents is essentially incremental. Thus it is likely that, while you are editing the application, only a small number of changes have been made between two consecutive code verifications. There are textual authoring tools, for example, that validate the code whenever the user stops typing, although it is possible that the difference between one and another checking is of just a single character. In visual authoring, on the other hand, validation may be performed after the addition, editing or removal of a given visual component. It is noticeable that in both situations it is not strictly necessary to reconstruct the entire data structure representing that instance of code and validate it completely. This creates the opportunity to manage such a structure in memory and perform the validation only in the modified and affected parts of the document. This type of validation that checks only the pieces of code changed and those affected by the changes is commonly referred to in the literature as *incremental validation* [6].

In the context of digital TV systems, we can intuitively see the influence of incremental validation in the conventional process of authoring, but this is not the only situation where such validation can offer performance gains. The fact that set-top boxes (device that executes interactive applications) are environments with limited computing resources raises the need for validation techniques that avoid redundant checks in such a limited hardware. A particular case of this need is the validation of applications created within the set-top box in TV viewers' households (Social TV) [7], since this validation must be done without affecting performance. On the other hand, systems such as ISDB-Tb [4] allow broadcasters to send editing commands to applications while they are being displayed, allowing what is known as live editing of content [8]. Before applying such edits, first it is necessary to validate them in order to make sure that the editing commands will result in consistent, accurate and coherent documents. From the above discussion, we can notice that the need for incremental validation mechanisms is present in various circumstances.

Nested Context Language (NCL) is a modular declarative hypermedia language focused on the definition of synchronization relationships between media objects. It is the standard language of ISDB-Tb for development of interactive applications and is also an international ITU-T recommendation for IPTV [9]. The complete validation of NCL documents cannot be made through traditional XML validation techniques, since NCL has a set of particularities that are not supported by these techniques. Section 3 brings up some of these particularities.

This work identifies several non-functional requirements that an NCL validation proposal must meet. Since NCL has been adopted by several countries in Latin America and in Africa as the standard for creation of interactive applications in open TV, a first requirement, which can be easily met, is the availability of multilingual error messages in the validation engine. Also, the code validation must be easily integrated to authoring tools, allowing its reuse in several environments. Thus, we come across our second requirement: reuse support and technological independence.

The fact that NCL is a modular language allows the defined modules to be grouped into profiles. For example, we cite two NCL profiles defined in the digital TV context: EDTV (Enhanced Digital TV) and BDTV (Basic Digital TV). So, the support to adjustment of the validation process to different profiles of the language is another requirement. Digital TV receivers have insufficient computational resources, with respect to both memory and processing. So, a digital TV middleware running in one of those receivers has limited computational power to execute interactive applications. An interesting approach under such conditions would be the middleware itself validating the code of the interactive application received, avoiding the possibility of running corrupted applications that would just waste resources. However, this validation process must be efficient, or it would be an obstacle, spending more resources than the trial of running a faulty application. This scenario just reinforces a fourth requirement: efficiency.

This work presents a detailed proposal for incremental validation of NCL documents, which meets the non-functional requirements raised in the previous paragraphs and carries out a study in order to quantify the gain of the incremental validation, compared to the traditional method. This validation proposal is also generic, that is, it can be adapted to various authoring environments, both visual and textual, and has a component-based architecture, allowing components to be replaced by others with the same interface, but still keeping the whole functional process.

This paper is organized as follows: Sect. 2 presents some related work. Section 3 discusses the particularities of the validation of NCL documents and presents the metalanguage used to support the incremental validation proposed herein. Section 4 illustrates an implementation of the proposed method as a proof of concept. Section 5 presents the integration between our implementation and the IDE Composer. Section 6 discusses a performance analysis, making a comparison between a non-incremental and an incremental approach. Finally, Sect. 7 brings the conclusions of this work.

2 Related work

As this paper presents an incremental validation mechanism that can be integrated to other authoring tools, this section is organized as follows: in Sect. 2.1 some source code validation mechanisms of different authoring tools are presented with focus on XML validation techniques and Sect. 2.2 refers to the XML incremental validation discussion, highlighting some works found in literature that address to this topic.

2.1 Source code validation in authoring tools

When one analyzes the several existing authoring environments for applications development, we notice that most of them has some built-in scheme to validate source code, and such schemes may vary from the simpler, which just performs the lexical verification of the keywords of the language [10], to those which implement more complex methods, such as code correction suggestion [11].

In the scope of imperative languages, it can be noticed that several authoring tools implement various code validation methods. Lua Eclipse [10] is a plug-in for the Eclipse IDE [12] designed for the development of Lua scripts. This environment features few validation resources, doing not much more than the verification of keywords of the language. Other tools, like CDT [13]—C/C++ development plug-in for Eclipse—and Visual Studio—proprietary development tool designed to the .NET platform—go beyond this simple verification and approach other syntactical and semantic aspects of the code. JDT [11]—another Eclipse plug-in, which offers a development environment for the Java language—even implements a scheme for suggestion and application of corrections in the source code.

For the code validation of declarative languages derived from XML, the most trivial is to think of using XML Schema validation [14], since it specifies validation rules for the elements and attributes of the language. There are several tools that implement this validation method. Intelligent Knowledge Management Environment (IKME) [15] is a static code validation tool that identifies syntactical errors with basis just on XML Schema of the language. XML Screamer [16] is another tool that uses XML Schema to validate XML documents and has an architecture that tries to optimize this process. Such optimizations are possible because of the integration between the SAX parser and the XML deserialization. For example, in a traditional scenario, a SAX parser throws an event whenever it finds a start tag. Then the deserialization catches this event and converts the information of the start tag for some business class related to the application. The XML Screamer integrates the scanning, parsing, validation and deserialization process in a single low level tool, making unnecessary creating and catching SAX events. It also carries out performance evaluations in order to prove the gain when using that approach. Finally, there are also validation libraries that use XML Schema, like the standard Java library for XML treatment and the Xerces API (Application Programming Interface) [17].

Despite being an XML-based language, the NCL code validation using this approach is insufficient, since NCL has some specificities not covered by XML Schema. NCL defines several references among its elements, and some of these references are only valid if both elements are in the same composition (feature called "perspective-based scope"). This particularity cannot be described using only the XML Schema. In practice, validating an NCL code using just XML Schema neglects some important referential and contextual particularities inherent to NCL. Section 3 presents more NCL particularities not covered by the XML Schema.

There are other validation processes for XML-based languages not based on XML Schema. SMIL Author [18], for instance, is an authoring environment that employs validation techniques not based on it and is able to identify temporal inconsistencies in the documents. It used Real-Time Synchronization Model (RTSM) as internal data model. SMIL Builder [19] is another authoring tool that does not use XML Schema and performs incremental validation on SMIL documents (in the next section this tool is described in more details). Because of the XML Schema limitation on describing some NCL features, our approach also does not use it and defines a proper metalanguage to describe NCL.

The two most important NCL authoring tools are NCL Eclipse [20] and Composer [21]. Both tools use NCL Validator [22] as validation component. The NCL Validator is an NCL document validation process consisting of three stages: (i) lexical validation; (ii) structural validation; and (iii) validation of contexts and references. Stage (i) checks whether the document structure is in accordance with the rules of a well-formed XML. Stage (ii) validates structural aspects, like the presence or absence of mandatory attributes or children. Finally, stage (iii) checks whether the references in the document are in accordance with the perspective-based scope schema defined for NCL. This tool also features support to multilingual error messages.

Besides performing an efficient validation process on NCL documents, some requirements met by the present proposal are not met by NCL Validator. As it does not make a clear separation of the language structure and the application source code, the NCL particularities are not generalized, resulting in the validation of each element of the language as a particular case. In practice, for each language element is defined a class and the referential checking is hardcoded on it. The perspective-based scope is, perhaps, the clearest example of one of the particularities that are not generalized in the source code. This non-separation of the structure of the language brings a second inconvenience: in order to validate different

profiles of the language, it is needed to change the source code and make a new compilation of the tool.

NCL Validator uses a DOM tree [23] as internal data modeling structure. This means that an authoring tool using NCL Validator needs to convert the NCL code instance to the respective DOM tree that represents it every time the code is checked (once the validator does not maintain state between validations), which demands more computational cost to the process. The process would possibly be improved if the authoring tool could work with a DOM tree to keep the document structure. This approach brings, however, some inconvenience. First, this forces the authoring tool to use the same data modeling used by NCL Validator. The ideal situation is when the validation tool is adapted to the authoring tool, not the contrary. A second inconvenience of this approach is the cost of keeping a DOM tree in memory always updated. This problem is particularly hard to solve when we deal with textual authoring, when it is impracticable to build a DOM tree while the document is not well formed, making the validation unfeasible.

As a third undesirable characteristic, NCL Validator neither employs nor allows any kind of technique that enables a partial or incremental validation on the NCL document. So, we notice that, despite focusing on the same target language, NCL Validator and the method proposed in this paper have significant differences in the validation process.

Dos Santos, in [24], discusses the validation and verification of hypermedia documents focusing on structural, referential and behavioral checks. Structural check consists in analyzing whether a given document satisfies the syntactic rules defined by the language grammar. An example of structural validation is the checking if a given element can be child of its element parent. Referential check consists in analyzing whether references between elements are valid. Behavioral check investigates whether the hypermedia document created describes a temporal scenario free of errors. For example, consider two elements A and B. If the end of element A ends element B, it is important that A ends after B starts. Another example of behavioral check is the verification if there is some node that is never started. An API called aNaa (API for NCL Authoring and Analysis) was developed following the MDA (Model Driven Architecture) approach which implements both structural and behavioral validation in NCL documents. Our approach is addressed for structural and referential validation and not for behavioral check. As in [24] is not covered the incremental validation topic, we assume that aNaa does not perform this type of validation.

Finally, we also cite NCL Inspector [25], which is an NCL code reviewing tool that allows the author to define new rules to be validated. These rules can be defined using the Java programming language or using XSLT documents. This tool, however, does not validate NCL code; it just validates the defined rules. This work does not focus on supporting specific validation rules defined by users, but in establishing general validation rules to NCL documents.

2.2 XML incremental validation

There are several works in literature that discuss XML incremental validation in many different scenarios. The fact that XML has become the de facto standard for storage and exchange of information in modern computer systems has increased the need for more efficient validation techniques of such documents. In this subsection is discussed some of these works in several contexts.

In [26] is proposed a validation approach based on an incremental data model called SXESP (Simplified XML Element Sequence Pattern) that checks whether a given XML document conforms to the rules expressed in XML Schema. The SXESP represents the XML tree through a sequence of elements using some operators to connect elements that have structural relations. An update **U** transforms the document **D** in a new document **D′**. The first step of incremental validation is to find the root element of the subtree to be validated. If **U** is the removal of the element **E**, then the subtree generated from **E** is moved to its parent and it is validated. If **U** is the editing of the element **E** or the adding of a subtree rooted in **E**, then the validation is performed from **E** and to its descendants. In order to persist the new document **D′** it is necessary that **U** to be a valid update, otherwise **U** is discarded and the system rollbacks the document to its original state.

Vianu and Papakonstantinou [6] present a method for incrementally check XML documents. The method analyses a given document with respect to a DTD or XML Schema and uses complex data structures to represent the document (a balanced tree is used for storing the children of each element). That work is focused on database management applications and its main concern is the structural check, ignoring any referential check between elements. The main issue addressed in this paper is the incremental validation of NCL documents. As discussed in Sect. 3, NCL has several particularities that ask for more sophisticated validation methods than those that only perform structural checks in the documents.

Thao and Munson [27] use the incremental approach to perform a three-way merge on XML documents in a versioning system. Each document version is represented by a delta (the changes between the original document and the derived one) and the merging algorithm creates a result document based on the original document and the deltas. When any modification is made in any node, the system marks that node. Thus, for presenting a specific document version, the system takes the original document, the list of marked elements and the delta of that version to compute the result document. That work carries out a performance evaluation of the proposed algorithm against commercial tools, prov-

ing the former to be faster, with lower memory utilization and more precise. In our proposal, we also mark modified elements to identify which of them need to be validated. In addition to marked elements, we use the concept of affected elements to identify those that were not directly modified, but may be affected by changes in the document.

In the context of authoring, the SMIL Builder [19] is an example of tool that performs incremental check in order to verify the temporal consistence of SMIL documents. Like NCL, SMIL is a hypermedia language focused on the definition of synchronism relations among media composing the application. This environment uses a structure called H-SMIL-Net [28] (a temporal extension of the Petri Net structure [29]) for internal representation of the SMIL document, which offers a structured data representation model. So, changing a part of the application code does not lead to the verification of the entire model, but only of that part affected by the change (incremental validation). The use of this data structure is justified by the need, in SMIL, for checking the definition of inconsistent temporal synchronisms [30]. H-SMIL-Net eases the perception of temporal inconsistencies, besides supporting incremental validation.

The visual nature of SMIL Builder reduces considerably the task of validating a document. Since every interaction between the author and the document are mediated by visual abstractions, the validation tool may infer that the code generated by the authoring tool will always be syntactically correct, constraining the validation process to the semantic verification of the temporal synchronisms defined by the author. Despite seeming a reduced work, the process of verifying these synchronisms is not trivial. The validation process described here is equivalent to that used by SMIL Builder in the sense that both are incremental. Besides the target language, one of the main differences between our proposal and the validation method employed in SMIL Builder is that the latter is strongly coupled to an authoring tool, whereas the former is not. Our proposal also performs structural checks, once we cannot assume that all authoring tools that will instantiate our validator will prevent the authors of structural mistakes during the authoring.

3 NCL validation

NCL inherits the NCM [31] causality and constraint paradigm, which specifies the spatio-temporal relationships defined among the medias that form an application. For interactive digital TV, the NCL profiles only support the causality paradigm, in which a set of actions is triggered when a set of conditions is satisfied.

Trying to provide a better structuring and encapsulation to applications, the NCM model introduces the notion of composite nodes. Composite nodes can be of two types: context and alternative. Context nodes cluster nodes that have any semantic relation between each other, as well as their respective relationships. The body of the document itself is a context node. The alternative nodes, on the other hand, are responsible for adapting the content of an application.

The composite nodes, besides better structuring an application, define the perspective of the elements inside it. So, the links in a composition are only capable of referencing elements inside the same composition. The perspective notion can also be understood as a nested hierarchy where an element is found. Due to this notion, we say that NCL has a perspective-based scope. Furthermore, the verification of NCL code must be contextual, that is, it must take the contexts perspective into consideration.

Among other NCL features, this perspective notion is one of the cases that hinder the validation of NCL documents with basis on XML Schema only. Actually, XML Schema would only be enough for syntactic and structural validation of the documents, since there are several particularities in NCL that it cannot express. For example, when checking the correctness of a link, it is necessary to be sure whether the nodes that are being associated are valid identifiers and are present in the same composition (which cannot be verified by an XML Schema specification alone).

Another feature recurrently employed in the NCL project is the multiple references that can be made among the elements of the language. The NCL project was conceived to allow a high reuse degree [32]. A media node, for example, makes reference to a descriptor that, in turn, makes reference to a region. Notice that both the descriptor and the region can be in separate files, making the validation even more difficult. A special type of reference is the one made possible by the *refer* attribute. If node A refers to node B by means of the *refer* attribute, then A will inherit all of the features of B, and they must necessarily be of the same type. A constraint to this reference is that it cannot happen if B is a *context* element ancestor of a *context* element of A, or if B is one element that also has the *refer* attribute referring to another node C.

A third characteristic of NCL that makes validation difficult is that the value of some attributes may depend on the value of another attribute (like the attributes *type* and *subtype* of the $<transition>$ element). In some cases, even the cardinality of the elements may depend on their parent elements (as occurs with the $<compoundCondition>$ element, that has different cardinalities depending whether its child of a $<causalConnector>$ or another $<compoundCondition>$) or on the value of an attribute (as in the cardinality of the $<bind>$ elements of a $<link>$).

The above singularities tend to increase the complexity of systematization of the validation process, eventually leading to the implementation of a validator that analyzes each element of a language individually, making it less generic. An inconvenience of a validator that does not generalize its

```
ELEMENT   (name, parent, cardinality, define_scope)

ATTRIBUTE (element_name, name, is_required, datatype_id)

DATATYPE  (datatype_id, regex)

REFERENCE (element_name, attr_name, ref_element_name,

ref_attr_name, perspective)
```

Listing 1. Skeleton of the metalanguage primitives.

validation method is the difficulty in maintaining its code, making unfeasible the adaptation of the validation to other versions or profiles of the language.

The main objective of this work is to develop a method that generalizes the treatment of part of the particularities of NCL, allowing the development of a validation process that deals with a reduced number of particular cases. Besides generalizing part of the process, we also look forward to giving support to the incremental validation of documents, which is normally a desirable feature for authoring tools.

3.1 NCL validation metalanguage

A way to make the validator aware of the NCL structure is to embed the structural aspects directly in the source code, using an abstract data structure. This approach, despite being very direct, has the problem of strongly coupling the source code to the NCL structure. This makes difficult, for example, to adapt the validation for different NCL profiles (or even versions).

An alternative to this strong coupling is to describe the language to be validated by means of a metalanguage. In this case, the validation must dynamically interpret a specification written in this metalanguage and run according to this definition. With this approach, making the validator able to accept different NCL profiles is just a matter of describing these profiles through the metalanguage, without changing the validator source code. Another important advantage of this approach is the fact that the generalization of the validation process is directly proportional to the expressive power of the metalanguage to describe NCL particularities. In other words, the use of a metalanguage can considerably reduce the number of particular cases that the validator must handle, minimizing the implementation effort and the codification errors margin.

In our previous work [33] we define a metalanguage for NCL description that can express its perspective-based scope, one of the main limitations of the use of XML Schema to validate NCL. Besides this characteristic, this language also describes several other NCL structural aspects. The remainder of this subsection is intended to explain it, highlighting the way it describes several NCL aspects.

The declarative metalanguage proposed is based on primitives that describe certain aspects of the language. We defined four primitives: ELEMENT, ATTRIBUTE, REFERENCE and DATATYPE. Each one of these primitives has a set of arguments and informs specific NCL aspects to the validator so that it can check the code. Listing 1 presents the skeleton of these primitives with their arguments.

The primitive ELEMENT defines the XML elements of the NCL. This primitive has four arguments: the name of the element, the name of the parent element, the cardinality (the number of occurrences) of the element related to this parent and a boolean value indicating whether this element defines a scope or not.

In NCL, one element can be child of different parent elements. This is the case, for example, of the element $<media>$, which can be child of $<context>$, $<body>$ or $<switch>$. Another interesting particular case happens with the element $<compoundCondition>$. This element can be either a child of a $<causalConnector>$ element or itself. The element $<causalConnector>$ can have 0 or 1 element $<compoundCondition>$, but a $<compoundCondition>$ can have as many children $<compoundCondition>$ as we want. This characteristic shows that for one single element, we can have different cardinalities for different parent elements. So, according to the proposed metalanguage, a declaration of an ELEMENT primitive is needed for each pair of parent and child elements.

The cardinality of the elements can be an integer or one of the following special characters: *, +, ?, #. There is also the operator A followed by the numerical indexes ($A1$, $A2$...). If the cardinality of an element is a number, this value indicates the exact number of times that this element should appear. The "*" indicates that the element can appear zero or more times. The symbol "+", on the other hand, indicates that the element must appear at least one time, and "?" tells that the element must appear zero or one time. The adoption of these symbols is based on the notation commonly used to describe regular expressions.

The symbol # has the following meaning: let us say that the element X has the elements Y and Z as children with cardinality #. Then, at least once either Y or Z must appear as child of X, with the possibility of both elements to appear simultaneously. This is the case, for example, of the element $<regionBase>$, which must have at least one element $<region>$ or an element $<importBase>$ as child.

The operator A followed by the indexes defines elements among which just one of them should appear exactly once, and the other should not appear. Let us say that the elements Y and Z, children of the element X, have cardinality $A1$. This means that the element X must have, mandatorily, either Y or Z as child, being allowed the existence of just one element Y or Z as child of X. Still following this example, if the elements P and Q are also children of X, but with cardinality $A2$, then, mandatorily, besides Y or Z, the element X must also have as child either P or Q, which can appear just once.

The primitive ATTRIBUTE specifies all attributes of the XML elements. For each attribute of a given element, it is needed a new ATTRIBUTE entry, even if different elements have the same attribute. An example of this is the *id* attribute, shared by most elements, which requires several declarations. The first argument is a boolean value that indicates whether this attribute is mandatory or not. Finally, the fourth argument must be the identifier of a data type defined by the primitive DATATYPE.

The DATATYPE primitive defines all data types that the attributes of the elements may have. This primitive has two arguments. The first one is the data type identifier and the second one is a regular expression that is capable of validating it. So, if we want to create a new data type (boolean, for instance), we could create a DATATYPE entry with this kind of identifier (BOOLEAN) and a regular expression capable of validating this new data type (^(true|false)$).

Finally, there is the REFERENCE primitive. This primitive can express references that can be made among the elements. In NCL, saying that an element A references an element B means that there is an attribute of A whose value equals an attribute that identifies the element B (some examples in NCL of attributes that identify an element are *id*, *alias* and *name*). The arguments of the REFERENCE primitive are: the name of the element which makes the reference; the attribute of the element which makes the reference; the name of the referred element; the attribute of the referred element; and the scope where the referred element must be found.

Summarizing, there are three forms in which a reference may occur. There are elements that can be referred from any scope of NCL code. The *id* attribute of an element $<descriptor>$, for example, can be referred by the *descriptor* attribute of any element $<media>$, regardless of the scope where the element $<media>$ is. In this case, we say that the scope of this reference is ANY.

Table 1 Attributes and children of the element $<regionBase>$

Element	Attribute	Content
regionBase	id, device, region	(importBase|region)+

In other case, a reference is valid only if both elements are in the same scope (or perspective, in the NCL jargon). The *component* attribute of an element $<port>$, for example, must make reference to the *id* attribute of one of the following elements: $<media>$, $<context>$ or $<switch>$. However, this reference will be valid only if the referred element is in the same perspective of the element $<port>$ (that is, if both elements are children of the same element $<context>$ or $<body>$). The scope of this reference will be SAME.

Finally, in a third form of reference, the scope is defined by the value of another attribute. Considering the element $<port>$ of the previous example, its *interface* attribute must refer to any of the interfaces of the element referred by the *component* attribute. So, if the *component* attribute of an element $<port>$ makes reference to the *id* attribute of an element $<media>$, then the value of the *interface* attribute should be: (i) the *id* attribute of an element $<area>$ child of the referred $<media>$; or (ii) the *name* attribute of an element $<property>$, child of the referred $<media>$. This kind of reference, where the scope of the element depends on the value of another attribute, is indicated by the following syntax: $ELEMENT.ATTRIBUTE. For the case of element $<port>$ mentioned, this scope would be $THIS.component. THIS is a syntactical sugar that represents the very referencing element. There is also the syntactical sugar GRANDPARENT, which represents the parent of the parent of the referencing element.

To exemplify the use of the metalanguage described in this subsection, we will take the element $<regionBase>$ as a use case. According to the definition in [4], this element has three attributes, where none of them is mandatory: *id*, *device* and *region*. Two possible children are defined: $<importBase>$ and $<region>$, with the condition that at least one of these children must be defined at least once. The element $<regionBase>$ is child of the element $<head>$, and a document may contain either several or no element $<regionBase>$. Table 1, extracted from [4] summarizes the attributes and child elements of the element $<regionBase>$.

Listing 2 illustrates part of the document that represents, in the metalanguage, the element $<regionBase>$ in an NCL profile. It shows the entries that define the element $<regionBase>$, its attributes, its child elements and the reference made by the *region* attribute to one of the elements $<region>$ defined in the document.

Since there is only one possible parent ($<head>$) for the element $<regionBase>$, an entry of the ELEMENT primitive is sufficient to define it (line 1). In an NCL document there can be zero or more elements of this type, so its cardi-

```
1: ELEMENT(regionBase,head,*,false)
2: ATTRIBUTE(regionBase,device,false,DEVICE)
3: ATTRIBUTE(regionBase,id,false,ID)
4: ATTRIBUTE(regionBase,region,false,STRING)
5: ELEMENT(region,regionBase,#,false)
6: ELEMENT(importBase,regionBase,#,false)
7: REFERENCE(regionBase,region,region,id,ANY)
8: DATATYPE(DEVICE,^(systemScreen\([0-9]+\)|systemAudio\([0-9]+\))$)
9: DATATYPE (ID,^([a-zA-Z][a-zA-Z0-9_]*)$)
10:DATATYPE(STRING,^(\S+)$)
```

Listing 2. Piece of the metalanguage that defines the element <regionBase>.

nality is defined by the operator *. The perspective concept does not apply to this element, justifying its last argument as false. Three ATTRIBUTE entries are needed to define the attributes (lines 2 and 4). The possible children of the element $<regionBase>$ are defined by two entries of the ELEMENT primitive (lines 5 and 6). Since at least one of these elements must appear at least once, the cardinality of both is defined by the operator #. It is defined a reference (REFERENCE primitive) which can be made by means of the *region* attribute, which makes reference to the *id* attribute of another element *region*. As the element $<region>$ can be referred from anywhere in the document, the last argument of this primitive is ANY (line 7). Finally, the types of each attribute declared in Listing 2 are defined by three DATATYPE entries (lines 8 and 10).

3.2 NCL incremental validation

Incremental validation consists in validating only the parts of the document that needs to be validated. It avoids the unnecessary checking of previously validated parts that have not been affected by recent modifications. We define incremental validation, based on the definition found in [26], as follows:

Definition 1 (*Incremental validation*) Being M the NCM data model and S(M) the set of all NCL documents whose structure follows the rules and restrictions defined by M, if D is a representation of an NCL document, a sequence of modifications U transforms D into U(D). Thus, considering $D \epsilon S(M)$, incremental validation consists in analyzing U and extracting from U(D) the subset of elements that needs to be validated so as to verify whether $U(D) \epsilon S(M)$.

According to Definition 1, the focus of incremental validation should be on checking small pieces of code rather than the whole document. Thus an important task to allow incremental validation is being able to infer, based on the set of document modifications, which segments of the document need to be further validated. In other words, we need to identify the parts of the document that have been modified since the latest validation and also those parts to be possibly affected by those modifications, thus eliminating the need for a complete rebuild of the NCL document representation, as is the case with the standard validation method.

Having identified the set of elements to be validated, it is necessary to decide which type of validation to apply in each case. Depending on the nature of the element and on the modification to be performed, it is quite possible that some elements may only need a structural or referential validation, thus discarding a thorough time-consuming document checking. Structural validation checks the element structure regarding the arguments of the defined ELEMENT, ATTRIBUTE and DATATYPE primitives while referential validation checks the references made regarding the arguments of the defined REFERENCE primitive.

To better organize the process of identifying the type of validation suitable for each element, we divided NCL elements into four distinct groups: A) independent elements; B) referencing elements; C) referred elements; and D) referencing and referred elements. Figure 1 illustrates these categories.

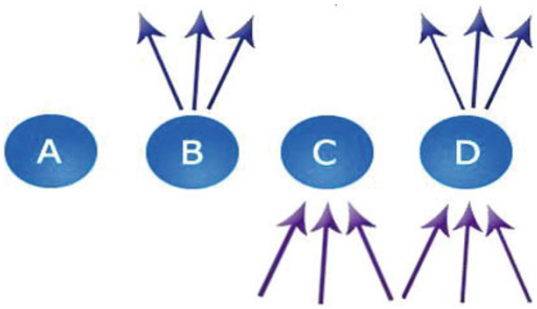

Fig. 1 Classification of NCL elements according to their references to/from other elements

```
<causalConnector id="connector1">
      <connectorParam name="valueTest"/>
            . . .
      <assessmentStatement comparator="eq">
            <attributeAssessmentrole="test"eventType="attribution"/>
            <valueAssessmentvalue="$valueTest"/>
      </assessmentStatement>
</causalConnector>
```

Listing 3. Example of validation of group A elements.

The elements in group A are those which do not reference any other element and are not referred by any other element, the precise reason why they are named independent. When we need to validate a modification in an element of this kind, all that is required is the structural checking of the element itself and of the subtree it spawns. It is not necessary to perform a referential validation, as the example in Listing 3 shows: if an editing changes the value of attribute *comparator* from *eq* to *lt*, in element $<assessmentStatement>$, the validation merely needs to check if the new value of the attribute belongs to the set of possible values and then validate the structure of the subtree it generates, which in this case comprises only the elements $<attributeAssessment>$ and $<valueAssessment>$ as well as their corresponding attributes.

It is important to note that whether type A children elements have references or not, it does not affect the type of validation to be made, since an edition on an element of this type does not affect any reference in the document, only its subtree structure. For example, changing $<assessmentStatement>$ in Listing 3 does not affect any reference made to its $<attributeAssessment>$ and $<valueAssessment>$ children.

On the other hand, when elements of group B are modified, the validation is a bit more complicated. Since these elements make reference to other elements, it is necessary to perform a referential validation besides the structural validation, including both the element and its associated subtrees. Consider the case of the element $<link>$ when referencing elements of the type $<causalConnector>$. According to Sect. 3, there should be as many $<bind>$ elements as there are roles defined by the referred connector. Thus if an editing in the document creates a link to a different connector, it is vital to verify whether the new connector does exist and the $<bind>$ elements are consistent with the roles defined in its scope. Listing 4 presents an excerpt of code where validation becomes mandatory since attribute *xconnector* is now referencing a new connector named *conn2*.

We now discuss the validation of elements belonging to group C, i.e. elements referred by others. When a member of this group is modified, the referencing elements also need to be checked. Two different types of validation are required in this case: the subtree generated by the modified element needs to be checked structurally, and the elements affected by the changes need to be checked referentially. Consider an element $<region>$ being referred by an element $<descriptor>$. If the id attribute of the element $<region>$ with value $rg1$ is modified, it is necessary to check if the new value of the *id* attribute is valid and it is also necessary to check whether the reference made by the $<descriptor>$ element is still valid for this modification. Listing 5 shows the code snippet where the validation described is applied.

Finally, in group D there are the elements that refer to other elements and are also referred by other ones. The subtree generated from the modified element should be checked structurally and referentially, and the referencing elements should have their references checked as well. Listing 6 shows an example of this type, in which a $<media>$ element refers to a $<descriptor>$ and is referred by an element $<bind>$. If a change occurs in $<media>$, it is necessary to check if the values of its attributes are correct and whether the reference to the element $<descriptor>$ and the reference from the element $<bind>$ remain valid.

From the above discussion, we conclude that the issue here is whether an element refers to other elements or not. Referential validation must be made when every element that makes reference or is referred by other elements is edited. In the first situation, it will be referentially validated, since its edition does not affect the elements referred by it. In the second situation, however, all the other elements that make reference to it must go through a reference validation as well. Structural validation is thus related to the fact that the element

```
<ncl>
   . . .
      <connectorBase>
            <causalConnector id="conn1">
                  <simpleCondition role="onBegin"/>
                  <simpleAction role="start"/>
            </causalConnector>
            <causalConnector id="conn2">
                  <simpleCondition role="onEnd"/>
                  <simpleAction role="stop"/>
            </causalConnector>
      </connectorBase>
<body>
        . . .
      <link xconnector="conn1">
            <bind role="onBegin" . . ./>
            <bind role="start" . . . />
      </link>
</ncl>
```

Listing 4. Example of validation of group B elements.

```
<ncl>
      <head>
            <regionBase>
                  <region id="rg1" top="5%" />
                  <region id="rg2" top="10%" left="30%" />
            </regionBase>
            <descriptorBase>
                  <descriptor id="d1" region="rg1" />
            </descriptorBase>
      </head>
</ncl>
```

Listing 5. Example of validation of group C elements.

was directly modified and then needs to have its structure checked again.

4 Implementation

As a proof of concept, we developed an incremental validation tool that was modeled using a component-based architecture, intending to provide a low coupling among its elements. Figure 2 illustrates this architecture, whereas the elements *Validation*, *Interpreter* and *Messages* represent the validator components with well-defined functions, and the element *Model* is the component that represents the NCL document to be validated. For integration with any other tools, it is used the *Adapter* design pattern [34] to convert the model used by the

```
<ncl>
      <head>
            <descriptorBase>
                  <descriptor id="d1" . . . />
            </descriptorBase>
      </head>
      <body>
            . . .
            <media id="media1" src="sample.mp3"descriptor="d1"/>
            <link xconnector="conn1">
            <bind component="media1".../>
      . . .
            </link>
      </body>
</ncl>
```

Listing 6. Example of validation of group D elements.

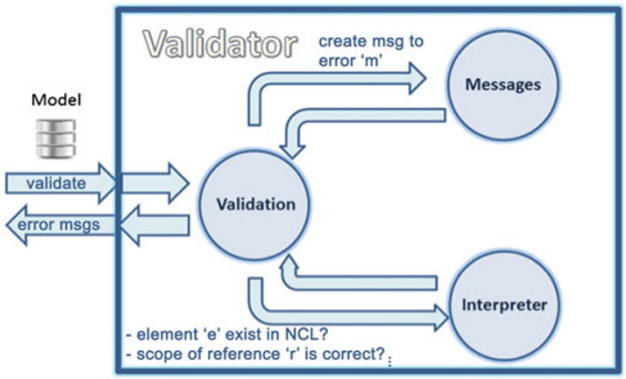

Fig. 2 The component-based architecture proposed for the validation tool

tool to our model. Section 5 presents the integration between our implementation and the IDE NCL Composer. In that section is discussed how a tool can integrate the implementation described here in order to support the NCL authoring. Figure 2 shows the proposed architecture for the validation tool.

A user usually edits a NCL document through an IDE that has its own model to maintain the document. It is unfeasible for the validation tool to understand the several existing IDE models, each of which with its own particularities. The Model component in Fig. 2 is the model used by the validator tool. A developer must create an adapter to track the IDE model and updates the validator one. Every edition on the Model component triggers a routine that checks which elements were affected by the edition. When the user submits a Model instance to be validated, the Validation component validates only the affected elements according to the type of validation needed, using the Interpreter component to support the validation. The Interpreter component is tied to the language structure through an instance of the metalanguage defined in Sect. 3.1. For each error found, a corresponding message is created in the Messages component to return to the user. The following sections delve in each component.

It is worth mentioning that, in our implementation, there are two phases clearly distinct: the first is the update of the model and the second is the validation itself. Note that the first phase takes place in a previous moment to the validation itself. Once the focus of this work is the later phase, that is, the (incremental) validation of NCL documents, it is beyond the scope to provide a deep discussion about which is the best model for representing NCL documents. Nevertheless, in order to better explain how the validator implemented works, in the next subsection will be detailed how our model is defined and how it can be updated. Also note that even if one can use other model completely different with the same interface, the incremental validation here described still would work. The performance tests discussed in Sect. 4.5 were carried out with focus on the validation phase.

4.1 Model

The model is the data structure that must be used by the Entity[1] so that the NCL document can be validated by the

[1] From now on we will use the term "Entity" as alias for a hypothetical authoring tool that uses the implementation described in this paper.

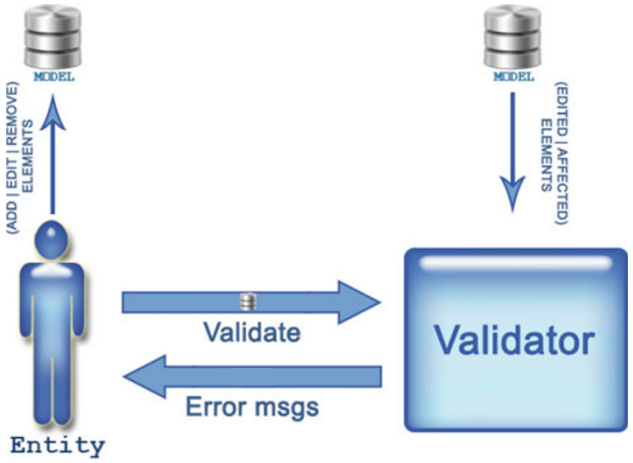

Fig. 3 *Vertical arrows* illustrate the flow of elements in the Model. *Horizontal arrows* illustrate the flow of the Model between the Entity and the Validator

tool.[2] It represents the document and performs important operations as the Entity interacts (through well-defined primitives) with the document. Figure 3 illustrates the role of the model in the architecture.

The vertical arrows illustrate the flow of elements in the model. The Entity is the data source, which feeds the model through the ADD/EDIT/REMOVE ELEMENTS and ADD CHILD and must guarantee that the model matches the state of the document to be validated. The Validator, in turn, receives the elements modified and affected by the operations performed in the model. This scheme is mainly responsible for making the incremental validation viable. It concentrates on the concepts discussed in Sect. 3.

The horizontal arrows illustrate the flow of the model between the Entity and the Validator. Every time that the Entity wants to validate an NCL document, it must forward a model instance to the Validator (through the Validator API), which, in turn, returns the messages informing the consistence of the document.

Each document element is represented in the model by an instance of the class *ModelElement.* This class has a set of data structures that keeps information about the elements. Each *ModelElement* has an attribute list, a child elements list, a pointer to its parent and a special list (called *references*) that keeps all the references made to it. The model, on the other hand, is represented by an instance of the class *Model*. This class maintains some data structures to represent NCL documents and offers an API that allows the Entity preserve the consistence between the model and the document. Figure 4 illustrates the *Model* class.

Whenever one of the primitive is invoked, the model performs some internal processing to verify the elements that

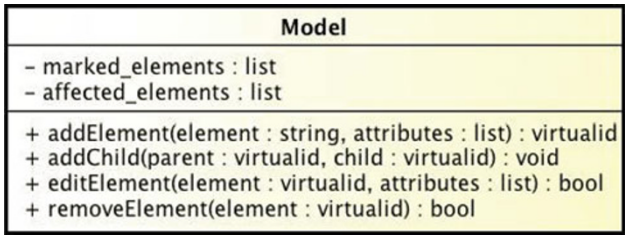

Fig. 4 Class that represents the Model component in Validator

need to be validated after the operation. The use of well-defined primitives to update the internal state suits the incremental nature of authoring, in which features are being added successively to the document. This also ensures that the model is always in a state that can be validated, i.e., the model is always in a state susceptible to validation before and after a primitive is invoked (because each primitive is an atomic operation). Consequently, this approach leaves room for heuristics that seek to optimize the exact moment that the validation process should be performed, e.g., a tool can expect a minimum number of updates be realized before validating the Model.

The Model maintains two main data structures that allow the identification of elements to be validated: a list of marked elements (*marked_elements*) and a list of affected elements (*affected_elements*). Whenever one element is added or edited, the Model inserts it into the marked elements list. Whenever one element is marked or removed, then all the elements that refer it are inserted into the affected list, unless they already are in the marked list (otherwise a double validation would be performed in these elements). Thus the Model can differentiate edited elements (marked list), elements affected by the editions (affected list) and other elements that do not need to be validated.

Despite the two lists discussed above, there is also a map of elements that performs the search of elements inserted in the Model in a constant time ($O(1)$). Such performance justifies the use of this map, instead of other options like a linked list (since its average search time is $O(N)$, compromising the Validator performance).

To complete the data structures maintained by the Model, there is a map that stores the references to elements that were not inserted into the Model. Considering the Listing 7, the *descriptor* elements identified as "*d*1" and "*d*2" are referring a *region* element identified as "*rg1*" that was not inserted into the Model. When the author creates the *region* "*rg1*" and inserts it into the Model, this should be able to identify that "*d*1" and "*d*2" do a reference to this *region* and then add them to the *references* list of "*rg1*" (list of elements that reference this *region*). A map of elements not inserted (*not_inserted_elements*) is used to store the identifiers of the elements that were not inserted and the list of elements that reference them.

[2] The term "tool", on the other hand, will be used as alias for the Validator implemented.

```
<ncl>
      <head>
            <regionBase>
                  <region id="rg0" . . . />
            </regionBase>
            <descriptorBase>
                  <descriptor id="d1" region="rg1" />
                  <descriptor id="d2" region="rg1" />
            </descriptorBase>
      </head>
</ncl>
```

Listing 7. Descriptors pointing to a region not created yet.

It is important to emphasize again that the instant that the primitives are called to update the Model and the instant that validation occurs are distinct. This enable in theory the possibility of one implementation of the validator allow the entity tool to interfere in the primitive calls (e.g., though *listeners*) making them more costly (this could happen while the document is being edited, for example) without affecting the validation time of the Model after. Although the focus of this work is on (incremental) validation, we briefly describe the complexity of the Model primitives in the next subsection.

4.1.1 Complexity analysis of Model's primitives

There are basically four operations to be realized when adding an element **E** to the Model: creating the **E** object of the class *ModelElement*; checking whether an element has made a reference to **E** before; checking whether **E** does references to other elements; inserting **E** into the marked list and the elements which reference it into the affected list. The instantiation of **E** is a time constant operation ($O(1)$). Checking whether an element has made a reference to it before its inclusion into the Model is a time constant operation too ($O(1)$). However, if there are **R** elements that do references to **E**, it is necessary to copy all of them to the *references* list of **E**, making the complexity of this operation be $O(\boldsymbol{R})$. On the other hand, to figure out whether **E** does any reference to others elements it is necessary to check its attributes. If **A** is the number of attributes of **E**, the complexity of this operation is $O(\boldsymbol{A})$. Finally, inserting **E** into the marked list is a time constant operation while inserting the elements that do references to it into the affected list is an $O(\boldsymbol{R})$ operation.

Briefly, we can say that the complexity of the primitive *addElement* is $O(\boldsymbol{R} + \boldsymbol{A})$. Given one element of the NCL current version (3.0), there are at most six other elements that can correctly do reference to it. Just to compare, NCL has 45 elements. This implies that in a real scenario **R** will always be lesser than **N**, unless the authors have purposely created an abnormal number of elements that do reference to one specific element, what will make **R** tending to **N**. Likewise if we have in mind that the element that have the highest number of attributes is the *descriptor*, which has only 18 attributes, we can conclude that the complexity $O(\boldsymbol{R} + \boldsymbol{A})$ will always be lesser than $O(\boldsymbol{N})$ in a real scenario.

The edition of elements consists basically in updating the attribute values of an element to new ones. However, it is necessary to ensure that the old references made to the element will be updated to the new ones. Considering **A1** as the number of attributes of an element **E** before its edition, **A2** as the number of attributes of **E** after its edition and **R1** as the number of elements that do reference to **E**. To update the references of **E**, it is necessary to process each attribute of **E**, what is an $O(\boldsymbol{A1} + \boldsymbol{A2})$ operation.

The worst case of the *editElement* primitive is when the identifier attribute of **E** is changed. In this case, it is necessary to update the list of elements that do reference to the old identifier of **E** ($O(\boldsymbol{R1})$), because these elements now do reference to an element that was not inserted into the Model, and check whether any element of the *not_inserted_map* has made a reference to the new identifier attribute of **E**. If **R2** is the number of elements that do reference to the new identifier of **E**, then copying them to *references* list of **E** is an $O(\boldsymbol{R2})$ operation. Finally, it is necessary to add **E** into the marked list ($O(1)$) and the elements that do reference to it into the affected list ($O(\boldsymbol{R1}+\boldsymbol{R2})$). So the complexity of the *editElement*, in the worst case, will be $O(\boldsymbol{A1} + \boldsymbol{A2} + \boldsymbol{R1} + \boldsymbol{R2})$. By the previous paragraph, we can say that in a real scenario such complexity will always be lesser than $O(\boldsymbol{N})$.

The *removeElement* primitive is simpler in comparison with the previous ones. It consists in removing one element **E** of the Model and adding all the elements of it *references*

list to the Model affected list. In that case, if **R** is the length of the *references* list, the complexity of this operation will be $O(\boldsymbol{R})$.

Finally, *addChild* primitive manipulates some pointers of the Model elements in order to associate the parent/child relation of these elements. If **P** is the parent element and **C** the child element, this primitive will add **C** to the *children* list of **P** and make the parent pointer of **C** points to **P**. After this operation, it is necessary to add both **P** and **C** to the marked list ($O(1)$).

4.2 Validation

The Validation component is the one responsible for performing the validation indeed. For each element in the marked list, it identifies the type of the element (according to the types defined in Sect. 3.2) and performs a validation regarding this type. It uses the Interpreter component knowledge about the language structure to identify the element type.

All the elements in the marked list, regardless of its type, must be structurally validated. Type A and C elements do not need to be referentially validated because they do not have any attribute that does reference to others elements. Type A and B elements can directly build error messages to all elements of its *references* list without checking the references, since they do not have any referenceable attribute. Thus, type D elements are the most costly ones to the validation, since both elements (beyond the structurally validation) and the elements that do reference to them must be referentially validated.

Last but not least, it is important to emphasize that due to the inability of the metalanguage (as of other languages, like XML Schema) in covering all NCL particularities, some minor features must be directly hardcoded on the source code of this component, e.g., the number of instantiated *roles* in a *link* element must match the number of defined *roles* in the *causalConnector* element.

4.3 Interpreter

The Interpreter component is the one responsible to knowing the rules and the structure of the target language. It uses the metalanguage specification to process several Validation component requests about issues found in the Model instance, e.g. an element E1 can be the parent of an element E2, an attribute A1 of an element E3 can refer to an attribute A2 of an element E4, an attribute A3 of an element E5 can have S as value, etc. This component is also used to know the type of each element, e.g. elements that do not have entries in the metalanguage primitive REFERENCE are type A, elements that have entries in the REFERENCE primitive but only on the first argument (element_name) are type B, etc.

The metalanguage specification can be stored in a text document and must be updated whenever the target language changes. In practice, separating the language structure and the validation source code allows the validation to be adaptable to others languages and other profiles of the same language (e.g., the BDTV profile of NCL) without affecting its source code.

This component, as it is often used as a source of information, shall have fast routines that do not affect the validation performance. Thereby one implementation should do an initial preprocessing on the metalanguage specification and organize the information in an efficient way instead of being frequently performing system calls and searches on the entire metalanguage document.

4.4 Message

The Message component is the one that manages all the messages from the validation process. For each error found during the validation, the Validation component requests a message for this component.

The independence of the validation process and the messages creation allows a more flexibility support to multilingual feature. During the instantiation of this component, it is informed which is the preferred language for the messages. If the preferred language is not supported, the messages will be created in the default language (English). The messages for each supported language are stored in text files (called "language files") that are parsed in the Validator constructor. The names of the language files are encoded according to the IETF BCP 47 [35] standard. This means that in order to add a new language to Validator, one just have to create a new file, name it according to the standard used and place it in the default directory where the Messages component searches for language files. It is not necessary to recompile any source code.

Finally, beyond the file name convention, the content of the language files needs to follow a simple syntax. Listing 8 presents a snippet of the language file that stores

```
3001 = There are more than one element with identifier "%s".
3002 = There are more than one element with alias "%s"
3003 = The '%s' attribute of element <%s> point to an invalid element.
…
```

Listing 8. A snippet of the file that stores English messages.

```
<ncl id="exemplo" xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
    <head>
        <regionBase>
            <region id="rg_screen" width="100%" height="100%"/>
        </regionBase>
        <descriptor id=”d1” region=”rg_screen”/>
    </head>
    <body>
        <port id=”p1” component=”v1”/>
        <media id=”v1” src=”A.mp4” descriptor=”d1”/>
    </body>
</ncl>
```

Listing9: File B.ncl.

the English messages. Note that each message has a unique identifier (left side of the equal symbol) and a set of parameters (‘%s’). The Validation component needs to know the identifier of the message that will be created and also needs to inform the parameters that will fill the “%s” gaps. This is the same approach adopted by the NCL Validator (actually, the language files currently being used are also the same for both tools).

4.5 Case study

This section develops a simple case study to illustrate the validator features described in this work. Considering that the author begins the development of the NCL document from the preexisting B.ncl of the Listing 9. The first step that one tool should do is to map the existing document to the Validator Model though the primitives described in Sect. 4.1.

Figure 5 illustrates the mapping of the B.ncl file to the validator Model. Beside the B.ncl file there is the primitive execution log (the attributes list was omitted of the primitives as matter of space). The tree at the right side of the picture represents the document structure (parent/child elements relationship) and each node symbolizes one *ModelElement* with it attributes and *references* list (the *ref_by* list in each node of the picture). The list of elements that were not inserted into the Model is represented by the Not Inserted bottom table, in which the left field is the identifier of the element (e.g., *id* and *name* attributes) and the right field are the elements that do reference to this identifier.

The marked and affected lists are detached in the bottom right part of the picture. We can note that the marked list has all the elements of the document while the affected list is empty. This happens to avoid repeated validations because, like we said, if one element is in the marked list it will already be referentially validated.

The Model is always in a state capable of validation before and after one primitive is executed. As we said, the decision of the exact moment that the validation will be performed is left to the tool. Considering, then, that the Model of Fig. 5 is committed to the validation described in Sect. 4.2, the Listing 10 shows the errors found in the presented NCL document.

Continuing the example, the author now adds more information to the NCL document. The Listing 11 illustrates the new NCL document (B++.ncl), to which was added the *descriptorBase* element. Figure 6 illustrates the mapping of this new NCL to the Model.

Because the Model stores the document state, it is not necessary to insert all the document elements again, only the modifications made, avoiding redundant processing of the primitives. When inserting the new element *descriptorBase,* is necessary to update the tree structure, placing the *descriptor* element as a *descriptorBase* child and this one as a *head* child. The marked list, then, will have the element *descriptor* (that did not leave the list in the last validation), the new element *descriptorBase* and its parent *head* that needs to validate its children cardinality. The affected list, on the other hand, will only have the *media* element, because this is the only element that can be affected by the modifications made (it does a reference to *descriptor*).

When submitting the Model to the validation again, we found that it does not have any errors, showing that NCL document is correct. The first validation acted as a conventional

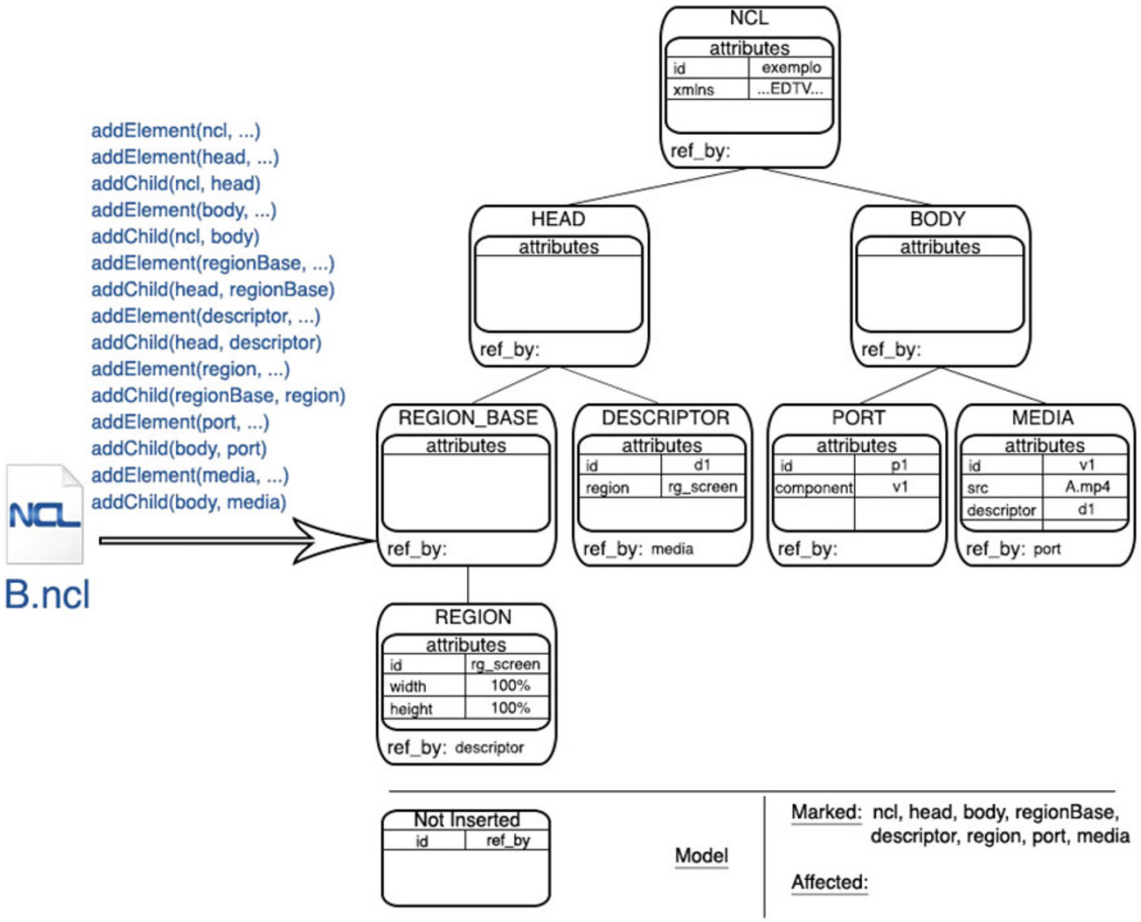

Fig. 5 B.ncl mapping to Model

```
-  Element ‘descriptor’ cannot have element ‘head’ as its parent.
```

Listing 10: Errors found in B.ncl.

validation because all the elements were both structurally and referentially validated. The second validation, however, shows the real gain with the incremental validation. In the case of a conventional validation, all the elements would be validated again independently of the modifications made on the document. This leads to a wastage of resources, since several elements that were not affected by the modifications made will pass again through the same unnecessary) tests batteries. In the incremental validation case, only the elements that can be affected by the modifications made are checked again, and these validations can be referential or structural (or both) for each element.

5 Integration with IDE composer

The Composer is an authoring tool based on multiple synchronized views of the same document. Each one of these views, in practice, is one plugin that can be added to the tool. Figure 7 shows the Composer IDE with six views: textual, structural, layout, outline, properties and debug.

The Composer architecture is composed by one microkernel that maintains an internal data structure that represents the NCL document being edited. Whenever one plugin modifies the model (adding, editing or removing elements) the microkernel sends a message to all the other plugins notifying them of these modifications. The Composer allows that each plugin performs reading operations without sending a message to the microkernel in order to forward it to the other plugins (this ensures that all plugins will always be aware of the current state of the Composer model). We can note that the Composer strategy of incremental modification on its model suits to the incremental validation described in this work.

In [21] is described the necessary steps in order to create a plugin to Composer. Here we just provide a brief description.

```
<ncl id="exemplo" xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
    <head>
        <regionbase>
            <region id="rg_screen" width="100%" height="100%"/>
        </regionbase>
        <descriptorBase>
            <descriptor id="d1" region="rg_screen"/>
        </descriptorBase>
    </head>
    <body>
        <port id=”p1” component=”v1”/>
        <media id=”v1” src=”A.mp4” descriptor=”d1”/>
    </body>
</ncl>
```

Listing11: B++.ncl.

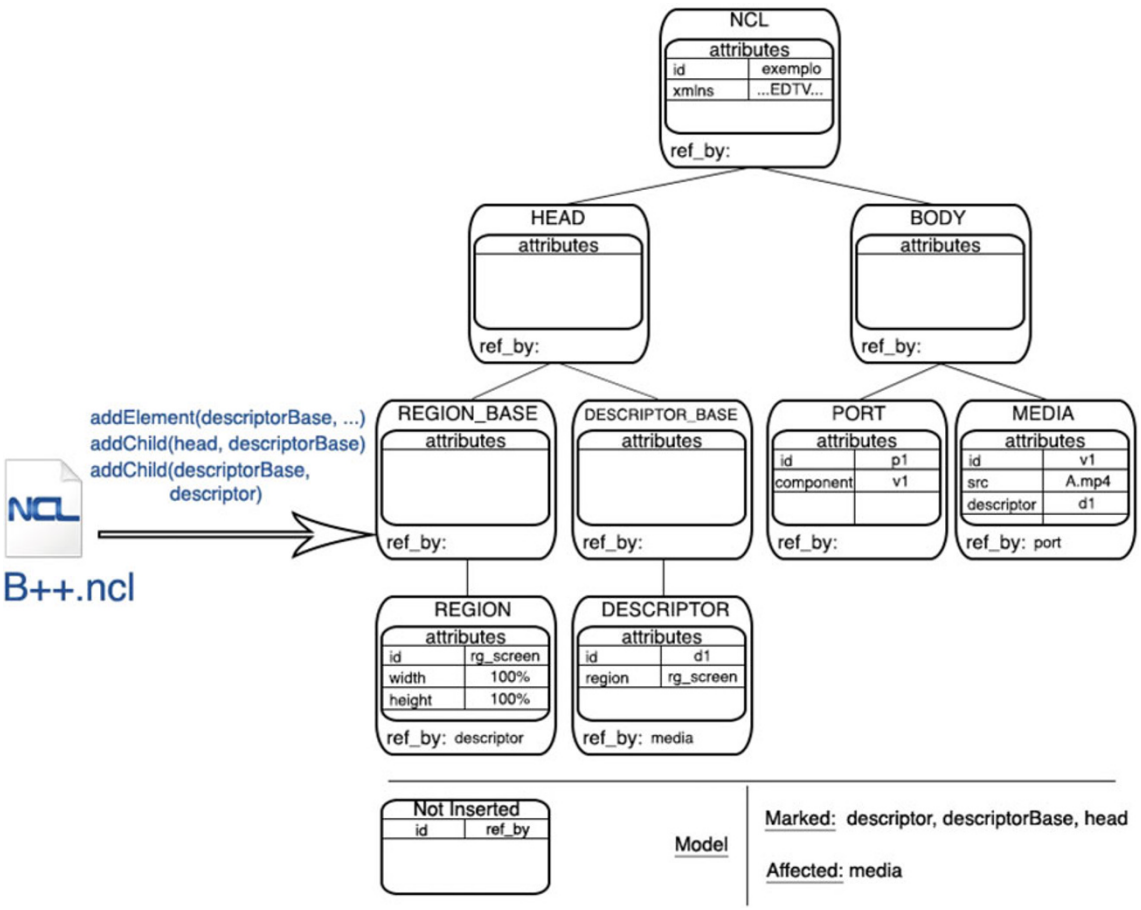

Fig. 6 B++.ncl

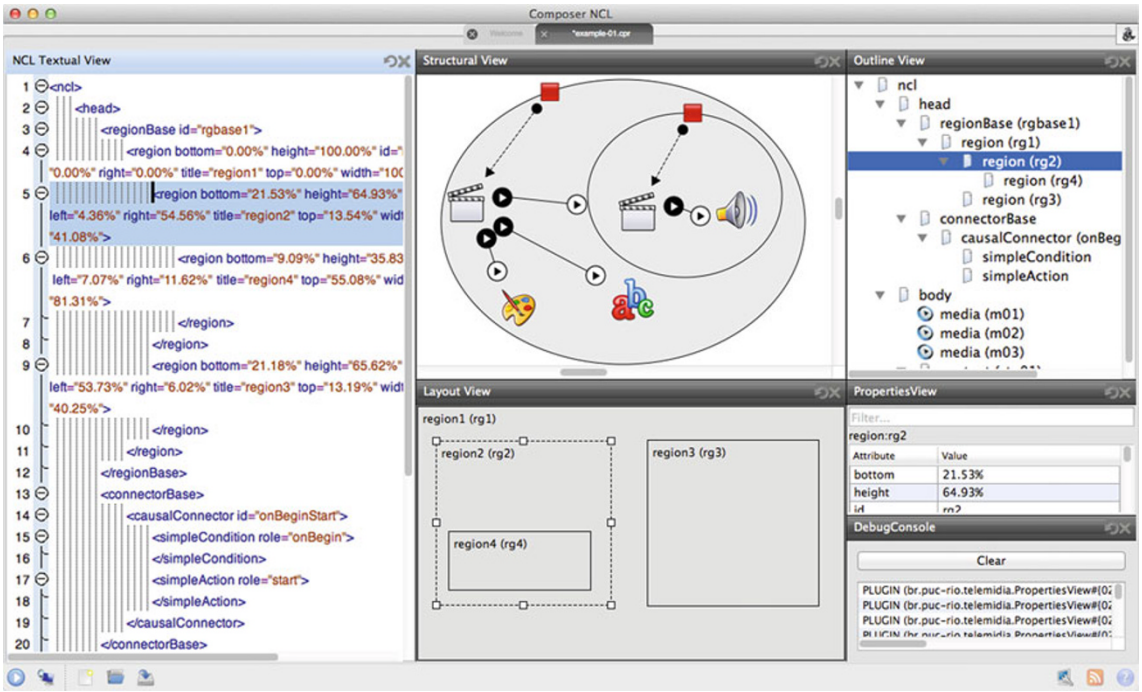

Fig. 7 IDE composer

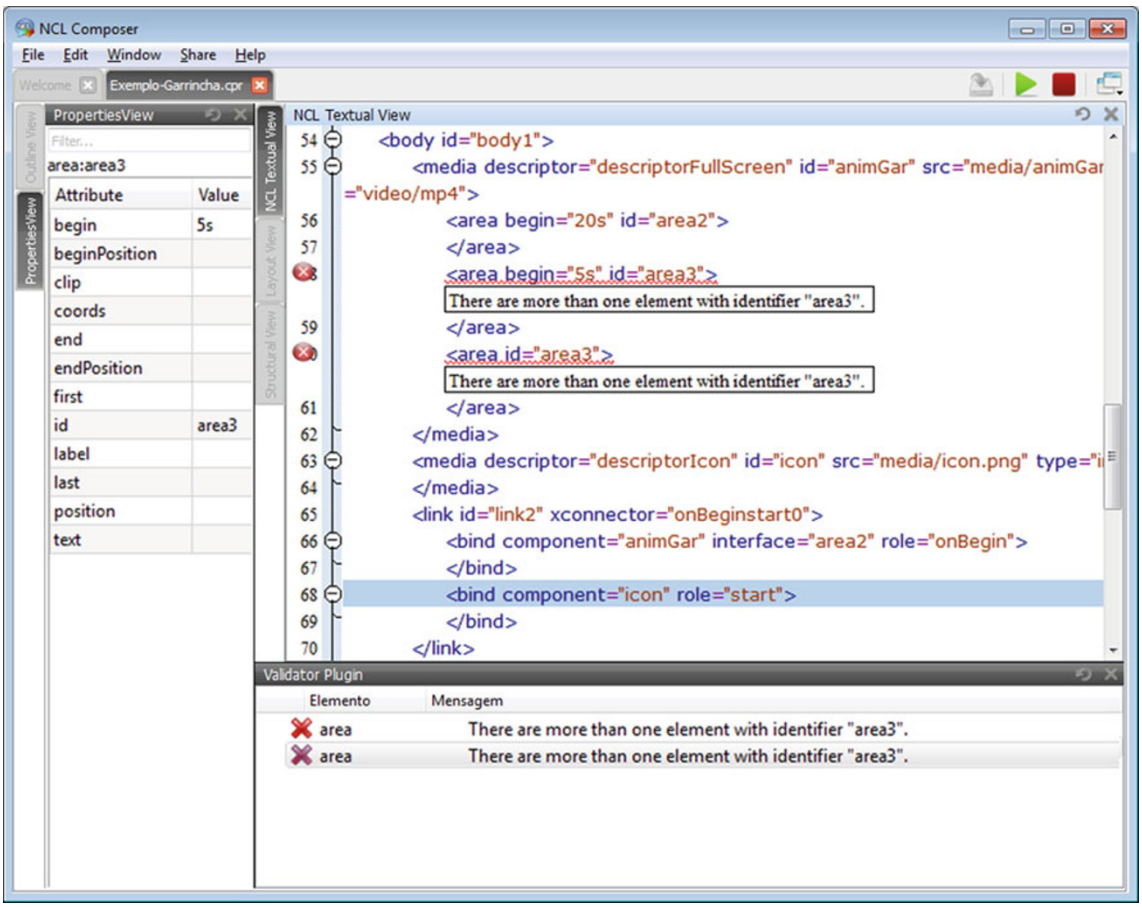

Fig. 8 Textual View incorporates error messages from ValidatorPlugin

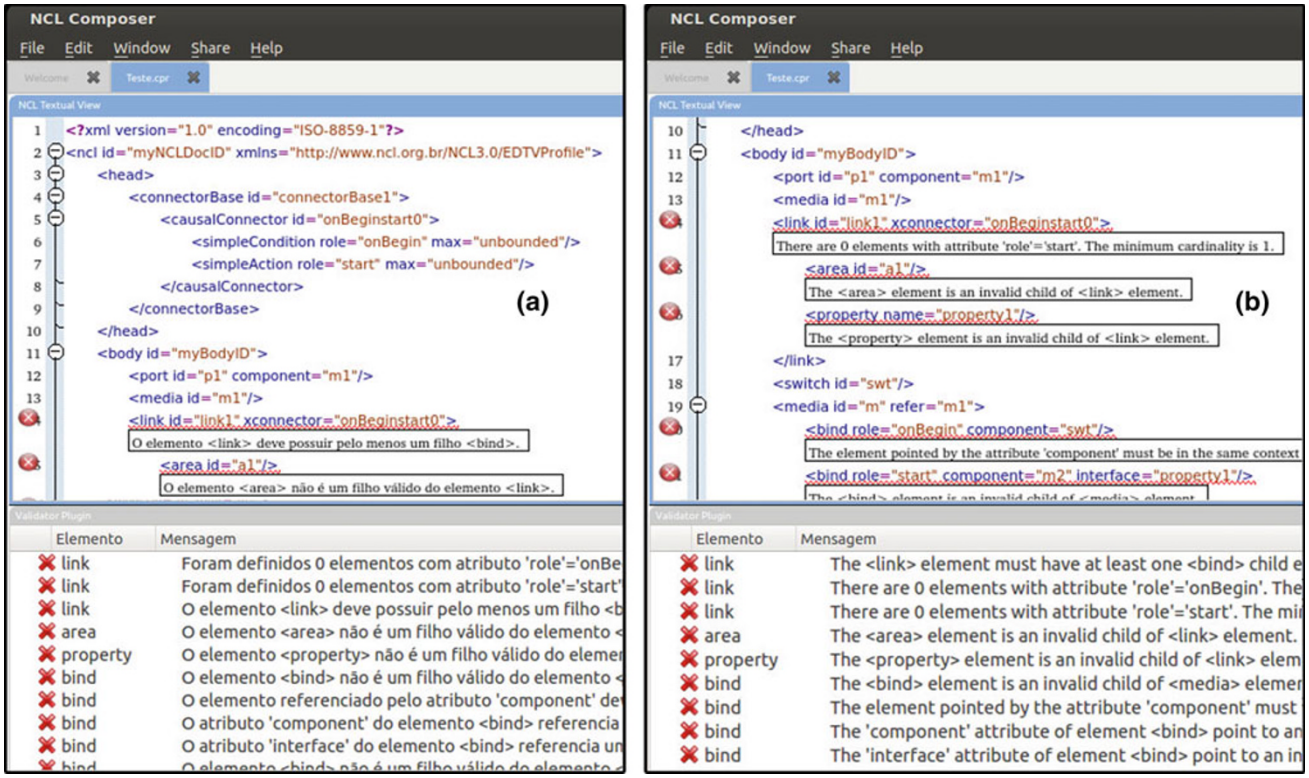

Fig. 9 Multilanguage validator plugin

The Composer offers an API that has two interfaces that must be implemented in order to create a plugin: *IPluginFactory* and *IPlugin*. The first one creates an instance of one plugin, storing information like version, developers, etc., while the second one is the plugin itself. The *IPlugin* interface has three main methods: *onEntityAdded*, *onEntityChanged* and *onEntityRemoved*. Such methods are executed when elements are added, edited and removed of the Composer model, respectively.

As discussed in the last section, each authoring tool has its own data structure to represent the NCL document being edited, and such structures are different from the ones maintained by the Validator Model described in Sect. 4.1. According to [34], the design pattern *Adapter* converts a class interface to a client interface, allowing different classes with incompatible interfaces work together (this is the exactly case of the Validator and the Composer interfaces). Thereby, the integration between the implemented validation and any authoring tool consist simply in creating an adapter that converts the tool model to the Validator Model and ensures that both are synchronized.

In order to integrate the Validator to the Composer, it was developed a validation plugin. Three classes were created: *ValidatorPluginFactory*, *ValidatorPlugin* and *ComposerNCLAdapter*. The *ValidatorPluginFactory* class implements the interface *IPluginFactory* and instantiates *ValidatorPlugin* objects that implements the *IPlugin* interface. *ValidatorPlugin* objects instantiates the *ComposerNCLAdapter* adapter that maps the methods *onEntityAdded*, *onEntityChanged* and *onEntityRemoved* to the Validator primitives (*addElement*, *addChild*, *editElement* and *removeElement*).

Whenever the validation plugin is notified about a modification made on the Composer model, it triggers the validation. Only the modified elements since the last validation and those affected by such modifications are validated, according to what described in Sect. 3.2. At the end of the validation the plugin emits a message informing the errors found to the other plugins. The emission of these messages allows other plugins to incorporate the error messages from the validator. Figure 8 shows the validation plugin running as well as the integration of its messages with textual view plugin.

Finally, the Composer has the internalization functionality. Currently there is support for two languages: English and Portuguese. Figure 9 shows the same error message being displayed in these two languages.

6 Incremental versus standard validation

This section presents a comparison between the methods of standard and incremental validation in terms of speed and responsiveness when applied to an NCL document according to the implementation described in earlier sections.

6.1 Experimental design

As a scenario, it is considered an NCL document of a given size (namely, number of elements) which is being modified by a user/programmer/application and that is validated in the background by means of either a standard or an incremental method. The time spent to validate the document was chosen as response variable, i.e., the time necessary to examine the document (in whole or in part) and return control back to the programmer. As factors that may influence the outcome of the experiment, there are the number of elements in the document, the number of modifications between each validation (the step), and the type of validation process. For each factor, two levels are defined resulting in a 2^3 experimental design as shown in Table 2.

We developed a program that generates an NCL document of a certain size, introduces modifications in that document and submits it to validation. It then measures the time necessary to validate the modified document. Each experiment was repeated 30 times and the means with a 95% confidence interval were calculated.

6.2 Results

Figure 10 shows the result of a document with 100 elements validated by means of a standard/incremental method. The standard validation has to inspect the whole document and thus has an average time of 35ms. For the incremental method, the step between validations is incremented by a factor of 5 elements each time and we observe that the response time is dependent on the size of the step. This time can be up to 7 times smaller than the time taken by standard validation method, for a step of 5 elements, and grows as the interval between validations stretches. Smaller steps yield more responsiveness, as control is given back promptly to the IDE, however, they can cause a certain performance penalty because of the frequent calls to the validation routine.

Table 2 A 2^3 experimental design

Factors	−1	1
No. of elements (A)	100	1,000
Step size (B)	5	15
Type of validation (C)	Incremental	Standard

Table 3 Influence of different factors on document validation time

Parameter	Estimated average	Variation (%)
q0	45.856	–
qA	31.7553	27.48
qB	1.4081	0.005
qC	40.5681	44.85
qAB	−0.0461	0.00
qAC	31.8378	27.62
qABC	0.0618	0.00

Table 3 summarizes the influence of each factor in the response variable. We notice that the validation method (A) and the number of elements in the document (C) were the two most influential factors alone, accounting, respectively, for 27.48% and 44.85% of the response variable. The combination of factors A and C also had an influence of 27.62%. The step size had a very small influence on the validation time.

6.3 Discussion

The experiments performed have shown that the incremental validation method is more advantageous than its standard counterpart. It is more intuitive, considering the nature of

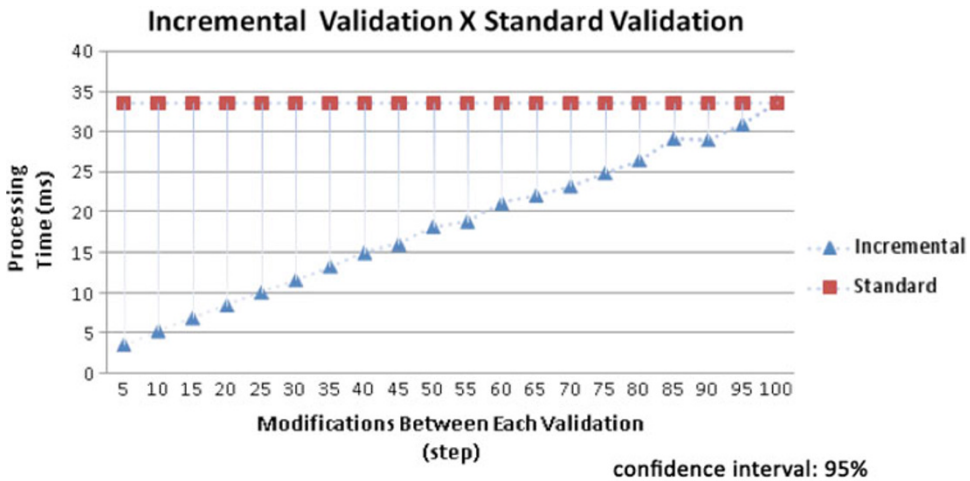

Fig. 10 Comparison between the time necessary to validate the whole document and the time to validate the document in smaller parts (incrementally)

the declarative authoring process, consumes less processing time as it validates only the portion of the document that has been modified, and, although it could potentially impose a certain performance penalty when using small incremental validation steps, it has revealed in practice a negligible overhead. Summarizing, the incremental validation is faster and more flexible than the standard method as our tests have demonstrated.

7 Conclusion

In the interactive digital TV (iTV) arena, application development and presentation usually occur in distinct places. The development stage is carried out at the broadcasters' and/or producers' premises whereas presentation occurs at viewers' households. In both cases, there is a clear need to validate application code in order to guarantee it is in accordance with current standards. This not only allows applications to run on any manufacturers' equipment (as long as it adheres to the standard) but it will also avoid wasting scarce computational resources on set-top boxes trying to execute erroneous applications. Nowadays, with the advent of Social TV and similar applications, which give the viewer the possibility to create and share his/her own content, the need for on-the-fly validation becomes more evident.

This work describes an incremental validation process for NCL, a declarative hypermedia language used for iTV applications development in Brazil, Argentina, Japan and several other countries. NCL is based on XML and is currently an ISDB-Tb standard and also an international ITU-T recommendation for IPTV.

The peculiarities of NCL have been discussed in the paper, especially those that make it particularly difficult or impractical to be validated by XML Schema or DTD, standards commonly used with other XML-based languages employed in iTV, such as SMIL, XHTML and BML. Given the incremental, almost interactive approach by which hypermedia documents are created, we believe that incremental validation, which is carried out gradually, as the user types a new line of code or adds a visual component to an IDE, is more natural and efficient, besides consuming less computational resources.

The incremental validation process proposed and implemented in this paper is based on a categorization of NCL language elements that highlights which parts of code need to be validated further after the developer makes an editing on the document. Our proposal is organized into two distinct stages: firstly, it analyzes the local effects of a code modification and looks for any side effects that it may have triggered. Secondly, it selects the type of validation to be employed in order to guarantee document consistency. The determination of side effects is especially important, since NCL has elements, such as links and connectors, that semantically and syntactically bind elements distributed throughout the NCL document and which may not be so obviously correlated. This feature is accomplished through a metalanguage that provides a formal annotation of these relationships between elements and media objects in NCL documents. The incremental method implemented here also allows the fine tuning of the interval between subsequent validations, given one may wish a more immediate system response (by validating each change) or a more conservative and efficient behavior (by validating a group of changes).

This work also reports an implementation of our proposal with focus on the instantiation of the method described. The validator implemented has a Model that maintains the state of the NCL document being created and performs important preprocessing operations in order to efficiently support incremental validation. Although the Model primitives work before the validation itself (thereby not affecting the validation time) we perform a complexity analysis to detect how costly they are. Thus the validation ensures the entire document validity by only applying a set of structural and referential tests on the elements affected by the user editions. This separation of the validation process in two distinct phases allows it to be made smoothly since its complexity is scattered to different moments of the application development. We also present a case study and illustrate how our validator is integrated with the iTV authoring tool NCL Composer.

In order to verify the performance of the proposed method, comparative tests confronting standard and incremental validation have been carried out. We chose as factors the number of elements in the document, the interval between subsequent validations and the type of the validation method itself (incremental or standard) resulting in a 2^3 experimental design. The means, collected with a 95% confidence interval revealed that incremental validation, as proposed and implemented in this paper, can perform NCL document validation up to seven times faster than the standard method, with negligible performance penalty due to the use of validation steps. It was also found that the type of validation and the number of elements in the document were the most prominent factors in the experiment, accounting for more than two-thirds of the validation time.

As future work, we intend to perform comparative tests of our validator against other NCL validation tools such as NCL Validator, which is probably the most used tool for NCL validation since it already comes integrated with the NCL Eclipse and Composer. It is interesting to check the memory usage and average validation time of our tool when compared to NCL Validator into a simulated environment with scarce computational resources. Finally, we also want to perform a qualitative evaluation of our tool with NCL programmers.

References

1. Soares LFG, Barbosa SDJ (2009) Programando em NCL Programando em NCL 3.0: Desenvolvimento de Aplicações para o Middleware Ginga, TV Digital e Web. Elsevier, Rio de Janeiro
2. W3C (2002) XHTML™ 1.0 The Extensible HyperText Markup Language, 2nd edn. http://www.w3.org/TR/xhtml1/. Accessed 27 March 2013
3. W3C (2008) Synchronized Multimedia Integration Language (SMIL 3.0). http://www.w3.org/TR/2008/REC-SMIL3-20081201/. Accessed 27 March 2013
4. Soares LFG, Rodrigues RF, de Resende Costa RM (2006) Nested context model 3.0 part 6–NCL (nested context language) main profile. Tech. rep., Departamento de Informtica da Pontifcia Universidade Catlica do Rio de Janeiro. http://bib-di.inf.puc-rio.br/techreports/2006.htm
5. Petre M (1995) Why looking isn't always seeing: readership skills and graphical programming. Commun ACM 38(6):33–44. doi:10.1145/203241.203251
6. Balmin A, Papakonstantinou Y, Vianu V (2004) Incremental validation of xml documents. ACM Trans Database Syst 29(4):710–751. doi:10.1145/1042046.1042050
7. Hess J, Ley B, Ogonowski C, Wan L, Wulf V (2011) Jumping between devices and services: towards an integrated concept for social tv. In: Proceedings of the 9th international interactive conference on interactive television, ACM, New York, NY, USA, EuroITV '11, pp 11–20. doi:10.1145/2000119.2000122
8. de Resende Costa RM, Moreno MF, Rodrigues RF, Soares LFG, (2006) Live editing of hypermedia documents. In: Proceedings of the ACM symposium on Document engineering (DocEng '06). ACM, New York, pp 165–172. doi:10.1145/1166160.1166202
9. ITU–International Telecommunication Union (2009) Nested context language (NCL) and Ginga-NCL for IPTV services
10. LuaEclipse (2008) An integrated development environment for the lua programming language. http://luaeclipse.luaforge.net/. Accessed 27 March 2013
11. Eclipse JDT (2001) Eclipse java development tools (jdt). http://www.eclipse.org/jdt/. Accessed 27 March 2013
12. Eclipse (2001) The eclipse foundation open source community website. http://www.eclipse.org/. Accessed 27 March 2013
13. Eclipse CDT (2002) Eclipse cdt (c/c++ development tooling). http://www.eclipse.org/cdt/. Accessed 27 March 2013
14. W3C (2004) XML Schema Part 1: Structures Second Edition. http://www.w3.org/TR/xmlschema11-1/. Accessed 26 March 2013
15. Shivadas A (2004) Intelligent correction and validation tool for xml. University of Kansas, Dissertation
16. Kostoulas MG, Matsa M, Mendelsohn N, Perkins E, Heifets A, Mercaldi M (2006) Xml screamer: an integrated approach to high performance xml parsing, validation and deserialization. In: Proceedings of the 15th international conference on World Wide Web, ACM, New York, WWW '06, pp 93–102,. doi:10.1145/1135777.1135796
17. Apache (1999) Apache xerces. http://xerces.apache.org/. Accessed 27 March 2013
18. Yang CC, Yang YZ (2003) Smilauthor: An authoring system for smil-based multimedia presentations. Multimedia Tools Appl 21(3):243–260. doi:10.1023/A:1025770817293
19. Bouyakoub S, Belkhir A (2011) Smil builder: An incremental authoring tool for smil documents. ACM Trans Multimedia Comput Commun Appl 7(1):2:1–2:30. doi:10.1145/1870121.1870123
20. Azevedo RGA, Soares Neto CdS, Teixeira MM, Santos RCM, Gomes TA (2011) Textual authoring of interactive digital tv applications. In: Proceddings of the 9th international interactive conference on Interactive television, ACM, New York, EuroITV '11, pp 235–244. doi:10.1145/2000119.2000169
21. Lima BS, Azevedo RGA, Moreno MF, Soares LFG (2010) Composer 3: Ambiente de autoria extensível, adaptável e multiplataforma. In: II Workshop de TV Digital Interativa, WTVDI (WebMedia'10)
22. LAWS (2009) Ncl validator. http://laws.deinf.ufma.br/nclvalidator/. Accessed 27 March 2013
23. W3C (1998) Document object model (dom) level 1 specification. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/. Accessed 27 March 2013
24. dos Santos JAF (2012) Multimedia and hypermedia document validation and verification using a model-driven approach. Universidade Federal Fluminense, Dissertation
25. Honorato GdSC, Barbosa SDJ (2010) Ncl-inspector: towards improving ncl code. In: Proceedings of the 2010 ACM Symposium on Applied Computing, ACM, New York, NY, USA, SAC '10, pp 1946–1947. doi:10.1145/1774088.1774500
26. Sun B, Yuan X, Kang H, Huang X, Guan Y (2010) Incremental validation of xml document based on simplified xml element sequence pattern. In: Proceedings of the 2010 Seventh Web Information Systems and Applications Conference, IEEE Computer Society, Washington, DC, USA, WISA '10, pp 110–114. doi:10.1109/WISA.2010.28
27. Thao C, Munson EV (2010) Using versioned tree data structure, change detection and node identity for three-way xml merging. In: Proceedings of the 10th ACM symposium on Document engineering, ACM, New York, NY, USA, DocEng '10, pp 77–86. doi:10.1145/1860559.1860578
28. Bouyakoub S, Belkhir A (2008) H-smil-net: A hierarchical petri net model for smil documents. In: Proceedings of the Tenth International Conference on Computer Modeling and Simulation, IEEE Computer Society, Washington, DC, UKSIM '08, pp 106–111. doi:10.1109/UKSIM.2008.54
29. Peterson JL (1981) Petri net theory and the modeling of systems. Prentice Hall PTR, Upper Saddle River
30. Yang CC (2000) Detection of the time conflicts for smil-based multimedia presentation. In: In Proc. of 2000 computer symposium (ICS-2000)-Workshop on Computer Networks, Internet, and Multimedia, pp 57–63
31. Soares LFG, Rodrigues RF (2005) Nested context model 3.0 part 1–NCM core. Tech. rep., Pontifícia Universidade Católica do Rio de Janeiro-Puc-Rio. http://bib-di.inf.puc-rio.br/techreports/2005.htm
32. Neto CdS Soares, Soares LFG, de Souza CS (2010) The Nested Context Language reuse features. J Braz Comput Soc 16:229–245. doi:10.1007/s13173-010-0017-z
33. Cerqueira Neto JR, Santos RCM, Soares Neto CS, Teixeira MM (2011) Método de validação estrutural e contextual de documentos ncl. In: Proceedings of the 17th Brazilian symposium on multimedia systems, WebMedia '11
34. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston
35. Davis M, Phillips A (2009) Tags for identifying languages. http://www.hjp.at/doc/rfc/rfc5646.html. Accessed 27 March 2013