# Relating Declarative Hypermedia Objects and Imperative Objects through the NCL Glue Language

Luiz Fernando Gomes Soares  Marcelo Ferreira Moreno  Francisco Sant'Anna

Pontifical Catholic University of Rio de Janeiro – PUC-Rio
Rua Marquês de São Vicente 225
22453-900 Rio de Janeiro, RJ, Brazil
+55 21 3527-1500 Ext: 3503

{lfgs, moreno, fanna}@inf.puc-rio.br

# Relating Declarative Hypermedia Objects and Imperative Objects through the NCL Glue Language

Luiz Fernando Gomes Soares  Marcelo Ferreira Moreno  Francisco Sant'Anna

Pontifical Catholic University of Rio de Janeiro – PUC-Rio
Rua Marquês de São Vicente 225
22453-900 Rio de Janeiro, RJ, Brazil
+55 21 3527-1500 Ext: 3503

{lfgs, moreno, fanna}@inf.puc-rio.br

## ABSTRACT

This paper focuses on the support provided by NCL (Nested Context Language) to relate objects with imperative code content and declarative hypermedia-objects (objects with declarative code content specifying hypermedia documents). NCL is the declarative language of the Brazilian Terrestrial Digital TV System (SBTVD) supported by its middleware called Ginga. NCL and Ginga are part of ISDB standards and also of ITU-T Recommendations for IPTV services.

The main contribution of this paper is the seamless way NCL integrates imperative and declarative language paradigms with no intrusion, maintaining a clear limit between embedded objects, independent of their coding content, and defining a behavior model that avoids side effects from one paradigm use to another.

## Categories and Subject Descriptors

I.7.2 [**Document and Text Processing**]: Document Preparation - *languages and systems, markup languages, multi/mixed media, hypertext/hypermedia, standards*.

D.3.2 [**Programming Languages**]: Language Classifications - *specialized application languages*.

## General Terms

Design, Languages, Standardization

## Keywords

NCL, Glue language, Declarative and Imperative code content, digital TV, intermedia synchronization, middleware.

## 1. INTRODUCTION

Declarative languages emphasize the high-level description of an application rather than its decomposition into an algorithmic implementation, as it is done when using imperative languages. Such declarative descriptions are easier to be devised and understood than imperative ones, which usually require programming expertise. Some declarative languages defines specific models to design applications targeted to specific domains (a declarative DSL ─ Domain Specific Language) offering a good balance between flexibility and simplicity, that is, losing some expressiveness but gaining still more simplicity.

Since an application can address problems in different domains, the use of declarative DSL objects (objects whose content are declarative-code spans using a specific DSL) acting in cooperation can be a good approach.

It is not unusual to see a declarative language embedding constructions defined in another declarative language to solve problems in different domains. This can be exemplified by the use of SMIL [1] constructs embedded in HTML (the so called HTML+SMIL) [2] or in SVG language [3], in order to provide these languages with abstractions to specify spatial and temporal synchronizations among their components. However, it is unusual to see components written in different declarative languages working in cooperation but maintaining their individuality, with a minimum intrusion, assuring that operations executed in one language environment do not affect the other environment.

Although it is unusual to see different declarative objects working in cooperation, problems requiring this solution are common. Take for example a digital TV application, written in a certain declarative language and sent to receivers. Each receiver in its turn could support an application written in other declarative language, for example, a recommender system, which could be integrated to the broadcasted application and then jointly sent to another receiver, in a typical Social TV application. Today, there are solutions addressing this typical example [11], but without a real integration between language environments. Transformations are performed on the received content in order to generate a unique content.

Despite the high level abstraction nature of declarative domain specific languages, it must be recognized that the richer expressiveness of general purpose imperative languages cannot be ignored, mainly for applications that produce many dynamic contents, as a result of complex operations. Therefore, imperative objects (objects whose content are imperative-code spans) may be the right solution to the development of certain tasks.


*DocEng'09,* September 16–18, 2009, Munich, Germany.

It is usual to see declarative DSLs embedding constructions defined in an imperative language to solve algorithmic problems, specially embedding imperative scripting languages. This can be exemplified by the use of ECMAScript with HTML [4] and BML [5]. However, the current approaches are usually very intrusive. The limit between the two programming paradigms is too flexible, making difficult code detaching, reuse and their conception by different programming teams. Moreover, the language's environments are so tightly coupled that unpredictable effects can be caused by an environment into another.

Recognizing that the richer expressiveness of general purpose imperative languages cannot be ignored, that the coexistence of declarative code chunks specified in different domain specific languages will be necessary in the digital TV domain, NCL (Nested Context Language [6]) is defined as a glue language to relate in time and space declarative objects, imperative objects and perceptual media objects (video, audio, image, text, etc.).

Indeed, as a glue language, NCL does not restrict or prescribe its object content types. In this sense, image objects (GIF, JPEG, etc.), video objects (MPEG, MOV, etc.), audio objects (MP3, WMA, etc.), text objects (TXT, PDF, etc.), imperative objects (Xlet, Lua, etc.), declarative objects (XHTML, SMIL, SVG, etc.), etc., are supported by the language. Which objects are supported depends only on which object players (engines) are coupled to the NCL formatter (player).

This paper focuses on how NCL provides support to relate imperative and declarative hypermedia objects. Declarative hypermedia objects are defined in the context of this paper as objects whose content is a hypermedia document specified using some declarative DSL). NCL is the declarative language of the Brazilian Terrestrial Digital TV System (SBTVD) supported by the SBTVD middleware called Ginga [7]. NCL and Ginga are part of ISDB (International Standard for Digital Broadcasting) standards (the previously known Japanese standard extended with Brazilian improvements). NCL and Ginga-NCL are also ITU-T H.761 Recommendation for IPTV services [9]. NCL and Ginga-NCL were designed at the TeleMidia Lab at PUC-Rio. The work has been coordinated by the authors of this paper that also chaired the ITU-T H.761 and the Brazilian DTV Middleware Working Group.

After presenting a motivation example of the NCL support to relate imperative objects and declarative hypermedia objects in the next section, this paper goes on briefly presenting some related work in Section 3. Section 4 discusses how object interfaces can be defined for all object types. Always using the example of Section 2 as a background, Section 5 presents the life cycle of media objects and how they can be related. Section 6 is reserved to final remarks.

## 2. MOTIVATION EXAMPLE

In order to illustrate how NCL relates imperative objects and declarative hypermedia objects, an example is detailed throughout this paper. This motivation example is introduced in this section also to illustrate the real need of different types of objects to work in collaboration in a DTV application.

The example uses multiple exhibition devices but is very simple. Consider the following application:

1. A film (a video animation) about a famous soccer player is broadcasted to be exhibited in a primary device (a TV set, in Figure 1).
2. During the video presentation, an advertisement about a soccer shoes will be played in a set of secondary devices (iPhones in Figure 1).
   a) When the advertisement is ready to be presented, an icon will appear in the TV set (right upper corner in Figure 1), during a certain period of time, in order to notify the existence of a secondary exhibition.
   b) During the same period of time, a soccer shoes icon will appear on secondary devices. If any viewer using a secondary device wants to buy soccer shoes, it must select the icon. After the selection the icon presentation will stop, an advertisement video and an HTML form will appear on the secondary device over a background picture. The selection also increments a counter, which holds the number of purchases performed by all viewers.
   c) A time is established for the advertisement purchase. When this time is over, the exhibition on secondary devices is stopped and the total number of purchases carried out by all viewers in this home network service is presented (in the TV set).

**Figure 1 – Multiple exhibition devices**

In order to implement this application, NCL will be used as a glue language for the following objects (called media-objects in NCL jargon) received by broadcasting (the film) and datacasting (all other objects and the NCL application itself), illustrated in Figure 2:

- The main video media object (the film);
- The icon (an image media object) that notifies the existence of secondary exhibition);
- The declarative hypermedia object representing the advertisement (NCLAdvert in Figure 2);
- The imperative object implementing the counter.

All these objects will be nested and related in the NCL application. The declarative hypermedia object could be implemented using any declarative language[1]. In order to illustrate how NCL can nest other NCL application, NCLAdvert will be implemented as another nested NCL application. This declarative hypermedia object, in its turn, relates a soccer shoes icon, the advertisement video, the background image and another declarative hypermedia object: an HTML object. Note then that we will have a declarative hypermedia object (NCLAdvert) also

[1] The reference implementation of Ginga supports XHTML, SMIL Tiny and NCL declarative hypermedia objects.

acting as a glue to relate other objects, including another declarative hypermedia object (HTML).

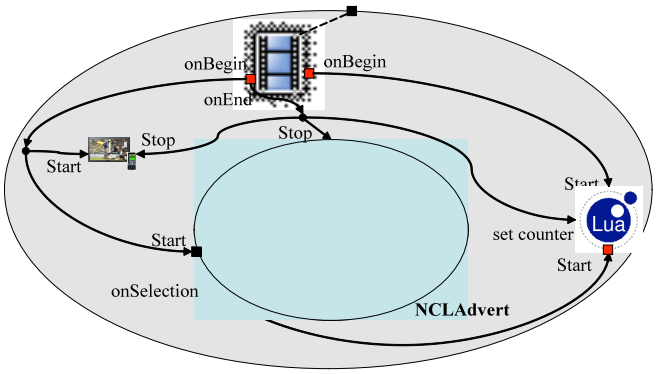


**Figure 2 – Structural view of the NCL parent application**

## 3. RELATED WORK

The modularization approach has been used in several W3C language recommendations. Modules are collections of semantically-related XML elements, attributes, and attribute values that represent a unit of functionality. Furthermore, a module specification may include a set of integration requirements, to which language profiles that include the module shall comply. It is not unusual to see declarative language profiles embedding modules defined by other declarative languages. As an example, in order to include temporal synchronization in its content, both HTML [2] and SVG [3] specific profiles include SMIL [1] modules. Note that this kind of integration does not define a clear limit between each language, since the elements and attributes imported are viewed by the programmer as new vocabulary added to the host language. In this solution only one language profile is defined; only one language paradigm guides the language functionality. In the case of the mentioned example, synchronization based on compositions with temporal semantics is the paradigm for declaratively specifying temporal synchronization.

This common integration is also found in NCL. NCL's Transition module and Metainformation module have the same functionality as SMIL homonyms, for example. However, NCL also allows embedding components written in different declarative languages working in cooperation but maintaining their individuality, with a very clear limit between the languages' paradigms. As an example, a parent NCL application allows embedding SMIL documents as one of its child media objects as well as other nested child NCL documents. In NCL media objects, temporal synchronization, for example, follows the event-driven paradigm and is specified by using NCL's link elements, while in SMIL child media objects maintains their paradigm for specifying temporal synchronization by using compositions with temporal semantics. Thus, the best of both approaches can be taken.

Like NCL, SMIL does not restrict its supported media types. This way, it is possible to have imperative and declarative content as new media types. However, SMIL does not define a standard general behavior for such media object players (engines), leaving details to particular SMIL implementations.

Example of systems that require different declarative hypermedia objects working in cooperation are not unusual. As aforementioned, in DTV domain it is common to receive applications written in a certain declarative language and to enrich this application with other content (information) probably created based on another declarative language. Today, there are solutions addressing this typical example, but without a real integration between language environments. In [11] the received content is transformed and joined with the enrichment content generating another application in the same language in which the enrichments are based.

Content enrichments can be achieved by using NCL Editing Commands [12]. Enrichments can be written in any desired declarative or imperative language, and can be generated by many users, which can collaborate in a wider social TV environment. The recommender system of [10] is being extended to adopt this approach.

As mentioned in Section 1, it is usual to see a declarative domain specific language embedding constructions defined in an imperative language to solve algorithmic problems. Three requirements should guide this integration. Indeed, the same requirements should also guide the integration between two declarative languages:

1) The languages should be modified as little as possible;

2) The border line between the two programming paradigms should be very well defined;

3) The relationship between the two language environments should be orthogonal in the sense that operations in one environment should not have unpredictable side effects in the other environment.

Probably the most widespread integration between declarative and imperative programming paradigms is by embedding ECMAScript code into XHTML documents. The ECMAScript language (through its dialect JavaScript) is an important component in the Web 2.0. However, the approach used in the integration between XHTML and ECMAScript is too intrusive, going against the loose-coupling principles raised in the previous paragraph:

1) Scripts are not self-contained XHTML entities exposing well behaved interfaces to the document. They actually export globals to be used throughout the document.

2) ECMAScript code spans are usually written inside documents, or, even more intrusively, inside XHTML tags or attributes.

3) ECMAScript has access and might change any part of the DOM tree of an XHTML document.

The Ambulant SMIL Player [13] added support for DOM mechanisms through the definition of "pseudo media objects", which contain imperative code chunks written in Python. As far as authors know, this is the first implementation of a SMIL player with support to imperative code spans embedded in SMIL documents. As with HTML, the free access to the DOM tree by scripts can compromise the whole document structure idealized at creation time. Such flexibility may be desired for complex tasks, but might be dangerous when used for everyday operations. The use of DOM to integrate imperative and declarative code spans is much more error-prone when dealing with documents that specify advanced multimedia relationships, like spatiotemporal relationships, since the document consistency can be lost after a structural modification.

Until its version 2.1, SMIL does not support imperative constructs in documents. In its version 3.0, through the State module [1], SMIL provides more control in applications by allowing explicit manipulation of variables and properties in a document. For instance, the attribute `expr` of media objects may contain an expression that avoids the playback of the object when evaluated to false, as in the following statement, where the audio is played only if the network connection is faster than 1Mbps:

Note however that this approach also mixes declarative and imperative code, characterizing a relaxed border between the two paradigms.

# 4. OBJECT INTERFACES

In NCL an interface shields the object content. No matter the object type, its content can only be exposed through its interfaces.

There are two types of media-object interfaces represented by NCL <area> and <property> elements.

## 4.1 Content Anchors

An <area> element defines a content anchor, that is, a subset of the information units that compose the object's content. What is an information unit of a media object depends on its type.

For a "video/xxx" media object, information units can be frames, or even more grainy, pixels in a frame. For an "audio/xxx" media object, information units can be audio samples. For a "text/xxx" media object, information units can be words, and so on. However, what defines a content anchor for declarative hypermedia-objects and imperative media-objects?

In an imperative media-object, imperative-code span may be associated with an <area> element through its *label* attribute. In this case the *label* value shall identify the code span (a function, a method, etc.).

A declarative hypermedia-object is handled by the NCL parent application as a set of temporal chains [15]. A temporal chain corresponds to a sequence of events (occurrences in time), initiated from the event that corresponds to the beginning of the declarative media-object presentation. As there are unpredictable events (events whose occurrence in time can only be determined during the media object presentation), as for example viewer interactions in a DTV application, the whole temporal chain can only be determined when its last unpredictable event occurs. Therefore, declarative media-object content cannot be determined a priori for all cases.

Figure 3 presents the temporal chain of the NCLAdvert declarative hypermedia-object introduced in Section 2. Note in the figure that there will be only one chain, which starts with the soccer shoes icon presentation. After starting, an unpredictable event (the user selection) can occur, making the chain to continue in the right table of Figure 3. Otherwise, it ends with the soccer shoes icon presentation ending, at 6 seconds (shown in the left table). It must be remarked that, albeit the declarative hypermedia-object has just one chain in this example, there are cases in which more than one chain can be defined. This will happen, for example, when a declarative hypermedia-object has more than one starting point.

| | |
|---|---|
| Start, Presentation, soccer shoes icon | **0s** |
| End, Presentation, soccer shoes icon | **6s** |

| | |
|---|---|
| Start, Selection, soccer shoes icon | **(0+X) s** |
| End, Presentation, soccer shoes icon | **(X+0)s** |
| Start, Presentation, background image | **(X+0)s** |
| Start, Presentation, soccer shoes video advert | **(X+0)s** |
| Start, Presentation, HTML form | **(X+0)s** |
| End, Presentation, soccer shoes video advert | **(X+13)s** |
| End, Presentation, HTML form | **(X+13)s** |

**Figure 3 – NCLAdvert temporal chain**

Although declarative hypermedia-object content cannot be determined a priori for all cases, content anchors may be defined. In NCL, sections in temporal chains may be associated with declarative hypermedia-object's child <area> elements using their *clip* attributes. In this case, the *clip* value is a triple "(chainId, beginOffset, endOffset)". The *chainId* parameter identifies one of the chains defined by the declarative hypermedia-object. The *beginOffset* and *endOffset* parameters define the begin time and the end time of the content anchor, with regards to the chain beginning time. When a declarative hypermedia-object defines just one temporal chain, like the example in Figure 3, the *chainId* parameter may be omitted. The *beginOffset* and *endOffset* may also be omitted, when they assume their default values: 0s and the chain end time, respectively.

A declarative hypermedia-object's content anchor can also refer to any content anchor defined inside the declarative code itself. In this case, the *label* attribute of the <area> element that defines the content anchor has a value such that the declarative hypermedia-object player is able to identify one of its internally defined content anchors. Thus, note that a declarative hypermedia-object can externalize content anchors defined inside its content to be used in relationships defined by the NCL parent object in which the declarative hypermedia-object is included.

Figure 4 presents content anchor definitions for child media objects of the parent NCL application of Figure 2. In Figure 4, media objects are defined by <media> elements. Content anchors are defined by <area> elements. The "anchor1" is an example of a video temporal anchor that starts 10 seconds after the beginning of the film and ends 20 seconds later. The "anchor2" defines a Lua function call identified by "show". Note that "anchor3" defines a chain identified by "interaction" in the declarative hypermedia-object "NCLAdvert". As the *beginOffset* and *endOffset* parameters are omitted, they assume their default values: 0s and the chain end time, respectively.

```
<body>
  <port id="entry" component="film"/>
  <media id="film" type="video/mpeg"
                       src="../media/ginga.mp4"
                       descriptor="videoDesc">
    <area id="anchor1" begin="10s" end="30s"/>
  </media>
  <media id="secIcon" src="../media/icon.png"
                       descriptor="imageDesc"/>
  <media id="counter" src="counter.lua"
                       descriptor="luaDesc">
    <property name="inc"/>
    <area id="anchor2" label="show"/>
  </media>
  <media id="NCLAdvert" type="application/x-ncl-
              ncl" src="../media/advert.ncl"
                       descriptor="nclDesc">
    <area id="anchor3" clip="interaction"/>
  </media>
  <media id="globalVar"
       type="application/x-ginga-settings">
    <property name="service.currentKeyMaster"/>
  </media>
...
</body>
```

**Figure 4 – Specification of <media> elements of the NCL application**

The declarative hypermedia-object player is in charge of interpreting the semantics associated with the object's content anchors. As an example, for a declarative media-object with NCL code, a temporal chain is identified by one of the NCL document entry points, defined by <port> elements[2], children of the document's <body> element. Figure 5 illustrates the <port id="interaction"…> element referenced by NCLAdvert's "anchor3" specified in Figure 4. Note that, as NCLAdvet defines only one entry point (<port> element), "anchor3" defines its *clip* attribute only as an example, since the unique entry point would have been assumed by default.

```
<body>
   <port id="interaction" component="icon"/>
   <media id="icon" src="../media/icon.png"
                       descriptor="iconDesc"/>
   <media id="background"
                src="../media/background.png"
                descriptor="backgroundDesc"/>
   <media id="shoes" src="../media/shoes.mp4"
                       descriptor="shoesDesc"/>
   <media id="ptForm" src="../media/ptForm.htm"
                       descriptor="formDesc"/>
…
</body>
```

**Figure 5 – Specification of <media> elements of NCLAdvert**

Usually, in declarative SMIL or SVG hypermedia objects it is not necessary to define content anchor's *clip* attribute, since only one temporal chain is defined by these objects. For these objects, anchors can be defined only if it is desired to start documents they define from a time lag from its specified beginning time.

In HTML declarative media-object, a content-anchor may specify in its *label* attribute a string that should be used by the media player to identify a content region to start the presentation of an HTML page.

[2] A <port> element always map to a child component interface.

In declarative hypermedia objects and imperative media objects, an <area> element may also be used just as an interface to be used as conditions of NCL links to trigger actions on other objects, as discussed in Section 5.

In NCL, every media object shall have an anchor with a region representing its whole content. This anchor is called the *whole content anchor* and it is declared by default in NCL documents. Every time an NCL component is referred without specifying one of its anchors, the *whole content anchor* is assumed, except for imperative media-objects as explained in what follows.

In a declarative hypermedia-object, the *whole content anchor* has a special meaning. It represents the presentation of any chain defined by the hypermedia-object. Every time a declarative hypermedia-object is started without specifying one of its content anchors, the *whole content anchor* is assumed, as usual, meaning that the presentation of every chain shall be started in parallel. Other declarative hypermedia-player behaviors are discussed in Section 5.1.

The *whole content anchor* has also a special meaning for imperative media-object. It represents the execution of any code span (function, methods, etc.) inside the imperative media-object. Another content anchor is also defined by default in imperative media-objects, called *main content anchor*. Every time an imperative media-object is started without specifying one of its content anchors or properties, the main content anchor is assumed and, as a consequence, the code span associated to it. In all other references to the imperative media-object without specifying one of its content anchors or properties, the whole content anchor is assumed.

## 4.2 Properties

Media objects (<media> elements) may have several embedded properties. Examples of these properties can be found among those that define media-object placements during a presentation, presentation durations, and others that define additional presentation characteristics: background, transparency, etc. Some properties have their values defined by the NCL engine, while others by application authors. However, in any case, when properties are used in relationships (exemplified in Section 5), they shall be explicitly declared in child <property> (interface) elements of a <media> element, as shown in Figure 4.

A <property> element defines the *name* attribute, which indicates the name of a property or property group, and an optional *value* attribute, defining an initial value for the *name* property. Properties defined in media objects acts like local variables, except properties defined in a special media object of type="application/ncl-settings", called settings object. This media object does not have content but only variables with different scope (global, channel, service, etc.). In Figure 4, the "globalVar" media object defines the "service.currentKeyMaster" variable that determines which media-object has the key navigation control in a DTV service.

Declarative hypermedia objects may have <property> elements used both to define usual presentation properties, and to externalize properties defined inside the hypermedia-object. Examples of the first group are usual properties to parameterize the hypermedia-object player behavior, like *left, top, soundLevel, background, etc*. In the second group are properties whose *name* attribute has a value such that the declarative hypermedia-object player is able to identify one of its internally defined properties.

As an example, for a declarative hypermedia-object with NCL code (<media type="application/x-ncl-NCL" …>) one of its <porperty> elements may be refer to a <port> element, child of its <body> element, through its *name* attribute (that must have the <port>'s *id* as its value). In its turn, the <port> element may be mapped to a <property> element defined by any object nested in the declarative NCL hypermedia-object, includind its settings object.

A <property> element defined in an imperative media-object may be mapped to a code span (function, method, etc.) or to a code attribute. In this case, the *name* attribute of the <property> element shall be used to identify the imperative-code span or the code attribute, respectively. When a <property> element is mapped to a code span (function, method, etc.) through its *name* attribute and values are assigned to the property, the code must be executed by the imperative media-object player with the assigned values interpreted as parameters passed to the code span. When a <property> element is mapped to an imperative-code attribute and values are assigned to the property, these values are reproduced in the associated imperative-code attribute.

Figure 4 shows an example of a <property> element of an imperative media object with Lua code ("counter"). This property identifies an "inc" code span that, when called, increments a counter.

In NCL, initial values may be assigned to properties when a <media> element is instantiated by an NCL player. Each NCL <media> element is associated with a <descriptor> element by its *descriptor* attribute, as shown in Figures 4 and 5. A <descriptor> element may have child <descriptorParam> elements, which are used to initialize implicit and explicitly declared properties.

# 5. RELATING OBJECTS: THE NCL GLUE

In NCL, relationships among media-objects are event-driven. They are defined by <link> elements that specify which events participate in the relationship and which is the relation.

## 5.1 Events

Content anchors and properties define events. An *event* denotes any occurrence in time with finite or infinitesimal duration [14], as usual in DTV domain [8]. A presentation event is defined by the presentation of a content anchor[3]. A selection event is defined by the selection of a content anchor. An attribution event is defined by the attribution of a value to a property.

Each event defines a state machine (see Figure 6) that must be maintained by the NCL player, based on information reported by every media object player.

A presentation event (associated with a media-object's content anchor) initializes in the sleeping state. At the beginning of the exhibition (or execution in the case of imperative objects) of its corresponding content anchor, the event goes to the occurring state. If the exhibition/execution is temporarily suspended, the event stays in the paused state, while this situation lasts. A presentation event may change from occurring to sleeping as a consequence of the natural end of the presentation/execution duration, or due to an action that stops the event. When the presentation/execution of an event is abruptly interrupted, through an abort command, the event goes to the sleeping state. The duration of an event is the time it remains in the occurring state. This duration may be intrinsic to the media object, explicitly specified by an author or derived from a relationship, as will be discussed in the next section.

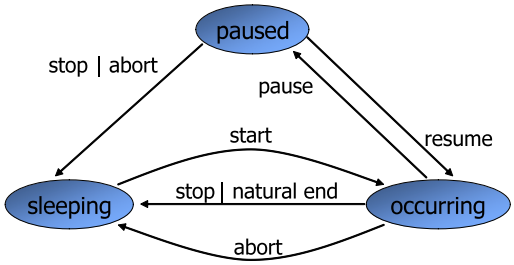


**Figure 6 -Event state machine**

It is important to remark that presentation events associated with *whole content anchors* of declarative hypermedia-objects or imperative media-objects have a different definition from other media objects, and very similar to presentation events defined on NCL composite objects. In these cases, a presentation event stays in the occurring state while at least one of the object's internal content anchors is being presented. It is in the paused state if at least one presentation event associated with one of the object's internal content anchors is in the paused state and all other presentation events associated with other content anchors are in the sleeping or paused state. Otherwise, the presentation event is in the sleeping state.

A selection event initializes in the sleeping state. It stays in the occurring state while the corresponding content anchor is being selected.

Attribution events stay in the occurring state while the corresponding property values are being modified.

If in an imperative media-object a *start* instruction is applied to a <property> element that calls the execution of a code span, no content anchor state is affected.

## 5.2 Relations and Relationships

In NCL it is possible to define constraint and causal relations. In NCL DTV profile [6] [7] [9] only causal relations are allowed. In causal relations, conditions defined on events must be satisfied in order to trigger action on events. Relations are defined using <connector> elements, as exemplified in Figure 7.

```
<connectorBase>
   <causalConnector id="onKeySelectionSet">
      <connectorParam name="var"/>
      <connectorParam name="keyCode"/>
      <simpleCondition role="onSelection"
               key="$keyCode" qualifier="or"/>
      <simpleAction role="set" value="$var"/>
   </causalConnector>
...
</connectorBase>
```

**Figure 7 – Example of causal relations in NCL**

[3] Presentation events may also be defined on NCL composite objects (represented by <body>, <context>, or <switch> elements of NCL [6]), representing the presentation of any content anchors defined by any object inside the composite object.

In Figure 7, a causal relation is defined specifying that when a remote control "key" (key="$keyCode") is selected (role="onSelection"), a value (value="$var") must be assigned (role="set"). Note in the example that both the key and the value are parameters to be defined. Note also that a relation does not identify which actors will play its roles. All these definitions are determined by relationships using the relation. Moreover, a role may be played by more than one actor, as exemplified in Figure 7 by the role="onSelection" whose corresponding condition will be considered satisfied if it is performed by any actor (qualifier="or").

Connector bases may be defined outside the NCL document specification (in another external document) and reused in relationships defined in other NCL documents that refer to these bases.

Relationships are defined by <link> elements and refer to relations through <link>'s *xconnector* attributes, as shown in Figure 8. The figure illustrates how media objects defined in Figure 4 can be related. The first link illustrates how a declarative hypermedia-object can be related with an imperative object, and the second one how a video media-object is related with a declarative hypermedia object and an imperative media-object.

```
<body>
...
 <link xconnector="onKeySelectionSet">
    <bind role="onSelection"
       component="NCLAdvert" interface="anchor3">
       <bindParam name="keyCode" value="RED"/>
    </bind>
    <bind role="set" component="counter"
                              interface="inc">
       <bindParam name="var" value="1"/>
    </bind>
 </link>
 <link xconnector="onEndStopStart">
    <bind role="onEnd" component="film"
                      interface="anchor1"/>
    <bind role="stop" component="secIcon"/>
    <bind role="stop" component="NCLAdvert"/>
    <bind role="start" component="counter"
                      interface="anchor2">
       <bindParam name="var" value="END"/>
    </bind>
 </link>
...
</body>
```

**Figure 8 – Example of relationships in NCL**

The first link refers to the connector defined in Figure 7 (xconnector="onKeySelectionSet"). Actors for each relation's role are defined through child <bind> elements.

The first <bind> element associates the role="onSelection" to a specific interface (interface="anchor3") of declarative hypermedia-objects instantiated from the <media id="NCLAdvert" interface="anchor3" ...>. This declarative content anchor specifies the clip="interaction" entry point (see Figure 4) that is mapped to the icon image (<port id="interaction" component="icon"/>), as defined in Figure 5. One instance of the declarative NCLAdvert media object will be created for each secondary exhibition devices (iPhones in Figure 1). This is specified by the NCLAdvert's descriptor (descriptor="nclDesc"), defined but not detailed in Figure 4. Since the role="onSelection" is defined by the <connector> element of Figure 7 with qualifier="or", any selection made on the soccer shoes icon will trigger the action role (role="set"). Moreover, the selection parameter key of Figure 7 is set to the RED key (<bindParam name="keyCode" value="RED"/>), meaning that the selection must be made using the RED remote control key.

Still in the first <link> element of Figure 7, the action triggered by the selection is specified to be applied to the "counter" imperative media-object (<bind role="set" component="counter" interface="inc">) in its interface (a property) "inc". As a result, a Lua procedure will be called, to increment a Lua variable named "counter"[4].

The second link specifies that when the film's "anchor1" presentation ends (<bind role="onEnd" component="film" interface="anchor1"/>), the presentation of the image that indicates secondary content must be stopped (<bind role="stop" component="secIcon"/>), all temporal chain of all declarative hypermedia objects must stop (<bind role="stop" component="NCLAdvert"/>) and the Lua procedure to show the counter final value must be called (<bind role="start" component="counter" interface="anchor2">).

Links defined in Figure 8 illustrate a one way communication from media objects (video and declarative hypermedia objects) to imperative objects. However, an imperative code may also command the start, stop, pause or resume of its associated content anchors through an API offered by the language [7]. These transitions may be used as conditions of NCL links to trigger actions on other objects of the same NCL document. Thus, a two-way synchronization can be established between the imperative code and the remainder of the NCL document.

## 5.3 Declarative Hypermedia Player and Imperative Player Behaviors

Declarative hypermedia objects and imperative media-objects have their life cycle controlled by the NCL application in which they are embedded. This implies an execution model different from when the declarative or imperative code runs under the total control of its own engine.

As with all media object players, imperative media-object and declarative hypermedia-object players must execute an initialization procedure when instantiated. However, different from other media players, imperative media-object initialization code is specified by the object author. This initialization procedure is executed only once, for each instance, and creates all code spans and data that may be used during the imperative-object execution. In particular, this procedure registers one (or more) event handler for communication with the NCL player. Note that at least the code span associated with the *main content anchor*, defined in Section 3.1, shall be created during the initialization procedure.

[4] The Lua code for this example is presented in Section 5.3.

As an example, Figure 9 illustrates the Lua code of the <media id="counter"…> element defined in Figure 4.

```
1  local counter = 0
2  function handler (evt)
3    if evt.class == 'ncl' and
4       evt.type == 'attribution' and
5        evt.name == 'inc' then
6      counter = counter + evt.value
7      event.post {
8        class   = 'ncl',
9        type    = 'attribution',
10       name    = 'inc',
11       action  = 'stop',
12       value   = counter,
13     }
14   elseif evt.class == 'ncl' and
15          evt.type  == 'presentation' and
16          evt.label == 'show' then
17     canvas:attrColor('yellow')
18     canvas:attrFont('vera', 24, 'bold')
19     canvas:drawText (9,9,
20         'Number of purchases: '..counter)
21     canvas:flush()
22   end
23 end
24 event.register(handler)
```

**Figure 9 – The Lua "counter" content**

When started, the Lua imperative media-object begins its initialization procedure shown in Figure 9, specified by the object author. In line 1 the counter is set to "0". Lines 2 to 23 contain the specification of the function that will handle attribution and presentation events coming from the NCL application (from links defined in Figure 8). Line 24 registers this function, that is, it leaves the media-object ready to receive events that will trigger this handling function.

After its initialization, the execution of an imperative media-object becomes event oriented in both directions. Any action commanded by the NCL player reaches the registered event handlers, and any NCL event state change notification is sent as an event to the NCL player.

Every media content players shall control event state machines associated with their events, reporting changes to their parent NCL player. A declarative hypermedia-object shall be able to reflect in its content anchors and properties behavior changes of its temporal chains. As an example, the NCL player of the NCLAdvert declarative hypermedia-object must inform its parent NCL Player when the soccer shoes icon is selected. This is done changing the event state machine associated with the "anchor3" content anchor, defined in Figure 4. Changes on this event state machine will then trigger actions defined by the first <link> element of Figure 8, under control of the parent NCL player.

Different from other media-object players, an imperative media-object player has not sufficient information to control by itself all event state machines, and shall rely on the imperative application content to command these controls. Thus an imperative media-object author must also take care of this task. As an example, when the attribution specified by the first <link> element of Figure 8 occurs (`<bind role="set" component="counter" interface="inc">`), the lua "counter" object handler function is called, and line 6 is executed, incrementing the counter. Note however that the <link> element only starts the attribution. Every time the handler function is called it must signalize the end of the attribution to the NCL formatter. This is an imperative media-object task that must be programmed by the media-object author, posting an end attribution event as specified in lines 7 to 13. As another example, when the "anchor2" presentation specified by the second <link> element of Figure 8 occurs (`<bind role="start" component="counter" interface="anchor2">`), the handler function is called, and lines 17 to 21 are executed, showing the counter result.

### *5.3.1 Starting Presentation Events*

The *start* instruction issued by an NCL player shall inform at least the following parameters to a media-object player: the media object to be controlled, its associated descriptor, a list of events (defined by the <media> element's <area> and <property> child elements, and by the default content anchors) that need to be monitored, and the presentation event that needs to be started.

If a media player receives a *start* instruction for an object already being presented (paused or not), it shall ignore the instruction and keep on controlling the ongoing presentation, except for declarative hypermedia objects and imperative media objects.

If a declarative hypermedia player receives a *start* instruction for a temporal chain already being presented (paused or not), it shall ignore the instruction and keep on controlling the ongoing presentation. However, if the start instruction is for a temporal chain that is not being presented, the instruction must be executed even if another temporal chain is being presented (paused or occurring).

Similarly, if an imperative media-object player receives a *start* instruction for an event associated with a content anchor and this event is in the *sleeping* state, it shall start the execution of the imperative code associated with the element, even though other portion of the media-object's imperative code is being in execution (paused or not). However, if the event associated with the target content anchor is in the *occurring* or *paused* state, the *start* instruction shall be ignored by the imperative-code player that keeps on controlling the ongoing execution.

As a consequence, different from what happens to other <media> elements, a <simpleAction> element with an *actionType* attribute equal to "stop", "pause", "resume" or "abort" shall be bound through a link to an imperative media object's interface or a declarative hypermedia-object's interface, which shall not be ignored when the action is applied. For other types of media-object, a <simpleAction> element with an *actionType* attribute equal to "stop", "pause", "resume" or "abort" does not need to identify an interface, and the action is applied to the content anchor being presented (paused or not).

As aforementioned, every time a declarative hypermedia-object is started without specifying one of its content anchors, the *whole content anchor* is assumed, as usual, but meaning that the presentation of every chain shall be started in parallel. As for imperative media-objects, every time they are started without specifying one of their content anchors or properties, the main content anchor is assumed and, as a consequence, the code span associated to it. In all other references to the imperative media-object without specifying one of its content anchors or properties, the whole content anchor is assumed.

In [7] a detailed behavior of media-objects presentation, selection and attribution event state machines can be found.

## 6. FINAL REMARKS

NCL is a glue language that can add functionalities written in another declarative or imperative language to an application. Joining the NCL facility of running on multiple devices in a cooperative environment, as that exemplified in Figure 1, together with the NCL support for live editing commands [12] several possibilities can be envisioned.

NCL editing commands allow including media objects and relationships in an NCL application during runtime. A device that is able to run an NCL player and other language engines can embed objects (and thus applications) written in these supported languages into NCL applications, even on-the-fly. So, it is possible, for example, to embed an application written in another language, for example a recommender system, into an NCL application being received by datacasting and played on real time, as is the case of DTV applications in agreement with the Brazilian terrestrial DTV system [7], or in any IPTV system in conformance with the ITU-T Recommendation H.761 for IPTV services [9].

The main contribution of this paper is the way NCL integrates imperative and declarative language paradigms with no intrusion, maintaining a clear limit between embedded objects, independent of their coding content, and defining a behavior model that avoids side effects from one paradigm use to another.

A future work is to go on exploring new applications that have other language media-objects running on secondary devices. Together with a SMIL group of CWI, we are planning some large experiments with devices able to run SMIL applications, such as recommender systems, and others.

Ginga-NCL open source implementation (the reference implementation of Brazilian DTV standard NCL player) can be obtained from http://www.softwarepublico.gov.br. Several examples of NCL applications can be obtained from http://club.ncl.org.br. All examples illustrated in this paper compose a unique NCL application (Figure 1) that can also be obtained from the last site.

## 7. ACKNOWLEDGMENTS

The authors would like to thank Marcio F. Moreno and Romualdo R. Costa for their valuable comments and their hard work implementation of all ideas here presented. This work has been supported by CNPq, MCT and FINEP.

## 8. REFERENCES

[1] W3C World-Wide Web Consortium. 2008. Synchronized Multimedia Integration Language – SMIL 3.0, W3C Recommendation. http://www.w3.org/TR/2008/ REC-SMIL3-20081201/

[2] W3C World-Wide Web Consortium. 1998. Timed Interactive Multimedia Extensions for HTML – HTML+TIME, W3C Recommendation. http://www.w3.org/TR/1998/ NOTE-HTMLplusTIME-19980918

[3] W3C World-Wide Web Consortium. 2003. Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation. http://www.w3.org/TR/2003/REC-SVG11-20030114/

[4] ETSI European Telecommunication Standards Institute. 2006. ETSI TS 102 812 V1.2.2 Digital Video Broadcasting "Digital Video Broadcasting (DVB); Multimedia Home Platform (MHP) Specification 1.1.1".

[5] ARIB Association of Radio Industries and Business. 2004. ARIB Standard B-24 Data Coding and Transmission Specifications for Digital Broadcasting, version 4.0, 2004.

[6] Soares L.F.G., Rodrigues R.F. 2006. Nested Context Language 3.0 Part 8 – NCL Digital TV Profiles. Technical Report. Departamento de Informática da PUC-Rio, MCC 35/06. http://www.ncl.org.br/documentos/NCL3.0-DTV.pdf.

[7] ABNT NBR Associação Brasileira de Normas Técnicas. 2007. Digital Terrestrial Television Standard 06: Data Codification and Transmission Specifications for Digital Broadcasting, Part 2 – GINGA-NCL: XML Application Language for Application Coding. http://www.abnt.org.br/imagens/Normalizacao_TV_Digital/ ABNTNBR15606-2_2007Ing_2008.pdf.

[8] ISO/IEC 13818-1, "Information technology — Generic coding of moving pictures and associated audio information: Systems", 1996, ISO/IEC.

[9] ITU-T Recommendation H.761, 2009. Nested Context Language (NCL) and Ginga-NCL for IPTV Services. Geneva, April 2009.

[10] Cattelan, R. G., Teixeira, C., Goularte, R., and Pimentel, M. D. 2008. Watch-and-comment as a paradigm toward ubiquitous interactive video editing. ACM Trans. Multimedia Comput. Commun. Appl. 4, 4 (Oct. 2008), 1-24. DOI= http://doi.acm.org/10.1145/1412196.1412201

[11] Cesar, P. S., Bulterman D.C.A., Jansen J., 2006. An Architecture for End-User TV Content Enrichment. Journal of Virtual Reality and Broadcasting, Volume 3(2006), no. 9.

[12] Costa R.M.R., Moreno M.F., Rodrigues R.F., Soares L.F.G. 2006. Live Editing of Hypermedia Documents. In Proceedings of ACM Symposium on Document Engineering (Amsterdam, Netherlands, 2006). DocEng 2006.

[13] Bulterman D.C.A., Jansen J., Kleanthous K., Blom K., Benden D. 2004. AMBULANT: A Fast, Multi-Platform Open Source SMIL Player. In Proceedings of ACM International Conference on Multimedia (New York, USA, 2004).

[14] Pérez-Luque M.J., Little T.D.C. "A Temporal Reference Framework for Multimedia Synchronization", *IEEE Journal on Selected Areas in Communications*, 14(1), January 1996.

[15] Soares L.F.G., Costa, R.M.R; Moreno,M.F. 2008. Graph-Based Schedulers for Resource Management and Presentation Control in a QoS Architecture for DTV Applications. Technical Report. Departamento de Informática da PUC-Rio, MCC 12/08.