PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 13/09

Nested Context Language 3.0

Part 13 - Ginga-NCL Implementors Guide v1.0

Luiz Fernando Gomes Soares, Marcelo Ferreira Moreno, Romualdo Resende Costa, Carlos Salles Soares Neto, Marcio Ferreira Moreno, Carlos Eduardo Freira Batista, Francisco Sant'Anna, Rafael Savignon, Felipe Nogueira, Guilherme Ferreira Lima, Bruno Seabra Lima, Roberto de Albuquerque Azevedo, José Geraldo de Sousa Jr., Eduardo Cruz Araújo, Alvaro Veiga, Felipe Nagato

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 13/09 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Março/2009

Nested Context Language 3.0

Part 13 - Ginga-NCL Implementors Guide v1.0

Luiz Fernando Gomes Soares, Marcelo Ferreira Moreno, Romualdo Resende Costa, Carlos Salles Soares Neto, Marcio Ferreira Moreno, Carlos Eduardo Freira Batista, Francisco Sant'Anna, Rafael Savignon, Felipe Nogueira, Guilherme Ferreira Lima, Bruno Seabra Lima, Roberto de Albuquerque Azevedo, José Geraldo de Sousa Jr., Eduardo Cruz Araújo, Alvaro Veiga, Felipe Nagato

Laboratório TeleMídia DI – PUC-Rio
Rua Marquês de São Vicente, 225, Rio de Janeiro, RJ - 22451-900.
`lfgs@inf.puc-rio.br`

Abstract. *This technical report provides operational guidelines for Ginga-NCL middleware implementations aiming at terrestrial and satellital DTV, and IPTV systems, which follows Norms ABNT 15606.2 and 15606-5, and ITU-T Recommendation H.761.*

Keywords:*, digital TV; middleware; declarative environment; NCL; Lua; Ginga-NCL.*

Resumo. *Este relatório técnico fornece o guia operacional para implementações do middleware Ginga-NCL, visando os sistemas de TV digital terrestre, por satélite e sistemas de IPTV que seguem as normas ABNT 15606.2 e 15606-5, e a Recomendação ITU-T H.761.*

Palavras chave: *TV digital; middleware; linguagem declarativa; NCL; Lua; Ginga-NCL.*

Nested Context Language 3.0
Part 13 - Ginga-NCL Implementors Guide v1.0

Laboratório TeleMídia

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rua Marquês de São Vicente, 225, Prédio ITS - Gávea

22451-900 – Rio de Janeiro – RJ – Brasil

http://www.telemidia.puc-rio.br

Table of Contents

Nested Context Language 3.0
Part 13 - Ginga-NCL Implementors Guide v1.0

Luiz Fernando Gomes Soares, Marcelo Ferreira Moreno, Romualdo Resende Costa, Carlos Salles Soares Neto, Marcio Ferreira Moreno, Carlos Eduardo Freira Batista, Francisco Sant'Anna, Rafael Savignon, Felipe Nogueira, Guilherme Ferreira Lima, Bruno Seabra Lima, Roberto de Albuquerque Azevedo, José Geraldo de Sousa Jr., Eduardo Cruz Araújo, Alvaro Veiga, Felipe Nagato

Laboratório TeleMídia DI – PUC-Rio
Rua Marquês de São Vicente, 225, Rio de Janeiro, RJ - 22451-900.

`lfgs@inf.puc-rio.br`

Abstract. *This technical report provides operational guidelines for Ginga-NCL middleware implementations aiming at terrestrial and satellital DTV, and IPTV systems, which follows Norms ABNT 15606.2 and 15606-5, and ITU-T Recommendation H.761.*

1. Introduction

Ginga-NCL is the declarative environment of Ginga middleware responsible that is for running NCL applications. This Technical Reference provides operational guidelines for Ginga-NCL middleware implementations aiming at terrestrial and satellital DTV and IPTV systems, which follows Norms ABNT 15606.2 and 15606-5, and ITU-T Recommendation H.761.

This Technical Report is firstly intended to be used by entities implementing receivers based on Ginga-NCL. Secondly, it is intended for developers of applications that use the Ginga functionalities and APIs. It shall be stressed however that the Ginga-NCL Operational Guidelines do not specify any implementation in a compliant receiver. The Operational Guidelines aim to ensure the same application behavior in different implementations of platforms supporting Ginga-NCL.

NOTE Ginga is a trademark of PUC-Rio and UFPB
NCL is a trademark of PUC-Rio

This report is organized as follows. Section 2 gives an historical evolution of the NCL versions. Section 3 presents a brief overview of the NCL 3.0 elements. Section 4 presents the identifiers for the modules and profiles of the NCL version 3.0. Section 5 discusses how the NCL 3.0 element and attribute semantics must be interpreted, which are their possible and default values, and the operational guidelines for an NCL formatter (user agent) when managing these elements and attributes. In Sections 6 the behavior of NCL media object players is established. Section 7 presents the guidelines for managing NCL editing commands and receiving data and events from DSM-CC object carousel. Section 8 presents the guidelines for NCLua media objects. Finally, Section 9 concludes the report.

2. NCL Historical Evolution

The first version of NCL [Antonacci 00, AMRS 00] was specified through an XML DTD – Document Type Definition [XML 1.0].

The second version of NCL, named NCL 2.0, was specified using XML Schema [SCHE 01]. Following recent trends, from version 2.0 on, NCL has been specified in a modular way, allowing the combination of its modules in language profiles.

Besides the modular structure, NCL 2.0 introduced new facilities to the previous version 1.0, among others:

- definition of hypermedia connectors and connector bases;
- use of hypermedia connectors for link authoring;
- definition of ports and maps for composite nodes, satisfying the document compositionality property;
- definition of hypermedia composite-node templates, allowing the specification of constraints on documents;
- definition of composite-node template bases;
- use of composite-node templates for authoring composite nodes;
- refinement of document specifications with content alternatives, through the <switch> element, grouping a set of alternative nodes;
- refinement of document specifications with presentation alternatives, through the <descriptorSwitch> element, grouping a set of alternative descriptors;
- use of a new spatial layout model.

NCL 2.1 brought some refinements to the previous version: a module for defining cost functions associated with media object duration was introduced; a module aiming at describing the selection rules of <switch> and <descriptorSwitch> elements was defined; and refinements in some NCL modules were made, mainly in the XTemplate module.

NCL 2.2 made minor refinements in some NCL 2.1 modules, concerning their element definitions, and introduced a different approach in defining NCL modules and profiles.

NCL 2.3 introduced two new modules for supporting base and entity reuse, and refined the definition of some elements in order to support the new features.

NCL 2.4 reviewed and refined the reuse support introduced in version 2.3, and the specification of the switch and descriptor switch elements. This version also split the Timing module introduced by NCL 2.1, creating a new module to encapsulate issues related with time-scaling operations (elastic time computation using temporal cost functions) in hypermedia documents.

The NCL 3.0 edition revised some functionalities contained in NCL 2.4. NCL 3.0 is more specific regarding some attribute values. This new version introduced two new functionalities, as well: Key Navigation and Animation functionalities. In addition, NCL 3.0 made depth modifications on the Composite-Node Template functionality and introduces some SMIL based modules to NCL profiles for transition effects in media presentation and for metadata definition. NCL 3.0 also reviewed the hypermedia connector specification in order to have a more concise notation. Relationships among imperative and declarative objects and other objects are also refined in NCL 3.0, as well as the behavior of imperative e declarative object players. Finally, NCL 3.0 also refined the support to multiple exhibition devices and introduced the support to NCL live editing commands.

NCM is the model underlying NCL. However, in its present version 3.0, NCL does not reflect all NCM 3.0 facilities yet. In order to understand NCL facilities in depth, it is necessary to understand the NCM concepts. With the aim of offering a scalable hypermedia model, with characteristics that may be progressively incorporated in hypermedia system implementations, the NCM and NCL family was divided in several parts.

The Nested Context Model is composed of Parts 1, 2, 3, and 4 of the collection:

- Part 1 – NCM Core
  concerned with the main model entities, which should be present in all NCM implementations[1].

- Part 2 – NCM Virtual Entities
  concerned mainly with the definition of virtual anchors, nodes and links.

- Part 3 – NCM Version Control
  concerned with model entities and attributes to support versioning.

- Part 4 – NCM Cooperative Work
  concerned with model entities and attributes to support cooperative document handling.

The NCL (Nested Context Language) specification is composed of Parts 5 to 12 of the collection:

- Part 5 – NCL (Nested Context Language) Full Profile
  concerned with the definition of an XML application language for authoring and exchanging NCM-based documents, using all NCL modules, including those for the definition and use of templates, and also the definition of constraint connectors, composite-connectors, temporal cost functions, transition effects and metainformation characterization.

- Part 6 – NCL (Nested Context Language) XConnector Profile Family
  concerned with the definition of an XML application language for authoring connector bases. One profile is defined for authoring causal connectors, another one for authoring causal and constraint connectors, and a third one for authoring both simple and composite connectors.

- Part 7 – Composite Node Templates
  concerned with the definition of the NCL Composite-Node Template functionality, and with the definition of an XML application language (XTemplate) for authoring template bases.

- Part 8 – NCL (Nested Context Language) Digital TV Profiles
  concerned with the definition of an XML application language for authoring documents

[1] It is also possible to have NCM implementations that ignore some of the basic entities, but this is not relevant so as to deserve a minimum-core definition.

aiming at the digital TV domain. Two profiles are defined: the Enhanced Digital TV (EDTV) profile and the Basic Digital TV (BDTV) profile.

- Part 9 – NCL Live Editing Commands
  concerned with editing commands used for live authoring applications based on NCL.
- Part 10 – Imperative Objects in NCL: The NCLua Scripting Language (this document)
  concerned with the definition of objects that contain imperative code and how these objects may be related with other objects in NCL applications.
- Part 11 – Declarative Hypermedia Objects in NCL: Nesting Objects with NCL Code in NCL Documents
  concerned with the definition of hypermedia objects that contain declarative code (including nested objects with NCL code) and how these objects may be related with other objects in an NCL application.
- Part 12 – Support to Multiple Exhibition Devices
  concerned with the use of multiple devices for simultaneously presenting an NCL document.
- Part 13 – Ginga-NCL Implementors Guide v1.0
  concerned with operational guidelines for Ginga-NCL middleware implementations.

In order to understand NCL, the reading of Part 1: NCM Core is recommended.

3. Overview of NCL Elements

NCL is an XML application that follows the modularization approach. The modularization approach has been used in several W3C language recommendations. A *module* is a collection of semantically-related XML elements, attributes, and attribute's values that represents a unit of functionality. Modules are defined in coherent sets. A *language profile* is a combination of modules. Several NCL profiles have been defined, among them those defined by Parts 5, 6, 7, and 8 of the NCL collection presented in Section 2. Of special interest are the profiles defined for Digital TV, the EDTVProfile (*Enhanced Digital TV Profile*) and the BDTVProfile (*Basic Digital TV Profile*). This section briefly describes the elements that compose these profiles. The complete definition of the NCL 3.0 modules for these profiles, using XML Schemas, is presented in [NCL Part 8]. Any ambiguity found in this text can be clarified by consulting the XML Schemas.

The basic NCL structure module defines the root element, called <ncl>, and its children elements, the <head> element and the <body> element, following the terminology adopted by other W3C standards.

The <head> element may have <importedDocumentBase>, <ruleBase>, <transitionBase> <regionBase>, <descriptorBase>, <connectorBase>, <meta>, and <metadata> elements as its children.

The <body> element may have <port>, <property>, <media>, <context>, <switch>, and <link> elements as its children. The <body> element is treated as an NCM context node. In NCM [NCL Part 1], the conceptual data model of NCL, a node may be a context, a switch or a media object. Context nodes may contain other NCM nodes and links. Switch nodes contain other NCM nodes. NCM nodes are represented by corresponding NCL elements.

The <media> element defines a media object specifying its type and its content location. NCL only defines how media objects are structured and related, in time and space. As a glue language, it does not restrict or prescribe the media-object content types. However, some types are defined by the language. For example: the "application/x-ncl-settings" type, specifying an object whose properties are global variables defined by the document author or are reserved environment variables that may be manipulated by the NCL document processing; and the "application/x-ncl-time" type, specifying a special <media> element whose content is the Universal Time Coordinated (UTC).

The <context> element is responsible for the definition of context nodes. An NCM context node is a particular type of NCM composite node and is defined as containing a set of nodes and a set of links [NCL Part 1]. Like the <body> element, a <context> element may have <port>, <property>, <media>, <context>, <switch>, and <link> elements as its children.

The <switch> element allows the definition of alternative document nodes (represented by <media>, <context>, and <switch> elements) to be chosen during presentation time. Test rules used in choosing the switch component to be presented are defined by <rule> or <compositeRule> elements that are grouped by the <ruleBase> element, defined as a child element of the <head> element.

The NCL Interfaces functionality allows the definition of node interfaces that are used in relationships with other node interfaces. The <area> element allows the definition of content anchors representing spatial portions, temporal portions, or temporal and spatial portions of a media object (<media> element) content. The <port> element specifies a composite node (<context>, <body> or <switch> element) port with its respective mapping to an interface of one of its child components. The <property> element is used for defining a node property or a group of node properties as one of the node's interfaces. The <switchPort> element allows the creation of <switch> element interfaces that are mapped to a set of alternative interfaces of the switch's internal nodes.

The <descriptor> element specifies temporal and spatial information needed to present each document component. The element may refer a <region> element to define the initial position of the <media> element (that is associated with the <descriptor> element) presentation in some output device. The definition of <descriptor> elements shall be included in the document head, inside the <descriptorBase> element, which specifies the set of descriptors of a document. Also inside the document <head> element, the <regionBase> element defines a set of <region> elements, each of which may contain another set of nested <region> elements, and so on, recursively; regions define device areas (e.g. screen windows) and are referenced by <descriptor> elements, as previously mentioned.

A <causalConnector> element represents a relation that may be used for creating <link> elements in documents. In a causal relation, a condition shall be satisfied in order to trigger an action. A <link> element binds (through its <bind> elements) a node interface with connector roles, defining a spatio-temporal relationship among objects (represented by <media>, <context>, <body> or <switch> elements).

The <descriptorSwitch> element contains a set of alternative descriptors to be associated with an object. Analogous to the <switch> element, the <descriptorSwitch> choice is done during the document presentation, using test rules defined by <rule> or <compositeRule> elements.

In order to allow an entity base to incorporate another already-defined base, the <importBase> element may be used. Additionally, an NCL document may be imported through the <importNCL> element. The <importedDocumentBase> element specifies a set of imported NCL documents, and shall also be defined as a child element of the <head> element.

Some important NCL element's attributes are defined in other NCL modules. The EntityReuse module allows an NCL element to be reused. This module defines the *refer* attribute, which refers to an element URI that will be reused. Only <media>, <context>, <body> and <switch> may be reused. The KeyNavigation module provides the extensions necessary to describe focus movement operations using a control device like a remote control. Basically, the module defines attributes that may be incorporated by <descriptor> elements. The Animation module provides the extensions necessary to describe what happens when a property value is changed. The change may be instantaneous, but it may also be carried out during an explicitly declared duration, either linearly or step by step. Basically, the Animation module defines attributes that may be incorporated by actions, defined as child elements of <causalConnector> elements.

Some SMIL functionalities are also incorporated by NCL. The <transition> element and some transition attributes have the same semantics of homonym element and attributes defined in the SMIL BasicTransitions module and the SMIL TransitionModifiers module. The NCL <transitionBase> element specifies a set of transition effects, defined by <transition> elements, and shall be defined as a child element of the <head> element.

Finally, the MetaInformation module is also incorporated, inheriting the same semantics of SMIL MetaInformation module. Meta-information does not contain content information that is used or display during a presentation. Instead, it contains information about content that is used or displayed. The Metainformation module contains two elements that allow describing NCL documents. The <meta> element specifies a single property/value pair. The <metadata> element contains information that is also related to meta-information of the document. It acts as the root element of an RDF tree: RDF element and its sub-elements (for more details, refer to W3C metadata recommendations [RDF 99]).

4. NCL: XML application declarative language for interactive multimedia presentations

Each NCL profile can group a subset of NCL modules, allowing the creation of languages according to user needs. Two profiles were defined for Ginga.

The *NCL 3.0 Enhanced DTV profile* includes the Structure, Layout, Media, Context, MediaContentAnchor, CompositeNodeInterface, PropertyAnchor, SwitchInterface, Descriptor, Linking, CausalConnectorFunctionality, ConnectorBase, TestRule, TestRuleUse, ContentControl, DescriptorControl, Timing, Import, EntityReuse, ExtendedEntityReuse KeyNavigation, Animation and TransitionBase modules of NCL 3.0, and also the BasicTransition, and Metainformation modules of SMIL2.0.

The *NCL 3.0 Basic DTV profile* includes the Structure, Layout, Media, Context, MediaContentAnchor, CompositeNodeInterface, PropertyAnchor, SwitchInterface, Descriptor, Linking, CausalConnectorFunctionality, ConnectorBase, TestRule, TestRuleUse, ContentControl, DescriptorControl, Timing, Import, EntityReuse, ExtendedEntityReuse and KeyNavigation modules.

The XML namespace identifier for the complete set of NCL 3.0 modules, elements and attributes, are contained within the following namespace:

- http://www.ncl.org.br/NCL3.0/

Profiles identify modules collectively. The following module collections are defined:

- http://www.ncl.org.br/NCL3.0/EDTVProfile

For the modules used by the NCL 3.0 Enhanced DTV profile.

- http://www.ncl.org.br/NCL3.0/BDTVProfile

For the modules used by the NCL 3.0 Basic DTV profile.

- http://www.ncl.org.br/NCL3.0/CausalConnectorProfile

For the modules used by the NCL 3.0 Causal Connector profile.

The following processing instructions must be written in an NCL document. They identify NCL documents that contain only elements defined in [NCL Part 8], and the NCL version to which the document conforms.

```
<?xml version="1.0" encoding="ISO-8859-15"?>
<ncl id="any string" xmlns="http://www.ncl.org.br/NCL3.0/profileName">
```

The <ncl> element *id* attribute may receive any string that ·matches· the NCName production in [Namespaces in XML] as a value.

The version number of an NCL document consists of a major number and a minor number, separated by a dot. The numbers are represented as a decimal number character string with leading zeros suppressed. The initial NCL standard version number is 3.0.

The profileName, in the URI path, must be EDTVProfile, BDTVProfile or CausalConnectorProfile.

5. NCL modules

The complete definition of these NCL 3.0 modules, using XML Schemas, is presented in "NCL –Nested Context Language 3.0 Part 8 – NCL Digital TV Profiles, which establishes the basis for ABNT Norms 15606-2 and 15606-5, and the ITU-T Recommendation H.761.Any ambiguity found in this text may be clarified by consulting the XML Schemas defined in [NCL Part 8].

For each NCL profile (Enhanced DTV profile and Basic DTV profile), a table is presented indicating the module elements and their attributes. For a given profile, the attributes and contents (child elements) of elements can be defined in the module itself or in the language profile that groups the modules. Therefore, the tables in this section show the attributes and contents that come from the profiles, besides those defined in the modules themselves. Tables in Section 5.1 show the attributes and contents that come from the NCL Enhanced DTV profile. Tables in Section 5.2 show the attributes and contents that come from the NCL Basic DTV profile. Element attributes that are required are underlined. In the tables, the following symbols are used: (?) optional (zero or one occurrence), (|) or, (*) zero or more occurrences, (+) one or more occurrences. The child element order is not specified in the tables.

Sections 5.3 to 5.25 discuss operation values and guidelines for each module.

5.1. NCL 3.0 Enhanced DTV profile

Table 5.1 – Extended Structure Module elements and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
ncl	*id, title, xmlns*	(head?, body?)
head		(importedDocumentBase?, ruleBase?, transitionBase?, regionBase*, descriptorBase?, connectorBase?, meta*, metadata*)
body	*id*	(port | property | media | context | switch| link | meta | metadata)*

Table 5.2 – Extended Layout Module elements and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
regionBase	*id, device, region*	(importBase|region)+
region	*<u>id</u>, title, left, right, top, bottom, height, width, zIndex*	(region)*

Table 5.3 – Extended Media Module element and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
media	*id, src, refer, instance, type, descriptor*	(area|property)*

Table 5.4 – Extended Context Module element and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
context	*id, refer*	(port|property|media|context|link|switch|meta|metadata)*

Table 5.5 – Extended MediaContentAnchor Module element and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
area	*id, coords, begin, end, text, position, first, last, label, clip*	empty

Table 5.6 – Extended CompositeNodeInterface Module element and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
port	*id, component, interface*	empty

Table 5.7 – Extended PropertyAnchor Module element and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
property	*name, value*	empty

Table 5.8 – Extended SwitchInterface Module elements and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
switchPort	*id*	mapping+
mapping	*component, interface*	empty

Table 5.9 – Extended Descriptor Module elements and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
descriptor	*<u>id</u>, player, explicitDur, region, freeze, moveLeft, moveRight, moveUp, moveDown, focusIndex, focusBorderColor, focusBorderWidth, focusBorderTransparency, focusSrc,focusSelSrc, selBorderColor, transIn, transOut*	(descriptorParam)*
descriptorParam	*<u>name</u>, <u>value</u>*	empty
descriptorBase	*id*	(importBase|descriptor|descriptorSwitch)+

Table 5.10 - Extended Linking Module elements and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
bind	*<u>role</u>, <u>component</u>, interface, descriptor*	(bindParam)*
bindParam	*<u>name</u>, <u>value</u>*	empty
linkParam	*<u>name</u>, <u>value</u>*	empty
link	*id, <u>xconnector</u>*	(linkParam*, bind+)

Table 5.11 – Extended CausalConnectorFunctionality module elements and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
causalConnector	*<u>id</u>*	(connectorParam*, (simpleCondition | compoundCondition), (simpleAction | compoundAction))
connectorParam	*<u>name</u>, type*	empty
simpleCondition	*<u>role</u>, delay, eventType, key, transition, min, max, qualifier*	empty
compoundCondition	*<u>operator</u>, delay*	((simpleCondition | compoundCondition)+,

		(assessmentStatement | compoundStatement)*)
simpleAction	*<u>role</u>, delay, eventType, actionType, value, min, max, qualifier, repeat, repeatDelay, duration, by*	empty
compoundAction	*<u>operator</u>, delay*	(simpleAction | compoundAction)+
assessmentStatement	*<u>comparator</u>*	(attributeAssessment, (attributeAssessment | valueAssessment))
attributeAssessment	*<u>role</u>, <u>eventType</u>, key, attributeType, offset*	empty
valueAssessment	*<u>value</u>*	empty
compoundStatement	*<u>operator</u>, isNegated*	(assessmentStatement | compoundStatement)+

Table 5.12 – Extended ConnectorBase module element and attribute used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
connectorBase	*id*	(importBase|causalConnector)*

Table 5.13 – Extended TestRule Module elements and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
ruleBase	*id*	(importBase|rule|compositeRule)+
rule	*id, <u>var</u>, <u>comparator</u>, <u>value</u>*	empty
compositeRule	*<u>id</u>, <u>operator</u>*	(rule | compositeRule)+

Table 5.14 – Extended TestRuleU<u>se</u> Module element and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
bindRule	*<u>constituent</u>, <u>rule</u>*	empty

Table 5.15 – Extended ContentControl Module elements and attributes in the Enhanced DTV profile

Elements	**Attributes**	**Content**
switch	*id*, *refer*	defaultComponent?, (switchPort | bindRule | media | context | switch)*)
defaultComponent	*component*	empty

Table 5.16 – Extended DescriptorControl Module elements and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
descriptorSwitch	*id*	(defaultDescriptor?, (bindRule | descriptor)*)
defaultDescriptor	*descriptor*	empty

Table 5.17 – Extended Import Module elements and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
importBase	*alias*, *documentURI*, *region*	empty
importedDocumentBase	*id*	(importNCL)+
importNCL	*alias*, *documentURI*	empty

Table 5.18 – Extended TransitionBase Module element and attribute used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
transitionBase	*id*	(importBase, transition)+

Table 5.19 – Extended BasicTransition module element and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
transition	*id*, *type*, *subtype, dur, startProgress, endProgress, direction, fadeColor, horRepeat, vertRepeat, borderWidth, borderColor*	empty

Table 5.20 – Extended Metainformation module elements and attributes used in the Enhanced DTV profile

Elements	**Attributes**	**Content**
meta	*<u>name</u>, <u>content</u>*	empty
metadata	*empty*	RDF tree

5.2. NCL 3.0 BasicDTV profile

Table 5.21 – Extended Structure Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
ncl	*id, title, xmlns*	(head?, body?)
head		(importedDocumentBase? ruleBase?, regionBase*, descriptorBase?, connectorBase?),
body	*id*	(port| property| media|context|switch|link)*

Table 5.22 - Extended Layout Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
regionBase	*id, device, region*	(importBase|region)+
Region	*<u>id</u>, title, left, right, top, bottom, height, width, zIndex*	(region)*

Table 5.23 – Extended Media Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
media	*<u>id</u>, src, refer, instance, type, descriptor*	(area|property)*

Table 5.24 – Extended Context Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
context	*<u>id</u>, refer*	(port|property|media|context|link|switch)*

Table 5.25 – Extended MediaContentAnchor Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
area	*id, coords, begin, end, text, position, first, last, label, clip*	empty

Table 5.26 – Extended CompositeNodeInterface Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
port	*id, component, interface*	empty

Table 5.27 – Extended PropertyAnchor Module elements and attributes in the Basic DTV profile

Elements	**Attributes**	**Content**
property	*name, value*	empty

Table 5.28 – Extended SwitchInterface Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
switchPort	*id*	mapping+
mapping	*component, interface*	empty

Table 5.29 – Extended Descriptor Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
descriptor	*id, player, explicitDur, region, freeze, moveLeft, moveRight, moveUp, moveDown, focusIndex, focusBorderColor; focusBorderWidth, focusBorderTransparency, focusSrc,focusSelSrc, selBorderColor*	(descriptorParam)*
descriptorParam	*name, value*	empty
descriptorBase	*id*	(importBase | descriptor | descriptorSwitch)+

Table 5.30 - Extended Linking Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
bind	*role*, *component*, *interface*, *descriptor*	(bindParam)*
bindParam	*name*, *value*	empty
linkParam	*name*, *value*	empty
link	*id*, *xconnector*	(linkParam*, bind+)

Table 5.31 – Extended CausalConnectorFunctionality module elements and attributes in the Basic DTV profile

Elements	**Attributes**	**Content**
causalConnector	*id*	(connectorParam*, (simpleCondition | compoundCondition), (simpleAction | compoundAction))
connectorParam	*name*, *type*	empty
simpleCondition	*role*, *delay*, *eventType*, *key*, *transition*, *min*, *max*, *qualifier*	empty
compoundCondition	*operator*, *delay*	((simpleCondition | compoundCondition)+, (assessmentStatement | compoundStatement)*)
simpleAction	*role*, *delay*, *eventType*, *actionType*, *value*, *min*, *max*, *qualifier*, *repeat*, *repeatDelay*	empty
compoundAction	*operator*, *delay*	(simpleAction | compoundAction)+
assessmentStatement	*comparator*	(attributeAssessment, (attributeAssessment | valueAssessment))
attributeAssessment	*role*, *eventType*, *key*, *attributeType*, *offset*	empty
valueAssessment	*value*	empty
compoundStatement	*operator*, *isNegated*	(assessmentStatement | compoundStatement)+

Table 5.32 – Extended ConnectorBase module element and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
connectorBase	*id*	(importBase|causalConnector)*

Table 5.33 – Extended TestRule Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
ruleBase	*id*	(importBase|rule|compositeRule)+
rule	*id, <u>var</u>, <u>comparator</u>, <u>value</u>*	empty
compositeRule	*<u>id</u>, <u>operator</u>*	(rule | compositeRule)+

Table 5.34 – Extended TestRuleUse Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
bindRule	*<u>constituent</u>, <u>rule</u>*	empty

Table 5.35 – Extended ContentControl Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
switch	*<u>id</u>, refer*	(defaultComponent?,(switchPort| bindRule|media| context | switch)*)
defaultComponent	*<u>component</u>*	empty

Table 5.36 – Extended DescriptorControl Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
descriptorSwitch	*<u>id</u>*	(defaultDescriptor?, (bindRule | descriptor)*)
defaultDescriptor	*<u>descriptor</u>*	empty

Table 5.37 – Extended Import Module elements and attributes used in the Basic DTV profile

Elements	**Attributes**	**Content**
importBase	*alias*, *documentURI*, *region*	empty
importedDocumentBase	*id*	(importNCL)+
importNCL	*alias*, *documentURI*,	empty

5.3. Structure module

The *xmlns* attribute of the <ncl> element declares an XML namespace — that is, it declares the primary collection of XML-defined constructs the document uses. Three values are allowed for the *xmlns* attribute: http://www.ncl.org.br/NCL3.0/EDTVProfile and http://www.ncl.org.br/NCL3.0/BDTVProfile, for the Enhanced and Basic DTV profiles, respectively, and http://www.ncl.org.br/NCL3.0/CausalConnectorProfile, for the Causal Connector profile. An NCL formatter shall know that the schemaLocation for these namespaces are, by default, respectively:

http://www.ncl.org.br/NCL3.0/profiles/NCL30EDTV.xsd,
http://www.ncl.org.br/NCL3.0/profiles/NCL30BDTV.xsd, and
http://www.ncl.org.br/NCL3.0/profiles/NCL30CausalConnector.xsd

5.3.1. Default values

There are not any default values.

5.3.2. Exception handling

1. Documents with *xmlns* attribute different from the three previously mentioned values shall be ignored by an implementation in conformance with [ABNT 15606-2, ABNT 15606-5 and ITU H.761].

2. *id* attributes whose values are not strings that match the NCName production in [Namespaces in XML] shall be ignored by an implementation in conformance with Standard ABNT NBR 15606 and ITU H.761.

5.4. Layout module

Each <regionBase> element is associated with a device where presentation will take place. In order to identify the association, the <regionBase> element defines the *device* attribute.

Each <regionBase> element is associated with a class of devices where presentation will take place. In order to identify the association, the <regionBase> element defines the *device* attribute, which may have the values: “systemScreen (i)” or “systemAudio(i)”, where *i* is an integer greater than or equal to 1.

There are two different types of device classes: active and passive. In an active class, a device is able to run media players. In a passive class, a device is not required to run media players, only to exhibit a bit map or a sequence of audio samples received from another device.

Multiple device support shall follow the guidelines established in “Nested Context Language 3.0: Part 12 – Support to Multiple Exhibition Devices” [NCL Part 12].

In SBTVD, systemScreen(1) and systemAudio(1) are reserved to passive classes, and systemScreen (2) and systemAudio(2) are reserved to active classes.

The interpretation of the region nesting inside a <regionBase> should be made by the software in charge of the document presentation orchestration (called *formatter*).

The position of a region, as specified by its *top*, *bottom*, *left*, and *right* attributes, is always relative to the parent geometry, which is defined by the parent <region> element or the total device area in the case of first nesting level regions. Attribute values may be non-negative “percentage” values, or integer pixel units. For pixel values, the author may omit the “px” unit qualifier (e.g. “100”). For percentage values, on the other hand, the “%” symbol shall be indicated (e.g. “50%”). The percentage is always relative to the parent’s width, in the case of *right*, *left* and *width* definitions, and parent’s height, in the case of *bottom*, *top* and *height* definitions.

The *top* and *left* attributes are the primary region positioning attributes. They place the left-top corner of the region in the specified distance away from the left-top edge of the parent region (or the device left-top edge in the case of the outermost region). Sometimes, explicitly setting the *bottom* and *right* attributes is helpful. Their values state the distance between the region’s right-bottom corner and the right-bottom corner of the parent region (or the device right-bottom edge in the case of the outermost region) (see Figure 5.1).

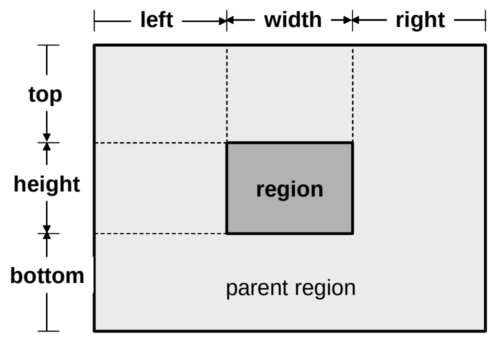

Figure 5.1 – Region positioning attributes

The *zIndex* attribute specifies the region superposition precedence, where regions with greater *zIndex* values are stacked on top of regions with smaller *zIndex* values. If two presentations generated by elements A and B have the same stack level then, if the display of an element B starts later than the display of an element A, the presentation of B is stacked on top of the presentation of A (temporal order); otherwise, if the display of the elements starts at the same time, the stacked order is chosen arbitrarily by the formatter.

5.4.1. Default values

1. The <regionBase> element that defines a passive class may also have a *region* attribute. If the attribute is not specified the exhibition will take place only on the passive class devices.

2. When a nested region doesn't specify a positioning or size value and this value cannot have its value computed from the other attributes, it shall assume the same value of the corresponding parent absolute attribute's value. In particular, when a first level region doesn't specify any positioning or size values, it will be assumed to be the whole device presentation area.

3. When not specified, the *zIndex* attribute shall be set equal to zero.

5.4.2. Exception handling

1. When the user specifies *top*, *bottom* and *height* information for the same <region>, spatial inconsistencies can occur. In this case, the *top* and *height* values shall have precedence over the *bottom* value. Analogously, when the user specifies inconsistent values for the *left*, *right* and *width* <region> attributes, the *left* and *width* values shall be used to compute a new *right* value.

2. Child regions must be completely contained in the area established by their parent regions. When some portion of the child region lies outside its parent region, the child region shall be ignored (considered as if it is not specified).

5.5. Media module

The Media module defines basic media object types. For defining media objects, this module defines the <media> element. Each media object has two main attributes, besides its *id* attribute: *src*, which defines the URI of the object content, and *type*, which defines the object type.

Note that media objects with the same *src* value and with its URI scheme different from "ncl-mirror" have the same content to be presented; however, the content of each object can be presented in a different moment in time, depending on which time the media objects were started. On the other hand, if the URI scheme is equal to "ncl-mirror", the media object whose *src* attribute defines this scheme and the media object referred by the scheme shall have the same content presentation in the same moment in time, when both media objects are being presented, independent from their starting time.

5.5.1. Continuous media object in TS elementary streams

As usual, if more than one media object has the same *src* attribute with a value referring to a content transported in the transport stream (TS) and are started, more than one presentation shall be started. Moreover, as usual, these objects can have different presentation regions that can be redimensioned by an application. However, it should be remarked that the number of media objects that can be presented in the SBTVD video plan depends on the receiver implementation. From the H.264 players implemented in hardware it is only required one content presentation at a time. To allow more than one video media object on the video plan is optional. If the number media objects of a certain type surplus the maximum allowed number, the start of exceeding media objects shall be ignored.

Regarding the video plan of the base exhibition device, if, and only if, there are not any media object being presented in this plan referring (through its *src* attribute) to a video elementary stream of a tuned transport stream (no matter in which application of the private base that represents this TV channel), a video ES that is not referred by a media object shall be presented, and in full screen. This video ES shall be the one referred by the last media object that had its presentation stopped in this video plan, or else, the primary video ES of the tuned transport stream, as defined by [ABNT 15604].

If, and only if there are not any media object referring (through its *src* attribute) to an audio stream of a tuned transport stream in any application of the private base that represents a TV channel being presented in the base exhibition device, an audio ES that is not referred by a media object shall be presented, and in full sound level. This audio ES shall be the one referred by the last media object that had its presentation stopped or else, the primary audio ES of the tuned transport stream, as defined by [ABNT 15604].

5.5.2. Special NCL object types

Five special types are defined: application/x-ginga-NCL (or application/x-ncl-NCL); application/x-ginga-NCLua (or application/x-ncl-NCLua; application/x-ginga-NCLet (or application/x-ncl-NCLet); application/x-ginga-settings (or application/x-ncl-settings); and application/x-ginga-time (or application/x-ncl-time).

Four very special objects are those that allow imperative and hypermedia declarative code embedded in an NCL document: text/html; application/x-ginga-NCL (or application/x-ncl-NCL); application/x-ginga-NCLua (or application/x-ncl-NCLua; application/x-ginga-NCLet (or application/x-ncl-NCLet). As such, they deserve to be discussed in the following three sections. The presentation of media objects also deserves a deep analysis, as presented in Section 6.

5.5.2.1. Declarative Hypermedia objects embedded in NCL presentations

Declarative hypermedia objects (with NCL code or coded with another declarative language) may be inserted into NCL documents. The way to add a declarative hypermedia object into an NCL document is to define a <media> element, whose content (located through the *src* attribute) is the declarative code to be executed. Both EDTV and BDTV profiles of NCL 3.0 allow the <media> element of text/html and application/x-ginga-NCL (or application/x-ncl-NCL) types to be nested in an NCL document.

5.5.2.1.1. NCL objects embedded in NCL applications

NCL objects embedded in NCL applications shall follow the guidelines established in "Nested Context Language 3.0: Part 11 – Declarative Hypermedia Objects in NCL: Nesting Objects with NCL code in NCL Documents" [NCL Part 11].

5.5.2.1.2. XHTML objects embedded in NCL applications

Any XHTML-based media object implementation shall support all common XML markups and stylesheet properties for the BML for basic services ("fixed terminal profile"), ACAP-X and DVB-HTML, as defined in ITU-T J.201 Recommendation [ITU J.201]. Common features of ECMAScript native objects and DOM APIs are optionals.

Although an XHTML-based browser shall be supported, the use of XHTML elements to define relationships (including XHTML links and stream events) should be dissuaded when authoring NCL documents.

HTML-based objects embedded in NCL applications shall follow the guidelines established in “Nested Context Language 3.0: Part 11 – Declarative Hypermedia Objects in NCL: Nesting Objects with NCL code in NCL Documents” [NCL Part 11].

5.5.2.2. Lua coded objects embedded in NCL applications

NCLua objects embedded in NCL applications shall follow the guidelines established in “Nested Context Language 3.0: Part 10 – Imperative Objects in NCL: The NCLua Scripting Language [NCL Part 10].

5.5.2.3. Java coded objects embedded in NCL applications

NCLet objects embedded in NCL applications shall follow the guidelines established in “Nested Context Language 3.0: Part 10 – Imperative Objects in NCL: The NCLua Scripting Language [NCL Part 10].

5.5.2.4. NCL *settings*

The application/x-ginga-settings type (or application/x-ncl-settings type) shall be applied to a special <media> element whose properties are global variables defined by the document author or reserved environment variables that may be manipulated by the NCL document processing.

The *user.location* property deserves a special attention depending on which country adopts Ginga. It shall be the country code concatenated with the country post code number (only numbers, without hyphen, underscore or slash characters), in the case of an implementation for Brazil.

5.5.3. Default values

1. The *type* attribute is optional (except for <media> elements with no *src* attribute defined) and should be used to guide the player (presentation tool) choice by the formatter. When the *type* attribute is not specified, the formatter should use the content extension specification in the *src* attribute to choose the player. Which players are associated with each *type* value is defined in Chapter 7 of this Operational Guidelines.

2. For media objects with the *src* attribute whose value identifies the “sbtvd-ts” scheme, the specific part of the scheme, more precisely, the program_number.component_tag, can be substituted by the following reserved words:

video	Corresponding to the primary video ES being presented on the video plan, as defined by [ABNT 15604].
audio	Corresponding to the primary audio ES, as defined by [ABNT 15604].
text	Corresponding to the primary text ES, as defined by [ABNT 15604].
video(i)	Corresponding to the ith smaller video ES *pid* listed in the PMT of the tuned services.
audio(i)	Corresponding to the ith smaller audio ES *pid* listed in the PMT of the tuned services.
text(i)	Corresponding to the ith smaller text ES *pid* listed in the PMT of the tuned services.

5.5.4. Exception handling

1. References to streaming video or audio resources shall not cause tuning. References that imply tuning to access a resource shall behave as if the resource were unavailable.

2. Except for <media> elements of application/x-ginga-settings (or application/x-ncl-settings) and application/x-ginga-time (or application/x-ncl-time) types, the attribute *src* should be explicitly declared. If this attribute is not specified the resource represented by the <media> element shall be assumed as unavailable.

3. Any action on a <media> element representing an unavailable resource shall be ignored by the NCL formatter. Any condition or assessment based on a <media> element representing an unavailable resource shall be considered as false.

4. If the number of media objects of a certain type surplus the maximum allowed number for that type in a particular exhibition device, the start of exceeding media objects shall be ignored.

5.6. Context module

No comments.

5.7. MediaContentAnchor module

The MediaContentAnchor module defines the <area> element.

For media objects of application/x-ginga-NCL (or application/x-ncl-NCL) type, the *clip* and *label* attribute values shall follow the guidelines established in "Nested Context Language 3.0: Part 11 – Declarative Hypermedia Objects in NCL: Nesting Objects with NCL code in NCL Documents" [NCL Part 11].

NOTE In later version of ABNR NBR 15606-2, the *label* attribute could assume the "(chainId, beginOffset, endOffset)" triple value, and other string values. In order to be complaint with ITU

Recommendation, a new attribute is added, the *clip* attribute, to specify the special value: "(chainId, beginOffset, endOffset)".

For media objects of application/x-ginga-NCLua (or application/x-ncl-NCLua) type, the *label* attribute values shall follow the guidelines established in "Nested Context Language 3.0: Part 10 – Imperative Objects in NCL: The NCLua Scripting Language [NCL Part 10].

5.7.1. Default values

1. In NCL, every node (<media> or <context> elements) must have an anchor with a region representing the whole content of the node. This anchor is called the *whole content anchor* and is declared by default in NCL. Every time an NCL component is referenced without specifying one of its anchors, the *whole content anchor* must be assumed.

2. In temporal content anchors, if the *begin* attribute of an <area> element is defined, but the *end* attribute is not specified, the end of the whole media content presentation shall be assumed as the anchor ending. On the other hand, if the *end* attribute is defined, but without an explicit *begin* definition, the start of the whole media content presentation shall be considered as the anchor beginning. Analogous behavior is expected from the *first* and *last* attributes. In the case of a <media> element of the application/x-ginga-time type, the *begin* and *end* attributes shall be always defined and shall assume an absolute value of the Universal Time Coordinated (UTC).

5.7.2. Exception handling

No comments.

5.8. PropertyAnchor module

The <property> element defines the *name* attribute, which indicates the name of the property or property group. A parent element cannot have <property> elements with the same *name* attribute values.

The <body>, <context>, and <media> elements may have several embedded properties which are not explicitly declared. However, when an embedded property is used in a relationship, it must be explicitly declared as a <property> (interface) element.

The value of the *contentId* property (associated to a continuous-media object whose content is defined referring to an elementary stream) is defined by the middleware system. It shall have the "null" value initially and be set to the identifier value transported in the NPT reference descriptor (in a field identified by the same name: contentId), as soon as the associated continuous-media object is started.

The value of the *standby* property is also defined by the middleware system. It shall be set to "true" while an already started continuous-media object content referring to an elementary stream is temporarily interrupted by another interleaved content, in the same elementary stream.

NOTE The *standby* property may be set to “true” automatically by the middleware when the identifier value transported in the NPT reference descriptor (in a field identified by the same name: contentId) signalled as non-paused is different from the *contentId* property value.

USE CASE The *standby* property can be used to pause an application when the continuous-media object content referring to an elementary stream transporting the main video of a TV program is temporarily interrupted by another interleaved content, for example an advertisement (TV commercial). The same property can then be used to resume the application.

When the *visible* property associated with a <context> or <body> element is equal to “true”, the *visible* property of each child element of the composition shall be taken into account to define how each of these child element will be exhibited.

When the *visible* property associated with a <context> or <body> element is equal to “false”, all child elements of the composition shall be exhibited as hidden. In particular, when a document has its <body> element with its *visible* property set to “false” and its presentation event is in the *paused* state, the document is said to be in stand-by. When there is only one application in execution and this application is in stand-by, the service’s main video shall be dimensioned to 100% of the screen, and the main audio shall be set to 100% of its volume.

It should be remarked that an object with a *visible* property equal to “false”, that is, an object exhibited as hidden, may not transit selection event machines defined by its content anchors to the “occurring” state while the *visible* property value persists as “false”.

5.8.1. Default values

1. The *value* attribute of a <property> element is optional and defines an initial value for the property declared as *name*. When the value is not specified, the property assumes as its initial value the one defined in homonym attributes of its node’s associated descriptor or region. When the *value* is specified, it has precedence over the value defined in homonym attributes of its node’s associated descriptor or region.

2. If the *left, right, top, bottom, width* or *height* properties are not defined and cannot be inferred from property values defined on <property>, <descriptor> and its child elements, or <region> elements, they shall assume the “0” value.

5.8.2. Exception handling

1. When a formatter treats a change in a property group it shall only test the process consistency at its end.

2. When the user specifies *top*, *bottom* and *height* information for the same <region>, spatial inconsistencies can occur. In this case, the *top* and *height* values shall have precedence over the *bottom* value. Analogously, when the user specifies inconsistent values for the *left*, *right* and *width* <region> attributes, the *left* and *width* values shall be used to compute a new *right* value.

3. Child regions must be completely contained in the area established by their parent regions. When some portion of the child region lies outside its parent region, the child region shall be ignored.

4. If two or more <property> elements with the same *name* attribute are defined as child elements of the same <media> element, only the last *value* defined shall be taken into account. The others shall be ignored.

5.9. CompositeNodeInterface module

The CompositeNodeInterface module defines the <port> element, which specifies a composite node port with its respective mapping to an interface (*interface* attribute) of one and only one of its components (specified by the *component* attribute).

5.10. SwitchInterface module

The SwitchInterface module allows the creation of <switch> element interfaces (see [NCL Part 8]), which may be mapped to a set of alternative interfaces of the switch's internal objects, allowing a link to anchor on the component chosen when the <switch> is processed. This module introduces the <switchPort> element, which contains a set of *mapping* elements. A *mapping* element defines a path from the <switchPort> to an interface (*interface* attribute) of one of the switch's components (specified by its *component* attribute).

5.10.1. Default values

1. A reference to an internal switch component shall be made through a <switchPort> element or, by default, to the <switch> element without specifying any <switchPort>. In this case, it is considered as if the reference is made to a default <switchPort> that contains a set o mapping elements to all child objects of the switch and referring to their *whole content anchors*.

5.11. Descriptor module

A <descriptor> element may have <descriptorParam> child elements, which are used to parameterize the presentation control of the object associated with the descriptor element.

A *descriptor* attribute is associated with any media object through <media> elements and through link endpoints (<bind> elements).

The *plan* parameter of a <descriptorParam> element defines in which plan of a structured screen an object will be placed. The value of this attribute can be: “background”, “video” and “graphic”, following the plan definition of ABNT NBR 15606-1.

The *reusePlayer* and *playerLife* attributes offer additional support for media-objects's player management, and can be used or be ignored by a particular middleware implementor. The *playerLife* attribute specifies what will happen to a player instance at the end of a media object presentation. Maintaining a player instance demands memory space however decreases the player loading time and the probability for synchronization mismatches, as a consequence. The *reusePlayer* attribute allows using the same player instance to more than one media object presentation, including using a player left in the memory space by the *playerLife* attribute.

5.11.1. Default values

1. Table 5.38 summarizes the default values for reserved parameter/attribute names.

Table 5.38 – Reserved parameter/attribute default values

Parameter/attribute name	**Value**
background	"transparent"
visible	"true"
transparency	"0"
plan	"video",for media with *src* attribute referring to a TS's PES, "graphics", for all other cases.
fit	"fill"
scroll	"none"
soundLevel, balanceLevel, trebleLevel, bassLevel	"1"
zIndex	"0"
fontColor	"white"
reusePlayer	"false"
playerLife	"close"

5.11.2. Exception handling

1. If several values are specified for the same parameter/attribute, the value defined in a <property> element has precedence over the one defined in a <descriptorParam> element, which has precedence over the value defined in an attribute of the corresponding <descriptor> element (including the *region* attribute).

5.12. Linking module

A <link> element must have an *xconnector* attribute, which refers to a hypermedia connector URI. The reference must have the format: *alias#connector_id*, or *documentURI_value#connector_id*, for connectors defined in an external document; or simply *connector_id*, for connectors defined in the document itself.

5.12.1. Default values

1. The *interface* attribute of a <bind> child element of a <link> element may refer to any node interface, that is, an anchor, a property or a port, if it is a composite node. The

interface attribute is optional. When it is not specified, the association will be done with the whole node content.

5.12.2. Exception handling

1. A <link> element must be ignored if the *xconnector* attribute refers to an inexistent hypermedia connector.

5.13. Connectors functionality

The Connectors Functionality defines the CausalConnectorFunctionality module that groups a set of basic Connector Functionality's modules, in order to make it easy to define a language profile. This is the case of the EDTV and BDTV profiles.

Relations in NCL are based on events. An event is an occurrence in time that can be instantaneous or have a measurable duration. NCL 3.0 defines the following types of events: presentation, selection, attribution, and composition (see [NCL Part 8]).

Each event defines a state machine that must be maintained by the NCL formatter, as shown in Figure 5.2. Moreover, every event has an associate attribute, named *occurrences*, which counts how many times the event transits from the occurring to the sleeping state during a document presentation. Events like presentation and attribution have also an attribute named *repetitions*, which counts how many times the event must be automatically restarted (transited from the sleeping to the occurring state) by the formatter. This attribute can contain the "indefinite" value, leading to an endless loop of the event occurrences until some external interruption.

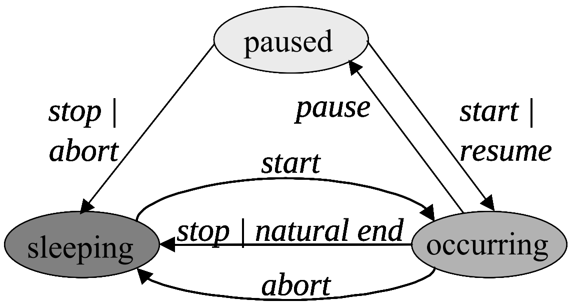

Figure 5.2 - Event state machine

The composition event is defined by the presentation of the structure of a composite node. The feature to present the composite node structure is optional. However, it should be emphasized that it is very useful in presenting the nested organization of a composition, since this is the organization of an application (a DTV program).

Relations are defined based on event states, changes on the event state machines, on event attribute values, and on node (<media>, <body>, <context> or <switch> element) property values. The CausalConnectorFunctionality module allows only the definition of causal relations, defined by the <causalConnector> element of the CausalConnector module. A <causalConnector> element represents a causal relation. In a causal relation, a condition shall be satisfied in order to trigger an action.

The <compoundCondition> element has a Boolean *operator* attribute ("and" or "or") relating its child elements. When an "and" compound condition relates more than one trigger condition (i.e. a condition that is satisfied only in an infinitesimal time instant – as for example, the end of an object presentation), the compound condition must be considered true in the instant immediately after all the trigger conditions are satisfied.

As with <simpleCondition> elements the role cardinality specifies the minimal (*min* attribute) and maximal (*max* attribute) number of participants that may play the role (number of binds) when the <causalConnector> is used for creating a link. If minimal and maximal cardinalities are not informed, "1" must be assumed as the default value for both parameters. When the maximal cardinality value is greater than one, several participants may play the same role. When it has the "unbounded" value, the number of binds is unlimited.

Table 5.39 – Reserved action *role* values associated to event state machines

Role Value	**Action Type**	**Event Type**
start	*start*	*presentation*
stop	*stop*	*presentation*
abort	*abort*	*presentation*
pause	*pause*	*presentation*
resume	*resume*	*presentation*
set	*start*	*attribution*

The <compoundAction> element has an *operator* attribute ("par" or "seq") relating its child elements. It is important to mention that when the sequential operator is used, actions must be fired in the specified order. However, an action does not need to wait the previous one to be finished in order to be fired.

5.13.1. Default values

1. If an *eventType* value of a <simpleCondition> element is "selection", the role can also define to which selection apparatus (for example, keyboard or remote control keys) it refers, through its *key* attribute. If this attribute is not specified, the selection via a pointer device (mouse, touch screen, etc.) shall be assumed.

   NOTE When a same selection apparatus is pressed, more than one <simple condition> may be considered satisfied, if this selection apparatus is defined in the *key* attribute of the <simpleCondition> and the interfaces bounded by <link> elements referring to the <simpleCondition> (through the *role* attributes of their <bind> elements) are being presented.

2. In a <simpleCondition> element and in a <simpleAction>, the role cardinality specifies the minimal (*min* attribute) and maximal (*max* attribute) number of participants that may play the role (number of binds) when the <causalConnector> is used for creating a <link>. If minimal and maximal cardinalities are not informed, "1" shall be assumed as the default value for both parameters.

3. A *qualifier* attribute should inform the logical relationship among binds of the same simple condition. If it is not specified, the default value "or" shall be assumed.

4. A *qualifier* attribute should inform the logical relationship among binds of the same simple action. If it is not specified, the default value “par” shall be assumed.

5. If an *eventType* value of a <attributeAssessment> element is “selection”, the role can also define to which selection apparatus (for example, keyboard or remote control keys) it refers, through its *key* attribute. If this attribute is not specified, the selection via a pointer device (mouse, touch screen, etc.) shall be assumed.

6. if the *eventType* value of a <attributeAssessment> element is “attribution” the *attributeType* is optional and has the value “nodeProperty” as default.

5.13.2. Exception handling

1. The minimal role’s cardinality value shall always be a positive finite value, greater than zero and lesser than or equal to the maximal cardinality value, otherwise the link shall be ignored.

2. If an *eventType* value is “attribution”, the <simpleAction> shall also define the value that shall be assigned, through its *value* attribute. If the *value* is specified as “$anyName” (where $ is a reserved symbol and anyName is any string, except reserved role names), the assigned value shall be retrieved from the property associated with the *role*=“anyName” and defined by a <bind> child element of the <link> element that refers the connector. If this value cannot be retrieved, no attribution shall be made.

3. If an *eventType* value is “attribution”, and the <simpleAction> defines the value that shall be assigned as “$anyName”, the value to be attributed shall be the value of a property (<property> element) of a component of the same composition where the link (<link> element) that refers to the event is defined, or a property of the composition where the link is defined, or a property of an element that can be reached through a <port> element of the composition where the link is defined, or even a property of an element that can be reached through a port (elements <port> or <switchPort>) of a composition nested in the same composition where the link is defined. Otherwise, no attribution shall be made.

4. An <attributeAssessment> element defines an *offset* attribute whose value may be added to the value of the variable referred by the corresponding *attributeType* attribute. The *offset* value shall have the same type and shall be specified with the same unit of the value to which it will be added, otherwise the *offset* value shall be ignored.

5. Selection events may only be defined on information units of a media object being presented, that is on information units whose associated presentation event is in the “occurring” state.

5.14. TestRule module

The TestRule module allows the definition of rules that, when satisfied, select alternatives for document presentation. These rules may be simple, defined by the <rule> element, or composite, defined by the <compositeRule> element.

Comparisons in simple rules are done based on the binary representation of the variable's value to be compared and the binary representation of the other value used in the comparison.

A <rule> element defined as child of a <compositeRule> element may have its *id* attribute omitted.

5.14.1. Exception handling

1. In simple rules, the *comparator* attribute relates the variable to a value. The variable type and the value type shall be equal; otherwise the rule definition shall be ignored.

5.15. TestRuleUse module

No comments.

5.16. ContentControl module

The ContentControl module specifies the <switch> element. NCL formatters shall delay the switch evaluation to the moment when a link anchoring in the switch needs to be evaluated. Test rules used to choose the switch component to be presented shall be defined by the TestRule module.

The ContentControl module also defines the <defaultComponent> element, whose *component* attribute identifies the default element that should be selected if none of the bindRule rules is evaluated as true.

During a document presentation, from the moment a <switch> is evaluated on, it is considered resolved until the end of the current switch presentation, that is, while its corresponding presentation event is in the "occurring" or "paused" state. The chosen alternative can be referred through any <switchPort> element mapped to one of its interfaces.

When a <context> is defined as a child of a <switch> element, the <link> elements recursively contained in the <context> must be considered by an NCL player only if the <context> is selected after the switch evaluation. Otherwise, the <link> elements should be considered disabled and must not interfere with the document presentation.

5.16.1. Exception handling

1. If the <defaultComponent> element is not defined in a <switch> element and if none of the bindRule rules is evaluated as true to a component bound by a <mapping> element child of the <switchPort> from which the <switch> element is referred, no component is selected for presentation and the NCL formatter shall behave as if the component does not exist.

5.17. DescriptorControl module

The DescriptorControl module specifies the <descriptorSwitch> element. Analogous to the <switch> element, the <descriptorSwitch> choice shall be done using test rules defined by the TestRule module. The descriptorSwitch evaluation must be delayed until the object referring the descriptorSwitch needs to be prepared to be presented.

The DescriptorControl module also defines the <defaultDescriptor> element, whose *descriptor* attribute identifies the default element that should be selected if none of the bindRule rules is evaluated as true.

During a document presentation, from the moment on a <descriptorSwitch> is evaluated, it is considered resolved until the end of the presentation of the <media> element that was associated to it, that is, while any presentation event associated with this <media> element is in the “occurring” or “paused” state.

5.17.1. Exception handling

1. If the <defaultDescriptor> element is not defined in a <descriptorSwitch> element and if none of the bindRule rules is evaluated as true, no descriptor is selected for presentation and the NCL formatter shall behave as if the descriptor does not exist.

5.18. Timing module

This module defines the *freeze* attribute for specifying what happens with a <media> element at the end of its presentation. When *freeze* is specified with a value equal to “true” the last image map of the object must be frozen indefinitely, that is, until its end is determined by an external event (for example, coming from a <link> evaluation), or by the *explicitDur* value for that object.

The Timing module also defines the *explicitDur* attribute specifying the presentation duration of an object represented by a <media> element. Note that the *explicitDur* attribute gives the presentation duration of an object and not the presentation duration of the object’s content. If the *explicitDur* value is greater than the content presentation duration what must happen on the end of the content presentation depends on the *freeze* attribute previously mentioned. If the *explicitDur* value is smaller than the content presentation duration, the content presentation is cut. Note that a player may, optionally, make elastic time adjustments on the media content in order to make the content presentation duration as close as possible to the *explicitDur* value.

5.18.1. Default values

1. When not specified, the value of the *freeze* attribute value shall be considered as “false”.

2. When not specified, the value of the *explicitDur* attribute value shall be considered as equal to the natural content presentation duration.

5.19. Import module

The <importBase> element has two attributes: *documentURI* and *alias*. The *documentURI* refers to a URI corresponding to the NCL document containing the base to be imported. The *alias* attribute specifies a name to be used as prefix when referring to elements of the imported base. The alias name shall be unique in a document and its scope is constrained to the document that has defined the *alias* attribute.

The <importNCL> element does not "include" the referred NCL document but only makes the referred document visible to have its components reused by the document that has defined the <importNCL> element. New relationships can be defined among new nodes created by reuse, but no new relationships can be defined inside imported context nodes since, when reused, a node cannot have its content modified, but can only be related to other nodes.

The <importNCL> element has two attributes: *documentURI*, and *alias*. The *documentURI* refers to a URI corresponding to the document to be imported. The *alias* attribute specifies a name to be used when referring to an element of this imported document. As in the <importBase> element, the name shall be unique (type=ID) and its scope is constrained to the document that has defined the *alias* attribute. The reference would have the format: *alias#element_id*. It is important to note that the same alias should be used when referring to elements defined in the imported document bases (<regionBase>, <connectorBase>, <descriptorBase>, etc.).

5.19.1. Exception handling

1. The <importBase> operation is transitive, that is, if *baseA* imports *baseB* that imports *baseC*, then *baseA* imports *baseC*. However, the *alias* defined for *baseC* inside *baseB* shall not be considered by *baseA*.

2. The <importNCL> element operation has also the transitive property, that is, if *documentA* imports *documentB* that imports *documentC*, then *documentA* imports *documentC*. However, the *alias* defined for *documentC* inside *documentB* shall not be considered by *documentA*.

5.20. EntityReuse module

The EntityReuse module allows an NCL element to be reused. Only <media>, <context>, <body> and <switch> may be reused.

When an element declares a *refer* attribute, all attributes and child elements defined by the referred element are inherited. For <media> elements, new child <area> and <property> elements may be added, and a new attribute, *instance*, may also be defined.

The referred element and the element that refers to it shall be considered the same, regarding their data specification.

5.20.1. Exception handling

1. An element that refers to another element cannot be reused; that is, its *id* cannot be the value of any *refer* attribute. When an element has the *refer* attribute with a value corresponding to an *id* of an element that refers to another one, the element shall be considered as nonexistent.

2. If the referred node is defined within an imported document *D*, the *refer* attribute value shall have the format "alias#id", where "alias" is the value of the *alias* attribute associated with the *D* import. Otherwise, the element that contains the *refer* attribute shall be considered as nonexistent.

3. A <media> element may only refer to another <media> element; a <switch> element may only refer to another <switch> element; a <context> or <body> element may only refer to another <context> or <body> element. In all other cases, the element that contains the *refer* attribute shall be considered as nonexistent.

4. All attributes and child elements defined by a referring element shall be ignored by the NCL formatter, except the *id* attribute that shall be defined. The only other exception is for <media> elements, in which new child <area> and <property> elements may be added, and a new attribute, *instance*, may be defined.

5. If the new added <property> element has the same *name* attribute of an already existing <property> element (defined in the reused <media> element), the new added <property> shall be ignored. Similarly, if the new added <area> element has the same *id* attribute of an already existent <area> element (defined in the reused <media> element), the new added <area> shall be ignored.

6. A <body>, <context> or <switch> elements may not include more than one element from the set composed of the referring object and corresponding referred objects. If this is the case, all reffered objects shall be considered as nonexistent.

5.21. ExtendedEntityReuse module

The ExtendedEntityReuse module defines the *instance* attribute.

The referred element and the element that refers to it shall be considered independent objects regarding their presentation, if the *instance* attribute receives a "new" value.

The referred element and the element that refers to it shall be considered the same regarding their presentation, if the *instance* attribute receives a "instSame" or "gradSame" value. The following semantics shall be respected:

- Assume the set of <media> elements composed of the referred <media> element and all the referring <media> elements. If any element of the subset formed by the referred <media> element and all other <media> elements having the *instance* attribute equal to "instSame" or "gradSame" is scheduled to be presented, all other elements in this subset, which are not child descendents of a <switch> element, are also assumed as scheduled for presenting, and additionally, when they are being presented, they shall be represented by the same presentation instance. Descendent elements of a <switch> element shall also have the same behavior, if all rules needed to present these elements are satisfied; otherwise they shall not be scheduled for presenting.

- If the *instance* attribute is equal to “instSame”, all scheduled nodes of the subset shall be immediately presented through a unique instance (start instruction applied on all subset elements).

- If the *instance* attribute is equal to “gradSame”, all scheduled nodes of the subset shall be presented through a unique instance, but now gradually, as start instructions are applied, coming from a link, etc.

- The common instance being presented shall notify all events associated with the <area> and <property> elements defined in all <media> elements of this subset that were scheduled for presenting.

- On the other hand, <media> elements in the set that have *instance* attribute values equal to “new” shall not be scheduled for presenting. When they are individually scheduled for presenting, no other element in the set is affected. Moreover, new independent presentation instances shall be created at the start of each individual presentation.

5.21.1. Default values

1. The *instance* attribute has “new” as its default value.

5.22. KeyNavigation Module

The KeyNavigation module provides the extensions necessary to describe focus movement operations using a control device like a remote control.

The *focusIndex* attribute specifies an index for the <media> element to which the focus may be applied.

When an element on focus is selected by pressing the “activate (select or enter) key”, the focus control shall be passed to the <media> element renderer (player). The player can then follow its own rules for navigation. The focus control shall be passed back to the NCL formatter when the “back key” is pressed. In this case, the focus goes to the element identified by the *service.currentFocus* property of the *settings* node (<media> element of application/x-ginga-settings type). In a multiple device environment, the hierarchical rules for input key control and exhibition device control shall follow the guidelines established in “Nested Context Language 3.0: Part 12 – Support to Multiple Exhibition Devices” [NCL Part 12].

Note The focus control may also be passed by setting the service.currentKeyMaster property of the *settings* node (<media> element of application/x-ginga-settings type). This may be done through a link action, through an imperative code command executed by an imperative-code node (NCLua object), or by the player of a node that has the current control.

5.22.1. Default values

1. When a <descriptor> element does not define the *focusIndex* attribute, it shall be considered as if no focus could be set.

2. When the *focusBorderColor,* the *focusBorderWidth,* the *focusBorderTransparency,* or the *selBorderColor* attribute are not defined, default values shall be assumed. These values are specified in properties of the <media> element of application/x-ginga-settings (or application/x-ncl-settings) type: *defaultFocusBorderColor, defaultFocusBorderWidth, defaultFocusTransparency*, *defaultSelBorderColor*.

5.22.2. Exception handling

1. In a certain presentation moment, if the focus has not been already defined, or is lost, a focus will be initially applied to the element that is being presented whose descriptor has the smallest index value.

2. Values of *focusIndex* attribute shall be unique in an NCL document. Otherwise, the repeated attributes shall be ignored if in a certain moment there are more than one <media> element to gain the focus.

3. When a <media> element refers to another <media> element (using the *refer* attribute), the *focusIndex* specified by the <descriptor> element associated with the referred <media> element shall be ignored.

4. When the focus is applied to an element with the visible property set to false, or to an element that it is not being presented, the current focus does not move.

5. When the focus is applied to an element with the visible property set to false, every selection on the element shall be ignored.

5.23. Animation module

Since NCL animation can be computationally intensive, it is only supported by the EDTV profile and only the properties that define numerical values and colors can be animated. An NCL formatter following the BDTV profile may ignore the animation attributes.

5.23.1. Default values

1. When setting a new value to a property the change is instantaneous by default (*duration*=″0″).

2. When setting a new value to a property the change from the old value to the new one can be linear by default (*by*=″indefinite″).

5.24. Transition module

The Transition module has just one element called <transition>. Each <transition> element defines a single transition template and shall have an *id* attribute so that it may be referred inside a <descriptor> element.

Transitions are classified according to a two-level taxonomy of types and subtypes. The *type* attribute is required and is used to specify the general transition. The *subtype* attribute provides transition-specific control.

Only the default subtype for the five required transition types listed in Table 5.40 shall be supported, the others, defined in [SMIL 2.1], are optional.

Table 5.40 – Required transition types and subtypes

Transition type	Default transition subtype
barWipe	leftToRight
irisWipe	rectangle
clockWipe	clockwiseTwelve
snakeWipe	topLeftHorizontal
fade	crossfade

The Transition module also defines attributes to be used in <descriptor> elements to use the transition templates defined by <transition> elements: *transIn* and *transOut* attributes.

All transitions defined in the Transition module accept four additional attributes coming from [SMIL 2.1] that may be used to control the visual appearance of the transitions: *horRepeat; vertRepeat; borderWidth;* and *borderColor*.

5.24.1. Default values

1. Te *subtype* attribute is optional. If this attribute is not specified then the transition reverts to the default subtype for the specified transition type, as shown in Table 40.

2. The default value for the *dur* attribute is 1 second.

3. The *startProgress* attribute specifies the amount of progress through the transition at which to begin execution. The default value is 0.0.

4. The *endProgress* attribute specifies the amount of progress through the transition at which to end execution. The default value is 1.0.

5. The *direction* attribute specifies the direction the transition will run. The default value is “forward”.

6. The default value for the *fadeColor* attribute is “black”.

7. The default value for both the *transIn* and the *transOut* attributes is an empty string, which indicates that no transition shall be performed.

8. The *horRepeat* attribute specifies how many times to perform the transition pattern along the horizontal axis. The default value is 1.

9. The *vertRepeat* attribute specifies how many times to perform the transition pattern along the vertical axis. The default value is 1.

10. The *borderWidth* attribute specifies the width of a generated border along a wipe edge. The default value is 0.

11. The *borderColor* attribute specifies the content of the generated border along a wipe edge. The default value for this attribute is the color "black".

5.24.2. Exception handling

1. If the named transition's type is not supported by the NCL formatter, the transition shall be ignored.

2. The *subtype* attribute, if specified, shall be one of the transition supported subtypes that is appropriate for the specified type, otherwise the transition shall be ignored.

3. The *endProgress* attribute values are real numbers in the range [0.0,1.0], and the value of this attribute shall be greater than or equal to the value of the *startProgress* attribute. If *endProgress* value is specified as less than the *startProgress* value, the transition effect shall be ignored. If *endProgress* is equal to *startProgress*, then the transition remains at a fixed progress for the duration of the transition.

4. Note that not all transitions will have meaningful reverse interpretations. For instance, a crossfade is not a geometric transition, and therefore has no interpretation of reverse direction. Transitions that do not have a reverse interpretation should have the *direction* attribute ignored and the default value of "forward" assumed.

5. If the value of the *type* attribute is not "fade", or the value of the *subtype* attribute is not "fadeToColor" or "fadeFromColor", then the *fadeColor* attribute shall be ignored.

6. If the value of the *transIn* attribute or the *transOut* attribute does not correspond to the value of the XML identifier of any one of the transition elements defined, then the value of the attribute shall be considered to be the empty string and therefore no transition should be performed.

5.25. Metainformation module

No comments.

6. Media objects in NCL presentations

As the Ginga-NCL architecture and implementation is a choice of each receiver developer, the goal of this section is just to define the expected behavior of a media player when controlling objects that take part in an NCL document.

A media player (or its adapter) must be able to receive presentation commands, to control the events' state machines of the media object, and answer queries from the formatter. This section describes how a media player must behave for each expected instruction issued by the formatter.

6.1. Expected behavior of basic media players

This section deals with media players for <media> elements whose types are different from "application/x-ginga-NCL" (or "application/x-ncl-NCL"), "application/x-ginga-NCLua" (or "application/x-ncl-NCLua") and text/html.

6.1.1. start instruction for presentation events

Before sending a *start* instruction, the formatter should find the more appropriate media player to be called based on the content type to be exhibited. For this sake, the formatter takes into consideration the *player* attribute of the <descriptor> element associated with the media object to be exhibited. If this attribute is not specified, the formatter shall take into account the *type* attribute of the <media> element. If this attribute is not specified either, the formatter shall consider the file extension specified in the *src* attribute of the <media> element.

The *start* instruction issued by a formatter shall inform the following parameters to the media player: the media object to be controlled, its associated descriptor, a list of events (presentation, selection or attribution) that need to be monitored by the media player, the presentation event that needs to be started (called here main-event), an optional offset-time and an optional delay-time.

The media object shall be derived from a <media> element, whose *src* attribute shall be used, by the media player, to locate the content and start its presentation. If the content cannot be located, or if the media player does not know how to handle the content type, the media player shall finish the starting operation without performing any action.

The descriptor shall be chosen by the formatter following the directives specified in the NCL document. If the *start* instruction results from a link action that has a descriptor explicitly declared in its <bind> element (*descriptor* attribute of the <link> element's children <bind> element), the resulting descriptor informed by the formatter shall merge the attributes of the bind descriptor with the attributes of the descriptor specified in the corresponding <media> element, if this attribute was specified. For the common attributes, the <bind> descriptor information shall superpose the <media> descriptor data. If the <bind> element does not contain an explicit descriptor, the descriptor informed by the formatter shall be the <media> descriptor, if this attribute was specified. Otherwise, a default descriptor for that *type* of <media> shall be chosen by the formatter.

The list of events to be monitored by a media player should also be computed by the formatter, taking into account the NCL document specification. It shall check all links in which the media object and the resulting descriptor participate. When computing the events to be monitored, the formatter shall take into account the media-object perspective, i.e., the path of <body> and <context> elements to reach the <media> element. Only links contained in these <body> and <context> elements should be considered to compute the monitored events.

The offset-time parameter is optional, it has “zero” as its default value, and it is meaningful only for continuous media or static media with explicit duration. In this case, this parameter defines a time offset from the beginning (beginning-time) of the main-event, from which the presentation of the main-event shall be immediately started (i.e., it commands the player to jump to the beginning-time + offset-time). Obviously, the offset-time value shall be lower than the main-event duration. If the offset-time is greater than zero, the media player shall put the main-event in the *occurring* state, but the event *starts* transition shall not be notified. If the offset-time is zero, the media player shall put the main-event in the *occurring* state and notify the occurrence of the *starts* transition. Events that would have their end-times before the beginning-time of the main-event and events that would have their beginning times after the end-time of the main-event do not need to be monitored by the media player (the formatter should do this verification when building the monitored event list). Monitored events that would have beginning-times before the beginning-time of the main-event and end-times after the beginning-time of the main-event shall be put in the *occurring* state, but their *starts* transitions shall not be notified (links that depend on this transition shall not be fired). Monitored events that would have their end times after the main-event beginning-time, but before the start time (beginning-time + offset-time) shall have their *occurrences* attribute incremented but the *starts* and *stops* transitions shall not be notified. Monitored events that have beginning-times before the start time (beginning-time + offset-time) and end time after the start time shall be put in the *occurring* state, but the corresponding *starts* transition shall not be notified.

The delay-time is also an optional parameter and its default value is “zero” too. If greater than zero, this parameter contains a time to be waited by the media player before starting the presentation. This parameter shall only be considered if the offset-time parameter is equal to “zero”.

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. In this case, the <simpleAction> element that has caused the *start* instruction shall not cause any transition on the corresponding event state machine.

6.1.2. stop instruction

The *stop* instruction only needs to identify a media object already being controlled. To identify the media object means to identify the <media> element, the corresponding descriptor and the media-object perspective. Therefore, if a <simpleAction> element with an *actionType* attribute equal to “stop” is bound through a link to a node interface, the interface shall be ignored when the action is performed.

If the object is not being presented (none of the events in the object list of events is in the *occurring* or *paused* state) and the media player is not waiting due to a delayed *start* instruction, the *stop* instruction shall be ignored. If the object is being presented, the main-event (the event passed as a parameter when the media object was started) and all monitored events in the *occurring* or in the *paused* state with end time equal or prior to the main-event end time shall transit to the *sleeping* state, and their *stops* transitions shall be notified. Monitored events in the *occurring* or in the *paused* state with end time posterior to the main-event end time shall be put in the *sleeping* state, but their *stops* transitions shall not be notified and their *occurrences* attribute shall not be incremented. The object content presentation shall be stopped. If the *repetitions* event attribute is greater than zero, it shall be decremented by one and the main-event presentation shall restart after the repeat delay time (the repeat delay shall have been passed to the media player as the start delay parameter). If the media object is waiting to be presented after a delayed *start* instruction and a *stop* instruction is issued, the previous *start* instruction shall be removed.

NOTE When all media objects referring to video elementary streams target to the video plan are in the sleeping state a video ES that is not referred by any media object shall be presented, and in full screen. An elementary video stream can be redimensioned only using a media object in presentation. The same happens with the audio. When all media objects referring to audio elementary streams are in the sleeping state, an audio ES shall be presented with 100% of its volume.

6.1.3. abort instruction

The *abort* instruction only needs to identify a media object already being controlled. If a <simpleAction> element with an *actionType* attribute equal to "abort" is bound through a link to a node interface, the interface shall be ignored when the action is applied.

If the object is not being presented and is not waiting to be presented after a delayed *start* instruction, the *abort* instruction shall be ignored. If the object is being presented, the main-event and all monitored events in the *occurring* or in the *paused* state shall transit to the *sleeping* state, and their *aborts* transitions shall be notified. Any content presentation shall stop. If the *repetitions* event attribute is greater than zero, it shall be set to zero and the media object presentation shall not restart. If the media object is waiting to be presented after a delayed *start* instruction and an *abort* instruction is issued, the previous *start* instruction shall be removed.

6.1.4. pause instruction

The *pause* instruction only needs to identify a media object already being controlled. If a <simpleAction> element with an *actionType* attribute equal to "pause" is bound through a link to a node interface, the interface shall be ignored when the action is applied.

If the object is not being presented (the main-event, passed as a parameter when the media object was started, is not in the *occurring* state) and the media player is not waiting for the start delay, the instruction shall be ignored. If the object is being presented, the main-event and all monitored events in the *occurring* state shall transit to the *paused* state and their *pauses* transitions shall be notified. The object presentation shall be paused and the pause elapsed time shall not be considered as part of the object duration. As an example, if an object has an explicit duration of 30 s and, after 25 s it is paused, even if the object stays paused for 5 min, after resuming the object main-event shall stay occurring for 5 s. If the main-event is still not occurring because the media player is waiting for the start delay, the media object shall wait for a resume instruction to continue waiting for the remaining start delay.

6.1.5. resume instruction

The *resume* instruction only needs to identify a media object already being controlled. If a <simpleAction> element with an *actionType* attribute equal to “resume” is bound through a link to a node interface, the interface shall be ignored when the action is applied.

If the object is not paused (the main-event, passed as a parameter when the media object was started, is not in the *paused* state) or the media player is not paused (waiting for the start delay), the instruction shall be ignored. If the media player is paused waiting for the start delay, it shall resume the wait from the instant it was paused. If the main-event is in the *paused* state, the main-event and all monitored events in the *paused* state shall be put in the *occurring* state and their *resumes* transitions shall be notified.

6.1.6. start instruction for attribution events

The *start* instruction may be applied to an object property independently from the fact whether the object is being presented or not (in this last case, although the object is not being presented, its media player shall be already instantiated). In the first case, the *start* instruction needs to identify the media object being controlled, a monitored attribution event and a value to be assigned to the attribute wrapped by the event. In the second case, the instruction shall also identify the <descriptor> element that will be used when presenting the object (as it is done for the *start* instruction for presentation). When setting a value to the attribute, the media player shall set the event state machine to the *occurring* state, and after finishing the attribution, again to the *sleeping* state, generating the *starts* transition and afterwards the *stops* transition.

For every monitored attribution event, if the media player changes by itself the corresponding attribute value, it shall also proceed as if it had received an external *set* instruction.

6.1.7. addEvent instruction

The *addEvent* instruction is issued in the case of receiving an *addInterface* NCL editing command (see Section 9). The instruction needs to identify a media object already being controlled and a new event that shall be included to be monitored. All rules applied to the intersection of monitored events with the main-event shall be applied to the new event. If the new event start time is prior to the object current time and the new event end time is posterior to the object current time, the new event shall be put in the same state of the main-event (*occurring* or *paused*), without notifying the corresponding transition.

6.1.8. removeEvent instruction

The *removeEvent* instruction is also issued in the case of receiving an *removeInterface* NCL editing command. The instruction needs to identify a media object already being controlled and a monitored event that should be no more controlled. The event state shall be put in the *sleeping* state without generating any transition.

6.1.9. Natural end of a presentation

Events of an object that has an explicit or an intrinsic duration normally end their presentations naturally, without needing external instructions. In this case, the media player shall transit the event to the *sleeping* state and notify the *stops* transition. The same shall be done for monitored events in the *occurring* state with the same end time of the main-event or with unknown end time, when the main-event ends. Events in the *occurring* state with end time posterior to the main-event end time shall be put in the sleeping state but without generating the *stops* transition and without incrementing the *occurrences* attribute. It is important to remark that, if the main-event corresponds to an object internal temporal anchor, when this anchor presentation finishes, the whole media object presentation shall finish.

NOTE An application author should take into account that the natural end of contents received as elementary streams can be notified only some time later, due to the delay of descriptors carrying this information.

6.2. Expected behavior of media players after instructions applied to composite objects

This section applies only for objects represented by <context>, <body> and <switch> elements.

6.2.1. Binding a composite node

A <simpleCondition> or <simpleAction> with *eventType* attribute value equal to "presentation" may be bound by a link to a composite node (represented by a <context> or <body> element) as a whole (i.e. without an interface being informed). Analogously, an <attributeAssessment> with *eventType* attribute value equal to "presentation" and *attributeType* equal to "state", "occurrences" or "repetitions" may be bound by a link to a composite node (represented by a <context> or <body> element) as a whole, and the attribute value should come from the event state machine of the presentation event defined on the composite node. If a <simpleAction> with *eventType* attribute value equal to "presentation" is bound by a link to a composite node (represented by a <context> or <body> element) as a whole (i.e. without an interface being informed), the instruction shall also be reflected to the event state machines of the composite child nodes, as explained in the following subsections.

6.2.2. Starting a context presentation

If a <context> or <body> element participates on an action role whose action type is "start", when this action is fired, the *start* instruction shall also be applied to all presentation events mapped by the <context> or <body> element's ports.

If the author wants to start the presentation using a specific port, it shall in addition indicate the <port> id as the <bind> *interface* value. In this case, the *start* instruction shall also be applied to the presentation event mapped by the <context> or <body> element's port.

6.2.3. Stopping a context presentation

If a <context> or <body> element participates in an action role whose action type is "stop", when this action is fired, the *stop* instruction shall also be applied to all presentation events of the composite child nodes.

If the composite node contains links being evaluated (or with their evaluation paused), the evaluations shall be suspended and no action shall be fired.

6.2.4. Aborting a context presentation

If a <context> or <body> element participates in an action role whose action type is "abort", when this action is fired, the *abort* instruction shall also be applied to all presentation events of the composite child nodes.

If the composite node contains links being evaluated (or with their evaluation paused), the evaluations shall be suspended and no action shall be fired.

6.2.5. Pausing a context presentation

If a <context> or <body> element participates in an action role whose action type is "pause", when this action is fired, the *pause* instruction shall also be applied to all presentation events of the composite child nodes that are in the occurring state.

If the composite node contains links being evaluated, all evaluations shall be suspended until a resume, stop or abort action is issued.

If the composite node contains child nodes with presentation events already in the paused state when the pause action is issued, these nodes shall be identified because if the composite receives a resume instruction, these events shall not be resumed.

6.2.6. Resuming a context presentation

If a <context> or <body> element participates in an action role whose action type is "resume", when this action is fired, the *resume* instruction shall also be applied to all presentation events of the composite child nodes that are in the paused state, except those that were already paused before the composite has been paused.

If the composite contains links with paused evaluations, they shall be resumed.

6.3. Expected behavior of hypermedia players in NCL applications

NCL media objects (<media> elements of "application/x-ginga-NCL" or "application/x-ncl-NCL" type) and HTML-based objects (<media> elements of "text/html" type) embedded in NCL applications shall follow the guidelines stablished in "Nested Context Language 3.0: Part 11 – Declarative Hypermedia Objects in NCL: Nesting Objects with NCL code in NCL Documents" [NCL Part 11].

6.4. Relation between the presentation-event state machine of a node and the presentation-event state machine of its parent node

This section applies for objects represented by <context>, <body>, <switch> elements a <media> element of "application/x-ginga-NCL" (or "application/x-ncl-NCL") type.

Whenever the presentation event of a node (media or composite) goes to the occurring state, the presentation event of the composite node (or NCL node of "application/x-ncl-NCL" or "application/x-ginga-NCL" type) that contains the node shall also enter in the occurring state.

When all child nodes of a composite node (or an NCL node of "application/x-ncl-NCL" or "application/x-ginga-NCL" type) have their presentation events in the sleeping state, the presentation event of the composite node (or NCL node of "application/x-ncl-NCL" or "application/x-ginga-NCL" type) shall also be in the sleeping state.

Composite nodes (or NCL node of "application/x-ncl-NCL" or "application/x-ginga-NCL" type) do not need to infer *aborts* transitions from their child nodes. These transitions in presentation events of composite nodes shall occur only when instructions are applied directly to the composite node (or the NCL node of "application/x-ncl-NCL" or "application/x-ginga-NCL" type) presentation event.

When all child nodes of a composite node (or an NCL node of "application/x-ncl-NCL" or "application/x-ginga-NCL" type) have their presentation events in a state different from the *occurring* state and at least one child node have its main-event in the *paused* state, the presentation event of the composite node (or the NCL node of "application/x-ncl-NCL" or "application/x-ginga-NCL" type) shall also be in the paused state.

If a <switch> element is started, but it does not define a default component and none of the <bindRule> referred rules is evaluated as true, the switch presentation shall not enter in the occurring state.

6.5. Expected behavior of imperative players in NCL applications

Imperative players for NCL media objects of "application/x-ginga-NCLua" (or "application/x-ncl-NCLua") and "application/x-ginga-NCLet" (or "application/x-ncl-NCLet") types shall follow the guidelines established in Section 5 of the Technical Report "Nested Context Language 3.0: Part 10 - Imperative Objects in NCL: The NCLua Scripting Language", [NCL Part 10]. This section is an authorized copy of reference [NCL Part 10], due to its importance in the definition of the bridge between Ginga-NCL and an imperative environment.

Document authors may define NCL links to start, stop, pause, resume or abort the execution of an imperative code. An imperative player (the language engine) shall interface the imperative execution environment with the NCL formatter.

Analogous to perceptual media content players (video, audio, image, etc.), imperative-code players shall control event state machines associated with the imperative object. As an example, if the code finishes its execution, the player shall generate the stops transition in the event presentation state machine corresponding to the code execution. However, different from media content players, an imperative-code player does not have sufficient information to control by itself all event state machines, and shall rely on the imperative application content to command these controls.

NCL links may be bound to imperative object interfaces (<area> and <property> elements, and the default content anchors).

If an external link starts, stops, pauses, resumes or aborts the presentation of an anchor representing an <area> element or the *main* content anchor, callbacks in the imperative code shall be triggered. The way these callbacks are defined is responsibility of each imperative code associated with the imperative object.

On the other hand, an imperative code may also command the start, stop, pause or resume of its associated content anchors through an API offered by the language. These transitions may be used as conditions in 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.

An imperative code may also be synchronized with other objects through <property> elements. When the <property> element is mapped to a code span (function, method, etc.) through its *name* attribute, a link action "start" applied to the property shall cause the code execution, with the set values interpreted as parameters passed to the code span. When the <property> element is mapped to an imperative-code attribute the action "start" shall assign the value to the attribute. As usual, the event state machine associated with the property shall be controlled by the imperative-object player, but sometimes, commanded by the imperative application.

A <property> element defined as a child of a <media> element representing an imperative object may also be associated with an NCL link assessment role. In this case, the NCL formatter shall query the property value in order to evaluate the link expression. If the <property> element is mapped to a code attribute, the code attribute value shall be returned by the imperative-object player to the NCL formatter. If the <property> element is mapped to a code span, the code shall be executed and its output value shall be returned by the imperative-object player to the NCL formatter.

6.5.1. Imperative-Object Execution Model

The lifecycle of an imperative object is controlled by the NCL formatter. The formatter is responsible for triggering the execution of an imperative object and for mediating the communication among this object and other nodes in an NCL document.

As with all media object players, once instantiated, the imperative-object player shall execute an initialization procedure. However, different from other media players, this initialization code is specified by the author of the imperative code. 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 and, in particular, registers one (or more) event handler for communication with the NCL formatter. Note that at least the code span associated with the *main content anchor* shall be created during the initialization procedure.

After the initialization, the execution of the imperative object becomes event oriented in both directions. That is, any action commanded by the NCL formatter reaches the registered event handlers, and any NCL event state change notification is sent as an event to the NCL formatter (as for example, the natural end of a code span execution). The imperative-object player is then ready to perform any instruction as discussed in the next sections.

6.5.2. Instructions to Presentation Events

NCL formatters may control imperative-object players issuing instructions that may cause changes on state machines of presentation events (code span executions). On the other hand, any state changes on these presentation event state machines are notified to the NCL formatter.

6.5.2.1. start instruction

The *start* instruction issued by a formatter shall inform the following parameters to the imperative-object player: the imperative 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 by the imperative-object player, the content-anchor *label,* or by default the *main* content anchor, identifying the associated imperative code to be started, and an optional delay-time. From the *src* attribute, the imperative-object player tries to locate the imperative code and start its execution. If the content cannot be located, the player shall finish the starting operation, without performing any action.

The descriptor shall be chosen by the formatter following the directives specified in the NCL document. If the *start* instruction results from a link action that has a descriptor explicitly declared in its <bind> element (*descriptor* attribute of the <link> element's children <bind> element), the resulting descriptor informed by the formatter shall merge the attributes of the bind descriptor with the attributes of the descriptor specified in the corresponding <media> element, if this attribute was specified. For the common attributes, the <bind> descriptor information shall superpose the <media> descriptor data. If the <bind> element does not contain an explicit descriptor, the descriptor informed by the formatter shall be the <media> descriptor, if this attribute was specified. Otherwise, a default descriptor for that imperative-object *type* of <media> shall be chosen by the formatter.

The list of events to be monitored by an imperative-object player should also be computed by the formatter, taking into account the NCL document specification. The formatter shall check all links where the imperative object and the resulting descriptor participate. When computing the events to be monitored, the formatter shall take into account the media-object perspective, i.e., the path of <body> and <context> elements to reach the <media> element. Only links contained in these <body> and <context> elements should be considered to compute the monitored events.

As with any other <media> element, the delay-time is an optional parameter and its default value is "zero". If greater than zero, this parameter contains a time to be waited by the imperative-object player before starting the code execution.

Different from what is performed on other <media> elements, if an imperative-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 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, differently from what happens for other <media> elements, a <simpleAction> element with an *actionType* attribute equal to "stop", "pause", "resume" or "abort" shall be bound through a link to a NCLua node interface, which shall not be ignored when the action is applied.

Since neither the formatter nor the imperative-code player has any other knowledge about the imperative-object's content anchors, except their *id* and *label* attributes, they do not know which other content anchors shall have their associated event put in the occurring state when a content anchor is started or is being in execution. Therefore, except for the event associated with the *whole content anchor*, it is the responsibility of the imperative-code span, as soon as it is started, to command the imperative-code player to change the state of any other event state machine that is related to the event state machine associated to the started code and to inform if a transition associated with a change shall be notified. Similarly, it is the responsibility of the imperative-code span to command any event state change, and to inform if the associated transition shall be notified, if the code-span execution starts another code span associated with a content anchor.

Differently from other <media> elements, if any content anchor is started and the event associated with the *whole content anchor* is in *sleeping* or *paused* state, it shall be put in the *occurring* state and the corresponding transition shall be notified.

6.5.2.2. stop instruction

The *stop* instruction needs to identify an imperative code span already being controlled. To identify the imperative code span means to identify the corresponding <media> element, the corresponding descriptor, a <media> element's interface and the imperative-object perspective.

The *stop* instruction issued by an NCL formatter shall be ignored by an imperative-object player if the imperative code span associated with the specified interface is not being executed (if the corresponding event is not in the *occurring* or *paused* state) and the imperative-object player is not waiting due to a delayed *start* instruction. If the imperative-object interface is being executed, its corresponding presentation event shall transit to the *sleeping* state, and their *stops* transitions shall be notified. The imperative code associated with the interface shall be stopped. If the *repetitions* event attribute is greater than zero, it shall be decremented by one and the imperative code associated with the interface shall restart after the repeat delay time (the repeat delay shall have been passed to the media player as the start delay parameter). If the imperative object is waiting to be presented after a delayed *start* instruction and a *stop* instruction is issued, the previous *start* instruction shall be removed.

For the same reason discussed in the *start* instruction, except for the event associated with the *whole content anchor*, it is responsibility of the stopped-code span, before it stops, to command the imperative-code player to change the state of any other event state machine that is related with the event state machine associated to the stopped code, and to inform if a transition associated with a change shall be notified.

Different from other <media> elements, if any content anchor is stopped and all other presentation events are in the *sleeping* state the *whole content anchor* shall be put in the *sleeping* state. If a content anchor is stopped and at least one other presentation event is in the *occurring* state the *whole content anchor* shall remain in the *occurring* state. In all other cases, if a content anchor is stopped the *whole content anchor* shall be put in the *paused* state. If the *stop* instruction is applied to an imperative object without specifying the node's interface, the *whole content anchor* is assumed. In this case, *stop* instructions shall be issued for all other content anchors.

6.5.2.3. abort instruction

The *abort* instruction needs to identify an imperative code span already being controlled. To identify the imperative code span means to identify the corresponding <media> element, the corresponding descriptor, a <media> element's interface and the imperative-object perspective.

If the imperative code associated with the object's interface is not being executed and is not waiting to be executed after a delayed *start* instruction, the *abort* instruction shall be ignored. If the imperative code associated with the object's interface is being executed, its associated event, in the *occurring* or in the *paused* state, shall transit to the *sleeping* state, and their *aborts* transitions shall be notified. If the *repetitions* event attribute is greater than zero, it shall be set to zero and the imperative-code execution shall not restart. If the imperative code associated with the object's interface is waiting to be executed after a delayed *start* instruction and an *abort* instruction is issued, the previous *start* instruction shall be removed.

For the same reason discussed in the *start* instruction, except for the event associated with the *whole content anchor*, it is the responsibility of the aborted-code span, before it aborts, to command the imperative-code player to change the state of any other event state machine that is related to the event state machine associated to the aborted code, and to inform if a transition associated with a change shall be notified.

Differently from other <media> elements, if any content anchor is aborted and all other presentation events are in the *sleeping* state, the *whole content anchor* shall be put in the *sleeping* state. If a content anchor is aborted and at least one other presentation event is in the *occurring* state, the *whole content anchor* shall remain in the *occurring* state. In all other cases, if a content anchor is aborted, the *whole content anchor* shall be put in the *paused* state. If the *abort* instruction is applied to an imperative object without specifying the node's interface, the *whole content anchor* is assumed. In this case, *abort* instructions shall be issued for all other content anchors.

6.5.2.4. pause instruction

The *pause* instruction needs to identify an imperative code span already being controlled. To identify the imperative code span means to identify the corresponding <media> element, the corresponding descriptor, a <media> element's interface and the imperative-object perspective.

If the imperative code associated with the object’s interface is not being executed (and not in the *paused* state) and is not waiting to be executed after a delayed *start* instruction, the instruction shall be ignored. If the imperative code associated with the object’s interface is being executed, its associated event in the *occurring* shall transit to the *paused* state, and the pause elapsed time shall not be considered as part of the object duration. If the imperative code associated with the object’s interface is waiting to be executed after a delayed *start* instruction, the imperative-object’s interface shall wait for a resume instruction to continue waiting for the remaining start delay.

For the same reason discussed in the *start* instruction, except for the event associated with the *whole content anchor*, it is the responsibility of the paused-code span, before it pauses, to command the imperative-code player to change the state of any other event state machine that is related to the event state machine associated to the paused code, and to inform if a transition associated with a change shall be notified.

Differently from other <media> elements, if any content anchor is paused and all other presentation events are in the *sleeping* state or *paused* state the *whole content anchor* shall be put in the *paused* state. If a content anchor is paused and at least one other presentation event is in the *occurring* state, the *whole content anchor* shall remain in the *occurring* state. If the *pause* instruction is applied to an imperative object without specifying the node’s interface, the *whole content anchor* is assumed. In this case, *pause* instructions shall be issued for all other content anchors that are in the occurring state.

6.5.2.5. resume instruction

The *resume* instruction needs to identify an imperative code span already being controlled. To identify the imperative code span means to identify the corresponding <media> element, the corresponding descriptor, a <media> element’s interface and the imperative-object perspective.

If the imperative code associated with the object’s interface is not paused or the imperative-object player is not paused (waiting for the start delay), the instruction shall be ignored. If the imperative-object player is paused waiting for the start delay, it shall resume the wait from the instant it was paused. If the imperative code associated with the object’s interface is paused, its associated event shall transit to the *occurring* state, and their *resumes* transitions shall be notified.

For the same reason discussed in the *start* instruction, except for the event associated with the *whole content anchor*, it is the responsibility of the paused-code span, before it pauses, to command the imperative-code player to change the state of any other event state machine that is related to the event state machine associated to the paused code, and to inform if a transition associated with a change shall be notified.

Differently from other <media> elements, if any content anchor is resumed, the *whole content anchor* shall be set to the *occurring* state. If the *resume* instruction is applied to an imperative object without specifying the node’s interface, the *whole content anchor* is assumed. If the *whole content anchor* is not in the *paused* state due to a previous receive of a *pause* instruction, the *resume* instruction is ignored. Otherwise, *resume* instructions shall be issued for all other content anchors that are in the *paused* state, except those that were already paused before the *whole content anchor* received the *paused* instruction.

6.5.2.6. Natural end of a code execution

Events of an imperative object normally end their execution naturally, without needing external instructions. In this case, immediately before ending, the code span shall command the imperative-code player to change the state of any other event state machine that is related to the event state machine associated to the ending code, and to inform if a transition associated with a change shall be notified. The ending presentation event shall transit to the *sleeping* state, and their *stops* transitions shall be notified. If the *repetitions* event attribute is greater than zero, it shall be decremented by one and the imperative code associated with the interface shall restart after the repeat delay time (the repeat delay shall have been passed to the media player as the start delay parameter).

Differently from other <media> elements, if any content anchor execution ends and all other presentation events are in the *sleeping* state, the *whole content anchor* shall be put in the *sleeping* state. If a content anchor execution ends and at least one other presentation event is in the *occurring* state, the *whole content anchor* shall remain in the *occurring* state. In all other cases, if a content anchor execution ends, the *whole content anchor* shall be set to the *paused* state.

6.5.3. Instructions to Attribution Events

NCL formatters may also send instructions that may cause changes on state machines of attribution events (code span executions). Similarly to presentation events, any state changes on attribution event state machines are notified to the NCL formatter.

Although imperative-node properties may be associated with code spans, the execution of these spans does not change any state machine associated with content anchors of the imperative object.

6.5.3.1. start instruction

The *start* instruction issued by a formatter may be applied to an imperative object's property independently from the fact whether the object is being in execution (the *whole content anchor* is in the *occurring* state) or not (in this latter case, although the object is not being executed, its imperative-object player shall have already been instantiated). In the first case, the *start* instruction needs to identify the imperative object, a monitored attribution event, and, if it is the case, a value to be passed to the imperative code wrapped by the event. In the second case, the instruction shall also identify the <descriptor> element that will be used when executing the object (as it is done for the *start* instruction for presentation). When setting a value to an attribute, the imperative-object player shall set the event state machine to the *occurring* state*,* and after finishing the attribution, again to the *sleeping* state, generating the *starts* transition and afterwards the *stops* transition.

Note again that, if a *start* instruction is applied to a <property> element that calls the execution of a code span, no content anchor state is affected.

For every monitored attribution event, if an imperative-object's code span changes by itself the corresponding attribute value, it shall also command the imperative-code player that shall proceed as if it had received an external *start* instruction.

6.5.3.2. stop, abort, pause and resume instructions

With the exception of the *start* instruction, discussed in the previous section, all other instructions have the same effect on the corresponding property attribution as they have on any property attribution of any type of object.

The *stop* instruction only stops the property attribution procedure, bringing the attribution event state machine to the *sleeping* state.

The *abort* instruction stops the property attribution procedure, bringing the attribution event state machine to the *sleeping* state and the property value to its original one.

The *pause* instruction only pauses the property attribution procedure, bringing the attribution event state machine to the *paused* state.

Finally, the *resume* instruction only resumes the property attribution procedure, bringing the attribution event state machine to the *occurring* state.

7. Live Editing and NCL Stream Events

The core of the Ginga-NCL presentation engine is composed of the NCL Formatter and its Private Base Manager module.

The NCL Formatter is in charge of receiving an NCL document and controlling its presentation, trying to guarantee that the specified relationships among media objects are respected. The formatter deals with NCL documents that are collected inside a data structure known as *private base*. Ginga associates a private base with a TV channel. NCL documents in a private base may be started, paused, resumed, stopped and may refer to each other.

The Private Base Manager is in charge of receiving NCL document editing commands and maintaining the active NCL documents (documents being presented).

The DSM-CC is adopted in SBTVD-T for carrying editing commands in MPEG-2 TS elementary streams, when commands come from the terrestrial broadcast channel. DSM-CC stream events and DSM-CC object carousel protocol are the basis for document handling in the Ginga presentation engine in agreement with SBTVD-T.

Editing commands are codified as DSM-CC stream events. The Ginga presentation engine shall be able to manage at least the Editing Command stream event, whose syntax is shown in Figure 7.1

Syntax	Number of bits
StreamEventDescriptor ( ) {	
descriptorTag	**8**
descriptorLength	**8**
eventId	**8**
reserved	**31**
eventNPT	**33**
privateDataLength	**8**
commandTag	**8**
sequenceNumber	**7**
finalFlag	**1**
privateDataPayload	**8 to 2008**
FCS	**8**
}	

Figure 7.1 - Editing command stream event descriptor

Event objects are used to map stream event names into stream event ids. Event objects are used to inform Ginga about DSM-CC stream events that can be received. Event names allow specifying types of events, offering a higher abstraction level for middleware applications when registering and handling DSM-CC stream events. The Private Base Manager, as well as NCL execution-objects (e.g. NCLua), must register themselves as listeners of stream events they handle, using event names.

NCL document files, and NCL media-object's contents are organized in file system structures. XML-based editing *command parameters* can be directly transported in the payload of a stream event descriptor or, alternatively, organized in file system structures to be transported, each one, in an object carousel. A DSM-CC carousel generator is used to join the file systems and stream event objects into a data elementary stream.

When an NCL document editing command needs to be sent, a DSM-CC event object must be created, mapping the string "*nclEditingCommand*" into a selected stream event id (see [NCL Part 9]), and putting it as an object in a DSM-CC object carousel. One or more DSM-CC stream event descriptors with the previous selected id are then created and sent in another MPEG-2 TS elementary stream. These stream events usually have their time reference set to zero, but can be postponed to be executed at a specific time. The Private Base Manager must register itself as an "*nclEditingCommand*" listener and is notified when this stream event arrives. The received *commandTag* is then used by the Private Base Manager to interpret the complete *command string* semantics. If the XML-based *command parameter* is short enough it is transported directly in the stream event descriptors payload. Otherwise, the privateDataPayload carries a set of reference pairs. In the case of pushed files (NCL documents or nodes), each pair relates a set of file paths with their respective location in the transport system. In the case of pulled files received from an interactivity channel or sited in the receiver itself, no reference pairs have to be sent, except the {uri, "null"} pair associated with the NCL document or XML node specification that is commanded to be added.

Receivers that only implement the NCL Basic DTV profile cannot handle the following commands: *pauseDocument*, *resumeDocument*, *addTransition*, *removeTransition*, *addTransitionBase* and *removeTransitionBase*.

Ginga associates a private base with a TV channel. When a channel is tuned, its corresponding private base is opened and activated by the Private Base Manager; other private bases shall be deactivated. For security reasons, only one private base may be active at a time. The simplest and most restricted way to manage private bases is to have only one private base associated with a TV channel open at a time. However, the number of private bases that may be kept open is a specific middleware implementation decision.

NCL resident applications are managed in a specific private base.

Add commands always have NCL entities as their arguments (XML-based command parameters). Whether the specified entity already exists or not, document consistency shall be maintained by the NCL formatter, in the sense that all entity attributes classified as required shall be defined.

7.1. Resource Identification

The "addDocument" e "addNode" editing commands must be used to map the server data structures to the data structure used in the receiver, when XML documents referred by these commands (NCL document files or other XML document files, respectively) are transmitted through an object carousel. In this case, in the same object carousel that carries the XML document specification, an event object must be transmitted in order to map the name "*nclEditingCommand*" to the eventId of the DSM-CC stream event descriptor that carries the addDocument or addNode editing command. The privateDataPayload of the stream event descriptor shall carry a set of {uri, id} reference pairs. The uri parameter of the first pair shall have the “x-sbtvd” schema (optional) and the absolute path of the NCL document or the NCL node specification (the path in the data server). The corresponding id parameter in the pair shall refer to the NCL Document or NCL Node specification IOR (carouselId, moduleId, objectKey; see ISO/IEC 13818-6) in the object carousel. If other file systems have to be transmitted using other object carousels in order to complete the editing command with media content (as it is usual in the case of addDocument or addNode commands), other {uri, id} pairs shall be present in the command. In this case, the uri parameter shall have the “x-sbtvd” schema (optional) and the absolute path of file system root (the path in the datacast server), and the corresponding id parameter in the pair shall refer to the IOR (carouselId, moduleId, objectKey; see ISO/IEC 13818-6) of any root child file or directory in the object carousel (the IOR of the carousel service gateway).

Usually, the transmission of files systems using other object carousels different from the one that carries the event object is necessary when the file systems that represent the application cannot be modeled by a unique tree data structure. An example can be found in [NCL Part9].

7.2. Default values

1. The *baseId* identifier of a private base associated with a TS channel assumes the *tsid* (transport stream identifier) of this TV channel, which shall be obtained from the *tsid* field of the PAT table.
2. When the *baseId* parameter of an *nclEditingCommand* transported in a transport stream (TS) is not specified, it shall assumed the same *tsid* of TS where it is transported.
3. When the *baseId* parameter of an *nclEditingCommand* coming from an NCLua object running in a certain private base is not specified, it shall assumed the same *baseId* value of this private base.
4. In an addDocument *nclEditingCommand*, if all resources of the application are below the same root, the *id* parameter of the {uri, id} pair may be omitted.

7.3. Exception handling

1. If the *baseId* parameter of an *nclEditingCommand* transported in a TS stream with *tsid* identifier has a value different from the *tsid* value, it shall be ignored.
2. If the *baseId* parameter of an *nclEditingCommand* coming from an NCLua object running in a certain private base has a value different from the *baseId* value of this private base, it shall be ignored.
3. Receivers that only implement the NCL Basic DTV profile cannot handle the following *nclEditingCommands*: *pauseDocument*, *resumeDocument*, *addTransition*, *removeTransition*, *addTransitionBase* and *removeTransitionBase*.
4. An *nclEditingCommand* that can cause document inconsistency or referring to an inexistent NCL element or any other inexistent identifier shall be ignored.

8. NCLua API

The scripting language adopted by Ginga-NCL to implement procedural objects in NCL documents is *Lua* (<media> elements of application/x-ginga-NCLua type, or application/x-ncl-NCLua type).

Besides the Lua standard library, the following modules shall be implemented: canvas; event; settings; persistent.

8.1. The *canvas* module

A canvas offers a graphical API to be used in an NCLua application. Using the API, it is possible to draw lines, rectangles, font, images, etc.

8.1.1. Default values

1. In all **canvas: drawXXX** operations, the line width shall be assumed as 1 pixel.
2. In the **canvas:drawEllipse (mode: string; xc, yc, width, height, ang_start, ang_end: number)** operation the angle units shall be assumed as degrees.
3. In the **canvas:drawEllipse (mode: string; xc, yc, width, height, ang_start, ang_end: number)** operation the 0 degree angle is in the higher Y coordinate of the ellipse and the angle progression follows the clockwise motion.

8.2. The *event* module

This module offers an API for event handling. Using the API, the NCL formatter may communicate with an NCLua application asynchronously.

In the **event.register ([pos: number]; f: function; [class: string]; […: any]),** the initial register position is 1.

In the **event.post** of class='si' and type='epg', the hasInteractivity data-table subfield shall consider the compatible carousel_ descriptor, the NCL editing commands and the AIT tables in order to specify if the EPG event has (or has not) an application.

8.2.1. Default values

1. In *ncl class*, events may be directed to specific anchors or to the whole node, this is identified by *label* field, which assumes the whole node when absent.

8.2.2. Exception handling

In the **event.register ([pos: number]; f: function; [class: string]; […: any]),** when a *handler* is registered in a position occupied by another one, every handler position from that position on shall be incremented, in order to give place to the new insertion. When a *handler* is removed, all other handler positions, from the removed handler position on shall be decremented.

9. Final Remarks

In order to offer a scalable hypermedia model, with characteristics that may be progressively incorporated in hypermedia system implementations, NCM was divided in several parts, and also its declarative XML application language: NCL. Ginga-NCL is the declarative environment of Ginga middleware responsible for running NCL applications. This technical report deals with the operational guidelines for Ginga-NCL implementations aiming at terrestrial and satellital DTV, and IPTV systems, which follows Norms ABNT 15606.2 and 15606-5, and ITU-T Recommendation H.761.

References

[ABNT 15604] ABNT NBR 15604, *Televisão digital terrestre – Receptores*

ABNT NBR 15604, *Digital terrestrial television – Receivers*

Available: http://www.abnt.org.br/imagens/Normalizacao_TV_Digital/ABNTNBR15604_2007Ing_2008.pdf

[ABNT 15606-1] ABNT NBR 15606-1, *Televisão digital terrestre – Codificação de dados e especificações de transmissão para radiodifusão digital – Parte 1: Codificação de dados*

ABNT NBR 15606-1, *Digital terrestrial television — Data coding and transmission specification for digital Broadcasting - Part 1: Data coding specification*

Available: http://www.abnt.org.br/imagens/Normalizacao_TV_Digital/ABNTNBR15606-1_2007Ing_2008.pdf

[ABNT 15606-2] ABNT NBR 15606-2, *Televisão digital terrestre – Codificação de dados e especificações de transmissão para radiodifusão digital – Parte 2: Ginga-NCL para receptores fixos e móveis – Linguagem de aplicação XML para codificação de aplicações*

ABNT NBR 15606-2, *Digital Terrestrial TV – Data Coding and transmission specification for digital broadcasting – Part 2: Ginga-NCL for fixed and mobile receivers: XML application language for application coding.*

Available: http://www.abnt.org.br/imagens/Normalizacao_TV_Digital/ABNTNBR15606-2_2007Ing_2008.pdf

[ABNT 15606-5] ABNT NBR 15606-5, *Televisão digital terrestre – Codificação de dados e especificações de transmissão para radiodifusão digital – Parte 5: Ginga-NCL para receptores portáteis – Linguagem de aplicação XML para codificação de aplicações*

ABNT NBR 15606-5, *Digital Terrestrial TV – Data Coding and transmission specification for digital broadcasting – Part 5: Ginga-NCL for portable receivers: XML application language for application coding*

Available: http://www.abnt.org.br/imagens/Normalizacao_TV_Digital/ABNTNBR15606-5_2007Ing_2008.pdf

[Antonacci 00] Antonacci M.J. NCL: Uma Linguagem Declarativa para Especificação de Documentos Hipermídia com Sincronização Temporal e Espacial. Master Dissertation, Departamento de Informática, PUC-Rio, April 2000.

[AMRS 00] Antonacci M.J., Muchaluat-Saade D.C., Rodrigues R.F., Soares L.F.G. NCL: Uma Linguagem Declarativa para Especificação de Documentos Hipermídia na Web, VI Simpósio Brasileiro de Sistemas Multimídia e Hipermídia - SBMídia2000, Natal, Rio Grande do Norte, June 2000.

[ITU J.201] ITU Recommendation J.201:2004, *Harmonization of declarative content format for interactive television applications*.

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

[NCL Part 1] Soares L.F.G; Rodrigues R.F. Nested Context Model 3.0: Part 1 – NCM Core, Technical Report, Departamento de Informática PUC-Rio, May 2005, ISSN: 0103-9741.

[NCL Part 8] Soares L.F.G; Rodrigues R.F. Nested Context Language 3.0: Part 8 – NCL Live Editing Commands, Technical Report, Departamento de Informática PUC-Rio, December 2006, ISSN: 0103-9741.

[NCL Part 9] Soares, L.F.G.; Rodrigues, R.F.; Costa, R.R.; Moreno, M. F. *Nested Context Language 3.0: Part 9 – NCL Live Editing Commands*. Technical Report, Informatics Department, PUC-Rio, No. 36/06. Rio de Janeiro. December 2006. ISSN 0103-9741.

[NCL Part 10] Soares, L.F.G.; Sant'Anna, F.F; Cerqueira, R. F. G. *Nested Context Language 3.0: Part 10 – Imperative Objects in NCL: The NCLua Scripting Language*. Technical Report, Informatics Department, PUC-Rio, No. 02/08. Rio de Janeiro. January 2008. ISSN 0103-9741.

[NCL Part 11] Soares, L.F.G. *Nested Context Language 3.0: Part 11 – Declarative Hypermedia Objects in NCL: Nesting Objects with NCL code in NCL Documents*. Technical Report, Informatics Department, PUC-Rio, No. 02/09. Rio de Janeiro. January 2009. ISSN 0103-9741.

[NCL Part 12] Soares, L.F.G. *Nested Context Language 3.0: Part 12 – Support to Multiple Exhibition Devices*. Technical Report, Informatics Department, PUC-Rio, No. 03/09. Rio de Janeiro. January 2009. ISSN 0103-9741.

[RDF 99] Resource Description Framework (RDF) Model and Syntax Specification, Ora Lassila and Ralph R. Swick. W3C Recommendation, 22 February 1999.
Available at http://www.w3.org/TR/REC-rdf-syntax/

[SCHE 01] XML Schema Part 0: Primer, W3C Recommendation, in http://www.w3.org/TR/xmlschema-0/, May 2001.

[SMIL 2.1] W3C Recommendation, *Synchronized Multimedia Integration Language – SMIL 2.1 Specification*. December de 2005

[XML 1.0] Bray T., Paoli J., Sperberg-McQueen C.M., Maler E. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, in http://www.w3.org/TR/REC-xml, February 1998.