J Braz Comput Soc (2010) 16: 229–245
DOI 10.1007/s13173-010-0017-z

ORIGINAL PAPER

The Nested Context Language reuse features

Carlos de Salles Soares Neto · Luiz Fernando Gomes Soares · Clarisse Sieckenius de Souza

Received: 8 December 2009 / Accepted: 16 July 2010 / Published online: 21 August 2010

Abstract NCL, the standard declarative language of the Brazilian Terrestrial Digital TV System and ITU-T Recommendation for IPTV Services, provides a high level of reuse in the design of hypermedia applications. In this paper we detail how its design and conceptual model have succeeded in supporting reuse at a declarative level. NCL supports not only static but also running code reuse. It also allows for reuse inside applications, reuse between applications, and reuse of code spans stored in external libraries. For a specification language to promote reuse, however, it must have a number of usability merits. Aspects of NCL usability are thus analyzed with the Cognitive Dimensions of Notation framework.

Keywords Ginga-NCL · DTV · NCL · SBTVD-T · Middleware · Code reuse · Declarative programming · Cognitive Dimensions of Notation

1 Introduction

This paper discusses syntactic code reuse (specification reuse) and running code reuse (execution reuse), from now on called active code reuse, in the context of NCL (Nested Context Language), a declarative DSL (Domain Specific Language) for hypermedia document authoring.

NCL is the declarative language of the Brazilian Terrestrial Digital TV System (SBTVD), supported by the middleware called Ginga. NCL and Ginga are part of ISDB (International Standard for Digital Broadcasting) standards [1], and also an ITU-T Recommendation for IPTV services [2]. Given that NCL is a DSL primarily designed for interactive digital TV (iDTV) applications, we will frame the discussion in this context.

Code reuse, or software reuse, has been an important and persistent topic in programming language and software engineering disciplines since their earliest days. The main goal has been to decrease program development time by reusing well-tested previously developed code, thus promoting increased software reliability and a specification that is less error-prone in its whole. However, not only static code reuse but also reuse of running code has been in focus. For example, programming languages for parallel systems allow for reusing code in execution by different parallel control flows of an application, producing different behavior in each flow.

Software engineering has proposed several methodologies and techniques for supporting code reuse. Typically and historically, these approaches have been developed for imperative languages, although they can be equally useful for declarative ones. Declarative languages emphasize the high-level description of an application rather than its decomposition into an algorithmic implementation. Moreover, declarative languages usually define specific models to design applications targeted to specific domains (a declarative DSL), offering a good balance between flexibility and simplicity. In other words, one loses some of the expressiveness but gains increased simplicity. Supporting reuse in declarative DSL is, however, more difficult to achieve, since reuse cannot come

C.S. Soares Neto (✉) · L.F.G. Soares · C.S. de Souza
Department of Informatics, Pontifical Catholic University of Rio de Janeiro, Rua Marquês de São Vicente, 225, Gávea, CEP 22453-900 Rio de Janeiro, RJ, Brazil
e-mail: csalles@deinf.ufma.br

L.F.G. Soares
e-mail: lfgs@inf.puc-rio.br

C.S. de Souza
e-mail: clarisse@inf.puc-rio.br

C.S. Soares Neto
Department of Informatics, Federal University of Maranhão, Av. dos Portugueses, s/n, Campus do Bacanga, CEP 65080-040 São Luís, MA, Brazil

at the expense of providing and keeping a clear orientation towards the domain for which the language was designed.

In this paper we detail how the NCL design and its conceptual model have succeeded in supporting reuse at a declarative level. However, in order to move from supporting to promoting reuse, a specification language must itself have a number of usability merits. If it does not, programmers (or multimedia document authors in our specific context) are likely to abandon it for a more usable specification tool. So, we take the Cognitive Dimensions of Notation (CDN) framework [3, 4] and analyze aspects of NCL usability as an interface language for creating hypermedia applications. Finally, we also include in this paper a brief outline of good programming practices that promote increased reuse in NCL-based authoring processes, which can help existing and future adopters of this language.

The next sections are organized as follows. As an underlying principle, Sect. 2 argues in favor of reuse in several aspects of an iDTV application. Section 3 first briefly presents NCL elements and then discusses how code reuse can be applied within the same NCL application. In Sect. 4, we show how to import and reuse code across different NCL applications. Section 5 presents the methodology we have applied to analyze NCL reuse features based on the CDN framework. Section 6 overviews some relevant related work. Finally, Sect. 7 presents our concluding remarks.

2 Reuse on iDTV applications

An iDTV application is constituted by media assets (video, audio, text and image perceptual media objects, as well as other media objects with imperative and declarative code) and spatial and temporal relationships among these assets. As parallelism is inherent to iDTV applications, there are several situations in which it is desirable to provide reuse both of active code (in this case media objects being presented) and of static code (in this case, syntactic reuse of part of a hypermedia document).[1]

Reuse of a media object's content is essential in iDTV applications. It is advantageous to let authors edit image, audio, and video files, and then reuse these contents in different documents or different parts of the same document. Content reuse can be related with both: (a) content specification (static reuse), allowing several media objects that refer to the same content to be independently presented by players; and (b) content presentation (active reuse), allowing for a unique content presentation to be shared by several media object players. Moreover, it is convenient to support not only content reuse but also extensive reuse of all other presentation attributes (the whole media object specification). In brief, we are thus talking about syntactic (static) reuse, which means having several independent instances of the same code (independent media object exhibitions), and reuse of the same media object presentation (active code reuse).

Reuse of presentation characteristics is also helpful. Applying interface design patterns tends to enhance iDTV application usability. When the same presentation pattern is used, families of iDTV applications gain proper identity and viewers become acquainted with that format. For example, if several media objects must always be exhibited in the same screen region of the same device, region-specification (static) reuse forces this pattern. If not only the location but also the way in which the media object is presented (for example the level of image transparency, font color, sound level, etc.) is declared apart, static reuse of this detached code span forces a predefined style.

Moving the reuse focus to structured document authoring, many iDTV applications follow a hierarchical organization. For example, a TV series may be composed of the TV channel logo (image) and various episodes; each episode is composed of scenes that are composed of shots and possibly some (perhaps interactive) advertisements; and so on. In other words, an iDTV application is a recursive composition of objects (media objects and other compositions). Very often several iDTV applications follow the same structure (the same TV script). Therefore, it is also convenient to support structure reuse of a whole iDTV application or part of it, to make production easier and also to enforce a format that gives identity to TV programs.

Going to the next reuse claim, one of the biggest advantages of iDTV applications is their capacity to adapt content or the way content is presented, depending on viewer profiles, viewer locations, or exhibition platform profiles. Adaptations are based on rule accomplishments, and rules should be reusable across different adaptations.

Going further, as aforementioned, iDTV applications are composed of media objects and relationships among them. Thus, reducing or at least simplifying the definition of relationships is essential to decrease the application development time and the programming-error risk. Relationships are defined by relation types plus relation actors (in our case, object interfaces) that play roles defined by the relation types. Relation types are difficult to specify, and it is a good practice to define them apart from applications and let them be shared by relationships.

It is also very usual to find part of iDTV applications repeated in other applications. This is, for example, the case of advertisements embedded in applications. Therefore it is worthwhile to treat application specifications as a library from which they can be imported by other applications. Indeed, the hierarchical nesting of compositions previously

[1] As usual in the hypermedia context, we will use the term document to refer to an application specification.

mentioned and exemplified in the case of a TV series encapsulation can be thought of as a document nesting. In the example, each composition can comprise a document. For example, each scene specification of an episode can be seen as a document that can be imported and included into another document specifying the whole episode, which in turn can be imported by another document representing the whole series.

Moreover, it is desirable to treat application libraries as an information base from which only some aspects of interest can be imported. For example, it should be possible to import merely the layout of an application to another application.

As seen, in iDTV application specifications reuse features are supposed to embrace numerous aspects involved in authoring and presentation tasks. It should be stressed at this point that although this paper focuses on reuse in NCL [2], the same approach can be exploited in other time-based DSLs targeted at the iDTV domain.

3 Reuse in a same NCL application

NCL is an XML application that follows the modularization approach. The NCL design naturally favors code reuse when detaching the several specification aspects of the authoring process of iDTV applications. This section begins with a brief presentation of NCL that will be useful in the discussion in the remainder of this section and in Sect. 4.

3.1 NCL overview

Very briefly presenting the main NCL elements (others will be described along the text), the general document structure is divided into <head> and <body> elements, as usual in W3C standards. The <head> contains all base elements (<regionBase>, <descriptorBase>, <ruleBase>, <transitionBase>, and <connectorBase>) whose child elements are targets of reuse by child elements of the <body>.

The <body> element contains <media>, <context>, <switch>, and <link> child elements. A <media> element defines a media object, specifying its type and its content location. The <context> element defines a context object. A context object is a composite that contains a set of objects (media, context, or switch) and a set of links (defining relationships among objects). The <switch> element allows the definition of alternative objects (represented by <media>, <context>, and <switch> elements) to be chosen during presentation time. Test rules used in choosing the switch component are defined by <rule> or <compositeRule> elements, defined in the <head> part of the document.

An object's interfaces are used in relationships with other objects' interfaces. The <area> element allows for the definition of a content anchor representing a spatial, temporal, or spatiotemporal segment of a media object's content, and also a code span in imperative media objects. The <property> element is used for defining an object's property (a local variable) or a group of object's properties as one of the object's interfaces. The <port> element of a <context> allows externalizing an interface of any of its internal child objects.

The temporal and spatial information needed to present each media object may be defined by <descriptor> elements. The element may refer to a <region> element to define the initial position of a <media> element in some output device. A <descriptor> element may also refer to a <transition> element to define transition effects in the beginning or end of a media object presentation. The <head> part of a document encloses the definition of <descriptor>, <region>, and <transition> elements.

A <causalConnector> element represents a relation that may be used for creating <link> elements. In a causal relation, a condition role shall be satisfied in order to trigger an action role. A <link> element binds an object's interfaces to connector roles (conditions or actions), defining a spatiotemporal relationship among objects.

Table 1 shows, in the first column, an NCL element that may reuse features defined in another NCL element presented in the third column, by using its attribute specified in the second column. However, these are not all of the reuse cases supported by NCL, as discussed in the following sections.

Table 1 Reuse in NCL

Element	Attribute	Reused element
media	refer	media
context	refer	context
descriptor	region	region
descriptor	transIn	transition
descriptor	transOut	transition
bind	descriptor	descriptor
link	xconnector	causalConnector
bindRule	rule	rule
switch	refer	switch
regionBase	region	region

3.2 Content reuse

In NCL, a media object specification defines its content, its interfaces (content anchors and properties), and how (and where) the content must be exhibited. In this section let us pay attention only to the content.

A media object defines its content by reference through its *src* attribute. The *src* value locates content independently from where it is (for example, stored in a local or remote file

```
<media id="video1"
       src="../media/movie.mp4"
       descriptor="descVideo1"/>
<media id="video2"
       src="../media/movie.mp4"
       descriptor="descVideo2"/>
<media id="video3"
       src="ncl-mirror://video2"
       descriptor="descVideo3"/>
```

Fig. 1 Content reuse

system, in a data carousel transmitted by some pushed data protocol, or in streaming pushed data). Therefore, an NCL application is composed by an NCL document, which describes the application structure and semantics, and by media contents defined on the outside.

The same content can be referred and reused by more than one distinct media object. Usually, these objects have independent exhibitions. Thus, for example, if a 100 s video is referred by two objects that are started within a 50 s lag, two 100 s exhibitions will be started from their beginning with a superposition of the 50 s end part of one onto the 50 s initial part of the other.

However, NCL also allows for a media object to define its content as a mirror of another media object's content. In the case of parallel presentations of the two objects, they must be identical. Going back to the previous example, if one content is defined as a mirror of the other one, the last instantiated (mirroring) media object would only have a duration of 50 s, starting from the second half part.

Figure 1 illustrates the first reuse case with "video1" and "video2" objects, and then the second reuse case with "video2" and "video3" media objects. The *descriptor* attribute is discussed in the next section.

3.3 Layout reuse

An important and time-consuming issue in hypermedia document design is the spatial positioning of media contents. The same occurs with the initialization of other presentation parameters specifying how a media object must be presented. These two issues are addressed with first class entities in an NCL document, namely <region> and <descriptor>.

Each set of <region> elements constitutes a <regionBase>, defining rectangular regions in a presentation device (a TV screen, for example) referred to by the *device* attribute of the <regionBase> element.[2] Regions can be superposed, as defined by their *zIndex* attribute.

Figure 2 exemplifies a scenario with two exhibition device classes: the base device, declared by default from

```
1:  <regionBase>
2:   <region id="backgroundRg"
             left="0%" top="0%" width="100%"
             height="100%"zIndex="1">
3:     <region id="centerRg"
               left="25%" top="25%" width="50%"
               height="50%" zIndex="2"/>
4:     <region id="buttonRg"
               right="2%" top="2%" width="5%"
               height="5%" zIndex="2"/>
5:   </region>
6:  </regionBase>
7:  <regionBase
             device="systemScreen(2)">
8:    <region id="merchandizingRg"
              left="25%" top="25%"
              height="50%" width="50%"/>
9:  </regionBase>
```

Fig. 2 Region bases for multiple devices

lines 1 to 6; and the "systemScreen(2)" class of devices, defined from lines 7 to 9. One or more devices can join a class. The base device has a "backgroundRg" region defined covering the whole screen. Two child regions are defined inside the "backgroundRg": "centerRg", corresponding to a half and centralized screen; and "buttonRg" where a button will be exhibited in the right upper corner. Devices registered in the "systemScreen(2)" class have only one region defined: "merchandizingRg", corresponding to a half-cut and centralized screen on this class of devices.

Figure 2 could be used in a scenario in which a video media object is exhibited in a central region ("centerRg") of a TV set, over (greater zIndex) a background image that fills the whole screen. At a certain moment an icon would appear in the upper right corner ("buttonRg"). If an interactivity button was pressed using a secondary device (for example, a remote control with a small screen), an advertisement would be exhibited in a central region ("merchandizingRg") of the secondary device screen. Note that in this scenario the advertisement would be presented only to those that had engaged an interaction, without annoying other viewers and making the watching experience more individual.

In NCL a <descriptorBase> element defines how media objects must be initially presented, detaching the content from the content presentation specification. This allows for a new reuse feature, since several media objects can share the same presentation initialization style.

Of course, the initial position of a media object is part of how an object must start its presentation. Thus a <descriptor> element can define by itself a region on a device screen or refer to a <region> element. Since several descriptors can share the same region, referring to a <region> element is another useful reuse feature.

Figure 3 adds a descriptor base to the scenario of Fig. 2. The "centerDs" descriptor refers to the "centerRg" region, also specifying that the sound level must be set to 50% of

[2]It should be remarked that NCL provides support to display an iDTV application on multiple exhibition devices working cooperatively under the NCL player control.

```
1:  <descriptorBase>
2:   <descriptor id="centerDs"
                 region="centerRg">
3:    <descriptorParam
       name="soundLevel" value="0.5"/>
4:   </descriptor>
5:   <descriptor id="buttonDs"
                 region="buttonRg">
6:    <descriptorParam
       name="transparency"
       value="0.2"/>
7:   </descriptor>
8:   <descriptor id="merchandizingDs"
                 region="merchandizingRg"/>
9:  </descriptorBase>
```

Fig. 3 Descriptor base

```
1:  <media id="merchandizing"
           descriptor="merchandizingDs"
           src="index.html"/>

2:  <!-- An alternative -->

3:  <media id="merchandizing"
           src="index.html">
4:   <property name="device"
                value="systemScreen(2)"/>
5:   <property name="left"
                value="25%"/>
6:   <property name="top" value="25%"/>
7:   <property name="height"
                value="50%"/>
8:   <property name="width"
                value="50%"/>
9:  </media>
```

Fig. 4 Alternatives for <media> element definition

its recorded level (lines 2 to 4). The "buttonDs" descriptor refers to the "buttonRg" region and also specifies that the image transparency must be set to 20% (lines 5 to 7). The "merchandizingDs" descriptor (lines 8 and 9) only refers to the "merchandizingRg" region in the secondary device.

Usually, NCL promotes reuse referring to identifiers of the reused parts, as location values used in the *src* attributes, or *id* values, when the reference is to an NCL entity, as in Fig. 1 (*descriptor* attribute values) and Fig. 3 (*region* attribute values). As in most XML application languages, NCL entities must have a distinct identifier in a document (the *id* attribute value).

It is important to stress that NCL allows for, but does not impose, reuse practices. Reference to <descriptor> and <region> elements are optional. A <descriptor> element could define a region using <descriptorParam> elements, thus without referring to <region> elements. A <media> element can also define all necessary presentation properties using <property> elements, without reference to a <descriptor> element. Indeed, descriptors are used only to initialize these properties. Figure 4 illustrates the <media> element whose content is the advertisement to be shown in the scenario illustrated in Figs. 2 and 3 (line 1), and an alternative for this <media> element definition without referring to descriptors (lines 3 to 9).

```
1:  <media id="mainVideo"
           descriptor="centerDs"
           src="sbtvd-ts://0x01.0x05"/>
2:  <media id="button"
           descriptor="buttonDs"
           src="media/redButton.png"/>
3:  <media id="merchandizing"
           descriptor="merchandizingDs"
           src="index.html"/>
4:  <media id="buttonRef"
           refer="button" instance="new"/>
```

Fig. 5 Media object reuse

3.4 Media object reuse

As stated in Sect. 2, when defining a media object it is necessary to specify its content location and its interface and how it must be presented. However, there is another way. A <media> element can be defined referring another one, inheriting all the media object definition (content, properties, and anchors).

Figure 5 extends the previous example scenario. In it, the main TV video (*id*="mainVideo") refers to the "centerDs" descriptor, as previously explained (line 1). Its *src* attribute specifies that the content must be retrieved from the transport stream, and more precisely, from the transport elementary stream with program_number.component_tag= "0×01.0×05". Line 2 defines the "button" media object, and line 3 the "merchandizing" HTML media object.

In line 4 of Fig. 5, the "buttonRef" media object refers to the "button" media object, inheriting all its definition (in this case, properties defined by the "buttonDs" descriptor). The *instance* attribute is explained in what follows.

In the NCL conceptual model, document structuring is made using contexts. A context is a special object that groups media objects and other contexts, recursively. The single constraint applicable is that a context cannot be included recursively in itself. A context can also contain relationships among its components. In NCL a context is represented by the <body> and the <context> elements. The <body> element is just a name for the ancestral of all contexts in an NCL document.

The context nesting creates what NCL calls a perspective for an object, a concept similar to the "scope" in general-purpose languages. An object's perspective is the nesting structure from the most external context (represented by the <body> element) to the object. In a perspective, an object inherits all relationships defined in ancestral contexts referring directly or indirectly to it.

Object reuse (in spite of being a media object or a context) allows an object to pertain to more than one perspective, and thus to more than one context. This is a very important reuse feature since a single object in different perspectives may have different behaviors (depending on the inherited relationships), promoting the concept of active code reuse for media objects.

In a <media> element specification that refers to (and thus reuses) another <media> element, the *instance* attribute allows for defining whether syntactic (static) reuse of code is desired (*instance*="new", see Fig. 5), or if active code reuse is the goal. In the last case, reuse may be either instantaneous (instance="instSame") or gradual (instance="gradSame"), as follows.

Assume that media object A refers to media object B.

(1) If *instance*="new", a new independent instance of A will be created when A is started, inheriting all defined code for B, independently of the fact that B is in exhibition or not. A inherits B interfaces, but can add new ones. Moreover, all interfaces take part only in the relationships defined in the perspective where the object was created.
(2) In the case of active code reuse (*instance* equal to "instSame" or "gradSame"), A and B are the same object.
   (i) If *instance*="instSame", interfaces for this same object come from the A and B specifications. Moreover, relationships defined both in the A perspective and in the B perspective are enabled.
   (ii) If *instance*="gradSame", the single difference is that each object representation must receive an action for its activation. Once active, its interfaces are incorporated to the common object in presentation. Likewise, relationships defined in the perspective of an activated media object become enabled.

Running media object code reuse has shown to be very useful, even more so than static code reuse.

3.5 Relation reuse

Unlike other XML languages, NCL detaches the relation and the relationship concepts [5], as is usual in ADLs (Architecture Description Languages [6]).

Relations are defined in a relation base in the head part of the document (<head> element). Relations define roles and the glue relating roles. NCL allows defining any kind of relation, but reserved words have been defined to simplify the definition of causal temporal and spatial relations. Causal relations are defined by <causalConnector> elements. In causal relations, conditions defined over roles must be satisfied in order to trigger actions to be applied in roles (the same or others).

Relationships (represented by NCL <link> elements) can be defined referring to a relation and defining actors to play the relation roles. Thus relation reuse is natural in NCL. Although this is the only way to define relationships in NCL 3.0, a syntactic sugar can be added to the language to allow relation and relationship definition within a single element, as discussed in Sect. 7.

As an example of relation reuse, assume that during a multimedia presentation the beginning of a video presentation must start an image presentation in parallel. Assume also that in a certain moment of the presentation the beginning of an audio track must start, also synchronously, an animation. These two relationships can share (reuse) the same relation: "on begin X role then start Y role". Based on this relation, the first relationship may bind the X role to the video and the Y role to the image. On the other hand, the second relationship may bind the X role to the audio track and the Y role to the animation.

It must be remarked that relationships may be defined among any media object type: perceptual media objects, media objects with imperative code (Lua code, for example), or media objects with NCL or other declarative language code (e.g., HTML).

3.6 Structure reuse

The context concept, defined in Sect. 3.4, is important not only to structure documents but also to encapsulate presentation semantics. Since a context contains not only objects but also relationships among them, a context actually defines a document embedded into another document: with a well-defined temporal and spatial semantics defined by the context's links.

Structuring an NCL document using contexts is a good programming practice. There are several examples (see www.club.ncl.org.br) that demonstrate its usefulness, some of them already discussed in this paper.

Just like media objects, contexts can be reused by another context definition. However, only static code reuse is allowed. Both contexts are considered independent new instances. There is no equivalent to "instSame" or "gradSame" values for context reuse.

Structure reuse is associated with the minimum and the maximum abstraction level exposed by the language notation, and by how many notation details can be encapsulated.

3.7 Rule and transition effect reuse

In the <head> of an NCL document rule bases and transition bases can be defined. Transitions are visual or acoustic effects that can be employed in the beginning or in the end of media object presentations. Rules make up a logic dialect that can be used to support content and content presentation adaptations.

Figure 6 illustrates an example of transition base (lines 2 to 4). Line 3 specifies the "fade" transition effect that must be applied to a content presentation during 3 seconds. Lines 12 and 13 show how a <descriptor> element refers to this transition to apply its effect in the beginning (*transIn*) of a media object presentation

```
1:  <head>
2:   <transitionBase>
3:    <transition id="fade3s" type="fade" dur="3s"/>
4:   </transitionBase>
5:   <ruleBase>
6:    <compositeRule id="canPlayPtVideoR" operator="and">
7:     <rule id="ptR" var="system.language" comparator="eq" value="pt"/>
8:     <rule id="processR" var="system.CPU" comparator="gt" value="0.4"/>
9:    </compositeRule>
10:  </ruleBase>
11:  <descriptorBase>
12:   <descriptor id="merchandizingDs" region="merchandizingRg" transIn="fade3s"/>
13:  </descriptorBase>
14: </head>
15: <body>
16:  <switch id="merchandizing">
17:    <bindRule rule="canPlayPtVideoR" constituent="vMerchandizing"/>
18:    <defaultComponent component="fMerchandizing"/>
19:    <media id="vMerchandizing" descriptor="merchandizingDs"
           src="merchandizing.mp4"/>
20:    <media id="fMerchandizing" descriptor="merchandizingDs" src="index.html"/>
21:  </switch>
22: </body>
```

Fig. 6 Transition and rule bases

It is likewise for the separation of the <descriptor> and <region>; detaching transitions effects from the descriptors allows their reuse. It is very common to have the definition of a few transition effects used by several media objects during an iDTV presentation. However, unlike what happens with regions, a <descriptor> element must always refer to a transition if the effect is requested. There is no way to define transitions as parameters of a <descriptor> element. Maybe this is a language concept that should be revised in the next version of the language.

As mentioned, rules express criteria that allow content and content presentation adaptations [7]. Rules can be simple or compound. Simple rules compare a variable to a value or to another variable. Compound rules are made up by the "or" and "and" expressions of rules (simple or compound).

In Fig. 6 the "canPlayPtVideoR" compound rule tests if a device can play a Portuguese video (lines 6 to 9), evaluating two system variables.

In NCL, <switch> elements contain alternative objects that can be chosen based on which rule is satisfied. Similar to <context> elements, <switch> elements can also be syntactically reused. As a <switch> example, Fig. 6 defines the <switch id="merchandizing" . . . > (lines 16 to 21). Line 17 specifies that if the exhibition device can play the Portuguese video ("canPlayPtVideoR" rule), the "vMerchandizing" media object must be selected. Otherwise, the "fMerchandizing" default media object is the correct choice.

Similarly to <switch> elements, <descriptorSwitch> elements contain alternative descriptors that can be chosen based on which rule is satisfied. Similarly to <descriptor> elements, <descriptorSwitch> elements can also be referred by media objects.

It is important to note that it is impossible to reuse a rule in order to define other rules. Moreover, rules can refer to global properties (properties of a special media object type: "application/x-ncl-settings") without having them explicitly declared. On the other hand, any other use of a property requires that the property be explicitly declared. Rules are rare exceptions in the NCL reuse coherence. Maybe this is a language concept that should be revised in the next version of the language.

4 Reuse across NCL applications

In all previously mentioned cases of Sect. 3, reuse was discussed within the scope of a single NCL application. However, it is possible to import information bases defined in other NCL applications and reuse their components as if they have been defined in the importing document. In addition, the whole structure of a document, or part of it, can be imported and reused. This allows for improving the final document structure more extensively than seen before, spreading its scope over several documents. For illustration, think of an advertisement (an NCL application) that may be embedded and exhibited in several iDTV applications (defined by other NCL documents).

4.1 Nesting NCL documents

There are two alternatives to reusing an entire document specification (that is, its structure and presentation layout). The first one amounts to importing a document and reusing it as a context object in the new document. The second one amounts to treating the document as a media object with NCL declarative code in the new document.

Importing document A as a context of document B allows for the creation of an independent copy of A inside B, as in any context reuse. The imported document must be referred to as "alias#docID" (where alias is an arbitrary string

(a) Document A.ncl

```
1: <?xml version="1.0"
        encoding="ISO-8859-1"?>
2: <ncl id="A" (...) >
3:    (...)
4: </ncl>
```

(b) Importing Document A.ncl into B.ncl

```
1: <?xml version="1.0"
        encoding="ISO-8859-1"?>
2: <ncl id="reuse-NCL" (...) >
3:  <head>
4:   <importedDocumentBase>
5:    <importNCL documentURI="A.ncl"
                 alias="docA"/>
6:   </importedDocumentBase>
7:  </head>
8:  <body>
9:  (...)
10: <context id="referDocA"
             refer="docA#A"/>
11: </body>
12: </ncl>
```

Fig. 7 Importing and nesting an NCL document

associated with the document URI and informed in the importing document's head; docId is the *id* attribute value associated with the imported document). Parts of an imported document can also be reused, but this discussion is left to Sect. 4.3.

Figure 7(a) illustrates an NCL document with *id*="A". The content of "A" is deliberately omitted without any prejudice. Figure 7(b) shows an NCL document with *id*="B" that imports "A" defining an alias ("docA") for this imported document in the "B" scope (line 5). The "referDocA" context reuses the "A" document definition referring to the "A" document *id* ("docA#A"). This is sufficient to create a copy of "A" able to be related with links defined in the "B" document. Note that when the context is exhibited, its node must follow the same layout defined in the "A" document.

The reuse of an NCL document as an NCL media object with NCL code ("application/x-ncl-NCL" type) allows for presenting the reused object in a region of an exhibition device screen, as if this region were the (virtual) available screen. Figure 8 illustrates this case, where the same document of Fig. 7(a) is now defined to be presented in a region specified by the "centerDs" descriptor.

4.2 Importing information bases

Every base defined in the <head> element of an NCL document can be imported by a corresponding base in another document by using the <importBase> element of NCL. This element defines an alias for the imported base and refers to the imported base through its URI (document's_URI#base_id).

Region bases and descriptor bases can be defined in files apart from NCL application specification files and then imported by these documents as if they were layout libraries.

(a) Document A.ncl

```
1: <?xml version="1.0"
        encoding="ISO-8859-1"?>
2: <ncl id="A" (...) >
3:    (...)
4: </ncl>
```

(b) Importing A.ncl into a region of C.ncl

```
1: <?xml version="1.0"
       encoding="ISO-8859-1"?>
2: <ncl id="C" (...) >
3:  <body>
4:   <media id="cDocA" src="A.ncl"
          type="application/x-ncl-NCL"
          descriptor="centerDs"/>
5:    (...)
6:  </body>
7: </ncl>
```

Fig. 8 Importing an NCL document to be exhibited in a region defined in the importing document

This is very useful in families of applications that should have the same presentation standard as a family identity and also improves usability. This is typical in iDTV dramas and other TV series.

Transition bases may also be imported and reused by descriptors of another document. The same works for rule bases. Practical experience shows that there is a trend of using the same set of rules by an author in several of its authored documents.

Connector base importing is the most common practice in NCL applications. Authors and organizations tend to add each relation they define in a common base shared by all their applications. Usually, <causalconnector> elements are defined by NCL language experts and put into a common connector base in order to be imported by naïve programmers into their applications. Mostly, a convention is defined by a company to identify their connectors in order to facilitate their reuse. It is rare in usual iDTV applications to need connectors different from the usual ones [1].

Importing and reusing connectors has an additional advantage of reducing the language learning curve, since connectors are one of the language's most difficult concepts. Importing and reusing connectors allows authors to produce iDTV applications from the very beginning of the language learning.

4.3 Importing objects of an NCL application

Section 4.1 discussed how a document can be imported and reused as a whole. In addition, parts of a document can also be reused. Indeed, any NCL object (media objects, contexts, and switches) can be reused.

In all these cases the reuse is similar to the whole document reuse. The single difference is to refer to the reused object identifier instead of the document identifier.

Fig. 9 Importing bases and documents

```
1:  <?xml version="1.0" encoding="ISO-8859-1"?>
2:  <ncl id="importing" (...) >
3:     <head>
4:        <descriptorBase>
5:           <importBase alias="desc" documentURI="descriptors.ncl"/>
6:        </descriptorBase>
7:        <connectorBase>
8:           <importBase alias="conn" documentURI="connectors.ncl"/>
9:        </connectorBase>
10:       <importedDocumentBase>
11:          <importNCL alias="docA" documentURI="A.ncl"/>
12:       </importedDocumentBase>
13:    </head>
14:    <body>
15:       <context id="refDoc" refer="docA#someContext"/>
16:       <media id="refMediaA" instance="new" refer="docA#someMedia"/>
17:       <media id="image" src="someImage.png" descriptor="desc#centerDs"/>
18:       (...)
19:       <link xconnector="conn#onBeginStopStart">
20:          <bind component="image" role="onBegin"/>
21:          <bind component="refMediaA" role="stop"/>
22:          <bind component="refDoc" role="start"/>
23:       </link>
24:    </body>
25: </ncl>
```

Figure 9 shows several importing examples. Two bases are imported by the document: the descriptor base in "descriptors.ncl" (line 5), and the connector base in "connectors.ncl" (line 8). In addition, the "A.ncl" is imported (line 11): its body content and all its bases.

In the body of the document in Fig. 9, the "someContext" imported context is reused (line 15) to be presented in the same regions defined in the imported "A" document. The "refMediaA" media object reuses the "someMedia" (line 16) media object defined in the "A" document. The "image" media object (line 17) with the content defined by the "someImage.png" file must be initially exhibited following the imported "centerDs" descriptor from "descriptors.ncl".

Still in Fig. 9, note that the <link> element (lines 19 to 23) refers to the imported "onBeginStopStart" connector. As can be noted, the nomenclature used is sufficient for an author to understand what the relation means. Thus, the <link> element only establishes the binds, defining that when the "image" media object begins to be presented (line 20), "refMediaA" media object must be stopped (line 21), and "refDoc" (line 22) context must start to be presented.

5 Analyzing NCL usability

Among possible methods to evaluate how usable NCL reuse features are for its intended users, there are empirical methods (involving empirical observations of how people actually use the features provided by NCL in real task situations or realistic lab settings) and analytic methods (derived from theories, models or frameworks, in varying degrees of formality). A combination of methods is clearly the best choice to gain insight and understanding with respect to how and why NCL supports reuse in practical contexts. However, one must decide how to combine them: for example, should they be applied independently of each other, or sequentially? Our choice in this research is the latter, starting with an analytic method. Among the advantages of this choice, two are particularly relevant. First, analytic methods can help detect specific features of NCL that we (its designers) were not aware of. Once detected, these features can then be empirically tested with NCL users in different contexts of use. Second, analytic methods can help establish theoretical or pretheoretical connections among (classes of) features, helping analysts in probing the nature and consequences of NCL features and relations among them. This possibility allows us to establish connections between NCL and other programming or specification languages, a factor that plays a major role in recruiting participants for empirical tests, for instance. To mention but one, if we were to find that NCL shares certain features with event-oriented programming languages, for example, empirical tests should perhaps discriminate between participants with and without prior knowledge of such languages.

A commonly used analytic tool for evaluating the usability of computer languages in the last few years has been the Cognitive Dimensions of Notation (CDN) framework [3, 4]. Although CDN originally consists of design principles for analyzing the usability of information artifacts, it only takes viewing computer languages as information artifacts themselves to gain considerable leverage in understanding the cognitive effort they may occasionally impose to programmers with different backgrounds and levels of expertise. It has, indeed, been lately criticized for not being "scientific according to standards normally applied in research" [8]; however, for researchers who acknowledge the

importance of interpretation and interpretive frameworks in science, this criticism does not apply. CDN can and, as we will show, does provide important insights on how to interpret the meaning and impact of syntactic and semantic features of programming languages on human cognition. It takes the analyst, however, an extra step compared to using noninterpretive predictive approaches in scientific research. We must define how each cognitive dimension is to be systematically applied to the object of analysis before we appreciate the results of such applications. So, we briefly list and define each dimension proposed by the CDN framework, stating how we apply it to the analysis of NCL. This application can be viewed as our subsidiary contribution for other researchers interested in analyzing other iDTV specification languages, and possibly other kinds of specification languages as well.

CDN is a framework commonly applied when thinking with diagrams [9], especially when designing Visual Programming Languages (VPLs). It has also been applied to evaluate different tools and programming languages. For example, Z formal specifications in the TranZit environment [10] and end user programming techniques in Interactive Football Playbook (IFP) [11] have been analyzed with CDN. In fact, because end user programming and development activities highlight the importance of usability and low cognitive loads in software development tools, CDN has gained popularity as an evaluation framework in this context [12], of which Web mash-ups [13, 14] and game scripting [15] are two noteworthy instances.

5.1 Method of analysis

The first step in our analysis was to cast NCL as an information artifact of some sort. Viewing a specification language as an artifact is easy. Artifacts are non-natural objects, man-made objects designed for a particular end. This being undoubtedly the case with NCL, we must only show that it is an information artifact. We take an information artifact to be one that not only carries information (which most artifacts arguably do, given that they carry meaning), but whose primary purpose is to represent and support information processing. This definition seems to accommodate the whole spectrum of artifacts that CDN has been used to analyze to date, and comfortably locates NCL within it for the purposes of the analysis that follows.

Next, computer languages are artifacts that have a dual nature. They represent information in a referential sense, and they also construct information in a generative sense [16]. In other words, they may not only refer to pre-existing information (for example a programming language construct like '`if a = 1 then ...`' tests an existing condition to verify whether `a` is equal to 1 or not), but they may also create representations as a consequence of computational execution (for example a construct like '`if a = 1 then b = false`' causes the assignment of value '`false`' to `b`, something that did not necessarily exist before). In practical terms, this dual nature leads us to include, in the analysis of computer languages, not only the linguistic constructs that it offers for specifying computer representations and processes, but also its operational semantics (i.e., what effects language constructs bring about when they are interpreted by a computer). We propose to add a third aspect of computer languages in the analysis: the programming infrastructure that supports programmers in creating and interpreting language constructs. Although the latter may be certainly considered an external factor that is not really intrinsic to the language being analyzed (for we can always use different editors and CASE tools to produce programs, regardless of the language we are working with), we claim that program editors, for example, underline and in an extensive way make explicit certain features of programming languages. This is especially true for NCL and its declarative abstractions.

In Fig. 10 we show how we organize the space of analysis. On the upper part of the figure, we see the NCL document, which is produced with the support of computer language editors. These typically focus on the lexical and syntactic aspects of computer languages, helping their users to perceive certain features of the language, avoid certain errors, establish certain relations, and so on. On the lower part of the figure, we see the NCL presentation, which results from a computer interpretation of the NCL code. This is carried out by computer interpreters called NCL Players that produce representations and behavior that help their users perceive the result of language constructs, their meaning,

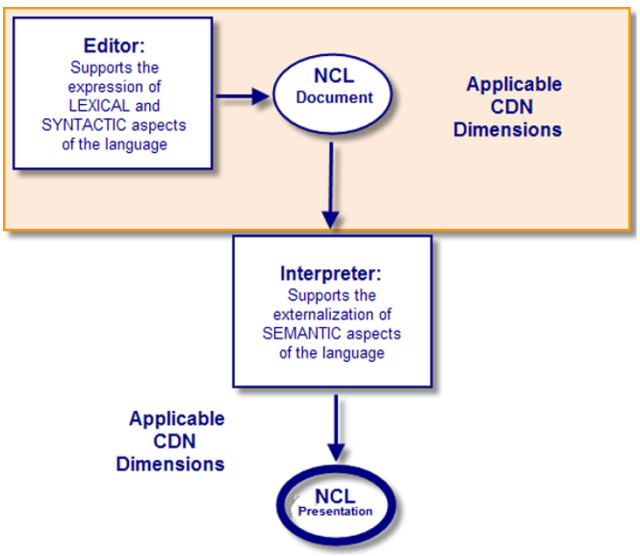

Fig. 10 Organizing the application of CDN dimensions to NCL features

scope, and the range of effects for which they can be used. Together, and only together, these two determine the usability of NCL as an information artifact. So, this is why we will include aspects of editors and interpreters in our analysis of NCL reuse with the CDN framework. Note that, as shown in Fig. 10, certain dimensions will be applied to both static and active code reuse, whereas others will only make sense (or be remarkably more relevant) in terms of one or the other. Note also that in the very definition of CDN dimensions, as is the case of visibility, for example, one must consider certain computational environment characteristics, lest we are not able to decide how elements of the language are "accessed" or "made visible". So, we think that our scope of analysis is sufficiently justified.

The editors we are considering in this analysis are: a non-specialized XML text editor, the NCL Eclipse [17], and the Composer [18]. The characteristics evaluated in the first environment are intrinsically related to NCL notation, focusing on an almost neutral environment that does not provide specific NCL features. NCL Eclipse [17], however, is a textual authoring tool with few graphical features that never overlap with text exhibition. It has been proposed to support author interaction directly with the NCL notation. Composer [18] is an authoring tool that makes use of four synchronized graphical views. Composer users can focus on specific views depending on the task they are currently performing. Obviously, NCL usability evaluation is different in each of these three environments, since they constitute different overall systems (notation + environment). The interpreter we are considering is the same in all three cases, NCL Player, the reference implementation of the Ginga standard [1, 19].

The thirteen CDN dimensions used in our analysis are defined in [3], and quoted below, along with notes on the specific interpretation we give to them in this paper.

- Visibility: "ability to view components easily".

Is all needed information easily identified and accessible when authors are editing part of an NCL document? How do editors improve visibility?

- Abstraction Gradient: "types and availability of abstraction mechanisms".

How easy is it to support composition and encapsulation in NCL? How do editors take advantage of the specification abstraction levels?

- Closeness of Mapping: "closeness of representation to domain".

How closely does the NCL document specification correspond to its presentation? How do editors help with this mapping?

- Consistency: "similar semantics are expressed in similar syntactic forms".

Is it possible to infer other NCL constructs once a basic subset of this language is learned? Are there exceptions or special cases to be learned?

- Diffuseness: "verbosity of language".

How many first-class NCL elements must be defined in a document for producing perceptible visual results in NCL presentations?

- Error Proneness: "the notation invites mistakes and the system gives little protection".

Does NCL notation induce programming or semantic errors? Do editors create new types of induced errors?

- Hard Mental Operations: "high demand on cognitive resources".

Which NCL authoring tasks become easier when one uses external annotations? How do editors make use of external annotations? Which NCL authoring tasks can be made easier by means of external cognitive aids? Which aids are these? Do editors and player provide them?

- Hidden Dependencies: "important links between entities are not visible".

How are dependencies identified in NCL documents? Which editor features are designed to help find dependencies?

- Premature Commitment: "constraints on the order of doing things".

In which order must NCL elements be created during document authoring? Do editors offer predefined orders or do they enable authors to create things in whichever order they wish?

- Progressive Evaluation: "work to date can be checked at any time".

Is it possible to get feedback for partially defined presentations of an NCL document? Is it possible to play incomplete documents?

- Role Expressiveness: "the purpose of an entity is readily inferred".

How obvious is the role of each NCL element in the whole solution?

- Secondary Notation: "extra information in means other than formal syntax".

Which are the comment and documentation types that can be found in NCL documents? How do editors handle comments and metadata?

- Viscosity: "resistance to change".

How much effort is required to make a change in an NCL document? What kind of changes are harder to commit?

5.2 Usability issues associated with NCL reuse

In this section we describe some typical needs authors have when creating NCL hypermedia applications. The section extends the analysis made in Sects. 3 and 4 by evaluating NCL usability in the context of using three authoring environments. Instead of evaluating the NCL notation expressiveness per se, we now evaluate how the notation communicates (to users) its design principles and the intent of its designers with the support of different computer environments that provide the necessary infrastructure for NCL programming.

The order in which the cognitive dimensions are discussed was chosen for fluency only. The reader should refer to the previous subsection to recall our interpretation of each cognitive dimension in the context of this research. As in previous sections, the term element below refers to an XML element, and the term attribute to an attribute of an XML element. As usual, XML attribute names appear in italics.

5.2.1 Visibility

Content reuse in NCL has the potential to introduce a visibility problem because an author may not see any media content (images, videos, etc.) by just reading an NCL document. However, this is an unavoidable problem, found in all textual languages designed for hypermedia document authoring.

NCL Eclipse supports preview of contents, so that images, audio, and videos can be easily displayed during the authoring phase. This helps authors to integrate contents mentally with their referring media objects. Although the latter is related to the visibility dimension, it is clear that it also alleviates the load of cognitive operations (another CDN dimension) inasmuch as it supports recognition rather than recall in retrieving objects by their names.

As NCL is a textual language for specifying hypermedia documents of any size, it will always pose some cognitive problem when a large document is being edited. This remains true even if more concise graphic abstractions are employed.

As mentioned in Sect. 3, running media object code reuse has shown to be very useful, even more so than static code reuse. However, in this case, the reuse can lessen the visibility problem, since the same object can be repeated in different parts of a code, only with its interfaces of interest.

Since relationships in NCL are always defined referring to a previously defined relation, this can also cause visibility cognitive problems. However, syntactic sugar can be added to the language to lessen this problem, as discussed in Sect. 7.

As discussed in Sect. 3, there is no way to define transitions as parameters of a <descriptor> element. This means that transitions are more prone to visibility cognitive problems than regions. Although this has not been considered relevant in the NCL 3.0 EDTV profile [1], this is a language concept that should be revised in the next version of the language.

In most general-purpose XML editors, it is possible to collapse and expand the XML tree structure in such a way that code spans can be hidden or shown, as desired. The same feature is offered both on NCL Eclipse and Composer tools. This, in itself, provides some visibility control.

NCL Eclipse, in addition, supports hypertext navigation from one editing position to another. Shortcuts (hyperlinks) are defined by NCL Eclipse when an NCL element refers to another element. With this feature, an NCL author can quickly switch from document body to head, in order to reach nonvisible code spans. The environment also provides a shortcut to return to the previous editing position.

5.2.2 Abstraction gradient

Structure reuse is in a straight line with the abstraction gradient cognitive dimension. Having hypermedia compositions (context) as the central abstraction concept of NCL shows the importance of this dimension in this language design, and the importance of this dimension in itself.

NCL notation is designed to help authors with several abstraction levels. General-purpose XML editors provide no support to help the encapsulation of code fragments.

The Composer structural view, however, represents the document as a composite graph, where graph nodes represent media objects or contexts, and graph edges represent links. This view reveals the several abstraction levels specified in an NCL document and makes it easier to achieve structure reuse. For example, authors can easily identify similarities between structures at a particular level of abstraction, and choose to reuse code corresponding to the abstractions they see.

5.2.3 Closeness of mapping

The NCL importing feature relates to the closeness of mapping dimension in CDN, as the concept can be considered a "programming trick". To import a document is to make its head and body "visible" to the importing document. Once the importing is concluded, reuse can be done by referring to the document bases or part of the body of the imported document.

As previously mentioned, NCL Eclipse treats an NCL specification as a hypertext, representing references to reused elements as hyperlinks. Thus, the hypermedia environment is itself a metaphor of the applications it has been designed to create. However, mapping NCL code onto NCL presentations in this editor is still a complex operation.

So, reuse opportunities that depend on recognizing common presentation features when looking at NCL code are more difficult to do.

Composer, in turn, makes use of multiple synchronized graphical views, each one of them related to a particular aspect of editing tasks, or to particular user profiles. In this way the tool meets a wider range of user needs, and is apt to support code reuse and abstraction mappings more easily than other editors. In structural view, a document is depicted as a composite graph, which may not be natural for authors without prior programming skills. The temporal view, however, depicts a document as an object that unfolds and behaves differently over time, which can be more familiar to audiovisual content producers with few program skills and more acquainted with timeline editions. These two representations can help authors detect reuse opportunities, which are closely related to presentational characteristics of a document, while dealing with the NCL code specification.

5.2.4 *Consistency*

The understanding of the three basic elements of NCL (<media>, <link>, and <context>) guide the understanding of the other ones; what is in line with the consistency cognitive dimension, that is, how much can be inferred from the knowledge of only part of the notation. Secondarily, the <descriptor> and <region> elements can be optionally used together with <media> elements. Likewise, <causalConnector> elements are used together with links. Interface elements (<area>, <property>, and <port>) are used together with <media> or < context> elements.

It is important to mention again that it is impossible to reuse a rule in order to define other rules. This is a rare exception in NCL reuse coherence, as previously mentioned. Still regarding consistency, as presented in Sect. 3, rules can refer to global properties without having them explicitly declared. As mentioned in that section, this is a language concept that should be revised in the next version of the language.

NCL has several merits in terms of consistency. As more extensively discussed in Sect. 3, understanding a small set of core concepts and a few XML elements is sufficient for authors to anticipate and guess the whole. We believe that the editing and playback infrastructure is not as relevant for analyzing NCL usability for purposes of reuse as the strictly linguistic aspects of NCL discussed in previous sections.

5.2.5 *Diffuseness*

The auto-complete feature of NCL Eclipse suggests valid values when editing NCL attributes. When an author activates auto-complete while editing the *region* attribute of a <descriptor> element, for example, only identifiers of <region> elements are presented. This quick reminder mechanism helps reduce the diffuseness problem by shortening codification distances between related elements, that is, the editor aggregates elements dispersed in the NCL code.

In Composer spatial view, authors are able to view and set all media positions applicable to any given point in presentation time. The spatial view helps decrease diffuseness by aggregating document representations on screen that are in one-to-one correspondence with the final exhibition spatial characteristics.

However, the diffuseness of NCL code can create problems when reusing an element because it introduces dependency chains (i.e., one element reuses another, which reuses another, and so on). Some common element chains are:

- media → descriptor → region
- media → descriptor → transition
- switch → rule → bindRule

This NCL feature is also related to hidden dependencies, further discussed in this section. With respect to diffuseness, we only remark that NCL "verbosity" somehow contributes to aggravating hidden dependency problems, and that our current editing and playback infrastructure does not yet provide robust solutions for them.

5.2.6 *Error proneness*

Two types of errors can be distinguished. The first is simply a defective NCL code, which makes an NCL document presentation impossible. Such errors should be detected during the authoring phase. NCL Eclipse provides efficient and extensive code validation support, which allows authors to easily find and remove programming errors. The second type of errors is semantic, which are found in properly coded documents that do not meet the author's expectations or intent.

NCL documents typically have several elements that refer to other elements. This is a welcome feature for reuse, but it can also create the need to check nonvisible code. When specifying a link, for example, the author needs to check the role cardinality in the referred connector specification. This information is not immediately available and can be misleading. The referring mechanism can lead to various other kinds of mistakes. Avoiding this is a matter of language design and programming infrastructure, tightly related to the previous discussion of the visibility dimension.

5.2.7 *Hard mental operations*

NCL coding and reuse still poses considerable problems related to hard mental operations. In spite of facilitating reuse in many other respects, there remain a few cases where improvement is clearly needed. Let us take NCL regions as an

example. A region is typically defined by attributes such as left, top, width, and height, which can have values in pixels or percentage (of the parent region). It is not so simple to mentally visualize the region area when reading the region attribute values. The region also carries the overlap information (*zIndex* attribute), stating which region must be presented when two or more of them occupy the same screen area. Authors usually need to display the desired content in order to find its appropriate position and dimension. Moreover, an object's exhibition area may change during a presentation. Therefore, defining and managing region geometries are considerably difficult tasks, and usually require associated tools. The problem is worse in reuse situations where related information is scattered throughout several elements.

Content preview in its corresponding screen area may help authors while performing such hard mental operations. As already mentioned, Composer's layout view supports region specifications by graphical manipulations. Basically, Composer displays a rectangle for each screen area. Their position and dimensions can be changed using drag-and-drop operations. The layout view also allows for screen area editing with respect to specific points on the presentation timeline, helping authors to make changes to previously defined parameters.

5.2.8 Hidden dependencies

As expected, when authors edit an NCL element, other elements that refer to it will change as well. In isolation, a referred element does not carry information about which other elements refer to it. Dependencies are only visible in the referring element.

NCL Eclipse provides a reverse hyperlink mechanism, previously mentioned, with which users can see the elements that refer to an element being edited. Thus, authors can check for dependencies in both directions, possibly identifying which is the impact of a change.

The synchronized views of Composer provide immediate feedback about changes to authors, helping to reduce hidden dependency effects.

There are several examples of hidden dependency problems specifically related to reuse. For example, when an attribute of a region changes, all associated descriptors collaterally receive the effects of this change. Likewise, changes in descriptors have impact on media objects that refer to them. Of course, this hidden dependency effect can be avoided if authors specify all presentation attributes using <property> elements, as discussed in Sect. 3.

5.2.9 Premature commitment

NCL Eclipse forces premature commitment because it validates the document as the editing goes. This gives authors an instantaneous feedback on codification errors and discourages the creation of documents that are purposefully incomplete or invalid. On the other hand, requiring a predefined order of specification of referred elements can annoy authors who usually choose to work using their own strategies. Thus, in NCL Eclipse, authors can turn off the warning messages generated by the code validation feature, and build specifications in a different order than the one that produces correct and playable code. The problem is, as will be emphasized in the next sub-section, that authors will not be able to see what their partial codification looks like as a presentation. The NCL Player requires strict syntactic constructs to be in place for it to play back a document.

Composer still has a primitive mechanism for code validation when compared to NCL Eclipse, and it does not advance the problem of playing back partially specified NCL code.

5.2.10 Progressive evaluation

Neither NCL Eclipse nor Composer supports any kind of progressive evaluation, or provides partial feedback for incomplete programs. Both environments are coupled to an external NCL interpreter that displays only valid and syntactically complete documents. This is an important usability issue, not only with respect to reuse, but also in general. In broad terms, it forces a particular top-down compositional strategy on the part of authors, which may go against the talent and preference of individual authors. It is thus an important item in the list of requirements for improved NCL usability in authoring tools.

5.2.11 Role expressiveness

The role expressiveness dimension helps to explain why NCL is easily learned by non-programmers and how reuse features help in this task. There are <media> elements to represent specific media objects (content). Causal sentences are expressed by first-class <link> elements, which reuse temporal relation specifications defined in connectors, which are also first-class elements (<causalConnector>). The same pattern is repeated for all elements needed to specify a document. Each element has a clearly defined role in the NCL data conceptual model.

Composer relies heavily on the role expressiveness of the NCL data model in providing its several authoring views. The auto-complete feature of NCL Eclipse also explores the role expressiveness extensively. Code suggestions are offered depending on the document position where authors are currently working. When giving suggestions inside the <ncl> element, for example, NCL Eclipse shows only two possible child elements: <head> or <body>. The same happens when the tool suggests attribute values, in particular, reference values for elements in reuse.

5.2.12 Secondary notation

XML comments provide a secondary notation to NCL. NCL also allows for the definition of metadata, through its <meta> and <metadata> elements. This can enhance the source code with extended semantic information. Secondary notation can help documentation about reuse features.

NCL Eclipse supports code highlighting, using different fonts and colors for element and attribute names and attribute values. The environment makes use of other types of secondary notation existing in NCL, such as XML comments and metadata. NCL Eclipse also supports in its current version its own documentation notation.

The current textual view of Composer does not support code highlighting. In other Composer's views, there could be extra annotations, but they are not implemented in the current version. XML comments are not explored by any of Composer's views except the textual view.

Secondary notation is ignored by the NCL interpreter.

5.2.13 Viscosity

In a way viscosity is a cognitive dimension affected by most (or all) of the previous dimensions discussed in this section. This dimension evaluates how difficult it is to make changes in an object of interest manipulated by an information artifact. In the context of NCL reuse, this practically amounts to asking how difficult it is to reuse objects. The previous subsections suggest that NCL editing and playback environments are still viscous mainly because of some issues with diffuseness, hard mental operations, hidden dependencies, and premature commitment. Mechanisms to control visibility and levels of abstraction, to support different kinds of mapping between representations and domain objects, to decrease error proneness, to increase role expressiveness, and to provide consistent language design and flexibility in incorporating secondary notation help users deal with current viscosity issues in NCL programming.

6 Related work

The NCL abstraction level is not close to that of low-level languages, such as HTML, but it is not close to that of higher level ones either, in particular to modeling languages in the Web field. Some reuse support is often considered in model-driven approaches. However, as NCL has an abstraction level related to its conceptual model, NCL should support reuse in its own declarative level.

There has been a lot of work reported in the literature about reuse provisioning in imperative languages [20, 21]. Although some declarative hypermedia authoring languages include reuse among their design principles, much less related work is found specifically about these languages. This section focuses only on reuse in declarative languages for hypermedia authoring.

In hypermedia authoring languages, at least the layout reuse is under concern since Style Sheets [22] were introduced for HTML. Likewise, content specification referring to an external file is common in several languages, such as SMIL (Synchronized Multimedia Integration Language), and, with lesser flexibility, especially with respect to anchor definition, in HTML.

Scalable Vector Graphics (SVG) [23], an XML-based W3C standard for describing vector graphics in two dimensions, allows for geometrical shape reuse through its <def> and <use> elements. Geometrical shapes are defined by <def> elements. To render them, an author must define the <use> element with *xlink:href*, *x* and *y* attributes. The *xlink:href* attribute specifies the geometrical shape identifier, as defined by the <def> element. The *x* and *y* attributes specify the position where the geometric shape must appear on the screen. In addition to this reuse feature, the <def> element also allows an author to include (import and reuse) elements from external namespaces, making SVG an easily extensible language.

SMIL [24], also an XML-based W3C standard, allows for several reuse features similar to the NCL ones. Like NCL, the specification of the presentation layout and the transition effects are made in the <head> part of a SMIL document. However, unlike NCL, the layout specification must be done in a single element, instead of the <region> and the <descriptor> division of NCL. This allows for extended reuse features in NCL. In SMIL, media objects and the document presentation semantics are defined in the <body> section, primarily including media objects in time containers (<par> and <seq> elements). SMIL does not allow for media object reuse, but only content reuse. Moreover, only static code reuse is allowed. No support is offered to active code reuse.

The synchronization paradigm of SMIL is based on compositions (containers) with temporal semantics (<par> and <seq> elements). There is no support for document structuring other than the temporal structure. Structure reuse is thus impossible. SMIL 3.0 has extended its synchronization paradigm to include relationships defined by "conceptual links" embedded into target content anchors. Unlike NCL relations, SMIL relations may not be reused.

Both SMIL and SVG show an impoverished import support, except for the previously mentioned SVG feature.

Neither SMIL, SVG, nor HTML provides any support for active code reuse. To the best of our knowledge, NCL is the only hypermedia authoring language with this feature.

7 Concluding remarks

The general NCL structure is composed of a header and a body. The header has several information bases, and the body specifies the organizational structure and the presentation semantics of a document. This general structure shows the designers' concern with reuse in NCL, in which language elements in the body frequently refer to elements in the header. This paper shows how an NCL author can benefit from the general structure of the language and create documents with higher levels of reuse.

The disconnection between hypermedia relations, on the one side, and relationships, on the other, is an important feature for NCL expressiveness and reuse. The definition of relation types is the most difficult task in authoring a document. However, once relations are defined, they can be reused in several relationships. As previously mentioned, it is common to see iDTV producers having a well-defined base of relations composed by expert programmers and shared among naive document authors. Visibility problems in using a predefined relation can be solved using a good naming strategy for relation identification. Terminology that is well known by authors, as exemplified in Sect. 3.5, is an asset in all authoring tasks.

It is rare to see an iDTV application that calls for connectors other than the usual ones. Several reserved words are defined by NCL in order to facilitate the creation of spatial and temporal relations. However, for the rare cases where new relation definitions are needed, a visibility problem persists. Relationships are created far from where relations are defined in the text. As opposed to other optional reuse features of NCL (like the layout definition), there is no option in this case. In order to bypass this problem, syntactic sugar has already been proposed for inclusion in future releases. This will allow for the joint definition of a relationship and its relation in a procedural code, written in quasi natural language, child of a pseudo link element. An NCL parser will then be responsible for translating this pseudo element into NCL <causalConnector> and <link> elements.

The definition of relationships as first-class entities, also allowing them to be part of hypermedia compositions (contexts), provides presentation semantics to these structures. In NCL, relationships that are external to contexts can act on them through well-defined interface points. Therefore, contexts encapsulate objects and their internal relationships. This concept is quite simple but very difficult to control [1]. It provides compositionality to contexts allowing for formal proofs of document properties [25], in addition to a rich potential for reuse.

Because of NCL support for document importing and nesting, and the ability to import information bases, the building process of large and complex documents becomes scalable, without bringing any collateral harm to the creation of simple and small documents. The reuse and import language features allow authors to rationalize and to easily understand the final presentation, grouping semantic information and accelerating inferential reasoning. This is another consequence of the encapsulation offered by contexts.

Detaching and spreading several authoring aspects into individual bases might be a point of criticism of NCL, since it seems that relevant interdependent parts of a document are being dispersed only to benefit reuse. However, this does not happen in practice because the information contained in each base is self-contained. The reference made to elements in a base helps to enrich the referring element with information that can be understood by itself (as first-class information) and not as partial information without meaning. Moreover, reuse is in the majority of cases an optional feature that can be avoided if necessary.

Concerning the editing and playback infrastructure of NCL as an information artifact, the CDN analysis suggests that for reuse in particular, but also for NCL specification in general, the environment currently provided for iDTV authors is still viscous. Viscosity arises mainly from issues with diffuseness, hard mental operations, hidden dependencies, and premature commitment. However, NCL merits in terms of language design consistency and multiple resources, visibility, abstraction, conceptual mapping, and role expressiveness, among others, contribute to alleviate problems.

The reuse of running code besides static code gives the language a unique reuse feature that is not found in any other declarative language for iDTV application authoring, as far as the authors know. This support makes the hypermedia application code cleaner, easier to understand, and less prone to errors [26].

The NCL language design and its conceptual model drive application authors to create documents with higher reuse degree. As a document increases in size along the authoring activity, there is a natural tendency to organize contents into contexts. This is in itself sufficient to promote reuse and to conjecture that sustained use of NCL leads to good programming practices.

Acknowledgements Carlos de Salles Soares Neto thanks CAPES for supporting his Ph.D. program. Luiz Fernando Soares and Clarisse de Souza thank CNPq for supporting their research financially.

References

1. ABNT NBR 15606-2:2007 (2009) Digital terrestrial television—Data coding and transmission specification for digital broadcasting—Part 2: Ginga-NCL for fixed and mobile receivers—XML application language for application coding. April 2009
2. Soares LFG, Rodrigues RF, Moreno MF (2007) Ginga-NCL: the declarative environment of the Brazilian digital TV system. J Braz Comput Soc 4(12):37–46

3. Blackwell AF, Green TRG (2003) Notational systems—the cognitive dimensions of notations framework. In: Carroll JM (ed) HCI models, theories and frameworks: toward a multidisciplinary science. Morgan Kaufmann, San Francisco, pp 103–134
4. Blackwell AF (2006) Ten years of cognitive dimensions in visual languages and computing. J Vis Lang Comput 17(4):285–287
5. Muchaluat-Saade DC, Rodrigues RF, Soares LFG (2002) XConnector: extending XLink to provide multimedia synchronization. In: II ACM symposium on document engineering—DocEng2002, McLean, USA
6. Clements PC (1996) A survey of architecture description languages. In: 8th international workshop on software specifications and design. IEEE Comput Soc, Washington
7. Soares LFG, Rodrigues RF, Cerqueira RFG, Barbosa SDJ (2009) Variable and state handling in NCL. Multimed Tools Appl. ISSN/ISBN: 13807501
8. Moody D (2009) Theory development in visual language research: beyond the cognitive dimensions of notations. In: IEEE symp. visual languages and human-centric computing, 2009. IEEE conference proceedings series. IEEE Press, New York, pp 151–154. doi:10.1109/VLHCC.2009.5295275
9. Blackwell AF, Whitley KN, Good J, Petre M (2002) Cognitive factors in programming with diagrams. Artif Intell Rev 15(1–2):95–114
10. Khazaei B, Triffitt E (2002) Applying cognitive dimensions to evaluate and improve the usability of *Z* formalism. In: SEKE '02: Proceedings of the 14th international conference on software engineering and knowledge engineering, July 2002
11. Neumann C, Metoyer RA, Burnett M (2009) End-user strategy programming. J Vis Lang Comput 20(1):16–29. doi:10.1016/j.jvlc.2008.04.005. ISSN 1045-926X
12. Guerra E, de Lara J, Malizia A, Diaz P (2009) Supporting user-oriented analysis for multi-view domain-specific visual languages. Inf Softw Technol 51(4):769–784. doi:10.1016/j.infsof.2008.09.005. ISSN 0950-5849
13. Le-Phuoc D, Polleres A, Hauswirth M, Tummarello G, Morbidoni C (2009) Rapid prototyping of semantic mash-ups through semantic web pipes. In: Proceedings of the 18th international conference on World Wide Web, Madrid, Spain, April 20–24, 2009
14. Ennals R, Gay D (2007) User-friendly functional programming for web mashups. In: ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on functional programming, October 2007
15. Kauhanen M, Biddle R (2007) Cognitive dimensions of a game scripting tool. In: Proceedings of the 2007 conference on future play, Toronto, Canada, November 14–17, 2007
16. Gelernter D, Jagganathan S (1990) Programming linguistics: a first course in the design and evolution of programming languages. MIT Press, Cambridge
17. Azevedo RGA, Lima BS, Soares Neto CS, Teixeira MM (2009) Uma abordagem para autoria textual de documentos hipermídia baseada no uso de visualização programática e navegação hipertextual. In: XV Simpósio Brasileiro de sistemas multimídia e Web—WebMedia 2009. Fortaleza, CE (available only in Portuguese)
18. Guimarães RL, Costa RMR, Soares LFG (2008) Composer: authoring tool for iTV programs. In: European interactive TV conference—EuroITV2008, Salzburg, Austria
19. ITU-T Recommendation H.761 (2009) Nested Context Language (NCL) and Ginga-NCL for IPTV services. Geneva
20. Johnson RE (1997) Components, frameworks, patterns. In: Proceedings of the 1997 symposium on software reusability. ACM, New York, pp 10–17. ISBN: 0-89791-945-9
21. Frakes WB, Fox CJ (1995) Sixteen questions about software reuse. Commun ACM 38(6):75-ff. ISSN: 0001-0782
22. Lie HW, Bos B (1997) Cascading style sheets. World Wide Web J 2(1):75–123. Special Issue on advancing HTML: style and substance. ISSN: 1085-2301. O'Reilly & Associates, Inc.
23. W3C (2008) Scalable Vector Graphics (SVG): XML graphics for the Web. http://www.w3.org/Graphics/SVG/.2008
24. W3C (2008) Synchronized Multimedia Integration Language (SMIL 3.0) W3C recommendation. http://www.w3.org/TR/2008/REC-SMIL3-200812.2008
25. Felix MF, Haeusler EH, Soares LFG (2002) Validating hypermedia documents: a timed automata approach. In: Monografias em ciência da computação—PUC-Rio, Brasil, 2002
26. Soares Neto CS, Souza CS, Soares LFG (2008) Linguagens computacionais como interfaces: um estudo com nested context language. In: Simpósio Brasileiro de fatores humanos em sistemas computacionais, Porto Alegre, RS, 2008 (available only in Portuguese)