INTERNATIONAL TELECOMMUNICATION UNION

TELECOMMUNICATION STANDARDIZATION SECTOR

STUDY PERIOD 2013-2016

STUDY GROUP 16

7rTD 130 (WP 2/16)

English only

Original: English

Question(s):	13/16	Geneva, 28 October – 08 November 2013

TD

Source:	Editor H.761
Title:	H.761 “Nested context language (NCL) and Ginga-NCL” (Rev.): Initial draft of revised text (Geneva, 28 October – 8 November 2013)

Summary

This document is the output text of Draft Recommendation ITU-T H.761 “Nested context language (NCL) and Ginga-NCL” agreed by the SG16 meeting (Geneva, 28 October – 08 November 2013), based on the discussions of COM16-C.425R1 (Brazil) "H.761: New NCL version and profile".

Contact:	Marcelo MORENO UFJF Brazil (Federative Republic of)	Tel: Fax: Email:	+55 32 2102 3311 moreno@ice.ufjf.br
Contact:	Luiz Fernando Gomes SOARES PUC-Rio Brazil (Federative Republic of)	Tel: Fax: Email:	+55 21 3527 1500 ext. 4330 +55 21 3527 1530 lfgs@inf.puc-rio.br

Table of Contents

Page

List of Figures

List of Tables

Page

Page

Electronic attachment: NCL 3.1 module schemas

Draft revised ITU-T H.761

Nested Context Language (NCL) and Ginga-NCL

AAP Summary

[To be added before Consent]

Summary

Recommendation ITU-T H.761 gives the specification of the Nested Context Language (NCL) and of an NCL presentation environment called Ginga-NCL to provide interoperability and harmonization among IPTV multimedia application frameworks.

NCL is a declarative glue language that holds media object presentations synchronized in time and space, no matter the types of the media objects. Ginga-NCL is an NCL presentation engine built as a component of a DTV middleware.

This Recommendation includes an electronic attachment containing NCL 3.0 module schemas used in the Enhanced DTV profile.

Introduction

Nested Context Language (NCL) is a declarative XML-based language initially designed aiming at hypermedia document specification for the Web. The language's flexibility, reusability, multi-device support, application content adaptability and, mainly, the language intrinsic ability for easily defining spatiotemporal synchronization among media assets, including viewer interactions, make it an outstanding solution for IPTV systems. NCL is also the declarative language used in the Japanese Brazilian terrestrial DTV standard (ISDB-T).

NCL is a glue language that holds media objects together in a multimedia presentation, no matter which object types they are. In this sense, media objects may be image objects (JPEG, PNG, etc.), video objects (MPEG, MOV, etc.), audio objects (MP3, WMA, etc.), text objects (TXT, PDF, etc.), imperative objects (with Lua code, etc.), other declarative objects (HTML, LIME, SVG, MHEG, nested NCL applications, etc.), etc. Which media objects are supported depends on which media players are embedded in the NCL Player (part of the Ginga-NCL environment). As an example, NCL treats an HTML document as one of its possible media objects. In this way, NCL does not substitute but embed XHTML-based documents. The same reasoning applies to other media content and multimedia content objects, and also to objects with content coded in any computer language.

Ginga-NCL is an NCL presentation engine built as a component of a DTV middleware. An open source reference implementation of Ginga-NCL is available under the GPLv2 licence (http://www.gingancl.org.br/index_en.html). This reference implementation was developed in a way that it can easily integrate a variety of media-object players for audio, video, image, text, etc., including imperative execution engines and other declarative language players.

A special NCL object type defined in Ginga-NCL is NCLua, an imperative media-object with Lua code as its content. Because of its simplicity, efficiency and powerful data description syntax, Lua is considered the default scripting language for Ginga-NCL. The Lua engine is small and written in ANSI/C, making it easily portable to several hardware platforms. The Lua engine is also distributed as free software under the Massachusetts Institute of Technology (MIT) licence (http://www.lua.org/license.html).

Recommendation ITU-T H.761

Nested context language (NCL) and Ginga-NCL

1 Scope

This Recommendation[1] specifies the Nested Context Language (NCL) and an NCL presentation environment, called Ginga-NCL, to provide interoperability and harmonization among IPTV multimedia application frameworks. To provide global standard IPTV services, it is foreseeable that a combination of different standard multimedia application frameworks will be used. Therefore, this Recommendation specifies the Nested Context Language, as one of those standards that compose the multimedia application frameworks, to provide interoperable use of IPTV services. Ginga-NCL is an NCL presentation environment that integrates NCL and Lua players. NCL and Lua frameworks can be used in other declarative environments, but if they are used together they shall follow the Ginga-NCL specification.

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[ITU-T H.222.0]	Recommendation ITU-T H.222.0 (2006) | ISO/IEC 13818-1:2007, *Information technology – Generic coding of moving pictures and associated audio information: Systems*.
[ITU-T H.750]	Recommendation ITU-T H.750 (2008), *High-level specification of metadata for IPTV services*.
[ITU-T J.200]	Recommendation ITU-T J.200 (2001), *Worldwide common core – Application environment for digital interactive television services*.
[ISO/IEC 13818-6]	ISO/IEC 13818-6 (1998), *Information technology – Generic coding of moving pictures and associated audio information – Part 6: Extensions for DSM-CC.* Plus its Amd.1 (2000), Amd.2 (2000), Amd.3 (2001).

3 Definitions

3.1 Terms defined elsewhere

This Recommendation uses the following terms defined elsewhere:

3.1.1 application [ITU-T J.200]: Information that expresses a specific set of observable behaviour.

3.1.2 application environment [ITU-T J.200]: The context or software environment in which an application is processed.

[1] This Recommendation includes an electronic attachment containing NCL 3.1 module schemas used in the Enhanced DTV profile.

3.1.3 application programming interface (API) [ITU-T J.200]: Software libraries that provide uniform access to system services.

3.1.4 character [ITU-T J.200]: Specific "letter" or other identifiable symbol, e.g., "A".

3.1.5 data carousel [ITU-T J.200]: A transmission scheme defined in [ISO/IEC 13818-6], with which data is transmitted repetitively. It can be used for downloading various data in broadcasting. It is the scheme of the DSM-CC User-to-Network Download protocol that embodies the cyclic transmission of data.

3.1.6 declarative application [ITU-T J.200]: An application which is started by, and primarily makes use of, a declarative information to express its behaviour.

3.1.7 declarative application environment [ITU-T J.200]: An environment that supports the processing of declarative applications.

3.1.8 digital storage media command and control (DSM-CC) [ITU-T J.200]: A control method defined in [ISO/IEC 13818-6], which provides access to files or streams for digital interactive services.

3.1.9 electronic program guide (EPG) [b-ITU-T H.770]: A service navigation interface which is used especially for programs.

3.1.10 element [ITU-T J.200]: A portion of document delimited by tags.

3.1.11 elementary stream (ES) [ITU-T H.222.0]: A generic term for one of the coded video, coded audio or other coded bit streams in PES packets. One elementary stream is carried in a sequence of PES packets with one, and only one, stream id.

3.1.12 execution engine [ITU-T J.200]: A subsystem in a receiver that evaluates and executes imperative applications consisting of computer language instructions and associated data and media content. An execution engine may be implemented with an operating system, computer language compilers, interpreters, and Application Programming Interfaces (APIs), which an imperative application may use to present audiovisual content, interact with a user, or execute other tasks, which are not evident to the user. A common example of an execution engine is the JavaTV software environment, using the Java programming language and byte code interpreter, JavaTV APIs, and a Java Virtual Machine for program execution.

3.1.13 locator [ITU-T J.200]: A linkage, expressed in the syntax provided in RFC 2396, which provides a reference to an application or resource.

3.1.14 markup language [ITU-T J.200]: A formalism that describes document structures, appearances, or other aspects. XHTML is an example of markup language.

3.1.15 normal play time (NPT) [ITU-T J.200]: The absolute temporal coordinates that represent the position in a stream at which an event occurs.

3.1.16 packet identifier (PID) [ITU-T H.222.0]: A unique integer value used to identify elementary streams of a program in a single or multi-program transport stream.

3.1.17 persistent storage [ITU-T J.200]: Memory available that can be read/written to by an application and may outlive the application's life. Persistent storage can be volatile or non-volatile.

3.1.18 plug-in [ITU-T J.200]: A set of functionalities that may be added to a generic platform in order to provide additional functionality.

3.1.19 presentation engine [ITU-T J.200]: A subsystem in a receiver that evaluates and presents declarative applications (consisting of content such as audio, video, graphics, and text) primarily based on presentation rules defined in the presentation engine. A presentation engine also responds to formatting information, or "markup", associated with the content, to user inputs, and to script statements, which control presentation behaviour and initiate other processes in response to user input and other events.

3.1.20 **receiver platform (platform)** [ITU-T J.200]: The receiver's hardware, operating system, and native software libraries.

3.1.21 **resource** [ITU-T J.200]: A network data object or a service that is uniquely identified in a network. An application resource or environment resource.

3.1.22 **service information (SI)** [ITU-T J.200]: Data which describes programs and services.

3.1.23 **transport stream** (TS) [ITU-T H.222.0]: The MPEG-2 transport stream syntax for the packetization and multiplexing of video, audio, and data signals for digital broadcast systems.

3.1.24 **uniform resource identifier (URI)** [ITU-T J.200]: An addressing method to access a resource in local storage or on the Internet.

3.2 Terms defined in this Recommendation

This Recommendation defines the following terms:

3.2.1 application life-cycle: Time period from the moment an application is loaded until the moment it is destroyed.

3.2.2 author: Person who writes NCL documents.

3.2.3 authoring tool: Tool to help authors create NCL documents.

3.2.4 attribute: Parameter that represents the character of a property.

3.2.5 declarative object content (or declarative media object content): Type of content that takes the form of a code written in some declarative language.

NOTE – An XHTML-based document, an MHEG application and an embedded NCL application are examples of declarative media objects.

3.2.6 element attribute (or attribute of an element): Property of an XML element.

3.2.7 event: Occurrence in time that may be instantaneous or have measurable duration.

3.2.8 hybrid application: A hybrid declarative application or a hybrid imperative application.

3.2.9 hybrid declarative application: Declarative application that makes use of imperative object content.

NOTE – An NCL document with an embedded NCLua object is an example of a hybrid declarative application.

3.2.10 **hybrid imperative application**: Imperative application that makes use of declarative content.

NOTE – A Java Xlet that creates and causes the display of an NCL document instance is an example of a hybrid imperative application.

3.2.11 **imperative application**: Application that is started by, and primarily makes use of, imperative information to express its behaviour.

NOTE – A Java program and a Lua program are examples of imperative applications.

3.2.12 **imperative application environment**: Environment that supports the processing of imperative applications.

3.2.13 imperative object content (or imperative media object content): Type of content that takes the form of an executable code written in some non-declarative language.

NOTE – A Lua script is an example of imperative object content.

3.2.14 media object (or media node): Collection of named pieces of data that may represent a media content, a multimedia content, or a program written in a specific language. Besides a media content, a media object contains a set of properties concerning its media content, like those specifying the position and size of the media content presentation, etc.

3.2.15 media player: Component of an application environment which decodes or executes a specific content type.

3.2.16 native application: An intrinsic function implemented by a receiver platform.

NOTE – A closed captioning display is an example of a native application.

3.2.17 NCL application: Set of information that consists of an NCL document (the application specification) and a group of data, including objects (media objects) accompanying the NCL document.

3.2.18 NCL document (or NCL content): An NCL application specification; an NCL code chunk.

3.2.19 NCL formatter: Software component that is in charge of receiving the specification of an NCL document and controlling its presentation, trying to guarantee that author-specified relationships among media objects are respected.

NOTE – NCL document renderer, NCL user agent, and NCL player are other names used with the same meaning of NCL formatter.

3.2.20 NCL node (or NCL Object): Refers to a <media>, <context>, <body>, or <switch> element of NCL.

3.2.21 NCL user agent: Any program that interprets an NCL document written in the document language according to the terms of this specification.

NOTE – A user agent may display a document, trying to guarantee that author-specified relationships among media objects are respected. The relation can be: read it aloud; cause it to be printed; convert it to another format, etc.

3.2.22 profile: Specification for a class of capabilities providing different levels of functionality in a receiver.

3.2.23 property element: NCL element that defines a property name and its associated value.

3.2.24 scripting language: Language used to describe an imperative object content that is embedded in other host language. For example, Lua is a scripting language for NCL documents as ECMAScript is for HTML documents.

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms:

ABNT	Brazilian Association for Technical Standards (Associação Brasileira de Normas Técnicas)
DTV	Digital Television
GIF	Graphics Interchange Format
HTML	HyperText Markup Language

HTTP	HyperText Transfer Protocol
ISDB-T_B	International Standard for Digital Broadcasting-Terrestrial TV with Brazilian Innovations
JPEG	Joint Photographic Experts Group
LIME	Lightweight Interactive Multimedia Environment
MIME	Multipurpose Internet Mail Extension
MNG	Multiple Network Graphics
MPEG	Moving Picture Experts Group
NCL	Nested Context Language
NCM	Nested Context Model
NPT	Normal Play Time
PES	Packetized Elementary Stream
PID	Packet Identifier
SMIL	Synchronized Multimedia Integration Language
TS	Transport Stream
URI	Universal Resource Identifier
URL	Universal Resource Locator
W3C	World-Wide Web Consortium
XHTML	eXtensible HyperText Markup Language
XML	eXtensible Markup Language

5 NCL and Ginga-NCL

Nested Context Language (NCL) is an XML application that allows authors to write interactive multimedia presentations. Using NCL, authors can describe the temporal behaviour of a multimedia presentation, associate hyperlinks (user interaction) with media objects, define alternatives for presentation (adaptation), and describe the layout of the presentation on multiple devices. NCL also allows for using Editing Commands (see Clause 9) coming from external sources, including those commands for live application generation.

Ginga-NCL is the logical subsystem of the Ginga system that processes NCL declarative applications (NCL documents). A key component of Ginga-NCL is the declarative content decoding engine (NCL formatter or NCL player). Another important module of Ginga is the Lua engine, which is responsible for interpreting NCLua objects, i.e., media objects with Lua code [b-H.IPTV-MAFR.14]. Lua is the scripting language of NCL.

Ginga-NCL deals with applications collected inside data structures known as private bases. A Private Base Manager component is in charge of receiving NCL document, Editing Commands and maintaining the NCL documents being presented. In Ginga-NCL, an application can be generated or modified on the fly, using NCL Editing Commands.

Figure 5-1 illustrates the Ginga-NCL presentation environment. Appendix I presents an overview of the whole Ginga architecture.

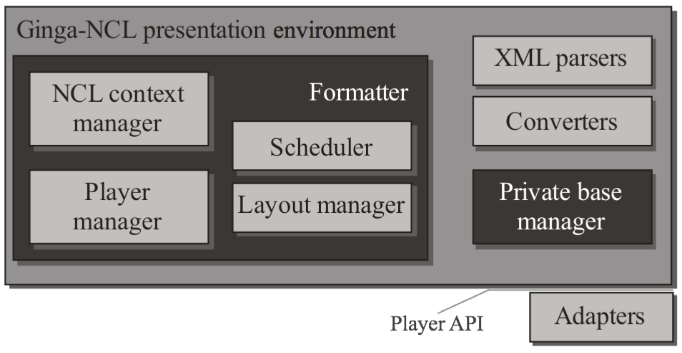

Figure 5-1 – Ginga-NCL presentation environment

6 Ginga-NCL harmonization with other IPTV declarative environments

An NCL application has a strict separation between its content and its structure. NCL itself does not define any media content. Instead, it defines the glue that holds media objects together in multimedia presentations.

An NCL document only defines how media objects are structured and related, in time and space. As a glue language, it does not restrict or prescribe the content types of its media objects. Which media objects are supported depends on the media players that are coupled in the NCL formatter. Among these players are the video and audio decoder/players that actuate in the video plane of a structured screen (see Clause 7.2.6), usually implemented in hardware in an IPTV receiver. In this way, note that video and audio streams of a service presented in the video plane are treated like all other media objects (exhibited in any presentation plane – see Clause 7.2.6) that may be related using NCL.

Another NCL media object that is required in a Ginga-NCL implementation is the HTML-based media object [b-W3C XHTML]. Therefore, NCL does not substitute, but embed HTML-based documents (or objects). As with other media objects, which HTML-based language will have support in an NCL formatter is an implementation choice, and, therefore, it will depend on which HTML browser will act as a media player integrated to the NCL formatter.

As a consequence, it is possible, for example, to have LIME browsers embedded in an NCL document player. It is also possible to receive an HTML-based browser code through datacasting and install it as a plug-in (usually as Lua objects).

It is also possible to have a harmonization browser implemented, and receiving the complementary part, if needed, as a plug-in, in order to convert the HTML player into one of the several IPTV browser standards.

Note that, in the extreme case, an NCL document may be reduced to having only one HTML media object. In this case, the NCL document player will act nearly like an HTML browser.

No matter the case, the HTML-based browser implementation shall be a consequence of the following requirements:

– minimization of the redundancy with existing NCL facilities;

– robustness;

– alignment with W3C specifications;

– rejection of non-conformant content;

– precise content layout control mechanisms;

– support of different pixel aspect ratios.

− support to Ginga-NCL's player API.

Although an HTML-based browser is required to be supported, the use of HTML elements to define relationships (including HTML links) is not recommended when authoring NCL documents. Structure-based authoring should be emphasized for the well-known reasons largely reported in the literature.

When any media player, in particular an HTML-based browser, is integrated to the Ginga-NCL formatter, it shall support the generic API discussed in Clause 8. Therefore, for some HTML-based browsers, an adapter module can be necessary to accomplish the integration.

Finally, for live editing, Ginga-NCL also supports event descriptors and Editing Commands defined by NCL.

Another NCL media object that must be supported by a Ginga-NCL implementation is the declarative NCL media object, i.e., a media object whose content is an NCL application. Therefore, NCL applications can be embedded in NCL parent applications, as well as HTML-based applications.

7 NCL: XML application declarative language for multimedia presentations

The modularization approach has been used in several XML-based language recommendations.

Modules are collections of semantically-related XML elements, attributes, and attribute values that represent a unit of functionality. Modules are defined in coherent sets. This coherence is expressed in that the elements of these modules are associated with the same namespace [b-W3C XMLNAMES1].

A language profile is a combination of modules. Modules are atomic, i.e., they shall not be subdivided when included in a language profile. Furthermore, a module specification may include a set of integration requirements to which language profiles that include the module shall comply.

NCL has been specified in a modular way, allowing for the combination of its modules in language profiles [b-NCL DTV]. Each profile may group a subset of NCL modules, allowing for the creation of languages according to the users' needs. Moreover, NCL modules and profiles can be combined with other language modules, allowing for the incorporation of NCL features into those languages, and vice-versa.

Commonly, there is a language profile that incorporates nearly all the modules associated with a single namespace. Other language profiles can be specified as subsets of the larger one. This is the case of the Enhanced DTV profile and the Raw DTV profile of NCL, focal points of this Recommendation.

The main purpose of being in conformance with a language profile is to enhance interoperability. The mandatory modules are defined in such a way that any document interchanged in a conforming language profile will yield a reasonable presentation. The document formatter, while supporting the associated mandatory module set, shall ignore all other (unknown) elements and attributes.

NCL edition 3.1 is partitioned into 14 functional areas, which are further partitioned into modules:

1) Structure:
   – Structure Module.
2) Components:
   – Media Module.

- Context Module.

3) Interfaces:
   - MediaContentAnchor Module.
   - CompositeNodeInterface Module.
   - PropertyAnchor Module.
   - SwitchInterface Module.
4) Layout:
   - Layout Module.
5) Presentation Specification:
   - Descriptor Module.
6) Timing:
   - Timing Module.
7) Transition Effects:
   - TransitionBase Module.
   - Transition Module.
8) Navigational Key:
   - KeyNavigation Module.
9) Presentation Control:
   - TestRule Module.
   - TestRuleUse Module.
   - ContentControl Module.
   - DescriptorControl Module.
10) Linking:
    - Linking Module.
11) Connectors:
    - ConnectorCommonPart Module.
    - ConnectorAssessmentExpression Module.
    - ConnectorCausalExpression Module.
    - CausalConnector Module.
    - CausalConnectorFunctionality Module.
    - ConnectorBase Module.
12) Animation:
    - Animation Module.
13) Reuse:
    - Import Module.
    - EntityReuse Module.
    - ExtendedEntityReuse Module.
14) Meta-Information:
    - Metainformation Module.

7.1 Identifiers for NCL 3.1 module and language profiles

Each NCL profile should explicitly state the namespace URI that is to be used to identify it.

Documents authored in language profiles that include the NCL Structure module can be associated with the "application/x-ncl+xml" mime type. Documents using the "application/x-ncl+xml" mime type are required to be host language conformant.

The XML namespace identifiers for the complete set of NCL 3.1 modules, elements and attributes are contained within the following namespace: http://www.ncl.org.br/NCL3.1/.

Each NCL module has a unique identifier. The identifiers for NCL 3.1 modules shall comply with Table 7-1.

Modules may also be identified collectively. NCL 3.1 defines the following module collections are defined:

– modules used by the NCL 3.1 Language profile:
  http://www.ncl.org.br/NCL3.1/LanguageProfile
– modules used by the NCL 3.1 Enhanced DTV profile:
  http://www.ncl.org.br/NCL3.1/EDTVProfile
– modules used by the NCL 3.1 Raw DTV profile:
  http://www.ncl.org.br/NCL3.1/RawDTVProfile

Table 7-1 – The NCL 3.1 module identifiers

Modules	Identifiers
Animation	http://www.ncl.org.br/NCL3.1/Animation
CompositeNodeInterface	http://www.ncl.org.br/NCL3.1/CompositeNodeInterface
CausalConnector	http://www.ncl.org.br/NCL3.1/CausalConnector
CausalConnectorFunctionality	http://www.ncl.org.br/NCL3.1/CausalConnectorFunctionality
ConnectorCausalExpression	http://www.ncl.org.br/NCL3.1/ConnectorCausalExpression
ConnectorAssessmentExpression	http://www.ncl.org.br/NCL3.1/ConnectorAssessmentExpression
ConnectorBase	http://www.ncl.org.br/NCL3.1/ConnectorBase
ConnectorCommonPart	http://www.ncl.org.br/NCL3.1/ConnectorCommonPart
ContentControl	http://www.ncl.org.br/NCL3.1/ContentControl
Context	http://www.ncl.org.br/NCL3.1/Context
Descriptor	http://www.ncl.org.br/NCL3.1/Descriptor
DescriptorControl	http://www.ncl.org.br/NCL3.1/DescriptorControl
EntityReuse	http://www.ncl.org.br/NCL3.1/EntityReuse
ExtendedEntityReuse	http://www.ncl.org.br/NCL3.1/ExtendedEntityReuse
Import	http://www.ncl.org.br/NCL3.1/Import
Layout	http://www.ncl.org.br/NCL3.1/Layout
Linking	http://www.ncl.org.br/NCL3.1/Linking
Media	http://www.ncl.org.br/NCL3.1/Media
MediaContentAnchor	http://www.ncl.org.br/NCL3.1/MediaContentAnchor

Table 7-1 – The NCL 3.1 module identifiers

Modules	Identifiers
KeyNavigation	http://www.ncl.org.br/NCL3.1/KeyNavigation
PropertyAnchor	http://www.ncl.org.br/NCL3.1/PropertyAnchor
Structure	http://www.ncl.org.br/NCL3.1/Structure
SwitchInterface	http://www.ncl.org.br/NCL3.1/SwitchInterface
TestRule	http://www.ncl.org.br/NCL3.1/TestRule
TestRuleUse	http://www.ncl.org.br/NCL3.1/TestRuleUse
Timing	http://www.ncl.org.br/NCL3.1/Timing
TransitionBase	http://www.ncl.org.br/NCL3.1/TransitionBase
Transition	http://www.ncl.org.br/NCL3.1/Transition
Metainformation	http://www.ncl.org.br/NCL3.1/MetaInformation

Three SMIL modules [b-W3C SMIL 2.1] were used as the basis for the NCL Transition module and the NCL Metainformation module definitions.

7.1.1 NCL version information

The following processing instructions shall be written in an NCL document. They identify NCL documents that contain only the elements defined in this Recommendation, and the NCL version to which the document conforms.

```
<?xml version="1.0" encoding="UTF-8"?>
<ncl id="any string" xmlns="http://www.ncl.org.br/NCL3.1/profileName">
```

The *id* attribute of an <ncl> element may receive any string that matches the NCName type definition [Namespaces in XML:1999] as its value. That is, it may receive any string value that begins with a letter or an underscore and that only contains letters, digits, '-', '.' and '_'.

The version number of an NCL document specification consists of a major number and a minor number, separated by a dot. The numbers are represented in base 10 with leading zeros suppressed. The initial standard version number is 3.0.

New NCL versions shall be released in accordance with the following versioning policy. If receivers that conform to older versions can still receive and play a document based on the revised specification, the new version of NCL shall be released with the minor number updated. If receivers that conform to older versions cannot receive or play a document based on the revised specifications in its full functionalities, the major number shall be updated.

A specific version is specified in the URI path http://www.ncl.org.br/NCLx.y/profileName, where the version number "x.y" is written immediately after the "NCL".

The profileName, in the URI path, shall be EDTVProfile or RawDTVProfile.

7.2 NCL modules

7.2.1 General remarks

The main definitions made by the NCL 3.1 modules that are present in the NCL 3.1 Enhanced DTV profile are given in Clauses 7.2.2 to 7.2.15.

The complete definition of these NCL 3.1 modules, using XML Schemas, is presented in Annex A. Any ambiguity found in this text can be clarified by consulting the XML Schemas (see Clause 7.2.2.1).

As stated in the scope of this Recommendation, NCL can be used in other declarative environments besides Ginga-NCL. Constraints coming only from Ginga-NCL are always pointed out in a separate paragraph of the subclauses of Clause 7.2, mentioning the Ginga-NCL specification.

After discussing each module, a table is presented indicating the module elements and their attributes. The value of an attribute may not contain quotation marks ("), as usual in XML attributes' values. For a given profile, attributes and contents (child elements) of an element may be defined in the module itself or in the language profile that groups the modules. Therefore, tables in this clause show attributes and contents that come from NCL Enhanced DTV profile, besides those defined in the NCL modules themselves. Tables in Clause 7.3.2 show the attributes and contents that come from NCL Raw DTV profile, besides those defined in the NCL modules themselves. 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.

Additionally, an NCL application is presented in Appendix II, for example purposes only.

7.2.2 Structure functionality

The Structure functionality has just one module, called Structure, which defines the basic structure of an NCL document.

7.2.2.1 Structure module

The Structure module defines the <ncl> root element, the <head> element and the <body> element, following the terminology adopted by other W3C standards. The <body> element of an NCL document is treated as an NCM context node [b-NCM Core].

In NCM, the conceptual data model of NCL, a node may be a context, a switch, or a media object. All NCM nodes are represented by corresponding NCL elements. Context nodes (see Clause 7.2.3) contain other NCM nodes and links.

Most NCL elements have the *id* attribute. This attribute may receive as its value any string that matches the NCName type definition [Namespaces in XML:1999], i.e., it may receive any string value that begins with a letter or an underscore and that only contains letters, digits, '-', '.' and '_'. The *id* attribute uniquely identifies an element within a document. Its value is an XML identifier.

In particular, the <ncl> element shall define the *id* attribute, and the <body> element may define the *id* attribute.

The *xmlns* attribute of <ncl> declares an XML namespace, i.e., it declares the primary collection of XML-defined constructs used in the document. The attribute value is the URL identifying where the namespace is officially defined. Two values are allowed for the *xmlns* attribute: "http://www.ncl.org.br/NCL3.1/EDTVProfile", for the Enhanced DTV profile, and "http://www.ncl.org.br/NCL3.1/RawDTVProfile", for the Raw DTV profile. An NCL formatter shall know that the schemaLocation for these namespaces is, by default, respectively:

```
http://www.ncl.org.br/NCL3.1/profiles/NCL31EDTV.xsd,
http://www.ncl.org.br/NCL3.1/profiles/NCL31RawDTV.xsd
```

Child elements of <head> and <body> are defined in other NCL modules.

The elements of this module, their child elements, and their attributes shall comply with Table 7-2.

Table 7-2 – Extended Structure module used in the EDTV profile

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

7.2.2.1.1 Exception handling

– Documents with *id* attributes whose values are not strings that match the NCName production [Namespaces in XML] shall be ignored by an implementation in conformance with this Recommendation.

– Documents with *xmlns* attribute different from the three previously mentioned values and that are not in conformance with future versions of this Recommendation shall be ignored by an implementation in conformance with this Recommendation.

– All attributes other than *id* and *xmlns* of the <ncl>element should be ignored by the NCL player.

7.2.3 Components functionality

The Components functionality is partitioned into two modules, called Media and Context.

7.2.3.1 Media module

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

In an implementation in conformance with Ginga-NCL specification, the URIs (uniform resource identifiers) defined in Table 7-3 shall be supported.

Table 7-3 – Allowed URIs

Scheme	Scheme-specific-part	Use
file:	///file_path/#fragment_identifier	Local files
http:	//server_identifier/file_path/#fragment_identifier	Remote files downloaded using the HTTP protocol. It can also refer to streams using HTTP-based streaming protocols like MPEG DASH.
https:	//server_identifier/file_path/#fragment_identifier	Remote files downloaded using the HTTPS protocol
rtsp:	//server_identifier/file_path/#fragment_identifier	Streams using the RTSP protocol
rtp:	//server_identifier/file_path/#fragment_identifier	Streams using the RTP protocol
ncl-mirror:	//media_element_identifier	Content flow identical to the one in presentation by another media element
ts:	//server_identifier/program_number.component_tag	Elementary streams contained in a transport stream

An absolute URI by itself contains all information needed to locate its resource. Relative URIs are also allowed. Relative URIs are incomplete addresses that are applied to a base URI to complete the location. The portions omitted are the URI scheme and server, and potentially, part of the URI path.

The primary benefit of using relative URIs is that documents and directories containing them may be moved or copied to other locations without requiring changing the URI attribute values within the documents. This is especially interesting when transporting documents from the server part (usually broadcasters) to the receivers. Relative URI paths are typically used as a short means of locating media files stored in the same directory as the current NCL document, or in a directory close to it. They often consist of just the filename (optionally with a fragment identifier into the file). They may also have a relative directory path before the filename.

It should be emphasized that 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. Relative URI is also allowed in using “ts” scheme. In this case, *server_identifier* can be omitted, and the source of the tuned “ts” shall be assumed.

NOTE 1 – Media objects with the same *src* value and with the corresponding URI scheme different from "ncl-mirror" have the same content to be presented. Moreover, the content of each object can have its presentation started at different moments in time, depending on the time the media objects were started. In addition, their presentations are completely independent. 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 and at the same moment in time, if both media objects are being presented, independently from their starting time. Being different media objects, their properties may have different values, as, for example, those that define the presentation location.

Media objects with the same *src* values and whose URIs refer to the same elementary media stream being broadcasted (pushed data) shall have the same content presentation and in the same moment in time, if they are being presented. However, as being different media objects, their properties may have different values, as, for example, those defining the presentation location.

For media objects with the *src* attribute whose value identifies the "ts" scheme, the program_number.component_tag values, can be substituted by the following reserved words:

– video: The primary video ES of the tuned or identified services.
– audio: The primary audio ES of the tuned or identified services.
– text: The primary text ES of the tuned or identified services.
– video(i): The ith smaller video ES *component_tag* listed in the PMT of the tuned or identified services.
– audio(i): The ith smaller audio ES *component_tag* listed in the PMT of the tuned or identified services.
– text(i): The ith smaller text ES *component_tag* listed in the PMT of the tuned or identified services.

The allowed values for the *type* attribute shall follow MIME Media Types format (or, more simply, MIME types). A MIME type is a character string that defines the class of media (audio, video, image, text, application) and a media encoding type (such as jpeg, mpeg, etc.). MIME types may be registered or informal. Registered MIME types are controlled by the Internet Assigned Numbers Authority (IANA). Informal MIME types are not registered with IANA, but are defined by common agreement.

A <media> element whose *type* value is prefixed by “application/x-” may be used to specify a declarative hypermedia-object or an imperative media object in an NCL application. In this case, the object's content (located through the *src* attribute) shall be a declarative or a non-declarative code span to be executed, respectively.

In an implementation in conformance with Ginga-NCL specification, two special types are defined: "application/x-ginga-NCL", and "application/x-ginga-NCLua".

NOTE 2 – In an implementation in conformance with Ginga-NCL specification, "application/x-ginga-NCL" and "application/x-ginga-NCLua" special types may also be defined as "application/x-ncl-NCL" and "application/x-ncl-NCLua", respectively. However this is should be avoided since this is deprecated, and will likely be unsupported in future versions of NCL.

The "application/x-ginga-NCL" type shall be applied to <media> elements with NCL code content (therefore, an NCL application can embed another NCL application). The "application/x-ginga-NCLua" type shall be applied to <media> elements with Lua imperative code content (see Clause 10).

NCL media objects embedded in NCL applications and HTML-based media objects embedded in NCL applications shall follow the guidelines established in Clause 8.3 (see also [b-NCL Decl. Obj.]).

NCLua objects embedded in NCL applications shall follow the guidelines established in Clause 8.4.

Two other special types shall be supported by any NCL presentation engine: "application/x-ncl-time", and "application/x-ncl-settings".

NOTE 3 – In an implementation in conformance with Ginga-NCL specification, "application/x-ncl-settings" and "application/x-ncl-time" special types may also be defined as "application/x-ginga-settings" and "application/x-ginga-time", respectively. However this is should be avoided since this is deprecated, and will likely not be supported in future versions of NCL.

The application/x-ncl-time type shall be applied to a special <media> element (there may be only one in an NCL document), whose content is the absolute value of the Universal Time Coordinated (UTC). Note that any continuous <media> element with no source can be used to define a clock relative to the <media> element start time. . In this <media> element, <area> child elements can be defined whose *begin* and *end* attributes delimit a period of time from the start of the <media> element.

The content of a <media> element of application/x-ncl-time type is a string with the following syntax: Year":"Month":"Day":"Hours":"Minutes":"Seconds"."Fraction, where Year is an integer; Month is an integer in the [1,12] interval; Day is an integer in the [1,31] interval; Hours is an integer in the [0,23] interval; Minutes is an integer in the [0,59] interval; Seconds is an integer in the [0,59] interval; Fraction is a positive integer.

The "application/x-ncl-settings" type shall be applied to a special <media> element (there may be only one in an NCL document) whose properties are global variables defined by the document author or reserved environment variables that may be manipulated by the NCL player. Table 7-4 states the already defined variables, their semantics and their possible values. In the table, if the value is specified in italic, it means any value of the type in italic. If the value is a list, its elements must be separated by ',' and after the ',' there can be zero or more space characters.

Table 7-4 – Global variables

Group	Variable	Semantics	Possible values
system – set of variables managed by the receiver system; – they may be read, but they may not have their values changed by an NCL application, a Lua procedure or any other imperative or declarative procedure; – receiver's native applications may change the variables' values; – they shall persist during the receiver life cycle.	system.language	Audio language.	*"ISO 639-1 code | ISO 639-2 code"*
	system.caption	Caption language.	*"ISO 639-1 code | ISO 639-2 code"*
	system.subtitle	Subtitle Language.	"*ISO 639-1 code | ISO 639-2 code*"
	system.returnBitRate(i)	Bit rate of the ith network interface in Kbit/s.	"*positive real*"
	system.screenSize	Device screen size, in (lines, pixels/line), when a class is not defined.	"*positive integer, positive integer*"
	system.screenOrientation	Device screen orientation	"portrait" | "landscape"
	system.screenVideoSize	Resolution set for the device's screen video plane, in (lines, pixels/line), when a class is not defined	"*positive integer, positive integer*"
	system.screenBackgroundSize	Resolution set for the device's screen background plane, in (lines, pixels/line), when a class is not defined	"*positive integer, positive integer*"
	system.screenGraphicSize	Resolution set for the device's screen graphics plane, in (lines, pixels/line), when a class is not defined.	"*positive integer, positive integer*"
	system.audioType	Type of the device audio, when a class is not defined.	"mono" | "stereo" | "5.1"
	system.screenSize(i)	Screen size of the class (i) of devices in (lines, pixels/line).	"*positive integer, positive integer*"
	system.screenGraphicSize(i)	Resolution set for the screen graphics plane of the class (i) of devices, in (lines, pixels/line).	"*positive integer, positive integer*"
	system.audioType(i)	Type of the audio of the class (i) of devices.	"mono" | "stereo" | "5.1"
	system.devNumber(i)	Number of exhibition devices registered in the class (i).	"*positive integer*"
	system.classType(i)	Type of the class (i).	"passive" | "active"
	system.classMin(i)	Minimum number of devices in class(i)	"*positive integer*"

Table 7-4 – Global variables

Group	Variable	Semantics	Possible values
	system.classMax(i)	Maximum number of devices in class(i)	“*positive integer*” | “unbounded”
	system.info(i)	List of class (i)'s media players.	“*string*”
	system.classNumber	Number of classes that have been defined.	“*positive integer*”
	system.CPU	CPU performance in MIPS, regarding its capacity to run applications.	“*positive real*”
	system.memory	Minimum memory space in Mbytes provided to applications.	“*positive integer*”
	system.operatingSystem	Type of the operating system.	“*string*”
	system.luaVersion	Version of the Lua engine supported by the receiver.	“*string*”
	system.luaSupportedEventClasses	List of event classes supported by NCLua, separated by ‘,’	“*string*”
	system.nclVersion	NCL language version.	“*string*"
	system.nclProfiles	Language profiles supported by the receiver, separated by ‘,’	“*string*”
	system.gingaNclVersion	Ginga-NCL environment version.	“*string*”
	system.*	Any variable with the "system." prefix not listed in this table shall be reserved for future use.	
user – set of variables managed by the receiver system; – they may be read, but they may not have their values changed by an NCL application, a Lua procedure or any other	user.age	User age.	“*positive integer*”
	user.location	User location shall be the country code concatenated with the country post code. The country code specification shall follow the ISO 3166-1 alpha 3 format.	“*string*”
	user.genre	User genre.	"m"| "f"
	user.language	User language	*“ISO 639-1 code | ISO 639-2 code”*

Table 7-4 – Global variables

Group	Variable	Semantics	Possible values
imperative or declarative procedure; – receiver's native applications may change the variables' values; – they shall persist during the receiver life cycle.	user.*	Any variable with the "user." prefix not listed in this table shall be reserved for future use.	
si – set of variables managed by the middleware system; – they may be read but they may not have their values changed by an NCL application, a Lua procedure or any other imperative or declarative procedure; – they shall persist at least until the next channel tuning.	si.numberOfServices	Number of services available in the tuned channel for the local country. NOTE – The value of this variable should be obtained from the number of PMT tables specified in the PAT table of the transport stream received from the tuned channel (see [ITU-T H.222.0]). The variable value should take into account only the PMT tables whose field country_code are equal to the value of the user.location variable of the Settings node (media object of "application/x-ncl-settings" type).	"*positive integer*"
	si.channelNumber	Number of the tuned channel.	"*positive integer*"
	si.*	Any variable with the "si." prefix not listed in this table shall follow the rules specified for the group.	

Table 7-4 – Global variables

Group	Variable	Semantics	Possible values
metadata – set of variables managed by the middleware system; – they may be read but they may not have their values changed by an NCL application, a Lua procedure or any other imperative or declarative procedure; – they shall persist at least until the next channel tuning.	metadata.*	Any variable with the "metadata" prefix shall follow the rules specified for the group. Variables in this group shall follow the high-level specification of metadata for IPTV services [ITU-T H.750].	
default – set of variables managed by the receiver system; – they may be read and have their values changed by an NCL application, a Lua procedure or any other imperative or declarative procedure; – receiver's native applications may change the variables' values;	default.focusBorderColor	Default colour applied to the border of an element in focus.	"white" | "black" | "silver" | "gray" | "red" | "maroon" | "fuchsia" | "purple" | "lime" | "green" | "yellow" | "olive" | "blue" | "navy" | "aqua" | "teal"
	default.selBorderColor	Default colour applied to the border of an element in focus when activated.	"white" | "black" | "silver" | "gray" | "red" | "maroon" | "fuchsia" | "purple" | "lime" | "green" | "yellow" | "olive" | "blue" | "navy" | "aqua" | "teal"
	default.focusBorderWidth	Default width (in pixels) applied to the border of an element in focus.	"*integer*"

Table 7-4 – Global variables

Group	Variable	Semantics	Possible values
– they shall persist during all receiver life cycle, however, they shall be set to their initial values when a new channel is tuned.	default.focusBorderTransparency	Default transparency applied to the border of an element in focus.	"*Real value between 0 and 1*" | "*Real value in the range [0,100] ending with the character '%' (e.g., 30%)*" NOTE: "1" or "100%" means full transparency and "0" or "0%" means no transparency.
	default.*	Any variable with the "default." prefix not listed in this table shall be reserved for future use.	
service	service.currentFocus	The focusIndex value of the <media> element on focus.	"*positive integer*"
– set of variables managed by the NCL Player; – they may be read and have their values changed by an NCL application of the same service;	service.currentKeyMaster	Identifier (id) of the <media> element that controls the navigational keys; if the <media> element is not being presented or is not paused, the navigational key control pertains to the NCL Formatter.	"*string*"
– they may be read but they may not have their values changed by a Lua procedure or any other imperative or declarative procedure of the same service; however, variable changes may be done using NCL commands; – they shall persist at least during the service life cycle.	service.*	Any variable with the "service." prefix not listed in this table shall follow the rules specified for the group.	

Table 7-4 – Global variables

Group	Variable	Semantics	Possible values
channel – set of variables managed by the NCL player; – they may be read and have their values changed by an NCL application of the same channel; – they may be read but they may not have their values changed by a Lua procedure or any other imperative or declarative procedure; however, variable changes may be done using NCL commands; – they shall persist at least until the next channel tuning. NOTE – A channel is defined as a set of related services.	channel.keyCapture	Request of alphanumeric keys for NCL applications.	"*string*"
	channel.virtualKeyboard	Request of a virtual keyboard for NCL applications.	"true" | "false"
	channel.keyboardBounds	Virtual keyboard region (left, top, width, height).	"*positive integer, positive integer, positive integer, positive integer*"
	channel.*	Any variable with the "channel." prefix not listed in this table shall follow the rules specified for the group.	

Table 7-4 – Global variables

Group	Variable	Semantics	Possible values
shared – set of variables managed by the NCL Player; – they may be read and have their values changed by an NCL application; – they may be read but they may not have their values changed by a Lua procedure or any other imperative or declarative procedure; however, variable changes may be done using NCL commands; – they shall persist at least during the receiver life cycle.	shared.*	Any variable with the "shared" prefix shall follow the rules specified for the group.	

Table 7-5 shows some possible values of the *type* attribute for the Enhanced DTV profile and the associated file extensions for an implementation in conformance with Ginga-NCL specification. The required types shall be defined for each particular TV system. The *type* attribute is optional and should be used to guide the player's (presentation tool) choice by the formatter. When the *type* attribute is not specified, the formatter shall use the content extension specification in the *src* attribute to make the player's choice.

When there is more than one player for the type supported by the formatter, the *player* property of the <media> element may specify which one will be used for presentation. Otherwise the formatter shall use a default player for that type of media.

Table 7-5 – MIME media types for Ginga-NCL formatters

Media type	File extensions
text/html	htm, html
text/plain	txt
text/css	css
text/xml	xml
image/bmp	bmp
image/png	png
image/mng	mng
image/gif	gif
image/jpeg	jpg, jpeg
audio/basic	wav
audio/mp3	mp3
audio/mp2	mp2
audio/mpeg	mpeg, mpg
audio/mpeg4	mp4, mpg4
video/mpeg	mpeg, mpg
application/x-ginga-NCL	ncl
application/x-ginga-NCLua	lua
application/x-ncl-settings	*no src (source)*
application/x-ncl-time	*no src (source)*

The *instance*, *refer* and *descriptor* attributes of <media> elements are extensions defined in other modules and are discussed in the definition of these modules.

However, it should be stressed that a <media> element of application/x-ginga-NCL type may not have the *instance* and *refer* attributes.

The elements of the Media module, their child elements, and their attributes shall comply with Table 7-6.

Table 7-6 – Extended Media module

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

7.2.3.1.1 Default values

– The system.nclVersion and system.gingaNclVersion properties of the media object of "application/x-ginga-settings" (or "application/x-ncl-settings") type have "3.1"and "1.1" as default values, respectively.

7.2.3.1.2 Exception handling

– If a <media> element does not define the *src* and *type* attributes it shall be considered a media object with a continuous content. In the <media> element, <area> child elements can be defined whose *begin* and *end* attributes delimit a period of time from the start of the <media> element.

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

– 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.

– The "ncl-mirror" scheme shall not refer to a <media> element of application/x-ginga-NCL, and application/x-ginga-NCLua types. If a <media> element whose *src* attribute specifies the "ncl-mirror"scheme and this scheme refers to a <media> element of application/x-ginga-NCL or application/x-ginga-NCLua types, the <media> element shall be ignored.

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

– If the file associated to a media object is updated in an object carrousel, and if the media object is not being presented when the updated version of the file arrives, the new version must replace the previous one. Therefore, when the media object is started after the update is completed, it will load the contents of the latest version. If the media object is started during the update, it must load the contents from the latest version of the file that has completely arrived. That version must be kept throughout the update process, only being replaced at the end. If the file associated to a media object is updated in an object carrousel, and if the media object is already being presented when the updated version of the file arrives, the media object presentation shall not be affected. However, the updated version must be used in future presentations.

7.2.3.2 Context module

The Context module is responsible for the definition of context nodes (context objects) through <context> elements. 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. As usual, the *id* attribute uniquely identifies each <context> element within a document.

The *refer* attribute is an extension defined in the Reuse module (see Clause 7.2.14).

The elements of the Context module, their child elements, and their attributes shall be in agreement with Tables 7.7.

Table 7-7 – Extended Context module

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

7.2.4 Interfaces functionality

The Interfaces functionality allows for the definition of node (media object or composite object) interfaces that will be used in relationships with other node interfaces. This functionality is partitioned into four modules:

– MediaContentAnchor, which allows for content anchor (or area) definitions as interfaces of media nodes (<media> elements);

– PropertyAnchor, which allows for the definition of node properties as interfaces of nodes;

– CompositeNodeInterface, which allows for port definitions as interfaces of composite nodes (<context> and <switch> elements); and

– SwitchInterface, which allows for the definition of special interfaces for <switch> elements.

7.2.4.1 MediaContentAnchor module

The MediaContentAnchor module defines the <area> element, which allows for the definition of content anchors representing:

– spatial portions, through the *coords* attribute (as in XHTML);

– temporal portions relative to the beginning time of the content presentation, through *begin* and *end* attributes;

– temporal and spatial portions through *coords*, *begin* and *end* attributes;

– textual chunks, through the *beginText,beginPosition*, *endText and endPosition* attributes that define the string and the string's occurrence in the text, respectively;

– temporal portions based on the number of audio samples or video frames, through *first* and *last* attributes, which shall indicate the initial and final sample/frame;

– temporal portions, through *first* and *last* attributes based on Normal Play Time(NPT) values. When values of the *first* and *last* attributes of an <area> element are specified in NPT, they refer to the temporal base specified in the *contenId* attribute of the <media> element that contains the <area> element.

– Moreover, the <area> element also allows for defining a content anchor based on:

– the *label* attribute, which specifies a string that should be used by a media player to identify a content region;

– the *clip* attribute, which specifies a triple value that shall be used by a media player to identify a clip in the content of a declarative hypermedia object.

Except for the <media> element of the application/x-ncl-time type, the *begin* and *end* attributes shall be specified according with one of the following syntax:

i) Hours“:”Minutes“:”Seconds“.”Fraction, Hours is an integer in the [0,23] interval; Minutes is an integer in the [0,59] interval; Seconds is an integer in the [0,59] interval; Fraction is a positive integer; or

ii) Seconds“s”, where Seconds is a positive real number.

For the <media> element of the “application/x-ncl-time” type, the *begin* and *end* attributes shall be specified according with the following syntax:

Year“:”Month“:”Day“:”Hours“:”Minutes“:”Seconds“.”Fraction (according to the country time zone).

The NCL user agent is responsible for translating the value for the country time zone to the UTC time.

The first and last attributes shall be specified according to one of the following syntaxes:

a) Samples"s", where Samples is a positive integer;

b) Frames"f", where Frames is a positive integer;

c) NPT"npt", where NPT is the Normal Play Time value.

When values of the first and last attributes of an <area> element are specified in NPT, they refer to the temporal base specified in the *contentId* property of the <media> element that contains the <area> element.

For media objects of text type, the *beginText* and *beginPosition* attributes specifies the beginning of the text anchor. The text anchor ending may be specified using the *endTex* and *endPosition* attributes, which also define a string and the string’s occurrence in the text, respectively.

EXAMPLE Assume the text content: “AAA AA AA AAA”; and the attributes *beginText*=“AA”.

For *beginPosition*=1, the anchor begins in the underlined bold string: **<u>AA</u>**A AA AA AAA

For *beginPosition*=2, the anchor begins in the underlined bold string: A**<u>AA</u>** AA AA AAA

For *beginPosition*=3, the anchor begins in the underlined bold string: AAA **<u>AA</u>** AA AAA

For media objects of "application/x-ginga-NCL" type, the *clip* and *label* attribute values may be defined, and shall follow the guidelines established for any declarative hypermedia objects in NCL, as follows.

A declarative hypermedia object (a <media> element of a declarative *type* with prefix “application/x-”) is handled by the NCL parent application as a set of temporal chains. A temporal chain corresponds to a sequence of presentation events (occurrences in time, see definition of *events* in Clause 7.2.12), initiated from the event that corresponds to the beginning of the declarative hypermedia object presentation. Sections in these chains may be associated with declarative hypermedia object's <area> child elements using the *clip* attribute. The *clip* value is a triple of the form "(chainId, beginOffset, endOffset)". The *chainId* parameter identifies one of the chains defined by the declarative hypermedia object. The *beginOffset* and *endOffset* parameters have the same sintax defined for the *begin* and *end* attributes, and define the begin time and the end time of the content anchor, with regards the beginning time of the chain. When a declarative hypermedia object defines just one temporal chain, the *chainId* parameter may be omitted (and also the first comma in the list). The *beginOffset* and *endOffset* may also be omitted when they assume their default values: 0s or the chain end time, respectively (e.g., "(chainId,,50s)"..

For a declarative hypermedia object with NCL code (i.e. <media> element of "application/x-ginga-NCL" type), a temporal chain is identified by one of the NCL document entry points, defined by <port> elements (see clause 7.2.4.3),, children of the document's <body> element.

A declarative hypermedia object's content anchor can also refer to any content anchor defined inside the declarative code itself. In this case, the *label* attribute of the <area> element that defines the content anchor has a value such that the declarative hypermedia object player is able to identify one of its internally defined content anchors. For a declarative hypermedia object with NCL code (i.e. <media> element of "application/x-ginga-NCL" type), one of its <area> elements may refer to a <port> element, child of its <body> element, through its *label* attribute (that must have the <port> element's *id* as its value). In its turn, the <port> element may be mapped to an <area> element defined in any object nested in the declarative NCL hypermedia object. Note that a declarative hypermedia object can externalize content anchors defined inside its content to be used in links defined by the parent NCL object in which the declarative hypermedia object is included.

In a media object of "application/x-ginga-NCLua" type, an imperative-code span may be associated with an <area> element using the *label* attribute. In this case, the *label* value shall identify the code span. An <area> element may also be used just as an interface to be used as conditions of NCL links (set by Lua code) to trigger actions on other objects.

As usual, <area> elements shall have the *id* attribute, which uniquely identifies the element within a document.

In NCM, every node (media or context node) shall 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 documents. Except for media objects with imperative code content (e.g. <media> element of "application/x-ginga-NCLua" type), every time an NCL component is referred without specifying one of its anchors, the *whole content anchor* is assumed.

The <area> element and its attributes shall comply with Table 7-8.

Table 7-8 – Extended MediaContentAnchor module

Elements	Attributes	Content
area	*id, coords, begin, end, beginText, beginPosition, endText, endPosition, first, last, label, clip*	Empty

7.2.4.1.1 Default values

– If the *begin* attribute 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 behaviour is expected from the *first* and *last* attributes.

– In textual content anchors, if the end of the anchor region is not defined, the end of the text content shall be assumed. If the beginning of the content anchor region is not defined, the beginning of the text content shall be assumed.

– When a declarative hypermedia-object defines just one temporal chain, the *chainId* parameter may be omitted. The *begingOffset* and *endOffset* may also be omitted when they assume their default values: 0s and the chain end time, respectively.

– Except for media objects with imperative code content (<media element of "application/x-ginga-NCLua" type, for example), every time an NCL component is referred without specifying one of its anchors, the *whole content anchor* is assumed.

7.2.4.2 PropertyAnchor module

The PropertyAnchor module defines an element named <property>, which may be used for defining a node (media object, context or switch) property or a group of node properties as one of its interfaces (anchors). The <property> element defines the *name* attribute, which indicates the name of the property or property group, and the *value* attribute, an optional attribute that defines an initial value for the *name* property. The parent element shall not have <property> elements with the same *name* attribute values.

It is possible to have NCL document players (formatters) that define some node properties as node interfaces, implicitly. However, in general, it is a good practice to explicitly declare the interfaces.

A <media> element may have several embedded properties. Examples of these properties can be found among those that define the media object placement during a presentation, the presentation duration, and others that define additional presentation characteristics: device (defining the device class in which the presentation will take place), top, left, bottom, right, width, height, zIndex, plane or plan (defining in which plane of a structured screen an object will be placed), explicitDur, background (specifying the background colour used to fill the area of a region displaying a media that is not filled by the media itself), transparency (indicating the degree of transparency of an object presentation), rgbChromakey (defining the RGB colour to be set as transparent), visible (allowing the object presentation to be seen or hidden), fit (indicating how an object will be presented), scroll (which allows for the specification of how an author would like to configure the scroll in a region), style (which refers to a style sheet [b-W3C CSS2] with information for text presentation, for example), soundLevel, balanceLevel, trebleLevel, bassLevel, fontColor, fontFamily, fontStyle, fontSize, fontVariant, fontWeight, player, reusePlayer (which determines if a new player shall be instantiated or if a player already instantiated shall be used), playerLife (which specifies what will happen to the player instance at the end of the presentation), moveLeft, moveRight, moveUp, moveDown, focusIndex, focusBorderColor, focusBorderWidth, focusBorderTransparency, focusSrc, focusSelSrc, selBorderColor, transIn, transOut, freeze, etc. Application authors can also define other own properties associated to <media> elements.

All properties may assume as their initial values those defined in homonym attributes of their node-associated descriptor and region (see Clauses 7.2.6 and 7.2.5). However, it should be remarked that <descriptor>, <descriptorParam>, and <region> elements (see Clauses 7.2.5 and 7.2.6) are only additional options for defining initial values for properties. Every property (and their initial values) defined by those elements may be defined using only <property> elements. Moreover, when the value of a property is specified in a <property> element, it has precedence over the value defined in homonym attributes of its node's associated descriptor or region.

Every <property> element has a Boolean attribute called *externable* that shall be set to "true" by default. When the property is intended to be used in a relationship, it shall be explicitly declared as a <property> (interface) element and with the *externable* attribute equal to "true".

NOTE - If a property is defined in a <descriptor> or <region> element, its *externable* attribute shall be set to “false” by default.

Neither the *id* property (attribute) of a <media>, <context> or <switch> element nor the *src* property (attribute) of a <media> element shall have the *externable* associated attribute equal to “true”. In other words, they cannot be changed by an action of a <link> element.

If the left, right, top, bottom, width or height properties defined in <property> elements have values in percentages (%), the percentages refer to the screen size of the device where the media object will be exhibited.

For soundLevel, trebleLevel and bassLevel audio properties, their values must be interpreted relative to the recorded volume of the media. A setting of “0%” mutes the media. A value of ‘100%′ will play the media at its recorded volume (i.e., 0 dB).

The plane property defines in which plane of a structured screen an object will be placed. The plan property has the same meaning but must be avoided, since it is deprecated, and it will likely be unsupported in future versions of NCL. The number of planes depends on the DTV system. Usually we can have at least three planes: background, video and graphics. The graphics plane superposes the video plane that superposes the background plane. The *zIndex* property gives the superposition order in just one plane, i.e., the plane superposition order has precedence over zIndex values.

If a video stream of a tuned service, which is not referred by any *src* attribute of <media> elements, is being presented on the video plane, the first started media object referring to this stream gets control of this content presentation, i.e., no new content presentation is started. Any other further <media> element that refers to this content by using the *src* attribute, when started, begins a new presentation.

If there is no media object being presented on the video plane referring (through its *src* attribute) to a video stream of a tuned service (no matter in which application of the private base that represents this TV channel), the video streams that were previously being presented in this plane when there were no application running shall be presented, with the same previous video parameters, although not being referred by any media object in exhibition.

If an audio stream of a tuned service, which is not referred by any *src* attribute of <media> elements, is being presented, the first started media object referring to this stream gets control of this content presentation, i.e., no new content presentation is started. Any other further <media> element that refers to this content by using the *src* attribute, when started, begins a new presentation.

If there is no media object being presented referring (through its *src* attribute) to an audio stream of a tuned service (no matter in which application of the private base that represents this TV channel), the audio streams that were previously being presented when there were no application running shall be presented, with the same previous audio parameters, although not being referred by any media object in exhibition.

The *visible* property may also be associated with a <context> or <body> element. In these cases, when the property’s value is equal to “true”, the *visible* property of each child element of the composition shall be taken into account. When the property’s value is equal to “false”, all child elements of the composition shall be hidden, i.e., although the values of their *visible* properties do not change, the child elements are not visible. In particular, when a document has its <body> element with its *visible* property set to “false” and its presentation event in the *paused* state (see definition of *events* in Clause 7.2.12), the document is said to be in standby. When all applications in presentation are in standby, the video streams that were previously being presented in the video plane, when there was no application running, shall be presented, with the same previous video parameters; similarly, the audio streams that were previously being presented when there were no application running shall be presented, with the same previous audio parameters. When any application returns from its standby state, it regains the whole control of these audio and video streams, and the presentation resumes with the same media content presentation as before when the last application has entered in the standby state.

An object with a *visible* property equal to “false”, i.e., a hidden object, may not transit selection event machines defined by its content anchors to the “occurring” state (see Clause 7.2.12) while the *visible* property value persists as “false”.

Some properties have their values defined by the middleware system, as for example, the *contentId* property (associated to a continuous-media object whose content is defined referring to an elementary stream), which has the "null" string as its initial value, and is set to the identifier value transported in the NPT reference descriptor (in a field of the same name: contentId), as soon as the associated continuous-media object is started. Another example is the standby property that 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 5 – The standby property may be set to "true" when the identifier value transported in the NPT reference descriptor (in a field of the same name: contentId) signalized as non-paused is different from the *contentId* property value.

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

A group of node properties may also be explicitly declared as a single <property> (interface) element, allowing authors to specify the value of several properties within a single property. The following groups shall be recognized by an NCL formatter: *location*, grouping (left, top), in this order; *size*, grouping (width, height), in this order; and *bounds*, grouping (left, top, width, height), in this order. When a formatter treats a change in a property group, it shall only test the process consistency at its end.

The words device, top, left, bottom, right, width, height, zIndex, plane, plan, explicitDur, background, transparency, rgbChromakey, visible, fit, scroll, style, soundLevel, balanceLevel, trebleLevel, bassLevel, fontColor, fontFamily, fontStyle, fontSize, fontVariant, fontWeight, player, reusePlayer, playerLife, moveLeft, moveRight, moveUp, moveDown, focusIndex, focusBorderColor, focusBorderWidth, focusBorderTransparency, focusSrc, focusSelSrc, selBorderColor, transIn, transOut, freeze, location, size and bounds are reserved words for values of the *name* attribute of the <property> element. The possible values for these reserved property names shall comply with Table 7-9. Implementation in agreement with this Recommendation shall be able to handle all these properties.

Table 7-9 – Reserved parameter/attribute and possible values

Parameter/attribute name	Value	Default
top, left, bottom, right, width, height	Type: layout A real number in the range [0,100] ending with the character "%" (e.g., 30%), or an integer value specifying the attribute in pixels (a positive integer, in the case of width and height), ending optionally	If any of these properties is not defined and cannot be inferred from the NCL rules, they shall assume value "0"

Table 7-9 – Reserved parameter/attribute and possible values

Parameter/attribute name	**Value**	**Default**
	with “px” string (e.g. 30px).	
location	Type: position Two numbers separated by a comma, each one following the value rule specified for left and top parameters, respectively. There may be zero or more space characters after the comma that shall de ignored.	See first row
size	Type: size Two values separated by a comma. Each value shall follow the same rule specified for width and height parameters, respectively. There may be zero or more space characters after the comma that shall de ignored.	See first row
bounds	Type: sizePosition Four values separated by commas. Each value shall follow the same rule specified for left, top, width and height parameters, respectively. There may be zero or more space characters after the comma that shall de ignored.	See first row
plane	Type: plane "background", "video" or "graphic", following the plane definition of the DTV system.	"video", for media with *src* attribute referring to a TS's PES, "graphics", for all other cases.
baseDeviceRegion	Type: sizePosition. Four values separated by comma. Each value shall follow the same rule specified for left, top, width and height parameters, respectively. There can be zero or more space characters after the comma that shall de ignored.	See first row
device	Type: device. “systemScreen (i)” or “systemAudio(i)” value, where *i* is an integer greater than zero.	systemScreen (0)
explicitDur	Type: time i) Hours":"Minutes":"Seconds"."Fraction, where Hours is an integer in the [0,23] interval; Minutes is an integer in the [0,59] interval; Seconds is an integer in the [0,59] interval; and Fraction is a positive integer. ii) Seconds"s", where Seconds is a positive real number. iii) The null string.	For continuous media, the default value shall be set to the natural content presentation duration, otherwise it must be set to the “null” string

Table 7-9 – Reserved parameter/attribute and possible values

Parameter/attribute name	Value	Default
background	Type: color Reserved colour names: "white", "black", "silver", "gray", "red", "maroon", "fuchsia", "purple", "lime", "green", "yellow", "olive", "blue", "navy", "aqua", or "teal". The background value may also be the reserved value "transparent". This can be helpful to present transparent images, like transparent GIFs, superposed on other images or videos.	transparent
visible	Type: Boolean "true" or "false".	true
transparency	Type: percent A real number in the range [0,1] or a real number in the range [0,100] and ending with the character "%" (e.g., 30%), specifying the degree of transparency of an object presentation ("1" or "100%" means full transparency and "0" or "0%" means opaque).	0
rgbChromakey	Type: RGB888 An RGB 888 value, or the “null” string.	The “null” string
fit	Type: fit "fill", "hidden", "meet", "meetBest", or "slice" value. "fill": scale the object's media content so that it touches all edges of the box defined by the object's width and height attributes. "hidden": if the intrinsic height (width) of the media content is smaller than the height (width) attribute, the object shall be rendered starting from the top (left) edge and have the remaining height (width) filled up with the background colour; if the intrinsic height (width) of the media content is greater than the height (width) attribute, the object shall be rendered starting from the top (left) edge until the height (width) defined in the attribute is reached, and have the part of the media content below (to the right of) the height (width) clipped. "meet": scale the visual media object while preserving its aspect ratio until its height or width is equal to the value specified by the height or width attributes. The media content left-top corner is positioned at the top-left coordinates of the box; the empty space at the right or the bottom shall be filled up with the background colour. "meetBest": the semantic is identical to "meet" except that the image is not scaled greater than 100% in either dimension. "slice": scale the visual media content while	fill

Table 7-9 – Reserved parameter/attribute and possible values

Parameter/attribute name	Value	Default
	preserving its aspect ratio until its height or width are equal to the value specified in the height and width attributes and the defined presentation box is completely filled. Some parts of the content may get clipped. Overflow width is clipped from the right of the media object. Overflow height is clipped from the bottom of the media object.	
scroll	Type: scroll "none", "horizontal", "vertical", "both", or "automatic" value.	none
style	Type: style The locator of a stylesheet file, or the “null” string.	the “null” string
soundLevel, trebleLevel, bassLevel	Type: percent A real number in the range [0,1] or a real number in the range [0,100] and ending with the character "%" (e.g., 30%).	1
balanceLevel	Type balance A real number in the range [–1,1].	0
zIndex	Type: unsigned integer An integer number in the range [0,255], where regions with greater *zIndex* values are stacked on top of regions with smaller *zIndex* values.	0
fontColor	Type: color "white", "black", "silver", "gray", "red", "maroon", "fuchsia", "purple", "lime", "green", "yellow", "olive", "blue", "navy", "aqua", or "teal".	white
textAlign	Type: alignment. Sets the horizontal alignment of text: “left”, “right”, “center”, or “justified” value.	left
fontFamily	Type: fontFamily A prioritized list of font family names and/or generic family names.	DTV system dependent
fontStyle	Type: fontStyle Sets the style of the font ("normal" or "italic" value).	normal
fontSize	Type: fontSize The size of a font.	DTV system dependent
fontVariant	Type: fontVariant Displays text in a "small-caps" font or a "normal" font.	normal
fontWeight	Type: fontWeight Sets the weight of a font ("normal" or "bold" value).	normal
player	Type: string. Establishes the player to be used	-

Table 7-9 – Reserved parameter/attribute and possible values

Parameter/attribute name	Value	Default
reusePlayer	Type: Boolean "false", "true" value.	false
playerLife	Type: playerLife "keep" or "close" value.	close
moveLeft, moveRight, moveUp, moveDown, focusIndex	Type: optionalInteger Positive integer or the "null" string.	the "null" string
focusBorderColor;	Type: color "white", "black", "silver", "gray", "red", "maroon", "fuchsia", "purple", "lime", "green", "yellow", "olive", "blue", "navy", "aqua", or "teal".	DTV system dependent: the value defined by the *default.focusBorderColor*
selBorderColor	Type: color "white", "black", "silver", "gray", "red", "maroon", "fuchsia", "purple", "lime", "green", "yellow", "olive", "blue", "navy", "aqua", or "teal".	DTV system dependent: the value defined by the *default.selBorderColor*
focusBorderWidth	Type: unsigned integer An integer value specifying the attribute in pixels.	DTV system dependent: the value defined by the *default.focusBorderWidth*
focusBorderTransparency	Type: percent A real number in the range [0,1] or a real number in the range [0,100] ending with the character "%" (e.g., 30%), specifying the degree of transparency of an object presentation ("1" or "100%" means full transparency and "0" or "0%" means opaque).	DTV system dependent: the value defined by the *default.focusTransparency*
focusSrc, focusSelSrc	Type: optionalURI An URI or the "null" string	the "null" string
freeze	Type: Boolean "true" or "false" value.	false
transIn, transOut	Type: optionalTransitionList i) A semicolon-separated list of <transition> element identifiers defined in the <transitionBase> element; or ii) A semicolon-separated list of transitions in which each transition is defined by a list of parameters (type, subtype, dur, startProgress, endProgress, direction, fadeColor, horzRepeat, vertRepeat, borderWidth, borderColor) sepatared by commas. In the lists, there can be zero or more space characters after the commas or the semicolons, which shall be ignored. In a transition list, if a parameter does not exist, it must be declared as the null string.	the "null" string

NOTE. The parameters that define a transition specified in a transIn or transOut list shall follow the syntax and semantics of the corresponding attributes of the <transition> element defined in Clause 7.2.8.2.

Properties that have reserved color string names as values ("white", "black", "silver", "gray", red", "maroon", fuchsia", "purple", "lime", "green", "yellow", "olive", "blue", "navy", "aqua", or "teal") follow the CSS1 color standard, as defined in Table 7.10.

Table 7-10 – Reserved names for colour definition

Name	**Hexadecimal**	**R**	**G**	**B**	**Hue**	**Satur.**	**Light**	**Satur.**	**Value**
White	#FFFFFF	100%	100%	100%	0^o	0%	100%	0%	100%
Silver	#C0C0C0	75%	75%	75%	0^o	0%	75%	0%	75%
Gray	#808080	50%	50%	50%	0^o	0%	50%	0%	50%
Black	#000000	0%	0%	0%	0^o	0%	0%	0%	0%
Red	#FF0000	100%	0%	0%	0^o	100%	50%	100%	100%
Maroon	#800000	50%	0%	0%	0^o	100%	25%	100%	50%
Yellow	#FFFF00	100%	100%	0%	60^o	100%	50%	100%	100%
Olive	#808000	50%	50%	0%	60^o	100%	25%	100%	50%
Lime	#00FF00	0%	100%	0%	120^o	100%	50%	100%	100%
Green	#008000	0%	50%	0%	120^o	100%	25%	100%	50%
Aqua	#00FFFF	0%	100%	100%	180^o	100%	50%	100%	100%
Teal	#008080	0%	50%	50%	180^o	100%	25%	100%	50%
Blue	#0000FF	0%	0%	100%	240^o	100%	50%	100%	100%
Navy	#000080	0%	0%	50%	240^o	100%	25%	100%	50%
Fuchsia	#FF00FF	100%	0%	100%	300^o	100%	50%	100%	100%
Purple	#800080	50%	0%	50%	300^o	100%	25%	100%	50%

The <property> element and its attributes shall be in agreement with Table 7.11.

Table 7-11 – Extended PropertyAnchor module

Elements	**Attributes**	**Content**
property	name, value, externable	Empty

7.2.4.2.1 Other default values

– 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, or else a default value. When the *value* is specified, it has precedence over the value defined in homonym attributes of its node's associated descriptor or region.

– 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 "0".

– If the audio properties *soundLevel*, *trebleLevel* or *bassLevel* are not specified they shall assume “100%” as value.

– The values “systemScreen (1)” and “systemAudio(1)” of the device property are reserved to passive classes, and the values “systemScreen(2)”, “systemScreen(3)”, “systemAudio(2)” and “systemAudio(3)” are reserved to active classes.

7.2.4.2.2 Exception handling

– 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.

– When the left, right, top, bottom, width or height properties exceed the dimension of the exhibition device, only the content portion inside the device dimension shall be exhibited.

– If a <bind> element refers to a <property> element with the *externable* attribute equal to “false”, the <bind> element shall be ignored by the NCL formatter.

– The <body> and <context> elements should ignore <property> child elements whose *name* attribute has as value the words top, left, bottom, right, width, height, zIndex, plane, explicitDur, background, transparency, rgbChromakey, visible, fit, scroll, style, soundLevel, balanceLevel, trebleLevel, bassLevel, fontColor, fontFamily, fontStyle, fontSize, fontVariant, fontWeight, player, reusePlayer, playerLife, moveLeft, moveRight, moveUp, moveDown, focusIndex, focusBorderColor, focusBorderWidth, focusBorderTransparency, focusSrc, focusSelSrc, selBorderColor, transIn, transOut, freeze, location, size and bounds.

– When the user specifies *top*, *bottom* and *height* information for the same <media> element, 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* properties, the *left* and *width* values shall be used to compute a new *right* value.

– When the *left, right, top, bottom, width* or *height* properties exceed the dimension of the exhibition device, only the content portion inside the device dimension shall be exhibited.

– The *reusePlayer* and *playerLife* attributes offer additional support for media-objects’ player management, and can be used or be ignored by a particular middleware implementer. 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.

– If the *name* attribute of a <property> element is “src” or “id”, the <property> element shall be ignored. These properties can be defined only by attributes of the <media> element. For these properties the *externable* attribute shall always have the “false” value.

7.2.4.3 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).

The <port> element and its attributes shall comply with Table 7-12.

Table 7-12 – Extended CompositeNodeInterface module

Elements	Attributes	Content
port	<u>*id*</u>, <u>*component*</u>, *interface*	Empty

7.2.4.4 SwitchInterface module

The SwitchInterface module allows for the creation of <switch> element interfaces (see Clause 7.2.10), which may be mapped to a set of alternative interfaces of internal nodes, allowing a link to anchor on the chosen interface when the <switch> is processed (see [b-NCM Core]). 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 components (specified by its *component* attribute).

Every element representing an object interface (<area>, <port>, <property>, or <switchPort>) shall have an identifier (*id* attribute or *name* attribute).

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 mapping elements to each child object of the switch and referring to its *whole content anchor*.

A <switchPort> element may define a mapping to a subset of the switch's components. When a link is bound to a <switchPort> element and all the rules bound to mapped components are evaluated as false, the <defaultComponent> element shall be chosen; if the <defaultComponent> element is not defined no component shall be selected for presentation.

The <switchPort> element, its child elements, and its attributes shall comply with Table 7-13.

Table 7-13 – Extended SwitchInterface module

Elements	Attributes	Content
switchPort	<u>*id*</u>	mapping+
mapping	<u>*component*</u>, *interface*	Empty

7.2.5 Layout functionality

The Layout functionality has a single module, called Layout, which specifies elements and attributes that may define how objects will be initially presented inside regions of output devices. Indeed, this module may define initial values for homonym NCL properties defined by <media>, <body>, and <context> elements (see Clause 7.2.3).

7.2.5.1 Layout module

A <regionBase> element, which may be declared in the NCL document <head>, defines a set of <region> elements, each of which may contain another set of nested <region> elements, and so on, recursively.

The <regionBase> element may have the id attribute, and <region> elements shall have the id attribute. As usual, the id attribute uniquely identifies the element within a document and shall follow the NCName production [Namespaces in XML].

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 zero. The chosen class defines global environment variables: system.screenSize(i), system.screenGraphicSize(i), and system.audioType(i), as defined in Table 7-4 (see clause 7.2.3).

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

The <regionBase> element that defines a passive class may also have a region attribute. This attribute is used to identify a <region> element in a <regionBase> associated with an active class where the (parent) device that creates the bit map sent to the passive-class devices is registered. In the specified region the bit map must also be exhibited.

Multiple device support shall follow the guidelines established in [b_NCL Multi. Dev.] "Support to Multiple Exhibition Devices".

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

In an implementation in conformance with Ginga-NCL specification, the first nesting level (implicitly defined by the <regionBase>) shall be interpreted as defining the device area where the presentation would take place; the second nesting level as windows (that is, presentation areas in the screen) of the parent area; and the other levels as regions inside these windows and so on.

A <region> may also define the following attributes: *left*, *right*, *top*, *bottom*, *height*, *width*, and *zIndex*. All these attributes have the usual meaning.

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 positive "percentage" values, or positive 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 at 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 define 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 7-1.

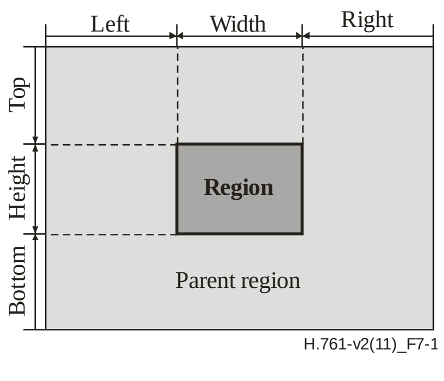

Figure 7-1 – Region positioning attributes

Regarding region sizes, when they may be specified by declaring *width* and *height* attributes using the "%" notation, the size of the region is relative to the size of its parent geometry as mentioned before. Sizes declared as absolute pixel values maintain those absolute values.

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.

The *left*, *right*, *top*, *bottom*, *height*, *width*, and *zIndex* attributes of a <region> element define initial values for the corresponding properties of an object, if these values are not declared in <property> and <descriptorParam> elements. If a property is defined in a <region> element, its *externable* attribute shall be set to “false” by default.

The Layout module also defines the *region* attribute to be used by a <descriptor> element (see Clause 7.2.6) to refer to a Layout <region> element.

The elements of this module, their child elements, and their attributes shall comply with Table 7-14.

Table 7-14 – Extended Layout module

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

7.2.5.1.1 Default values

– When the *device* attribute is not specified, the presentation shall take place in the same device that runs the NCL formatter.

– In a conformant implementation, systemScreen(1) and systemAudio(1) are reserved to passive classes, and systemScreen(2), systemScreen(3), systemAudio(2) and systemAudio(3) are reserved to active classes.

– If the *region* attribute of the <regionBase> element that defines a passive class is not specified the exhibition will take place only on the passive class devices.

– The intrinsic size of a region is equal to the size of the logical parent’s geometry. This means that, if a nested region doesn’t specify any positioning or size values, it will be assumed to have the same position and size values of its parent region. 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.

– When any of the *top*, *bottom*, *height*, *left*, *right*, and *width* <region> attributes is not specified and cannot have its value computed from the other attributes, its value shall be inherited from the corresponding parent absolute value.

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

– All attributes other than those defined in Table 7.14 for the <region> element should be ignored by the NCL player.

7.2.5.1.2 Exception handling

– 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.

– Child regions cannot stay outside 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 not specified).

– Any other attribute than *id, left, right, top, bottom, height, width,* and *zIndex,* shall be ignored by the NCL 3.1 Player.

7.2.6 Presentation Specification functionality

The Presentation Specification functionality has a single module called Descriptor. The purpose of this module is to specify temporal and spatial information needed to present each document component. This information is modelled by *descriptors*.

7.2.6.1 Descriptor module

The Descriptor module allows for the definition of <descriptor> elements, which contain a set of optional attributes, grouping temporal and spatial definitions, which should be used according to the type of object to be presented. The definition of <descriptor> elements shall be included in the head section of the document, inside the <descriptorBase> element, which specifies the set of descriptors of a document. The <descriptor> element shall have the *id* attribute and the <descriptorBase> element may have the *id* attribute, which, as usual, uniquely identifies the elements within a document.

A <descriptor> element may have temporal attributes, namely: *explicitDur* and *freeze*, defined by the Timing module (see Clause 7.2.7); an attribute called *player*; an attribute called *region*, which refers to a region defined by elements of the Layout module (see Clause 7.2.5); and key-navigation attributes, namely: *moveLeft*, *moveRight*, *moveUp*; *moveDown*, *focusIndex*, *focusBorderColor*; *focusBorderWidth*; *focusBorderTransparency*, *focusSrc*, *selBorderColor*, and *focusSelSrc*, defined by the KeyNavigation module (see Clause 7.2.9); and transition attributes, namely: *transIn* and *transOut* (see Clause 7.2.8). All these attributes are used to establish initial values for the corresponding properties of an object, if these values are not declared in <property> and <descriptorParam> elements. If a property is defined in a <descriptor> element and if it is not overwrited by a <property> element, its *externable* attribute shall be set to “false” by default.

A <descriptor> element may also have <descriptorParam> child elements, which are used to parameterize the presentation control of the object associated with the descriptor element. These parameters can, for example, redefine some attribute values defined by the region attributes. They can also define other media object property's initial values, such as values for *plane*; *rgbChromakey*; *background*; *visible*; *fit*; *scroll*; *transparency*; *style*; and also specific values for audio objects, such as values for *soundLevel, balanceLevel, trebleLevel* and *bassLevel properties*. Besides, <descriptorParam> child elements can determine if a new player shall be instantiated or if a player already instantiated shall be used (*reusePlayer*), and specify what will happen to the player instance at the end of the presentation (*playerLife*). Therefore, <descriptorParam> elements can establish initial values for the corresponding properties of an object, if these values are not declared in <property> elements. If a property is defined in a <descriptorParam> element, its *externable* attribute shall be set to “false” by default.

Besides the <descriptor> element, the Descriptor module defines a homonym attribute, which refers to an element of the document descriptor set. When a language profile uses the Descriptor module, it has to determine how *descriptors* will be associated with document components. Following NCM directives, this Recommendation establishes that the *descriptor* attribute is associated with any media node through <media> elements and through link endpoints (<bind> elements) (see Clause 8.2.1).

It should be remarked that the set of descriptors of a document may contain <descriptor> elements or <descriptorSwitch> elements, which allow for specifying alternative descriptors (see Clause 7.2.10).

The elements of the Descriptor module, their child elements, and their attributes shall comply with Table 7-15.

Table 7-15 – Extended Descriptor module

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)+

It must be stressed that <descriptor> and <region> elements are just "syntactic sugar" that promote reuse. All <media> element's properties may be defined using only <property> elements.

If several values are specified for the same property, 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).

7.2.7 Timing functionality

The Timing functionality defines the Timing module. The Timing module allows for the definition of temporal attributes for document components.

7.2.7.1 Timing module

Basically, the Timing module defines attributes for specifying what will happen with an object at the end of its presentation (*freeze*), and the ideal duration of an object (*explicitDur*). These attributes may be incorporated by <descriptor> elements.

When *freeze* is specified with a value equal to "true" the last image map of the object must be frozen indefinitely, i.e., 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 *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.

The *explicitDur* attributes shall be specified according with one of the following syntax:
i) Hours“:”Minutes“:”Seconds“.”Fraction, where Hours is an integer in the [0,23] interval; Minutes is an integer in the [0,59] interval; Seconds is an integer in the [0,59] interval; and Fraction is a positive integer
ii) Seconds”s”, where Seconds is a positive real number.

7.2.7.1.1 Default values

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

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

7.2.8 Transition Effects functionality

The Transition Effects functionality is divided into two modules: TransitionBase and Transition.

7.2.8.1 TransitionBase module

The TransitionBase module defines the <transitionBase> element that specifies a set of transition effects, and shall be defined as a child element of the <head> element.

The <transitionBase> element, its child elements, and its attributes shall comply with Table 7-16.

Table 7-16 – Extended TransitionBase module

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

7.2.8.2 Transition module

The Transition module is based on SMIL 2.1 specification [b-W3C SMIL 2.1]. It has just one element called <transition>.

In NCL 3.1 Enhanced DTV profile, the <transition> element is specified in the <transitionBase> element and allows a transition template to be defined. Each <transition> element defines a single transition template and shall have an *id* attribute so that it may be referred.

The attributes of the <transition> element come from SMIL BasicTransitions module specification: *type*, *subtype*, *dur*, *startProgress*, *endProgress*, *direction* and *fadeColor*.

Transitions are classified according to a two-level taxonomy of types and subtypes. Each of the transition types describes a group of transitions which are closely related. Within that type, each of the individual transitions is assigned a subtype which emphasizes the distinguishing characteristic of that transition.

The *type* attribute is required and it is used to specify the general transition.

The *subtype* attribute provides transition-specific control. This attribute is optional and, if specified, shall be one of the transition subtypes appropriate for the specified type. If this attribute is not specified, then the transition reverts to the default subtype for the specified transition type. Only the subtypes for the five required transition types listed in Table 7-17 shall be supported. The other subtypes defined in SMIL specifications [b-W3C SMIL 2.1] are optional

Table 7-17 – Required transition types and subtypes

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

The *dur* attribute specifies the duration of the transition. The duration is specified as Seconds“s”, where Seconds is a positive real number.

The *startProgress* attribute specifies the amount of progress through the transition at which to begin execution. Legal values are real numbers in the range [0.0,1.0]. For instance, one may want to begin a crossfade with the destination image being already 40% faded in. In this case, *startProgress* would be 0.4.

The *endProgress* attribute specifies the amount of progress through the transition at which to end execution. Legal 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* is equal to *startProgress*, then the transition remains at a fixed progress for the duration of the transition.

The *direction* attribute specifies the direction in which the transition will run. The legal values are "forward" and "reverse". The default value is "forward". 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.

If the value of the *type* attribute is "fade" and the value of the *subtype* attribute is "fadeToColor" or "fadeFromColor" (values that are optional in NCL), then the *fadeColor* attribute specifies the ending or starting colour of the fade.

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, which initialize values of the *transIn* and *transOut* properties. These properties may also be defined using <property> elements. Transitions specified with a *transIn* attribute will begin at the beginning of the media element's active duration (when the object presentation begins to occur). Transitions specified with a *transOut* attribute will end at the end of the media element's active duration (when the object presentation transits from occurring to sleeping state).

The value of the *transIn* and *transOut* attributes is a semicolon-separated list of transition identifiers. Each of the identifiers shall correspond to the value of the XML identifier of one of the transition elements previously defined in the <transitionBase> element. The purpose of the semicolon-separated list is to allow authors to specify a set of fall-back transitions if the preferred transition is not available. The first transition in the list shall be performed if the user-agent has implemented this transition. If this transition is not available, then the second transition in the list shall be performed, and so on.

All transitions defined in the Transition module accept four additional attributes (coming from the SMIL TransitionModifiers module specification) that may be used to control the visual appearance of the transitions. The *horzRepeat* attribute specifies how many times to perform the transition pattern along the horizontal axis. The *vertRepeat* attribute specifies how many times to perform the transition pattern along the vertical axis. The *borderWidth* attribute specifies the width of a generated border along a wipe edge. Legal values are integers greater than or equal to 0. If *borderWidth* value is equal to 0, then no border should be generated along the wipe edge. If the value of the *type* attribute is not "fade", then the *borderColor* attribute specifies the content of the generated border along a wipe edge. If the value of this attribute is a colour, then the generated border along the wipe or warp edge is filled with this colour. If the value of this attribute is "blend", then the generated border along the wipe blend is an additive blend (or blur) of the media sources.

The element of the Extended Transition Module, its child elements, and its attributes shall comply with Table 7-18.

Table 7-18 – Extended Transition module

Elements	Attributes	Content
transition	id, type, subtype, dur, startProgress, endProgress, direction, fadeColor, horzRepeat, vertRepeat, borderWidth, borderColor	Empty

7.2.8.2.1 Default values

– If the subtype attribute is not specified then the transition reverts to the default subtype for the specified transition type, as shown in Table 17.

– The default value for the *dur* attribute is 1s.

– The default value for the *startProgress* attribute is 0.0.

– The default value for that *endProgress* attribute is 1.0.

– The default value for the *direction* attribute is “forward”.

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

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

– The default value for the *horzRepeat* attribute is 1.

– The default value for the *vertRepeat* attribute is 1.

– The default value for the *borderWidth* attribute is 0.

– The default value for the *borderColor* attribute is the colour "black".

7.2.8.2.2 Exception handling

– If the named transition’s type is not supported by the NCL formatter, the transition should be ignored.

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

– The *endProgress* 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 should be ignored. If *endProgress* is equal to *startProgress*, then the transition remains at a fixed progress for the duration of the transition.

– Transitions that do not have a reverse interpretation should have the *direction* attribute ignored.

– 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.

– If any value in the list of the *transIn* attribute or the *transOut* attribute does not correspond to the value of an XML identifier of a transition element defined in the <transitionBase>, then this transition shall be ignored.

7.2.9 Navigational Key functionality

The Navigational Key functionality defines the KeyNavigation module that 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.

7.2.9.1 KeyNavigation module

The *focusIndex* property specifies an index for the <media> element to which the focus may be applied, when this element is in exhibition. The *focusIndex* property may be defined using a <property>, a <descriptorParam>, or a <descriptor> element, in this last case through a *focusIndex* attribute. When this property is not defined, the object is considered as if no focus could be set to it. 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 being presented with the smallest *focusIndex* value. Values of *focusIndex* attribute should be unique in an NCL document. Otherwise, the repeated properties will be ignored if at a certain moment there is more than one <media> element to gain the focus.

Assuming a <media> element on focus, its *moveUp* property specifies a value equal to the *focusIndex* value associated to an element to which the focus shall be applied when the "up arrow key" is pressed. Its *moveDown* property specifies a value equal to the *focusIndex* value associated to an element to which the focus shall be applied when the "down arrow key" is pressed. Its *moveRight* property specifies a value equal to the *focusIndex* value associated to an element to which the focus shall be applied when the "right arrow key" is pressed. Its *moveLeft* property specifies a value equal to the *focusIndex* value associated to an element to which the focus shall be applied when the "left arrow key" is pressed. These properties may be defined using <property>, <descriptorParam> or <descriptor> elements, in this last case through a *moveUp, moveDown, moveLeft, and moveRight* attributes, respectively.

The *focusSrc* property can specify an alternative media source to be presented, instead of the current presentation, if an element receives the focus. This attribute follows the same rules of the *src* attribute of the <media> element.

When an element receives a focus, the square box defined by the element positioning attributes shall be decorated. The *focusBorderColor* property defines the decorative colour.

The *focusBorderColor* property may receive the reserved colour names: "white", "black", "silver", "gray", "red", "maroon", "fuchsia", "purple", "lime", "green", "yellow", "olive", "blue", "navy", "aqua", or "teal".

The *focusBorderWidth* property defines the width in pixels of the decorative border (0 means that no border will appear, positive values mean that the border is outside the object content, and negative values mean that the border is drawn over the object content).

The *focusBorderTransparency* property defines the decorative colour transparency. The *focusBorderTransparency* shall be a real value between 0 and 1; or a real value in the range [0,100], ending with the character "%" (e.g., 30%), with "1" or "100%" meaning full transparency and "0" or "0%" meaning no transparency.

When an element on focus is selected by pressing the activation (select or enter) key, the *focusSelSrc* property can specify an alternative media source to be presented, instead of the current presentation. This property follows the same rules of the *src* attribute of the <media> element. When selected, the square box defined by the element positioning attributes shall be decorated with the colour defined by the *selBorderColor* property, the width of the decorative border defined by the *focusBorderWidth* property, and the decorative colour transparency defined by the *focusBorderTransparency* property.

The *focusSrc*, *focusBorderColor, focusBorderWidth, focusBorderTransparency, focusSelSrc,* and *selBorderColor* may be defined using <property>, <descriptorParam> or <descriptor> elements, in this last case through homonymous attributes, respectively. The *focusSrc* attribute and the *focusSelSrc* attribute can only specify perceptual media content (video, images, text and audio). The original media content behavior does not stop when substituted but continues running hidden (visible="false").

When an element on focus is selected by pressing the "activate (select, enter, etc.) key", if there is a <simpleCondition> element with its *role* attribute equal to "onSelection" without specifying the *key* attribute, this condition is considered satisfied if the element on focus is the one specified by the *component* attribute of the <simpleCondition> element. Therefore, the navigational keys act similarly to a pointer device (like mouse, etc.).

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* attribute of the *settings* node (<media> element of application/x-ncl-settings type). In a multiple device environment, the hierarchical rules for input key control and exhibition device control shall follow the guidelines established in [b_NCL Multi. Dev.] "Support to Multiple Exhibition Devices".

NOTE – When the NCL player begins an NCL application presentation, it shall receive the "CURSOR_DOWN", "CURSOR_LEFT", "CURSOR_RIGHT", "CURSOR_UP", "ENTER" and "BACK" key notifications, as well as notifications coming from pointer devices (for example, from a mouse, selection on a touch screen, etc.). From the moment the object on focus is selected (when the ENTER key is pressed or the pointer device makes the selection), or when the value of the service.currentKeyMaster attribute of the Settings node is changed to a value equal to the *id* of a valid media object, the NCL player shall stop receiving navigational key notifications (and notifications from all other keys it controls), except notifications coming from the "BACK" key selection. All previously controlled keys, including the "BACK" key, when pressed, shall now notify the media player of the media object on focus (however, the notification coming from the selection of the media object shall not be passed to the player of this selected object). Note thus that both the NCL player and the media player can handle "BACK" key notifications, however the NCL player cannot use "BACK" key notifications to trigger condition roles of links during this period of time, but only to control the key rights. When the media object on focus ends its presentation, the NCL player regains the control of the "CURSOR_DOWN", "CURSOR_LEFT", "CURSOR_RIGHT", "CURSOR_UP", "ENTER" keys (as well as the control of all keys that were previously allocated to it, and the control of the pointer devices), and can use "BACK" key notifications to trigger condition roles of links. During the presentation of the media object on focus, its media player may pass the control of the navigational keys (and the pointer devices) to its internal (child) media players, and so on. In this meantime if the "BACK" key is pressed, the navigational key control is passed back to the parent media player (the child media player loses the control) until the last "BACK" pressing, and so on, until the NCL player that has started the process regains the control. From then on, only this NCL player will receive navigational key notifications (and notifications coming from the keys it had previously allocated, and also notifications coming from pointer devices). When a media player receives the navigational key control and does not want this control, it can refuse it passing the control back, as if the "BACK" key had been pressed.

The focus control may also be passed by setting the service.currentFocus property, and the key control by setting the service.currentKeyMaster attribute of the *Settings* node (<media> element of "application/x-ncl-settings" type). This may be done through a link action, through an NCL Editing Command executed by an imperative-code node (for example, an NCLua object).The player of a node that has the current control may not directly change the service.currentKeyMaster property.

7.2.9.1.1 Default values

– When the *focusIndex* property is not defined, it shall be considered as if no focus could be set.

– When the *focusBorderColor*, *focusBorderWidth*, *focusBorderTransparency*, or *selBorderColor* attributes 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: *default.focusBorderColor, default.focusBorderWidth, default.focusTransparency, default.selBorderColor.*

7.2.9.1.2 Exception handling

– 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 that has the smallest index value.

– Values of *focusIndex* attribute should be unique in an NCL document. Repeated attributes shall be ignored if in a certain moment there is more than one <media> element to gain the focus.

– 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.

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

– If the *focusSrc* attribute or the *focusSelSrc* attribute receives an invalid value, they should be ignored and the NCL player must proceed as if this attributes are inexistent.

7.2.10 Presentation Control functionality

The purpose of the Presentation Control functionality is to specify content and presentation alternatives for a document. This functional area is partitioned into four modules: TestRule, TestRuleUse, ContentControl and DescriptorControl.

7.2.10.1 TestRule module

The TestRule module allows for the definition of rules that, when satisfied, select alternatives for document presentation. The specification of rules in NCL 3.1 is done in a separate module, because they are useful for defining either alternative components or alternative descriptors.

The <ruleBase> element specifies a set of rules, and shall be defined as a child element of the <head> element. These rules may be simple, defined by the <rule> element, or composite, defined by the <compositeRule> element.

Simple rules define an optional identifier (*id* attribute), a variable (*var* attribute), a value (*value* attribute), and a comparator (*comparator* attribute) relating the variable to the value. . A <rule> element defined as child of a <compositeRule> element may have its *id* attribute omitted. The variable type and the value type shall be the same, otherwise the rule definition shall be ignored by the NCL formatter. The variable shall be a property of the settings node (<media> element of application/x-ncl-settings type), that is, the *var* attribute shall have the same value of a <property> *name* attribute, defined as a child of the <media> element of application/x-ncl-settings type. The *comparator* attribute shall have one of the values: "eq", "ne", "gt", "lt", "gte", or "lte".

For string variables, comparisons in simple rules are done based on the binary representation of the variable's value and the binary representation of the value used in the comparison.

Composite rules have an identifier (*id* attribute) and a Boolean operator ("and" or "or" – *operator* attribute) relating their child rules. As usual, the *id* attribute uniquely identifies the <rule> and <compositeRule> elements within a document.

The elements of the TestRule module, their attributes, and their child elements shall comply with Table 7-19.

Table 7-19 – Extended TestRule module

Elements	Attributes	Content
ruleBase	id	(importBase|rule|compositeRule)+
rule	*id*, *var*, *comparator*, *value*	Empty
compositeRule	*id*, *operator*	(rule | compositeRule)+

7.2.10.1.1 Exception Handling

– If the value of the *comparator* attribute of the <rule> element is not the values: "eq", "ne", "gt", "lt", "gte", or "lte", the element shall be ignored.

– In <rule> elements, the *comparator* attribute relates the variable to a value. If the variable type and the value type are not equal, the< rule> element shall be ignored.

– In <rule> elements, the *var* attribute shall have the same value of a <property> element's name attribute, defined as a child of the <media> element of application/x-ncl-settings type, otherwise the rule definition shall be ignored.

7.2.10.2 TestRuleUse module

The TestRuleUse defines the <bindRule> element, which is used to associate rules with components of a <switch> or <descriptorSwitch> element, through its *rule* and *constituent* attributes, respectively.

The element of the TestRuleUse module and its attributes shall comply with Table 7-20.

Table 7-20 – Extended TestRuleUse module

Elements	Attributes	Content
bindRule	*constituent*, *rule*	Empty

7.2.10.3 ContentControl module

The ContentControl module specifies the <switch> element, allowing the definition of alternative document nodes to be chosen during presentation time. Test rules used to choose the switch component to be presented are defined by the TestRule module. NCL formatters shall delay the switch evaluation to the moment when a link anchoring in the switch needs to be evaluated. The rules are evaluated in the order they are referred in the <switch> element. During a document presentation, from the moment when 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 (see definition of *events* in Clause 7.2.12). The chosen alternative can be referred through a <switchPort> element mapped to one of its interfaces.

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

In order to allow links to anchor on the component chosen after evaluating the rules of a *switch*, a language profile should also include the SwitchInterface module, which allows for the definition of special interfaces, named <switchPort>.

As usual, <switch> elements shall have the *id* attribute, which uniquely identifies the element within a document. The *refer* attribute is an extension defined in the Reuse module (see Clause 7.2.14).

The ContentControl module elements, their attributes and child elements shall comply with Table 7-21.

Table 7-21 – Extended ContentControl module

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

7.2.10.3.1 Exception handling

– 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.

7.2.10.4 DescriptorControl module

The DescriptorControl module specifies the <descriptorSwitch> element, which contains a set of alternative descriptors to be associated with an object. The <descriptorSwitch> elements shall have the *id* attribute, which uniquely identifies the element within a document. Analogous to the <switch> element, the <descriptorSwitch> choice is done during presentation time, using test rules defined by the TestRule module. The rules are evaluated in the order they are referred in the <descriptorSwitch> element. NCL formatters shall delay the descriptorSwitch evaluation to the moment the object referring the descriptorSwitch needs to be prepared to be presented.

During a document presentation, from the moment on a <descriptorSwitch> is evaluated for a specific <media> element, it is considered resolved for that <media> element until the end of the presentation of this <media> element, i.e., while any presentation event associated with this <media> element is in the "occurring" or "paused" state (see definition of *events* in Clause 7.2.12).

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

The DescriptorControl module elements, their attributes, and their child elements shall comply with Table 7-22.

Table 7-22 – Extended DescriptorControl module

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

7.2.10.4.1 Exception handling

– 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.

7.2.11 Linking functionality

The Linking functionality defines the Linking module, responsible for defining links using connectors.

7.2.11.1 Linking module

A <link> element may have an *id* attribute, which uniquely identifies the element within a document, and shall have an *xconnector* attribute, which refers to a hypermedia connector URI. The reference shall have the format: *alias#connector_id*, or *documentURI_value#connector_id*, for connectors defined in an external document (see Clause 7.2.14); or simply *connector_id*, for connectors defined in the document itself.

The <link> element also contains child elements called <bind> elements, which allow associating nodes with connector roles (see Clause 7.2.12). In order to make this association, a <bind> element has four basic attributes. The first one is called *role*, which is used for referring to a connector role. The second one is called *component*, which is used for identifying the node. The third is an optional attribute called *interface*, used for making reference to the node interface. The fourth is an optional attribute called *descriptor*, used to refer to a descriptor to be associated with the node, as defined by the Descriptor module (see Clause 7.2.6).

NOTE – The *interface* attribute may refer to any node interface, i.e., an anchor, a property, a port (if it is a composite node), or a switchPort (if it is a switch node). The interface attribute is optional. When it is not specified, the association will be done with the whole node content, as explained in Clause 7.2.4, except for media objects with imperative code content, as explained in Clause 8.3.1.

If the connector element defines parameters (see Clause 7.2.12), the <bind> or <link> elements should define parameter values through child elements called <bindParam> and <linkParam>, respectively, both with *name* and *value* attributes. In this case, the *name* attribute shall refer to the name of a connector parameter while the *value* attribute shall define a value to be assigned to the respective parameter.

The elements of the linking module, their attributes, and their child elements shall comply with Table 7-23.

Table 7-23 – Extended Linking module

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

7.2.11.1.1 Exception handling

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

– If a <link> defines the same parameter through using the <linkParam> and <bindParam> elements, the definition by using <bindParam> element has precedence.

– If the number of participants specified by the <link> element for a same condition specified in the referred <causalConnector> is greater than the value of the *max* attribute or is lesser than the value of the *min* attribute of this condition, the <link> shall be ignored.

7.2.12 Connectors functionality

The NCL 3.1 Connectors functionality is partitioned into seven basic modules: ConnectorCommonPart, ConnectorAssessmentExpression, ConnectorCausalExpression, CausalConnector, ConstraintConnector, ConnectorBase, and CompositeConnector.

The Connectors functionality modules are totally independent from the other NCL modules. Besides the basic modules, the Connectors functionality also defines modules that group sets of basic modules, in order to make easier to define a language profile. This is the case of the CausalConnectorFunctionality module, used in the definition of the EDTV and RawDTV profiles. The CausalConnectorFunctionality module groups the following modules: ConnectorCommonPart, ConnectorAssessmentExpression, ConnectorCausalExpression, and CausalConnector.

The <causalConnector> element represents a relation that may be used for creating <link> elements in documents.

A connector specifies a relation independently of derived relationships, i.e., it does not specify which nodes (represented by <media>, <context>, <body>, and <switch> elements) will interact through the relation. A <link> element, in its turn, represents a relationship, of the type defined by its connector, interconnecting different nodes. Links representing the same type of relation, but interconnecting different nodes, may reuse the same connector, reusing all previous specifications. A connector specifies, through its child elements, a set of interface points, called *roles*. A <link> element refers to a connector and defines a set of binds (<bind> child elements of the <link> element), which associate each link endpoint (node interface) to a role of the used connector.

Relations in NCL are based on events. An event is an occurrence in time that may be instantaneous or have a measurable duration. NCL 3.1 defines the following types of events:

– presentation event, which is defined by the presentation of a subset of the information units of a media object, specified in NCL by the <area> element, or by the media node itself (whole content presentation). Presentation events may also be defined on composite nodes (represented by a <body>, <context>, or <switch> element), representing the presentation of the information units of any node inside a composite node;

– selection event, which is defined by the selection of a subset of the information units of a media object (which is specified in NCL by the <area> element, or by the media node's whole content anchor), being presented and visible; and
– attribution event, which is defined by the attribution of a value to a property of a node (represented by a <media>, <body>, <context>, or <switch> element). The property shall be declared in a <property> child element of the node with *externable* attribute equal to "true".

Each event defines a state machine that should be maintained by the NCL formatter (see Figure 7-2).

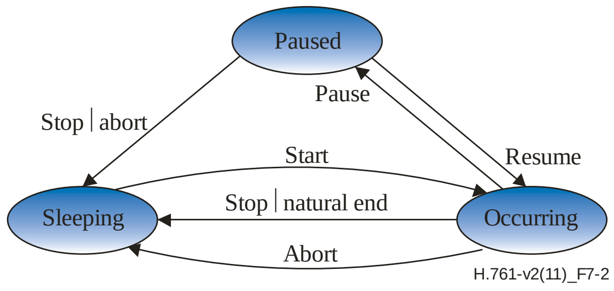

Figure 7-2 – Event state machine

Transition names for the event state machine shall comply with Table 7-24.

Table 7-24 – Transition names for an event state machine

Transition (caused by action)	Transition name
sleeping → occurring (start)	*starts*
occurring → sleeping (stop or natural end)	*stops*
occurring → sleeping (abort)	*aborts*
occurring → paused (pause)	*pauses*
paused → occurring (resume)	*resumes*
paused → sleeping (stop)	*stops*
paused → sleeping (abort)	*aborts*

A presentation event associated with a media node, represented by a <media> element, initializes in the sleeping state. At the beginning of the exhibition of its information units, the event goes to the occurring state. If the exhibition is temporarily suspended, the event stays in the paused state, while this situation lasts. A presentation event may change from occurring to sleeping as a consequence of the natural end of the presentation duration, or due to an action that stops the event. When the presentation of an event is abruptly interrupted, through an abort presentation command, the event also goes to the sleeping state. The duration of an event is the time it remains in the occurring state. This duration may be intrinsic to the media object, explicitly specified by an author (*explicitDur* attribute of a <descriptor> element, *explicitDur* name of a <descriptorParam> element, or *explicitDur* name of a <property> element), or derived from a relationship.

A presentation event associated with a composite node represented by a <body> or a <context> element stays in the occurring state while at least one presentation event associated with anyone of the composite child nodes is in the occurring state, or at least one composite node child link is being evaluated. It is in the paused state if at least one presentation event associated with anyone of the composite child nodes is in the paused state and all other presentation events associated with the composite child nodes are in the sleeping or paused state. Otherwise, the presentation event is in the sleeping state.

NOTE 1 – More details about the behaviour of presentation event state machines for media and composite nodes are given in Clause 8.

A presentation event associated with a switch node, represented by a <switch> element, stays in the occurring state while the switch child element chosen from the bind rules (selected node) is in the occurring state. It is in the paused state if the selected node is in the paused state. Otherwise, the presentation event is in the sleeping state.

A selection event initializes in the sleeping state. It stays in the occurring state while the corresponding anchor (subset of the information units of a media object) is being selected.

Attribution events stay in the occurring state while the corresponding property values are being modified. Obviously, instantaneous events, like attribution events for simple value assignments, stay in the occurring state only during an infinitesimal period of time.

Relations are defined based on event states, changes on the event state machines, and on node (<media>, <body>, <context> or <switch> element) property values.

7.2.12.1 CausalConnectorFunctionality module

The CausalConnectorFunctionality module allows only for the definition of causal relations, defined by the <causalConnector> element of the CausalConnector module.

A <causalConnector> element has a glue expression, which defines a condition expression and an action expression. When the condition expression is satisfied, the action expression shall be executed. The <causalConnector> element shall have the *id* attribute, which uniquely identifies the element within a document.

A condition expression may be simple (<simpleCondition> element) or composite (<compoundCondition> element), both elements defined by the ConnectorCausalExpression module.

The <simpleCondition> element has a *role* attribute, whose value shall be unique in the connector's role set. As aforementioned, a role is a connector interface point, which is associated to node interfaces by a link that refers to the connector. A <simpleCondition> also defines an event type (*eventType* attribute) and to which transition it refers (*transition* attribute). The *eventType* and *transition* attributes are optional. They may be inferred by the *role* value if reserved values are used. Otherwise, the *eventType* and *transition* attributes are required.

Reserved values used for defining <simpleCondition> roles are stated in Table 7-25. If an *eventType* value is "selection", the role can also define to which selection apparatus (for example, keyboard or remote control keys) it refers, through its *key* attribute. At least the following values (case sensitive) shall be accepted for the *key* attribute: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z", "*", "#", "MENU", "INFO", "GUIDE", "CURSOR_DOWN", "CURSOR_LEFT", "CURSOR_RIGHT", "CURSOR_UP", "CHANNEL_DOWN", "CHANNEL_UP", "VOLUME_DOWN", "VOLUME_UP", "ENTER", "RED", "GREEN", "YELLOW", "BLUE", "BACK", "EXIT", "POWER", "REWIND", "STOP", "EJECT", "PLAY", "RECORD", "PAUSE". If the *key* attribute is not specified, the selection via a pointer device (mouse, touch screen, navigational keys, in agreement with clause 7.2.9, etc.) shall be assumed.

NOTE 2 – 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.

Table 7-25 – Reserved condition role values associated to event state machines

Role Value	Transition Value	Event Type
onBegin	*starts*	*presentation*
onEnd	*stops*	*presentation*
onAbort	*aborts*	*presentation*
onPause	*pauses*	*presentation*
onResume	*resumes*	*presentation*
onSelection	*starts*	*selection*
onBeginSelection	*starts*	*selection*
onEndSelection	*stops*	*selection*
onBeginAttribution	*starts*	*attribution*
onEndAttribution	*stops*	*attribution*
onAbortAttribution	*aborts*	*attribution*
onPauseAttribution	*pauses*	*attribution*
onResumeAttribution	*resumes*	*attribution*

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>. The minimal cardinality value shall always be a positive finite value, greater than zero and lesser than or equal to the maximal cardinality value. When the maximal cardinality value is greater than one, several participants may play the same role, i.e., there may be several binds connecting diverse nodes to the same role. The "unbounded" value may be set to the *max* attribute, if the role may have unlimited binds associated with it. In these two latter cases, a *qualifier* attribute should be specified informing the logical relationship among the simple condition binds. As described in Table 7-26 the possible values for the *qualifier* attribute are: "or" and "and". If the qualifier establishes the "or" logical operator, the link action will be triggered whenever any condition occurs. If the qualifier establishes the "and" logical operator, the link action will be triggered after all the simple conditions occur.

Table 7-26 – Simple condition *qualifier* values

Role Element	Qualifier	Semantics
simpleCondition	*or*	True whenever any associated simple condition occurs.
simpleCondition	*and*	True immediately after all associated simple conditions have occurred.

A *delay* attribute may also be defined for a <simpleCondition> specifying that the condition will be true after a time delay from the moment the transition occurs.

The <compoundCondition> element has a Boolean *operator* attribute ("and" or "or") relating its child elements: <simpleCondition>, <compoundCondition>, <assessmentStatement> and

<compoundStatement>. A *delay* attribute may also be defined specifying that the compound condition will be true after a time delay from when the expression relating its child elements is true. The <assessmentStatement> and <compoundStatement> elements are defined by the ConnectorAssessmentExpression module.

NOTE 3 – When an "and" compound condition relates more than one trigger condition (that is, a condition that is satisfied only in an infinitesimal time instant – as for example, the end of an object presentation), the compound condition shall be considered true in the instant immediately after all the trigger conditions have been satisfied.

The ConnectorAssessmentExpression module defines four elements: <assessmentStatement>, <attributeAssessment>, <valueAssessment> and <compoundStatement>.

The <attributeAssessment> has a *role* attribute, which has to be unique in the connector role set. As usual, the *role* is a connector interface point, which is associated to node interfaces by a <link> that refers to the connector. An <attributeAssessment> also defines an event type (*eventType* attribute). If the *eventType* value is "selection", the <attributeAssessment> should also define to which selection apparatus (for example, keyboard or remote control keys) it refers, through its *key* attribute. If the *key* attribute is not specified, the selection via a pointer device (mouse, touch screen, etc.) shall be assumed. If the *eventType* value is "presentation" or “selection”, the *attributeType* is optional and, if present, it shall specify the event state ("state"); if the *eventType* is "attribution" the *attributeType* is optional and may have the value "nodeProperty" (default) or "state". In the first case, the event represents a node property to be evaluated; in the second case the event represents the evaluation of the corresponding attribution event state. An *offset* value may be added to an <attributeAssessment> before the comparison. For example, an offset may be added to an attribute assessment to specify: "the screen vertical position plus 50 pixels".

The <valueAssessment> element has a *value* attribute that may assume an event state value, or any value to be compared with a node property.

The <assessmentStatement> element has a *comparator* attribute that compares the values inferred from its child elements (<attributeAssessment> element and <valueAssessment> element). The *comparator* attribute shall have one of the values: "eq", "ne", "gt", "lt", "gte", or "lte".

The <compoundStatement> element has a Boolean *operator* attribute ("and" or "or") relating its child elements: <assessmentStatement> or <compoundStatement>. The *isNegated* attribute may also be defined to specify that the <compoundStatement> child element shall be negated before the Boolean operation is evaluated.

An action expression captures actions that may be executed in causal relations and may be composed of a <simpleAction> or a <compoundAction> element, also defined by the ConnectorCausalExpression module.

The <simpleAction> element has a *role* attribute, which has to be unique in the connector role set. As usual, the role is a connector interface point, which is associated to node interfaces by a <link> that refers to the connector. A <simpleAction> also defines an event type (*eventType* attribute) and which event state transition it triggers (*actionType*). The *eventType* and *actionType* attributes are optional. They can be inferred by the *role* value if reserved values are used; otherwise, *eventType* and *actionType* are required. Reserved values used for defining <simpleAction> *roles* are stated in Table 7-27.

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 to the connector.

NOTE 4 – Declaring the *role*="anyName" attribute in a <bind> element of a <link> implies having a role implicitly declared as <attributeAssessment role="anyName" eventType="attribution" attributeType="nodeProperty"/>. This is the only possible case of a <bind> element referring to a role that is not explicitly declared in a connector.

NOTE 5 – If value="$anyName", the value to be attributed is 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 of a property of the composition where the link is defined, or of a property of an element that can be reached through a <port> element of the composition where the link is defined, or even of 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. Each time an attribution is set, the attributed value shall be gotten from the property identified by the <bind> element of the link.

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. 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. In these two latter cases, the actions shall be executed in the same order of the bind sequence.

NOTE. Actions binding to the same role shall be fired in the specified <bind> element order. However, an action does not need to wait the previous one to be completed in order to be fired. This means that the order of the results of applying a sequence of actions can be different from the order of the actions in the sequence.

Table 7-27 – 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*
startAttribution	*start*	*attribution*
stopAttribution	*stop*	*attribution*
abortAttribution	*abort*	*attribution*
pauseAttribution	*pause*	*attribution*
resumeAttribution	*resume*	*attribution*

A *delay* attribute may also be defined for a <simpleAction> specifying that the action shall be triggered only after waiting for the specified time.

Besides all the aforementioned attributes, the <simpleAction> element may also have attributes defined in the Animation Functionality (*duration* and *by* attributes), if its *eventType* value is "attribution" (see Clause 7.2.13).

The <compoundAction> element groups child elements: <simpleAction> and <compoundAction>. The execution of the actions shall be performed in the order they are specified. A *delay* attribute may also be defined specifying that the fired compound action shall be applied after the specified delay.

The <compoundAction> element has also an optional attribute called *operator*, which is deprecated in NCL 3.1. It should be defined only in applications that also targets Ginga 1.0 (the Player of the old NCL 3.0 version). In this case, it shall have the “seq” value. In future versions of NCL this attribute can be unsupported. The actions shall be executed in the same order of the sequence of <compoundAction> element’s child elements.

NOTE. Actions shall be fired in the specified child elements order. However, an action does not need to wait the previous one to be completed in order to be fired. This means that the order of the results of applying a sequence of actions can be different of the order of the actions in the sequence.

The <causalConnector> element may have <connectorParam> child elements, which are used to parameterize connector attribute values. The ConnectorCommonPart module defines the type of the <connectorParam> element, which has *name* and *type* attributes. In order to specify which attributes receive parameter values defined by the connector, their values are specified as the parameter name preceded by the *$* symbol. For instance, in order to parameterize the *delay* attribute, a parameter called *actionDelay* is defined (*<connectorParam name="actionDelay" type="unsignedLong"/>*) and the value *"$actionDelay"* is used in the attribute (*delay*="$*actionDelay*").

The elements of the CausalConnectorFunctionality module, their attributes, and their child elements shall comply with Table 7-28.

Table 7-28 – Extended CausalConnectorFunctionality module

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, 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)+

The ConnectorBase module defines an element called <connectorBase>, which allows for grouping connectors. As usual, the <connectorBase> element should have the *id* attribute, which uniquely identifies the element within a document. The exact content of a connector base is specified by the language profile that uses the Connectors facility. However, since the definition of connectors is not easily done by inexperienced users, the idea is to have expert users defining connectors, storing them in libraries (connector bases) that may be imported, and making them available to others for creating links.

The element of the ConnectorBase module, its attributes, and its child elements shall comply with Table 7-29.

Table 7-29 – Extended ConnectorBase module

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

7.2.12.1.1 Default values

– In a <simpleCondition> element and in a <simpleAction> element, if minimal or maximal cardinalities are not informed, “1” shall be assumed as the default value.

– In a <simpleCondition> element, if the *qualifier* attribute is not specified, the default value “or” shall be assumed.

– If the eventType value of an <attributeAssessment> element is “attribution” the attributeType is optional and has the value “nodeProperty” as default.

– In a compound statement, if the *isNegated* attribute is not defined, the default value “false” shall be assumed.

7.2.12.1.2 Exception handling

– If the minimal role’s cardinality value is not a positive finite value, greater than zero and lesser than or equal to the maximal cardinality value, the link shall be ignored.

– In a <simpleAction> element, if an *eventType* value is “attribution”, the *value* attribute is specified as “$anyName”, and the value to be attributed cannot be retrieved, no attribution shall be made.

– In a <simpleAction> element, if an *eventType* value is “attribution” and the *value* attribute is specified 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 may be made.

– In a <compoundAction> any received attribute, except the *delay* attribute shall be ignored.

– In an <attributeAssessment> element, if the *offset* value does not have the same type of the *attributeType* attribute or if it is not specified with the same unit of the value to which it will be added, the *offset* value shall be ignored.

– In an <assessmentStatement> element, if the value of the *comparator* attribute is not “eq”, “ne”, “gt”, “lt”, “gte”, or “lte”, the element shall be ignored.

– All attributes other than those defined in Table 7.28 for the <simpleAction>element should be ignored by the NCL player.

7.2.13 Animation functionality

Animation in the cartoon sense is actually a combination of two factors: support for object drawing and support for object motion – or more correctly, support for object alteration as a function of time.

NCL is not a content format and, as such, does not have support for creating media object's content and it does not have a generalized method for altering media object's content. Instead, NCL defines a scheduling and orchestration format. This means that NCL cannot be used to make cartoons, but can be used to render cartoon objects in the context of a general presentation, and to change the timing and rendering properties of a cartoon (or any other) object as a whole, while it is being displayed.

The animation primitives of NCL allow values of node properties to be changed during an active explicitly declared duration. Since NCL animation can be computationally intensive only the properties that define numerical values and colours may be animated.

The Animation Functionality defines the Animation module that provides the extensions necessary to describe what happens when a node property value is changed.

7.2.13.1 Animation module

Basically, the Animation module defines attributes that may be incorporated by <simpleAction> elements of a connector, if its *eventType* value is "attribution". Two new attributes are defined: *duration* and *by*.

When setting a new value to a property, the change is instantaneous by default (*duration*=″0″), but the change may also be carried out during an explicitly declared duration, specified by the *duration* attribute.

Also, when setting a new value to a property, the change from the old value to the new one may be linear by default (*by*=″indefinite″), or carried out step by step, with the pace specified by the *by* attribute.

The *by* attribute represents a step to be used in incrementing or decrementing a <property> value towards the final value of an attribution. It has a string as value that must be a positive number or the “indefinite” string. When the value is “indefinite”, it should be used the smallest step the middleware implementation is able to use. When the <property> value is a tuple, the same step given in the *by* attribute must be used in every element of the tuple.

The combination of the *duration* and *by* attribute definitions gives how (discretely or linearly) the change shall be performed, and its transforming interval.

7.2.13.1.1 Default values

– When setting a new value to a property, the change is instantaneous by default (duration=“0”), if the duration attribute is not specified.

– When setting a new value to a property, the change from the old value to the new one, if not specified the opposite, is assumed to be linear by default (by=“indefinite”).

7.2.13.1.2 Exception handling

– If the value set to a property by an attribution event is different from the current property value and the by attribute is defined as “0”, it must be assumed as “indefinite”.

7.2.14 Reuse functionality

NCL allows for intensive reuse of its elements. The NCL Reuse functionality is partitioned into three modules: Import, EntityReuse and ExtendedEntityReuse.

7.2.14.1 Import module

In order to allow an entity base to incorporate another already-defined base, the Import module defines the <importBase> element, which 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 this 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 reference would have the format: *alias#element_id*. The import relation 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*.

When a language profile uses the Import module, the following specifications are allowed:

– the <descriptorBase> element may have a child <importBase> element referring to a URI corresponding to another NCL document containing the descriptor base (in fact its child elements) to be imported and nested. When a descriptor base is imported, the region bases, the transition base, and the rule base, when present in the imported document, are also automatically imported to the corresponding region and rule bases of the importing document;

– the <connectorBase> element may have a child <importBase> element referring to a URI corresponding to another connector base (in fact its child elements) to be imported and nested;

– the <transitionBase> element may have a child <importBase> element referring to a URI corresponding to another transition base (in fact its child elements) to be imported and nested;

– the <ruleBase> element may have a child <importBase> element referring to a URI corresponding to another NCL document containing the rule base (in fact its child elements) to be imported and nested;

– the <regionBase> element may have a child <importBase> element referring to a URI corresponding to another NCL document containing the region base (in fact its child elements) to be imported and nested. As the referred document URI can have more than one region base, the base to be imported must be identified by assigning its id to the baseId attribute. On importing a <regionBase>, an optional attribute, called region, may be specified within the <importBase> element. When present, the attribute shall identify the id of a <region> element declared in the <regionBase> element of the host document, which did the importing operation. As a consequence, all child <region> elements of the imported <regionBase> shall be considered as child <region> elements of the region referred by the <importBase>'s region attribute. If not specified, the child <region> elements of the imported <regionBase> shall be considered children of the host document <regionBase> element that did the importing operation.

The <importedDocumentBase> element specifies a set of imported NCL documents, and shall be defined as a child element of the <head> element. In addition, <importedDocumentBase> elements shall have the *id* attribute, which uniquely identifies the element within a document.

An NCL document may be imported through the <importNCL> element. All bases defined inside an NCL document, as well as the document <body> element, are imported all at once through the

<importNCL> element. With the exception of region bases, all other bases will be treated as if each one were imported by an <importBase> element. The imported <regionBase> elements are placed as direct children of the <head> element of the host NCL document. The imported <body> element will be treated as a <context> element. It should be stressed that 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. Thus, imported <body>, as well as any of its contained nodes, may be reused inside the <body> element of the importing NCL document.

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 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.). The <importNCL> element relation has also the transitive property, i.e., 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*. By definition, the import operation is not recursive.

When a document is imported, its <media> element of application/x-ginga-settings (or application/x-ncl-settings) type has no influence on the same type <media> element of the importing document, whose properties are those that are valid for the importing document.

The elements of the Import module, their child elements, and their attributes shall comply with Table 7-30.

Table 7-30 – Extended Import module

Elements	Attributes	Content
importBase	<u>*alias*</u>, <u>*documentURI*</u>, *region*, *baseId*	Empty
importedDocumentBase	*id*	(importNCL)+
importNCL	<u>*alias*</u>, <u>*documentURI*</u>	Empty

7.2.14.2 EntityReuse module

The EntityReuse module allows an NCL element to be reused. This module defines the *refer* attribute, which refers to an element *id* that will be reused. Only <media>, <context>, <body> and <switch> may be reused. An element that refers to another element cannot be reused; i.e., its *id* cannot be the value of any *refer* attribute.

NOTE – 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 imported document *D*.

When a language profile uses this module, it may add the *refer* attribute to:

– a <media> or <switch> element. In this case, the referred element shall be, respectively, a <media> or <switch> element, which represents the same node previously defined in the document <body> itself (in any nesting level) or in an external imported <body> (in any nesting level). The referred element shall directly contain the definition of all its attributes and child elements;

– a <context> element. In this case, the referred element shall be a <context> or an imported <body> element. The referred element shall directly contain the definition of all its attributes and child elements.

When an element declares a *refer* attribute, all attributes and child elements defined by the referred element are inherited, except the *id* attribute. All attributes and child elements defined by the referring element shall be ignored by the 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, named *instance*, may be defined. The added <area> and <property> elements are inherited by the referred element, and thus, to all elements that refer it. The *instance* attribute is defined in the ExtendedEntityReuse module.

The referred element and the element that refers to it shall be considered the same, regarding its data specification. The <body> element and <context> elements shall not have child elements referring to the same element, neither both the referring and referred elements.

The referred element and the element that refers to it shall also be considered the same regarding their presentation, if the *instance* attribute receives the "instSame" value. In this case, <link>elements that are bound to the referred or the referring <media> elements shall have the *descriptor* attribute in their respective <bind> elements ignored. 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.

Therefore, assuming the set of <media> elements composed of the referred <media> element and all the referring <media> elements, the following semantics shall be respected.

– If any element of the subset formed by the referred <media> element and all other <media> elements having the *instance* attribute equal to "instSame" is scheduled to be presented, all other elements in this subset, which are not child descendants of a <switch> element, are also assumed as scheduled for presenting at the same time through a unique instance (start instruction applied on all subset elements). The common instance in presentation shall notify all events associated with the <area> and <property> elements defined in all <media> elements of this subset that were scheduled for presenting. Descendant elements of a <switch> element shall also have the same behaviour, if all rules needed to present these elements are satisfied; otherwise, they shall not be scheduled for presenting.
– When an element with the *instance* attribute equal to “new” is scheduled for presenting, no other element in the set is affected. Moreover, new independent presentation instances shall be created at each individual presentation start.

It should be stressed that all <media> elements shall have the same behaviour previously described regarding reuse, including the <media> element of application/x-ginga-settings (or application/x-ncl-settings) type.

7.2.14.2.1 Default values

– If the value of the *instance* attribute is not defined, it shall assume the “new” string.

7.2.14.2.2 Exception handling

– 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.

– 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.

– A <media> element may only refer to another <media> element; a <switch> element may only refer to another <switch> element; a <context> 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.

– 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.

– A <body>, <context> or <switch> element 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 objects in the set shall be considered as nonexistent.

7.2.15 Metainformation functionality

Metainformation does not contain content information that is used or displayed during a presentation. Instead, it contains information about content that is used or displayed. The Metainformation Functionality is composed of the Metainformation module that comes from SMIL Metainformation module specification [b-W3C SMIL 2.1].

7.2.15.1 Metainformation module

The Metainformation module contains two elements that allow for the description of NCL documents. The <meta> element specifies a single property/value pair in the *name* and *content* attributes, respectively. The <metadata> element contains information that is also related to metainformation of the document. It acts as the root element of the resource description framework (RDF) tree. The <metadata> element may have as child elements: RDF elements and its sub-elements [b-W3C RDF].

The elements of the Metainformation module, their child elements, and their attributes shall comply with Table 7-31.

Table 7-31 – Extended Metainformation module

Elements	Attributes	Content
meta	*name*, *content*	Empty
metadata	*empty*	RDF tree

7.3 NCL language profiles for IPTV

Each NCL profile may group a subset of NCL modules, allowing the creation of languages according to user needs.

Any document in conformance with NCL profiles shall have the <ncl> element as its root element.

The NCL 3.1 Full profile, also called *NCL 3.1* Language profile, is the "complete profile" of the NCL 3.1 language. It comprises all NCL modules (including those discussed in Clause 7.2) and provides all facilities for declarative authoring of NCL documents.

The following NCL 3.1 module schemas used in the profiles of this Recommendation Draft are available as an electronic attachment:

– Animation module: NCL31Animation.xsd

– CausalConnector module: NCL31CausalConnector.xsd

- CompositeNodeInterface module: NCL31CompositeNodeInterface.xsd
- ConnectorAssessmentExpression Module: NCL31ConnectorAssessmentExpression.xsd
- ConnectorBase module: NCL31ConnectorBase.xsd
- ConnectorCausalExpression Module: NCL31ConnectorCausalExpression.xsd
- ConnectorCommonPart Module: NCL31ConnectorCommonPart.xsd
- ContentControl module: NCL31ContentControl.xsd
- Context module: NCL31Context.xsd
- Descriptor module: NCL31Descriptor.xsd
- DescriptorControl module: NCL31DescriptorControl.xsd
- EntityReuse module: NCL31EntityReuse.xsd
- ExtendedEntityReuse module: NCL31ExtendedEntityReuse.xsd
- Import module: NCL31Import.xsd
- KeyNavigation module: NCL31KeyNavigation.xsd
- Layout module: NCL31Layout.xsd
- Linking module: NCL31Linking.xsd
- Media module: NCL31Media.xsd
- MediaContentAnchor module: NCL31MediaContentAnchor.xsd
- Metainformation module: NCL31Metainformation.xsd
- NCL31CausalConnectorFunctionality.xsd
- PropertyAnchor module: NCL31PropertyAnchor.xsd
- Structure module: NCL31Structure.xsd
- SwitchInterface module: NCL31SwitchInterface.xsd
- TestRule module: NCL31TestRule.xsd
- TestRuleUse module: NCL31TestRuleUse.xsd
- Timing module: NCL31Timing.xsd
- TransitionBase module: NCL31TransitionBase.xsd
- Transition module: NCL31Transition.xsd

The profiles defined for digital TV are:

a) **NCL 3.1 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, TransitionBase, Transition and Metainformation modules of NCL 3.1. The tables in Clause 7.3.1 show each module element, already extended by the attributes and child elements inherited from other modules, for this profile (see XML Schemas in the electronic attachment NCL31EDTV.xsd of this Recommendation).

b) **NCL 3.1 Raw DTV profile**: includes the Structure, Media, Context, MediaContentAnchor, CompositeNodeInterface, PropertyAnchor, Linking, CausalConnectorFunctionality, ConnectorBase, EntityReuse, and ExtendedEntityReuse modules of NCL 3.1. Tables in 7.3.2 show each module element, already extended by the attributes and child elements inherited from other modules, for this profile (see XML schema in the electronic attachment NCL31RawDTV.xsd of this Recommendation).

7.3.1 Attributes and elements of the NCL 3.1 Enhanced DTV profile

Table 7.32 – Extended structure module elements and attributes used in the Enhanced DTV profile

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

Table 7.33 – Extended media module elements and attributes used in the Enhanced DTV profile

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

Table 7.34 – Extended context module elements and attributes used in the Enhanced DTV profile

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

Table 7.35 – Extended MediaContentAnchor module elements and attributes used in the Enhanced DTV profile

Elements	Attributes	Content
area	*id, coords, begin, end, beginText, beginPosition, endText, endPosition, first, last, label, clip*	*empty*

Table 7.36 – Extended PropertyAnchor module elements and attributes used in the Enhanced DTV profile

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

Table 7.37 – Extended CompositeNodeInterface module elements and attributes used in the Enhanced DTV profile

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

Table 7.38 – Extended SwitchInterface module elements and attributes used in the Enhanced DTV profile

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

Table 7.39 - Extended layout module elements and attributes used in the Enhanced DTV profile

Elements	Attributes	Content
regionBase	*id, device, region*	*((importBase|region)+, meta*, metadata*)*
Region	*id, left, right, top, bottom, height, width, zIndex*	*(region)**

Table 7.40 – Extended descriptor module elements and attributes used in the Enhanced DTV profile

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

Table 7.41 – Extended TransitionBase module elements and attributes used in the Enhanced DTV profile

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

Table 7.42 – Extended Transition module elements and attributes used in the Enhanced DTV profile

Elements	Attributes	Content
transition	*id, type, subtype, dur, startProgress, endProgress,*	*empty*

	direction, fadeColor, horzRepeat, vertRepeat, borderWidth, borderColor	

Table 7.43 – Extended TestRule Module elements and attributes used in the Enhanced DTV profile

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

Table 7.44 – Extended TestRuleUse module elements and attributes used in the Enhanced DTV profile

Elements	Attributes	Content
bindRule	*constituent, rule*	*empty*

Table 7.45 – Extended ContentControl module elements and attributes used in the Enhanced DTV profile

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

Table 7.46 – Extended DescriptorControl module elements and attributes used in the Enhanced DTV profile

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

Table 7.47 - Extended linking module elements and attributes used in the Enhanced 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 7.48 – Extended CausalConnector functionality module elements and attributes in the Enhanced 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, duration, by*	*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 7.49 – Extended ConnectorBase module element and attributes used in the Enhanced DTV profile

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

Table 7.50 – Extended Import module elements and attributes used in the Enhanced DTV profile

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

Table 7.51 – Extended Metainformation module elements and attributes used in the Enhanced DTV profile

Elements	Attributes	Content
meta	*name, content*	*empty*

metadata	*empty*	*RDF tree*

7.3.2 Attributes and elements of the NCL 3.1 Raw DTV profile

Table 7.52 – Extended structure module elements and attributes used in the Raw DTV profile

Elements	Attributes	Content
ncl	*id, xmlns*	*(head?, body?)*
head		*(connectorBase?)*
body	*id*	*(port| property| media| context| link)**

Table 7.53 – Extended media module elements and attributes used in the Raw DTV profile

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

Table 7.54 – Extended context module elements and attributes used in the Raw DTV profile

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

Table 7.55 – Extended MediaContentAnchor module elements and attributes used in the Raw DTV profile

Elements	Attributes	Content
area	*id, coords, begin, end, beginText, beginPosition, endText, endPosition, first, last, label, clip*	*empty*

Table 7.56 – Extended PropertyAnchor module elements and attributes used in the Raw DTV profile

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

Table 7.57 – Extended CompositeNodeInterface module elements and attributes used in the Raw DTV profile

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

Table 7.58 - Extended linking module elements and attributes used in the Raw DTV profile

Elements	Attributes	Content
bind	*role, component, interface*	*(bindParam)**

bindParam	*name, value*	*empty*
linkParam	*name, value*	*empty*
link	*id, xconnector*	*(linkParam*, bind+)*

Table 7.59 – Extended CausalConnector functionality module elements and attributes in the Raw 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, duration, by*	*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 7.60 – Extended ConnectorBase module element and attributes used in the Raw DTV profile

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

8 Media objects in NCL presentations

The presentation of an NCL document requires the synchronization of several media objects, which are specified by <media> elements. For each media object, a media player shall control both the presentation of its content and its NCL events. Moreover, the media player shall be able to receive presentation commands, to control all event state machines, and to answer queries coming from the formatter.

In order to favor the incorporation of third-party media players into an NCL presentation engine, a modular design is suggested, aiming at separating the media players from the presentation engine (NCL Player).

Figure 8.1 illustrates a modular organization for an NCL presentation environment. In order to use third-party with interfaces that are not compatible with the one required by the presentation engine, it is necessary to develop modules, called adapters, to make the necessary adaptations. In this case, the media player consists of an adapter together with the player itself.

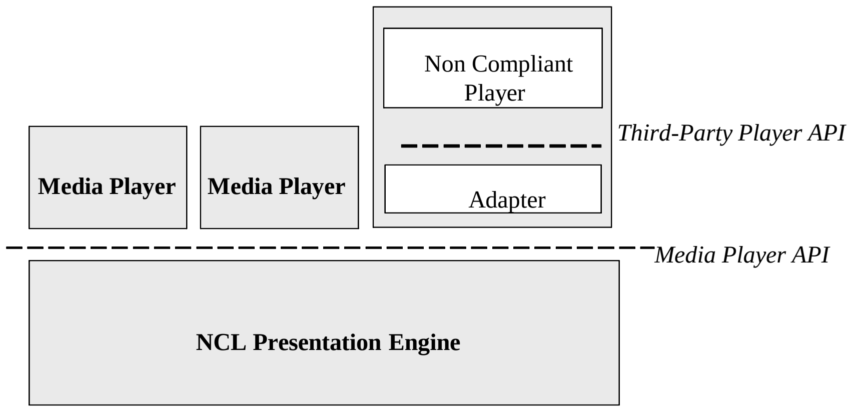

Figure 8-1 – APIs for integrating media players with an NCL presentation engine implementation

8.1 The Media Player API

Media players must control every internal event state machine and report any change on this machine to the NCL Player. Therefore, besides defining an interface to receive commands coming from the NCL Player, the Media Player API shall also define an interface to report internal event state machine changes to the NCL Player.

This clause does not specify how must be an instance of the Media Player API, but a type (a template) for a set of instances, which is language independent, enabling communication between software components that do not share a language.

Although an interface comprises every interaction between software components, what it is disclosed about the Media Player API, i.e., what it is documented in this Drat Recommendation, is more limited. It is exposed only what needs to be know in order to interact with media players.

Thus, this clause focuses on how software components interact, not on how they are implemented, restricting the documentation to phenomena that are externally visible, and exposing only what it is necessary to know.

EXAMPLE. Impementations could use synchronous or asynchronous communication between the NCL Player and media players. In asynchronous communication, for instance, some way to register and unregister callback functions is also part of the API. However, no matter the implementation, conversion to and from the Media Player API shall be considered.

8.1.1 Interface data types

The set of data types used in the Media Player API are based on the XML specification provided in Listing 8.1, except the *any* type, which represents any type that is implementation dependent.

```
<!--
XML Schema for the Media API data types
```

```
This is NCL
Copyright: 005 PUC-RIO/LABORATORIO TELEMIDIA, All Rights Reserved.
See http://www.telemidia.puc-rio.br

Public URI: http://www.ncl.org.br/NCL3.1/ancillary/mediaAPI.xsd
Author: TeleMidia Laboratory
Revision: 30/06/2013

Schema for the NCL media API data types.
-->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:mediaAPI="http://www.ncl.org.br/NCL3.1/MediaAPI"
  targetNamespace="http://www.ncl.org.br/NCL3.1/MediaAPI"
  elementFormDefault="qualified" attributeFormDefault="unqualified" >

  <!-- define the temporalAnchorAttrs attribute group -->
  <attributeGroup name="temporalAnchorAttrs">
    <attribute name="begin" type="string" use="optional"/>
    <attribute name="end" type="string" use="optional"/>
  </attributeGroup>

  <!-- define the textAnchorAttrs attribute group -->
  <attributeGroup name="textAnchorAttrs">
    <attribute name="beginText" type="string" use="optional"/>
    <attribute name="beginPosition" type="unsignedLong" use="optional"/>
    <attribute name="endText" type="string" use="optional"/>
    <attribute name="endPosition" type="unsignedLong" use="optional"/>
  </attributeGroup>

  <!-- define the sampleAnchorAttrs attribute group -->
  <attributeGroup name="sampleAnchorAttrs">
    <attribute name="first" type="string" use="optional"/>
    <attribute name="last" type="string" use="optional"/>
  </attributeGroup>

  <!-- define the coordsAnchorAttrs attribute group -->
  <attributeGroup name="coordsAnchorAttrs">
    <attribute name="coords" type="string" use="optional"/>
  </attributeGroup>

  <!-- define the labelAttrs attribute group -->
  <attributeGroup name="labelAttrs">
    <attribute name="label" type="string" use="optional"/>
  </attributeGroup>

  <!-- define the clip attribute group -->
  <attributeGroup name="clipAttrs">
    <attribute name="clip" type="string" use="optional"/>
  </attributeGroup>

  <!-- define the default attribute group -->

  <!-- define the values for defaults-->
  <simpleType name="defaultType">
    <restriction base="string">
      <enumeration value="wholeContentAnchor" />
      <enumeration value="mainContentAnchor" />
    </restriction>
  </simpleType>

  <attributeGroup name="defaultAttrs">
```

```
    <attribute name="specialAnchor" type="mediaAPI:defaultType" use="optional"/>
  </attributeGroup>

  <!-- define the values for transistions-->
  <simpleType name="transitionType">
    <restriction base="string">
      <enumeration value="starts" />
      <enumeration value="stops" />
      <enumeration value="pauses" />
      <enumeration value="resumes" />
      <enumeration value="aborts" />
    </restriction>
  </simpleType>

  <!-- define the value for the content locator-->
  <simpleType name="URIlistPrototype">
    <list itemType="anyURI"/>
  </simpleType>

  <!-- define the values for transistions event type-->
  <simpleType name="eventRestrictedPrototype">
    <restriction base="string">
      <enumeration value="presentation" />
      <enumeration value="selection" />
    </restriction>
  </simpleType>

  <!-- define the values for properties-->
  <simpleType name="propertyPrototype">
    <restriction base="string">
      <enumeration value="layout" />
      <enumeration value="position" />
      <enumeration value="size" />
      <enumeration value="sizePosition" />
      <enumeration value="plane" />
      <enumeration value="device" />
      <enumeration value="time" />
      <enumeration value="color" />
      <enumeration value="percent" />
      <enumeration value="RGB888" />
      <enumeration value="fit" />
      <enumeration value="scroll" />
      <enumeration value="style" />
      <enumeration value="balance" />
      <enumeration value="alignment" />
      <enumeration value="fontStyle" />
      <enumeration value="fontFamily" />
      <enumeration value="fontSize" />
      <enumeration value="fontVariant" />
      <enumeration value="fontWeight" />
      <enumeration value="playerLife" />
      <enumeration value="optinal integer" />
      <enumeration value="optionalURI" />
      <enumeration value="optionalTransition" />
      <enumeration value="unsignedInteger" />
      <enumeration value="date" />
      <enumeration value="short" />
      <enumeration value="anyURI" />
      <enumeration value="ID" />
      <enumeration value="IDREF" />
      <enumeration value="unsigned short" />
      <enumeration value="long" />
      <enumeration value="unsigned long" />
      <enumeration value="float" />
      <enumeration value="double" />
      <enumeration value="char" />
      <enumeration value="string" />
      <enumeration value="Boolean" />
      <enumeration value="octet" />
    </restriction>
  </simpleType>

  <!-- declare global elements in this module -->
  <element name="transition" type="mediaAPI:transitionType"/>

  <!-- define the <area> type-->
  <complexType name="anchorType">
    <attribute name="eventId" type="ID" use="required"/>
    <attribute         name="eventType"         type="mediaAPI:eventRestrictedPrototype"
use="required"/>
    <attributeGroup ref="mediaAPI:coordsAnchorAttrs" />
    <attributeGroup ref="mediaAPI:temporalAnchorAttrs" />
    <attributeGroup ref="mediaAPI:textAnchorAttrs" />
    <attributeGroup ref="mediaAPI:sampleAnchorAttrs" />
    <attributeGroup ref="mediaAPI:labelAttrs" />
    <attributeGroup ref="mediaAPI:clipAttrs" />
    <attributeGroup ref="mediaAPI:defaultAttrs" />
  </complexType>

  <!-- declare global elements in this module -->
  <element name="area" type="mediaAPI:anchorType"/>

  <!-- define the <property> type-->
  <complexType name="propertyType">
    <attribute name="eventId" type="ID" use="required"/>
    <attribute name="name" type="string" use="required" />
    <attribute name="value" type="string" use="optional"/>
    <attribute name="type" type="mediaAPI:propertyPrototype" use="required"/>
    <attribute name="externable" type="boolean" use="required"/>
  </complexType>

  <!-- declare global elements in this module -->
  <element name="property" type="mediaAPI:propertyType"/>

  <!-- define the <media> and <event> types-->
  <group name="mediaInterfaceElementGroup">
    <choice>
      <element ref="mediaAPI:area"/>
      <element ref="mediaAPI:property"/>
    </choice>
  </group>

  <complexType name="mediaType">
        <choice minOccurs="0" maxOccurs="unbounded">
          <group ref="mediaAPI:mediaInterfaceElementGroup"/>
        </choice>
        <attribute name="src" type="mediaAPI:URIlistPrototype" use="required"/>
  </complexType>

  <complexType name="eventType">
    <choice minOccurs="1" maxOccurs="1">
      <group ref="mediaAPI:mediaInterfaceElementGroup"/>
    </choice>
  </complexType>
  <!-- declare global elements in this module -->
  <element name="media" type="mediaAPI:mediaType"/>
  <element name="event" type="mediaAPI:eventType"/>
</schema>
```

Listing 8.2 – Data types for the parameters used in the Media Player API.

Although all exchanged information is textual, the specified attributes of the <media>, <area> and <property> data shall follow the types defined in Clause 7 of this Draft Recommendation, or are XML data types, or are the IDL (Interface Definition Language) data types presented in Table 8.1. The characters are coded using ISO 8859-1.

Table 8.1 – IDL data types used in this Draft Recommendation

Type	**Range**	**Minimum size in bits**
short	-2^{15} to $2^{15}-1$	16
unsigned short	0 to $2^{16}-1$	16
long	-2^{31} to $2^{31}-1$	32
unsigned long	0 to $2^{32}-1$	32
long long	-2^{63} to $2^{63}-1$	64
unsigned long long	0 to $2^{64}-1$	64
float	IEEE single-precision	32
double	IEEE double-precision	64
long double	IEEE double extended floating point	exponent of 15 bits and signed fraction of 64 bits
octet	0 to 255	8

8.1.2 Interface specification

In order to put a media object in execution, the NCL Player must firstly instantiate the appropriate media player (there can be more than one instance of a same media player), which shall follow the API defined in Listing 8.2 to communicate with the NCL Player.

Implemented by:	Operation (input parameters)
Media player	**prepare** (mediaType media)
Host NCL Player	**notifyAudioBuffer (**any buffer)
Host NCL Player	**notifyVideoBuffer (**any buffer)
Media player	**start** (ID eventId)
Media player	**addEvent** (eventType event)
Media player	**removeEvent** (ID eventId)
Media player	**stop** (ID eventId)
Media player	**abort** (ID eventId)
Media player	**pause** (ID eventId)
Media player	**resume** (ID eventId)
Host NCL Player	**notifyEventTransition** (ID eventId, transitionType transition)
Media player	**requestPropertyValue**(string name)
Host NCL Player	**notifyPropertyValue** (string name, string value)
Media player	**setPropertyValue**(string name, string value, string duration, string by)
Host NCL Player	**notifyError**(string message)

Listing 8.2 – Media Player API.

Since each media player instance controls only one media object, this object does not need to be identified in any operation. The NCL Player knows the media player instance from the moment it is created and from then on it begins to communicate directly with this instance. Similarly, media players know the NCL Player, for example through a registration process executed soon after the media player instantiation.

The *prepare* operation, issued by the NCL Player, shall inform the following parameters to the media player: the properties associated with the media object to which the media player has been created, the list of events (presentation, selection, attribution, etc.) that need to be monitored by the media player, and the location of the media content to be executed/presented.

Events that need to be monitored are identified by the *eventId* parameter. The events derived from the *whole content anchor* and *main content anchor* shall also be identified.

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 *prepare* operation without performing any action and report an error message. If the operation succeeds, the memory areas to be filled by the media player are identified and returned, depending on the type of media to be presented, through using the *notifyAudioBuffer* and *notifyVideoBuffer* operations.

Events can be added or removed from the list of events using the *addEvent* and *removeEvent* operations, respectively.

The *setPropertyValue* interface allows the NCL Player to set values to properties of the media object in execution controlled by the media player. For example, using the *setPropertyValue* the host language player can pass an input parameter used by the media player in running the presentation. When setting a new value to a property the change is instantaneous by default (parameter *duration*="0"), but the change may also be carried out during an explicitly declared duration, specified by the *duration* parameter. In this last case, the change from the old value to the new one may be linear by default (parameter *by*="indefinite"), or carried out step by step, with the pace specified by the *by* parameter.

The *requestPropertyValue* interface allows the NCL Player to get property values of the media-object in execution controlled by the media player. The value is returned when the media player notifies the NCL Player by means of the *notifyPropertyValue* interface.

Media players must notify the host language player of changes in the event state machines it controls. Event state changes are notified via *notifyEventTransition* interface.

8.1.3 Input Device Control Model

Some media players may gain control of the input devices that previously were controlled by the NCL Player. This control passing is notified to the media player via the *notifyInputControl* interface, as shown in Listing 8.4, with the corresponding data types defined in Listing 8.3.

EXAMPLE. NCL player delegates control of its input devices to its media player if the media object associated to the media player is in focus and the ENTER key is pressed (see Clause 7.2.9).

```
<!--
XML Schema for the Input Control API data types

This is NCL
Copyright: 005 PUC-RIO/LABORATORIO TELEMIDIA, All Rights Reserved.
See http://www.telemidia.puc-rio.br

Public URI: http://www.ncl.org.br/NCL3.1/ancillary/inputControlAPI.xsd
Author: TeleMidia Laboratory
Revision: 30/06/2013

Schema for the NCL media API data types.
-->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:inputControlAPI="http://www.ncl.org.br/NCL3.1/InputControlAPI"
  targetNamespace="http://www.ncl.org.br/NCL3.1/InputControlAPI"
  elementFormDefault="qualified" attributeFormDefault="unqualified" >


   <!-- define the values for deviceTye-->
  <simpleType name="someDevicePrototype">
    <restriction base="string">
      <enumeration value="remoteControl" />
      <enumeration value="keyboard" />
      <enumeration value="motionSensor" />
      <enumeration value="positionSensor" />
    </restriction>
  </simpleType>

  <simpleType name="deviceType">
    <union memberTypes="string inputControlAPI:someDevicePrototype"/>
  </simpleType>

  <!-- declare global elements in this module -->
  <element name="device" type="inputControlAPI:deviceType"/>

  <!-- define the value for the set of keys-->
  <simpleType name="keyListType">
    <list itemType="string"/>
  </simpleType>

  <!-- declare global elements in this module -->
  <element name="keyList" type="inputControlAPI:keyListType"/>
  <complexType name="sensorType">
    <attribute name="coords" type="string" use="optional"/>
    <attribute name="key" type="string" use="optional" />
  </complexType>

  <!-- declare global elements in this module -->
  <element name="sensor" type="inputControlAPI:sensorType"/>

</schema>
```

Listing 8.3 – Input Control API.

Implemented by:	Operation (input parameters)
Host NCL Player	**notifyInputControl** ()
Media player	**requestInputControl** (deviceType device, keyListType keyList)
Host NCL Player	**notifyInput** (deviceType device, sensorType sensor)
Media player	**nestInput** (unsignedInteger nestingLevel)

Listing 8.3 – Input Control API.

Upon receiving the notification, the media player must register which device types it wants to control (by default all keyboards, motion sensors, and remote controls) and which particular list of keys (in the case of keyboards and remote controls). This is done via the *requestInputControl* interface. From then on, each registered input is passed to the media player via the *notifyInput* interface: a key identification or the sensor position.

As the input control model is recurrent, a media player can also pass the control acquired to an internal entity, and so on (see Clause 7.2.9). However, each time a descendent entity gains control, the media player shall notify the host language system via the *nestInput* interface. When a BACK key is pressed, control must be passed back to the father entity, until reach the host language player that can now pass control to another plug-in.

It should be noted that the hierarchical input control may refer to any input device spread in a distributed environment (in a multiple device execution).

Any Ginga-NCL's media player has to implement the API of Listing 8.2 and follow the life-cycle defined in the following sub-clauses of Clause 8. Those media players that may control input devices have to implement the API of Listing 8.4.

8.2 Expected behaviour of basic media players

This clause deals with media players for <media> elements whose types are different from any media object containing hypermedia declarative code (for example, "application/x-ginga-NCL" type or "text/html" type) and different from any media object containing imperative code (for example, "application/x-ginga-NCLua" type).

A media object being presented is identified by the *id* attribute of the corresponding <media> element and the *id* of the <descriptor> elements that were associated with the media object, if there is any. This identification is called in this clause as *representationObjectId*. Since each media player instance controls only one media object presentation, the *representationObjectId* also identifies the corresponding media player.

8.2.1 *start* action on presentation events

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

Before issuing the *start* operation, the NCL Player shall inform the following parameters to the media player: the locators of the content of the media object to be controlled, a list of all properties associated with the media object, and a list of events (presentation, selection or attribution) that need to be monitored by the media player (defined by the <media> element's <area> and <property> child elements, and by the *whole content anchor*), by issuing a *prepare* operation. When the presentation needs to be started the *start* operation shall be issue, specifying the presentation event that needs to be started, called here *main event*.

The *src* attribute of the <media> element shall be used, by the media player, to locate the content. 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 procedure without performing any action. An *error* condition shall be notified.

The list of all properties associated with the media object must take into account the descriptors (if any) associated with the media object. The descriptors shall be chosen by the NCL Player following the directives specified in the NCL document. If the *start* action results from a link action that has a descriptor explicitly declared in its <bind> element (*descriptor* attribute of the child <bind> element of the <link> element), the resulting descriptor shall merge the attributes of the descriptor specified in the <bind> with the attributes of the descriptor specified in the corresponding <media> element, if this attribute is specified. For the common attributes, the information defined by the descriptor specified in the <bind> shall superpose the information defined by the descriptor specified in the <media> element. If the <bind> element does not contain an explicit descriptor, the resulting descriptor shall be the one specified by the <media> descriptor, if this attribute is specified; otherwise, the resulting descriptor does not exist. Based on this procedure, the resulting descriptor is used to initialize values of the properties associated with the media object. It should be reiterated that values defined in <property> child elements of the <media> element that specifies the media object superpose the corresponding values defined in the resulting descriptors.

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

Events that would have their end-times previous to 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 NCL Player should do this verification when building the monitored event list).

Monitored events that would have beginning-times before the start time of the *main event* and end-times after the start 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).

If a media player receives a *start* operation for an object already being presented (paused or not), it shall ignore the operation and keep on controlling the ongoing presentation. In this case, the <simpleAction> element that has caused the *start* action shall not cause any transition on the corresponding event state machine.

NOTE. If a video stream of a tuned service, which is not referred by any *src* attribute of <media> elements, is being presented on the video plane, the first started media object referring to this stream gets control of this content presentation, i.e., no new presentation is started. Any other further <media> element that refers to this content by using the *src* attribute, when started, begins a new presentation. Neverthless, the object presentation shall follow the same other procedures described in this clause.

If an audio stream of a tuned service, which is not referred by any *src* attribute of <media> elements, is being presented, the first started media object referring to this stream gets control of this content presentation, i.e., no new presentation is started. Any other further <media> element that refers to this content by using the *src* attribute, when started, begins a new presentation. Neverthless, the object presentation shall follow the same other procedures described in this clause.

8.2.2 *stop* action on presentation events

The *stop* action results in a *stop* operation to the corresponding media player. The *stop* operation does not need to identify any monitored event. 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 operation is issued.

If the object is not being presented (none of the events in the object’s list of events is in the *occurring* or *paused* state), the *stop* action 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 previous to the end time of the *main event* 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 end time of the *main event* shall be put in the *sleeping* state, but their *stops* transitions shall not be notified. The object’s content presentation shall be stopped.

NOTE. The *stop* operation shall transit the monitored events to the *sleeping* state no matter if a transition effect is being applied to the media object. In other words, the transition effect shall also be stopped. Transition effects shall never be applied after an object suffers a *stop* operation.

When there is no media object being presented on the video plane referring (through its *src* attribute) to a video stream of a tuned service, the video streams that were previously being presented in this plane when there were no application running shall be presented, with the same previous video parameters, even though not being referred by any media object in exhibition. A video stream content can only have its propertieschanged using a media object (referring to the video stream) in presentation. Similarly, when there is no media object being presented referring (through its *src* attribute) to an audio stream of a tuned service, the audio streams that were previously being presented when there were no application running shall be presented, with the same audio parameters, even though not being referred by any media object in exhibition. An audio stream content can only have its properties changed using a media object (referring to the audio stream) in presentation.

8.2.3 *abort* action on presentation events

The *abort* action results in an *abort* operation to the corresponding media player. The *abort* operation does not need to identify any monitored event. 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 operation is issued.

If the object is not being presented, the *abort* action shall be ignored. If the object is being presented, its *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; moreover, any content presentation shall stop.

8.2.4 *pause* action on presentation events

The *pause* action results in a *pause* operation to the corresponding media player. The *pause* operation does not need to identify any monitored event. 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 operation is issued.

If the object is not being presented (i.e., its *main event*, passed as a parameter when the media object was started, is not in the *occurring* state), the action 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. For example, if an object has an explicit duration of 30s, and after 25s it is paused, then even if the object stays paused for 7 minutes, after resuming the object, the *main event* shall stay occurring for 5s.

8.2.5 *resume* action on presentation events

The *resume* action results in a *resume* operation to the corresponding media player. The *resume* operation does not need to identify any monitored event. 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 operation is issued.

If the object is not paused (i.e., its *main event*, passed as a parameter when the media object was started, is not in the *paused* state), the action shall be ignored. 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.

8.2.6 *start* action on attribution events

The *start* action results in a *setPropertyValue* operation to the corresponding media player. The *start* action may be applied to an object only if the object is being presented. The *setPropertyValue* operation needs to identify a monitored attribution event, to define a value to be assigned to the property wrapped by the event, to define the duration of the attribution process, and to define the attribution step. When setting a value to the property, 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.

Durind setting a value to a property, if a media player receives a *requestPropertyValue* operation targeting the property, it must return (*notifyPropertyValue* operation) the initial value of the property, i.e., the one immediately before the activation of the corresponding *setPropertyValue* operation.

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 *setPropertyValue* operation.

8.2.7 *stop, abort, pause, and resume* actions on attribution events

The *stop, abort, pause and resume* actions resut in corresponding *stop, abort, pause and resume* operations to the corresponding media player. In this case the operations shall identify the attribution event being monitored.

The *stop* operation only stops the property attribution procedure, bringing the attribution event state machine to the *sleeping* state, and generating the *stops* transition.

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

The *pause* operation only pauses the property attribution procedure, bringing the attribution event state machine to the *paused* state, and generating the *pauses* transition.

Finally, the *resume* operation only resumes the property attribution procedure, bringing the attribution event state machine to the *occurring* state, and generating the *resumes* transition.

8.2.8 *addInterface* NCL Editing Command

The *addInterface* NCL Editing Command (see Clause 9) results in an *addEvent* operation to the corresponding media player. The operation needs to identify a new event that shall be included in the list of events. In the case of a monitored event, 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 previous 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.

8.2.9 *removeInterface* NCL Editing Command

The *removeInterface* NCL Editing Command results in an *removeEvent* operation to the corresponding media player. The operation needs to identify the event that should be no more controlled. In the case of a monitored event, the event state shall be put in the *sleeping* state before its removal, without generating any transition.

8.2.10 Natural end of a presentation

Presentation events of an object, with 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. When the *main event* presentation finishes, the whole media object presentation shall finish.

8.3 Expected behavior of declarative hypermedia players in NCL applications

Declarative hypermedia-objects (media objects whose content are a declarative code specified in some declarative programming language, for example media objects of "application/x-ginga-NCL" type or "text/html" type) have their life cycle controlled by their parent NCL application. This implies an execution model different from when the declarative code runs under the total control of its own engine.

A declarative hypermedia-object is handled by the NCL parent application as a set of temporal chains. A temporal chain corresponds to a sequence of presentation events, initiated from the event that corresponds to the beginning of the declarative hypermedia-object presentation. Sections in these chains may be associated with declarative hypermedia-object's <area> child elements using the *clip* attribute. For a declarative hypermedia-object with NCL code, a temporal chain is identified by one of the NCL document entry points, defined by <port> elements, children of the document's <body> element. A declarative hypermedia-object's content anchor can also refer to any content anchor defined inside the declarative code itself using the *label* attribute of an <area> child element As an example, for a declarative hypermedia-object with NCL code (i.e., <media> element of "application/x-ncl-NCL" type>) one of its <area> elements may refer to a <port> element. In its turn, the <port> element may be mapped to an <area> element defined in any object nested in the declarative NCL hypermedia-object. Thus, note that a declarative hypermedia-object can externalize content anchors defined inside its content to be used in links defined by the NCL parent object, in which the declarative hypermedia-object is included.

As usual in NCL, a declarative hypermedia-object shall have a content anchor called the *whole content anchor* declared by default in NCL documents. This content anchor, however, has a special meaning. It represents the presentation of any chain defined by the media-object. Every time a declarative hypermedia-object is started without specifying one of its content anchors, the *whole content anchor* is assumed, as usual, meaning that the presentation of all temporal chain shall be instantaneously started, in the order they are defined by <area> elements.

Document authors may define NCL links to start, stop, pause, resume or abort the execution of a declarative code. On the other hand, a declarative code may also command the start, stop, pause or resume of its associated content anchors and properties. These transitions may be used as conditions of NCL links to trigger actions on other objects of the same NCL parent application. Thus, a two-way synchronization can be established between a declarative code and the remainder of the NCL application.

NCL links may be bound to declarative hypermedia-object interfaces (<area> and <property> elements, and the default whole content anchor). A declarative player (the language engine) shall interface its declarative execution environment with the NCL Player. Analogous to basic media content players, declarative-code players shall control event state machines associated with the declarative media-object, reporting changes to their parent NCL player. A declarative hypermedia-object shall be able to reflect in its content anchors and properties behaviour changes in its temporal chains. As usual, declarative hypermedia-object player shall implement the media player API and the input control API of Listing 8.2 and 8.3, respectively.

8.3.1 *start* action on presentation events

Before issuing the *start* operation, the NCL Player shall inform the following parameters to the declarative hypermedia-object player: the locator of the content of the declarative hypermedia-object to be controlled, a list of all properties associated with the media object, ~~the media object identification during execution (*representationObjectId*);~~ and a list of events (presentation, selection or attribution) that need to be monitored by the hypermedia-object player (defined by the <media> element's <area> and <property> child elements, and by the *whole content anchor*), by issuing a *prepare* operation. When the presentation needs to be started the *start* operation shall be issue, specifying the event (defined by the *clip, label*, or the *whole content anchor*), which identifies the associated temporal chain sections, to be started, called here *main event*.

From the locator (*src* attribute of the media object), the player tries to locate of the content of the declarative hypermedia-object. If the content cannot be located, the player shall finish the starting procedure, without performing any action. An *error* condition shall be notified.

From the *start* operation on, the NCL Player shall follow the same procedure defined for basic media objects, defined in Clause 8.1.1, with the exception presented in the following three paragraphs. Similarly to basic media players (see Clause 8.2.1), the list of all properties associated with the declarative hypermedia-object must take into account the descriptors associated with the media object, if there is any.

If a declarative hypermedia-object player receives a *start* operation for a temporal chain already being presented (paused or not), it shall ignore the operation and keep on controlling the ongoing presentation. However, unlike what is performed on <media> elements of the basic types, if the start operation is for a temporal chain that is not being presented, the operation must be executed even if another temporal chain is being presented (paused or occurring). As a consequence, unlike what happens for the basic types of <media> elements, a <simpleAction> element with an *actionType* attribute equal to "stop", "pause", "resume" or "abort" shall be bound through a link to a declarative hypermedia-object's interface, which shall not be ignored when the action is applied.

Every time a declarative hypermedia-object is started without specifying one of its content anchors, the *whole content anchor* is passed in the *start* operation, meaning that the presentation of all temporal chains shall be started, in the order they are defined in the event list controlled by the hypermedia-object player.

Unlike what is performed on <media> elements of the basic types, 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.

Likewise the basic media players (see Clause 8.2.1), the list of events to be monitored by a declarative hypermedia-object player should also be computed by the NCL Player, taking into account the NCL document specification. The NCL Player shall check all links where the media object and the resulting descriptor (if any) participate.

Events that would have their end-times previous to 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 declarative hypermedia-object player (the NCL Player should do this verification when building the monitored event list).

Monitored events that would have beginning times before the start time of the *main event* and end-times after the start 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).

8.3.2 *stop* action on presentation events

The *stop* action results in a *stop* operation to the corresponding hypermedia-object player. The *stop* operation needs to identify a temporal chain already being controlled (or all of them). To identify the temporal chain means to identify the corresponding <media> element’s interface.

The *stop* operation issued by an NCL Player shall be ignored by a declarative hypermedia-object player if the temporal chain associated with the specified interface is not being presented (if none of the events in the object list of events is in the *occurring* or *paused* state). If the temporal chain associated with the specified interface is being presented, the *main event* (the event passed as a parameter when the temporal chain was started) and all monitored events of this temporal chain in the *occurring* or in the *paused* state with end time equal or previous to the end time of the *main event* 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.

Unlike the basic media types, if some 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 another 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* operation is applied to a declarative hypermedia-object specifying the *whole content anchor, stop* operations shall be issued for all temporal chains.

8.3.3 *abort* action on presentation events

The *abort* action results in an *abort* operation to the corresponding hypermedia-object player. The *abort* operation needs to identify a temporal chain already being controlled (or all of them). To identify the temporal chain means to identify the corresponding <media> element’s interface.

The *abort* operation issued by an NCL Player shall be ignored by a declarative hypermedia-object player if the temporal chain associated with the specified interface is not being presented (if none of the events in the object list of events is in the *occurring* or *paused* state). If the temporal chain associated with the specified interface is being presented, the *main event* (the event passed as a parameter when the temporal chain was started) and all monitored events of this temporal chain in the *occurring* or in the *paused* state shall transit to the *sleeping* state, and their *aborts* transitions shall be notified. The temporal chain presentation shall be stopped.

Unlike what is performed on <media> elements of the basic types, 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 stopped the *whole content anchor* shall be put in the *paused* state. If the *abort* operation is applied to a declarative hypermedia-object specifying the *whole content anchor, abort* operations shall be issued for all temporal chains.

8.3.4 *pause* action on presentation events

The *pause* action results in a *pause* operation to the corresponding hypermedia-object player. The *pause* operation needs to identify a temporal chain already being controlled (or all of them). To identify the temporal chain means to identify the corresponding <media> element's interface.

The *pause* operation issued by an NCL Player shall be ignored by a declarative hypermedia-object player if the temporal chain associated with the specified interface is not being presented. If the temporal chain associated with the specified interface is being presented, the *main event* (the event passed as a parameter when the temporal chain was started) and all monitored events of this temporal chain in the *occurring* shall transit to the *paused* state and their *pauses* transitions shall be notified. The temporal chain presentation shall be paused and the pause elapsed time shall not be considered as part of its duration.

Unlike what is performed on <media> elements of the basic types, 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* operation is applied to a declarative hypermedia-object specifying the *whole content anchor* is assumed, *pause* operations shall be issued for all other content anchors that are in the occurring state.

8.3.5 *resume* action on presentation events

The *resume* action results in a *resume* operation to the corresponding hypermedia-object player. The *resume* operation needs to identify a temporal chain already being controlled (or all of them). To identify the temporal chain means to identify the corresponding <media> element's interface.

The *resume* operation issued by an NCL Player shall be ignored by a declarative hypermedia-object player if the temporal chain associated with the specified interface is not in a paused satate. If the temporal chain is in the *paused* state, the *main event* and all monitored presentation events in the *paused* state shall be put in the *occurring* state and their *resumes* transitions shall be notified.

Unlike what is performed on <media> elements of the basic types, if any content anchor is resumed, the *whole content anchor* shall be set to the *occurring* state. If the *resume* operation is applied to a declarative hypermedia-object specifying the *whole content anchor* and the *whole content anchor* is not in the *paused* state due to a previous receive of a *pause* operation, the *resume* operation is ignored. Otherwise, *resume* operations 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* has received the *pause* operation.

8.3.6 *Natural end* of a temporal chain section presentation

Events of a declarative hypermedia-object normally end their execution naturally, without needing external instructions. In this case, the declarative hypermedia-object player shall transit the event to the *sleeping* state and notify the *stops* transition. The same shall be done for monitored events of the same temporal chain in the *occurring* state with the same end time of the *main event* or with unknown end time, when the *main event* ends. Event chains in the *occurring* state with end time posterior to the end time of the *main event* shall be put in the sleeping state but without generating the *stops* transition and without incrementing the *occurrences* attribute.

Unlike what is performed on <media> elements of the basic types, 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 the 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, when a content anchor execution ends, the *whole content anchor* shall be set to the *paused* state.

8.3.7 *start, stop, abort, pause and resume* actions on attribution events

All actions for attribution events have the same effect on the corresponding property attribution as they have on any property attribution of NCL objects of the basic types, as specified in Clauses 8.2.6 and 8.2.7.

8.3.8 addEvent and removeEvent instructions

The *addEvent* and *removeEvent* instructions have the same effect on the list of monitored events, as specified in Clauses 8.2.8 and 8.2.9.

8.4 Expected behaviour of imperative-object media players in NCL applications

In an implementation in conformance with Ginga-NCL specification, the support to the “application/x-ginga-NCLua” type is required, which allows for having Lua imperative code (file extension “.lua”) content associated to <media> element. Other imperative object types are optional, but their execution engine must follow the same semantics specified in this clause.

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 Player, and shall follow the Media Player API and the Input Control API defined in Clause 8.1.

As stated in Clause 7.2.4, imperative code span may be associated with an <area> element (using the *label* attribute). If external links start, stop, pause, resume, or abort the anchor presentation, callback functions in the imperative code span shall be triggered. The way these callbacks are defined is responsibility of each imperative code associated with the NCL imperative object.

As usual in NCL, an imperative object shall have a content anchor called the *whole content anchor,* which is declared by default. However, this content anchor has a special meaning. It represents the execution of any code span inside the imperative-code object. Another content anchor, called *main content anchor*, is also defined by default. Every time an imperative object is started without specifying one of its content anchors or properties, the *main content anchor* is assumed and, as a consequence, the code span associated to it. In all other references to the imperative object without specifying one of its content-anchors or properties, the *whole content anchor* shall be assumed.

Imperative objects can also define <property> child elements. The <property> element can be mapped to a code span (function, method, etc.) through its *name* attribute. In this case, a “start” link action 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 (for example, the object’s properties specifying the screen region in which the result of the code span execution will be placed), the action “start” shall assign the value to the attribute.

A <property> element defined as a child of a <media> element representing an imperative code may be associated with an NCL link assessment role. In this case, the NCL Player 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 player to the NCL Player. If the <property> element is mapped to a code span, it shall be executed and the output value resulting from the execution shall be returned to the NCL Player.

Analogous to perceptual (i.e., basic) media content players (video, audio, image, etc.), imperative-code players shall control event state machines associated with the imperative object. As an example, if a code span finishes its execution, the player shall generate the stops transition in the event presentation state machine corresponding to the code execution. However, unlike the basic media content players, an imperative-code player may not have sufficient information to control by itself its event state machines, and shall rely on programmed code, part of its imperative content, to accomplish these controls.

On the other hand, an imperative code span may also command the start, stop, pause, or resume of its <area> and <property> elements through an API offered by the imperative language (see Clause 10.3.2 for the NCLua case). The resulting transitions may be used as conditions of NCL links to trigger actions on other NCL objects of the same application. Thus, a two-way synchronization can be established between the imperative code and the remainder of the NCL application.

The lifecycle of an imperative object is controlled by the NCL Player. The NCL Player 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 players, once instantiated, the imperative-object media player shall execute an initialization procedure. However, unlike other media players, this initialization code must be specified by the author of the imperative code. This initialization procedure is executed only once, for each imperative-object instance. It creates all code spans and data that may be used during the imperative-object execution and, in particular, registers one (or more) event handlers for communication with the NCL Player.

After the initialization, the execution of the imperative object becomes event oriented in both directions; i.e., any action commanded by the NCL Player reaches the registered event handlers, and any event state change, controlled by the imperative-object media player, generates a notification that is sent to the NCL Player (as for example, the natural end of a code span execution). After the initialization, the imperative-object player is then ready to perform any operation as discussed in the next clauses.

8.4.1 *start* action on presentation events

Before issuing the the *start* insaction, the NCL Player shall inform the following parameters to the imperative-object player: the locator of the content of the media object to be controlled, a list of all properties associated with the media object, and a list of events that need to be monitored (defined by the <media> element's <area> and <property> child elements, and by the default content anchors), by issuing a *prepare* operation. When the presentation needs to be started the *start* operation shall be issue, specifying the event (defined by the *label*, or the *main content anchor*), which identifies the associated imperative code to be started.

From the locator (*src* attribute of the media object), 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. An *error* condition shall be notified.

Similarly to the basic media players (see Clause 8.2.1), the list of all properties associated with the imperative object must take into account the descriptors associated with this media object, if any.

Besides, the list of events to be monitored should be computed by the NCL Player taking into account the NCL document specification. It shall check all links where the imperative object and the resulting descriptor (if any) participate.

Unlike what is performed on basic types of <media> elements, if an imperative-object player receives a *start* operation 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* operation shall be ignored by the imperative-code player that keeps on controlling the ongoing execution. As a consequence, unlike what happens with <media> elements of the basic types, a <simpleAction> element with an *actionType* attribute equal to "stop", "pause", "resume" or "abort" shall be bound through a link to an imperative node interface, which shall not be ignored when the action is applied.

Since neither the NCL Player nor the imperative-code media player have 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 media 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. Moreover, 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.

Unlike what is performed on <media> elements of the basic types, 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.

8.4.2 *stop* action on presentation events

The *stop* action results in a *stop* operation to the corresponding imperative-object player. 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's interface.

The *stop* operation issued by an NCL Player 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). If the imperative-object interface is being executed, its corresponding presentation event shall transit to the *sleeping* state, and its *stops* transition shall be notified. The imperative code execution associated with the interface shall be stopped.

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.

Unlike what is performed on <media> elements of the basic types, 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* operation is applied to an imperative object specifying the *whole content anchor*, *stop* operations shall be issued for all other content anchors.

8.4.3 *abort* action on presentation events

The *abort* action results in an *abort* operation to the corresponding imperative-object player. The *abort* operation needs to identify an imperative code span already being controlled. To identify the imperative code span means to identify the corresponding <media> element's interface.

If the imperative code associated with the object's interface is not being executed, the *abort* operation 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 its *aborts* transition shall be notified.

For the same reason discussed in the *start* operation, 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.

Unlike what is performed on <media> elements of the basic types, 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* operation is applied to an imperative object specifying the *whole content anchor*, *abort* instructions shall be issued for all other content anchors.

8.4.4 *pause* action on presentation events

The *pause* action results in a *pause* operation to the corresponding imperative-object player. The *pause* operation needs to identify an imperative code span already being controlled. To identify the imperative code span means to identify the corresponding <media> element's interface.

If the imperative code associated with the object's interface is not being executed (and not in the *paused* state), the operation 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, its *pauses* transition shall be notified, and the pause elapsed time shall not be considered as part of the object duration.

For the same reason discussed in the *start* operation, 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.

Unlike what is performed on <media> elements of the basic types, 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* operation is applied to an imperative object specifying the *whole content anchor*, *pause* operations shall be issued for all other content anchors that are in the *occurring* state.

8.4.5 *resume* action on presentation events

The *resume* action results in a *resume* operation to the corresponding imperative-object player. The *resume* operation needs to identify an imperative code span already being controlled. To identify the imperative code span means to identify the corresponding <media> element's interface.

If the imperative code associated with the object's interface is not paused, the operation shall be ignored. If the imperative code associated with the object's interface is paused, its associated event shall transit to the *occurring* state, and its *resumes* transition shall be notified.

For the same reason discussed in the *start* operation, except for the event associated with the *whole content anchor*, it is the responsibility of the paused-code span, before it resumes, 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 resumed code, and to inform if a transition associated with a change shall be notified.

Unlike what is performed on <media> elements of the basic types, if any content anchor is resumed, the *whole content anchor* shall be set to the *occurring* state. If the *resume* operation is applied to an imperative object specifying the *whole content anchor* and the *whole content anchor* is not in the *paused* state due to a previous receive of a *pause* instruction, the *resume* operation is ignored. Otherwise, *resume* operations 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 *pause* operation.

8.4.6 Natural end of a code execution of presentation events

Events of an imperative object normally end their execution naturally, without needing external command. 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 its *stops* transition shall be notified.

Unlike what is performed on <media> elements of the basic types, 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.

8.4.7 *start* action on attribution events

The *start* action results in a *setPropertyValue* operation to the corresponding imperative-object media player. The *start* action issued by an NCL Player 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 both cases, the *setPropertyValue* instruction needs to identify a monitored attribution event, and, if it is the case, to define a value to be assigned to the property wrapped by the event, to define the duration of the attribution process, and to define the attribution step. When setting a value to an attribute, the imperative-object 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.

Durind setting a value to a property, if an imperative-object media player receives a *requestPropertyValue* operation targeting the property, it must return (*notifyPropertyValue* operation) the initial value of the property, i.e., the one immediately before the activation of the corresponding *setPropertyValue* operation.

If a *setPropertyValue* operation is applied to an event that calls the execution of a code span, no event associated to a content anchor has its state 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 *setPropertyValue* operation.

8.4.8 ***stop, abort, pause*** **and** ***resume*** **actions on attribution events**

With the exception of the *start* operation, discussed in the previous clause, all other operations have the same effect on the corresponding property attribution as they have on any property attribution of NCL objects of the basic types, as specified in Clause 8.2.7.

8.4.9 addEvent and removeEvent instructions

The *addEvent* and *removeEvent* instructions have the same effect on the list of monitored events, as specified in Clauses 8.2.8 and 8.2.9.

8.5 Expected behavior of media players after actions applied to composite objects

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

8.5.1 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>, <switch>, or <body> element) as a whole (i.e. without an interface being informed). As usual, the event state machine of the presentation event defined on the composite node shall be controlled as specified in 7.2.12. Analogously, an <attributeAssessment> with *eventType* attribute value equal to "presentation" and *attributeType* equal to "state" may be bound by a link to a composite node (represented by a <context>, <switch>, 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 action shall also be reflected to the event state machines of the composition's child nodes, as explained in the following sub-clauses.

8.5.2 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 without referring to any specific interface, the *start* operation shall also be applied to all presentation events mapped by the <context> or <body> element's ports. Moreover, the *start* operations shall be applied in the same order the <port> elements are defined in the composition.

If the author wants to start the presentation using a specific port, it shall, in addition, indicate the <port> *id* as the *interface* value of the corresponding <bind> element.

8.5.3 Stopping a context presentation

If a <context> or <body> element participates on an action role whose action type is "stop", when this action is fired without referring to any specific interface, the *stop* operation shall also be applied to all presentation events of the composition's 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.

8.5.4 Aborting a context presentation

If a <context> or <body> element participates on an action role whose action type is “abort”, when this action is fired without referring to any specific interface, the *abort* operation shall also be applied to all presentation events of the composition’s child nodes.

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

8.5.5 Pausing a context presentation

If a <context> or <body> element participates on an action role whose action type is “pause”, when this action is fired without referring to any specific interface, the *pause* operation shall also be applied to all presentation events of the composition’s child nodes that are in the occurring state.

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

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

8.5.6 Resuming a context presentation

If a <context> or <body> element participates on an action role whose action type is “resume”, when this action is fired without referring to any specific interface, the *resume* operation shall also be applied to all presentation events of the composition’s child nodes that are in the paused state, except those that were already paused when the composition has been paused.

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

8.6 Relation between the presentation-event state machine of a node and the presentation-event state machine of its parent-composite node

This clause applies for objects represented by <context>, <body>, and <switch> elements, and <media> elements of “application/x-ginga-NCL” type.

Whenever a presentation event of a child node (media or composite) goes to the *occurring* state, the presentation event of the composite node (or of the NCL node of “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 of an NCL node of “application/x- ginga-NCL” type) have their presentation events in the *sleeping* state, the presentation event of the composite node (or of the NCL node of “application/x- ginga-NCL” type) shall also be in the *sleeping* state.

Composite nodes (or NCL nodes of “application/x- ginga-NCL” type) do not need to infer *aborts* transitions from their child nodes. These transitions in presentation events of composite nodes (or of NCL nodes of “application/x- ginga-NCL” type) shall occur only when operations are applied directly to composite node presentation events (see Clause 8.4).

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

9 NCL Editing Commands

NCL Editing Commands (*nclEditingCommand*) may be issued externally to an NCL application execution or internally by the execution of an NCL application's imperative object (Clause 10 deals with events generated by NCLua objects).

NCL Editing commands allow changing an NCL application behaviour during runtime [b_NCL Live E.C.].

9.1 Private bases

The core of an NCL presentation engine is composed of the NCL Player and its Private Base Manager module.

The NCL Player 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 NCL Player deals with NCL applications that are collected inside a data structure known as *private base*. NCL applications in a private base may be started, paused, resumed, aborted, stopped, and may refer to each other.

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

NCL Editing Commands are wrapped in a structure called NCL event descriptors. NCL event descriptors have a structure composed basically of an *eventId* (identification), a time reference (*eventNPT)* and a private data field. The identification uniquely identifies the NCL Editing Command. The time reference indicates the exact moment to trigger the event (to execute the command). A time reference equal to zero informs that the event shall be triggered immediately after being received (events carrying this type of time-reference are commonly known as "do it now" events). The private data field provides support for event parameters (see Table 9.1).

Table 9.1 – Editing Command event descriptor

Syntax	Number of bits
EventDescriptor ( ) {	
eventId	16
eventNPT	33
privateDataLength	8
commandTag	8
sequenceNumber	7
finalFlag	1
privateDataPayload	8 to 1928
FCS	8
}	

The *commandTag* uniquely identifies the Editing Commands, as specified in Table 9.2. In order to allow sending a complete command in more than one event descriptor, all descriptors of the same command shall be numbered and sent in sequence (i.e., it cannot be multiplexed with other Editing Commands with the same *commandTag*), with the *finalFlag* equal to 1, except for the last descriptor that shall have the *finalFlag* field equal to 0. The *privateDataPayload* contains the Editing-Command parameters. Finally, the *FCS* field contains a checksum of the entire *privateData* field, including the *privateDataLength*.

NCL Editing Commands are divided in three subsets.

The first subset focuses on the private base operation (openBase, activateBase, deactivateBase, saveBase, and closeBase commands).

The second subset allows for application manipulation in a private base (to add, remove, and save an application in an open private base, and to start, pause, resume, and stop application presentations in an active private base).

The third subset defines commands for live editing in an open private base, allowing NCL elements to be added and removed, and allowing values to be set to NCL <property> elements. *Add* commands always have NCL elements as their arguments. The NCL elements are defined using an XML-based syntax notation defined in Clause 9.2, which is similar to the syntax notation used in the NCL 3.1 language schemas, with the exception of the *addInterface* command, in which the *begin* or *first* attribute of an <area> element may receive the "now" value, specifying the current NPT (Normal Play Time) of the node specified in the *nodeId* argument. Whether the specified NCL element already exists or not, document consistency shall be maintained by the NCL Player, in the sense that all element attributes stated as required shall be defined. There is just one exception to this rule, the *interface* attribute of a <bind> child element of a <link> elements may be left inconsistent, referring to an <area> element to be fulfilled by an *addInterface* command whose *begin* attribute has the "now" value. In this case, the <link> shall be evaluated as soon as the *addInterface* command is issued.

If the XML-based *command parameter* (command arguments) is short enough, it may be transported directly in the event descriptors' payload. Otherwise, the *privateDataPayload* carries a set of reference pairs. In the case of pushed files (NCL documents or NCL nodes), each pair is used to associate a set of file paths with their respective location (identification) in the transport system. In the case of pulled files or files 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.

Table 9.2 shows the *command strings* with their arguments (command parameters) surrounded by round brackets. The table also gives the unique identifier of each Editing Command (*commandTag*) and the command semantics.

Table 9.2 – Editing Commands for Ginga's private base manager

Command string	Command tag	Description
openBase (baseId, location, meta)	0x00	Opens an existing private base located with the *location* parameter. If the private base does not exist or the location parameter is not informed, a new base is created with the *baseId* identifier. The location parameter specifies the storage device in the receiver environment and the path for opening the base. The *meta* parameter contains information of accessible network domains from which application specification can come to be included in the private

Command string	Command tag	Description
		base. If not specified only application specifications coming from the same network domain that issued the *openBase* command shall be considered. If the private base is already opened, ignores the command.
activateBase (baseId)	0x01	Turns on an open private base. All its applications are then available to be started. If the private base is not opened, ignores the command.
deactivateBase (baseId)	0x02	Turns off an open private base. All its running applications shall be stopped. If the private base is not opened, ignores the command
saveBase (baseId, location)	0x03	Saves the whole private base content into a persistent storage device (if available). The *location* parameter shall specify the device and the path for saving the base. If the private base is not opened, ignores the command
closeBase (baseId)	0x04	Closes the open private base and disposes all private base content. If the private base is not opened, ignores the command
addDocument (baseId, {uri, id}+, meta)	0x05	Adds an NCL application to an open private base. The NCL application’s files can be: i) sent in the datacast network as a set of pushed files; for these pushed files, each {uri, id} pair is used to relate a set of file paths in the NCL document specification with their respective locations in a transport system; NOTE. The set of reference pairs shall be sufficient to enable Ginga mapping any file reference present in the NCL application specification to its concrete location in the receiver memory. ii) received from an IP network as a set of pulled files, or may be files already present in the receiver; for these pulled files, no {uri, id} pairs have to be sent, except the {uri, “null”} pair associated with the NCL document specification that the Editing Command requests to be added in baseId, if this NCL document is not received as a pushed file. The *meta* parameter contains information of accessible network domains with permission information to each domain. Permission to access to network resources is given by

Command string	Command tag	Description
		application authentication. If the application is already added, ignores the command
removeDocument (baseId, documentId)	0x06	Removes an NCL application from an open private base. If the application is not in the private base, or if the application is running, ignores the command.
startDocument (baseId, documentId, interfaceId, offset, nptBaseId, nptTrigger) NOTE. The offset parameter is a time value.	0x07	Starts playing an NCL document in an active private base, beginning the presentation from a specific document interface. The time reference provided in the *nptTrigger* field defines the initial time positioning of the application with regards to the NPT time base identified in the *nptBaseId* field. Three cases may happen: i) If *nptTrigger* is different from 0 and is greater than or equal to the current NPT value of the NPT time base identified by the *nptBaseId*, the document presentation shall wait until NPT has the value specified in *nptTrigger* to be started from its beginning time+*offset*. ii) If *nptTrigger* is different from 0 and is less than the current NPT value of the NPT time base identified by the *nptBaseId*, the application shall be started immediately from its beginning time+*offset*+(NPT – *nptTrigger*)$_{\text{seconds}}$ NOTE. Only in this case, the *offset* parameter value may be a negative time value, but *offset*+(NPT – nptTrigger)$_{\text{seconds}}$ shall be a positive time value. iii) If *nptTrigger* is equal to 0, the application shall start its presentation immediately from its beginning time+*offset* NOTE. If the *interfaceId* parameter is specified as “null”, all <port> element of the <body> element shall be triggered (started). If the *offset* parameter is specified as “null”, it shall assume the “0” as value. If the application is not in the private base, or if the application is running, ignores the command.
stopDocument (baseId, documentId)	0x08	Stops the presentation of an NCL application in an active private base. All application events that are occurring shall be stopped. If the application is not in the private base, or if the application isnot running, ignores the command.
pauseDocument (baseId, documentId)	0x09	Pauses the presentation of an NCL application in an active private base. All application events that are occurring shall be paused. If the application is not in the private base, or if the

Command string	Command tag	Description
		application is not running, ignores the command.
resumeDocument (baseId, documentId)	0x0A	Resumes the presentation of an NCL application in an active private base. All previously application events that were paused by the pauseDocument Editing Command shall be resumed. If the application is not in the private base, or if the application is not running, ignores the command.
saveDocument (baseId, documented, location)	0x2E	Saves an NCL application of an open private base into a persistent storage device (if available). The *location* parameter shall specify the device and the path for saving the application. If the NCL application to be saved is running in the open private base, first stops its presentation (all application events that are occurring shall be stopped). If the application is not in the private base, ignores the command.
addRegion (baseId, documentId, regionBaseId, regionId, xmlRegion)	0x0B	Adds a <region> element as a child of another <region> in the <regionBase> or as a child of the <regionBase> (*regionId*= "the null string") of an NCL document in an open private base. If the application is not in the private base, or the destination region base does not exist, ignores the command. If the <region> element already exists, first remove it.
removeRegion (baseId, documentId, regionId)	0x0C	Removes a <region> element from a <regionBase> of an NCL document in an open private base. If the application is not in the private base, or the destination region base does not exist, or the <region> element does not exist, ignores the command.
addRegionBase (baseId, documentId, xmlRegionBase)	0x0D	Adds a <regionBase> element to the <head> element of an NCL document in an open private base. If the XML specification of the regionBase is sent in a file system apart; the *xmlRegionBase* parameter is just a reference to this content. If the application is not in the private base, ignores the command. If the <regionBase> element already exists, first remove it.
removeRegionBase (baseId, documentId, regionBaseId)	0x0E	Removes a <regionBase> element from the <head> element of an NCL document in an open private base. If the application is not in the private base, or the <regionBase> element does not exist, ignores the command.
addRule (baseId, documentId,	0x0F	Adds a <rule> element to the <ruleBase> of an NCL

Command string	**Command tag**	**Description**
xmlRule)		document in an open private base. If the application is not in the private base, or the destination rule base does not exist, ignores the command. If the <rule> element already exists, first remove it.
removeRule (baseId, documentId, ruleId)	0x10	Removes a <rule> element from the <ruleBase> of an NCL document in an open private base. If the application is not in the private base, or the destination region base does not exist, or the <rule> element does not exist, ignores the command.
addRuleBase (baseId, documentId, xmlRuleBase)	0x11	Adds a <ruleBase> element to the <head> element of an NCL document in an open private base. If the XML specification of the ruleBase is sent in a file system apart; the *xmlRuleBase* parameter is just a reference to this content. If the application is not in the private base, ignores the command. If the <ruleBase> element already exists, first remove it.
removeRuleBase (baseId, documentId, ruleBaseId)	0x12	Removes a <ruleBase> element from the <head> element of an NCL document in an open private base. If the application is not in the private base, or the <ruleBase> element does not exist, ignores the command.
addConnector (baseId, documentId, xmlConnector)	0x13	Adds a <connector> element to the <connectorBase> of an NCL document in an open private base. If the application is not in the private base, or the destination connector base does not exist, ignores the command. If the <connector> element already exists, first remove it.
removeConnector (baseId, documentId, connectorId)	0x14	Removes a <connector> element from the <connectorBase> of an NCL document in an open private base. If the application is not in the private base, or the destination connector base does not exist, or the <connector> element does not exist, ignores the command.
addConnectorBase (baseId, documentId, xmlConnectorBase)	0x15	Adds a <connectorBase> element to the <head> element of an NCL document in an open private base. If the XML specification of the connectorBase is sent in a file system apart, the *xmlConnectorBase* parameter is just a reference to this content. If the application is not in the private base, ignores

Command string	Command tag	Description
		the command. If the <connectorBase> element already exists, first remove it.
removeConnectorBase (baseId, documentId, connectorBaseId)	0x16	Removes a <connectorBase> element from the <head> element of an NCL document in an open private base. If the application is not in the private base, if the <connectorBase> element does not exist, ignores the command.
addDescriptor (baseId, documentId, xmlDescriptor)	0x17	Adds a <descriptor> element to the <descriptorBase> of an NCL document in an open private base. If the application is not in the private base, or the destination descriptor base does not exist, ignores the command. If the <descriptor> element already exists, first remove it.
removeDescriptor (baseId, documentId, descriptorId)	0x18	Removes a <descriptor> element from the <descriptorBase> of an NCL document in an open private base. If the application is not in the private base, or the destination descriptor base does not exist, or the <descriptor> element does not exist, ignores the command.
addDescriptorSwitch (baseId, documentId, xmlDescriptorSwitch)	0x19	Adds a <descriptorSwitch> element to the <descriptorBase> of an NCL document in an open private base. If the XML specification of the descriptorSwitch is sent in a file system; the *xmlDescriptorSwitch* parameter is just a reference to this content. If the application is not in the private base, or the destination descriptor base does not exist, ignores the command. If the <descriptorSwitch> element already exists, first remove it.
removeDescriptorSwitch (baseId, documentId, descriptorSwitchId)	0x1A	Removes a <descriptorSwitch> element from the <descriptorBase> of an NCL document in an open private base. If the application is not in the private base, or the destination descriptor base does not exist, or the <descriptorSwitch> element does not exist, ignores the command.
addDescriptorBase (baseId, documentId, xmlDescriptorBase)	0x1B	Adds a <descriptorBase> element to the <head> element of an NCL document in an open private base. If the XML specification of the descriptorBase is sent in a file system apart; the *xmlDescriptorBase* parameter is just a reference to this content.

Command string	Command tag	Description
		If the application is not in the private base, ignores the command. If the <descriptorBase> element already exists, first remove it.
removeDescriptorBase (baseId, documentId, descriptorBaseId)	0x1C	Removes a <descriptorBase> element from the <head> element of an NCL document in an open private base. If the application is not in the private base, or the <descriptorBase> element does not exist, ignores the command.
addTransition (baseId, documentId, xmlTransition)	0x1D	Adds a <transition> element to the <transitionBase> of an NCL document in an open private base. If the application is not in the private base, or the destination transition base does not exist, ignores the command. If the <transition> element already exists, first remove it.
removeTransition (baseId, documentId, transitionId)	0x1E	Removes a <transition> element from the <transitionBase> of an NCL document in an open private base. If the application is not in the private base, or the destination transition base does not exist, or the <transition> element does not exist, ignores the command.
addTransitionBase (baseId, documentId, xmlTransitionBase)	0x1F	Adds a <transitionBase> element to the <head> element of an NCL document in an open private base. If the XML specification of the transitionBase is sent in a file system apart; the *xmlTransitionBase* parameter is just a reference to this content. If the application is not in the private base, ignores the command. If the <transitionBase> element already exists, first remove it.
removeTransitionBase (baseId, documentId, transitionBaseId)	0x20	Removes a <transitionBase> element from the <head> element of an NCL document in an open private base. If the application is not in the private base, or the <transitionBase> element does not exist, ignores the command.
addImportBase (baseId, documentId, docBaseId, xmlImportBase)	0x21	Adds an <importBase> element to the base (<regionBase>, <descriptorBase>, <ruleBase>, <transitionBase>, or <connectorBase> element) of an NCL document in an open private base. If the application is not in the private base, or the destination base does not exist, ignores the command.

Command string	**Command tag**	**Description**
		If the <importBase> element already exists, first remove it.
removeImportBase (baseId, documentId, docBaseId, documentURI)	0x22	Removes an <importBase> element, whose documentURI attribute is identified by the *documentURI* parameter, from the base (<regionBase>, <descriptorBase>, <ruleBase>, <transitionBase>, or <connectorBase> element) of an NCL document in an open private base. If the application is not in the private base, or the destination base does not exist, or the <importBase> element does not exist, ignores the command.
addImportedDocumentBase (baseId, documentId, xmlImportedDocumentBase)	0x23	Adds an <importedDocumentBase> element to the <head> element of an NCL document in an open private base. If the application is not in the private base, ignores the command. If the <importDocumentBase> element already exists, first remove it.
removeImportedDocumentBase (baseId, documentId, importedDocumentBaseId)	0x24	Removes an <importedDocumentBase> element from the <head> element of an NCL document in an open private base. If the application is not in the private base, or the <importDocumentBase> element does not exist, ignores the command.
addImportNCL (baseId, documentId, xmlImportNCL)	0x25	Adds a <importNCL> element to the <importedDocumentBase > element of an NCL document in an open private base. If the application is not in the private base, or the destination imported document base does not exist, ignores the command. If the <importNCL> element already exists, first remove it.
removeImportNCL (baseId, documentId, documentURI)	0x26	Removes an <importNCL> element, whose documentURI attribute is identified by the *documentURI* parameter, from the <importedDocumentBase > element of an NCL document in an open private base. If the application is not in the private base, or the destination imported document base does not exist, or the <importNCL> element does not exist, ignores the command.
addNode (baseId, documentId, compositeId, {uri, id}+, meta)	0x27	Adds a node (<media>, <context>, or <switch> element) to a composite node (<body>, <context>, or <switch> element) of an NCL application in an open private base. The XML specification of the node and its media content may be: i) sent in the datacast network as a set of pushed

Command string	Command tag	Description
		files; the {uri, id}pairs is used to relate file paths in the NCL document specification of the node with their respective locations in a transport system; NOTE. The set of reference pairs shall be sufficient to enable Ginga mapping any file reference present in the XML specification to its concrete location in the receiver memory. ii) received from an IP network as a set of pulled files, or may be files already present in the receiver; for these pulled files, no {uri, id} pairs have to be sent, except the {uri, “null”} pair associated with the XML node specification that the Editing Command requests to be added in compositeId, if this XML document is not received as a pushed file. The *meta* parameter contains information of accessible network domains with permission information of each domain. If the application is not in the private base, or the destination composite node does not exist, or the node is being presented, ignores the command. If the node already exists and it is not being presented, first remove it.
removeNode(baseId, documentId, compositeId, nodeId)	0x28	Removes a node (<media>, <context>, or <switch> element) from a composite node (<body>, <context>, or <switch> element) of an NCL application in an open private base. If the application is not in the private base, or the destination composite node does not exist, or the node does not exist, or the node is being presented, ignores the command.
addInterface (baseId, documentId, nodeId, xmlInterface)	0x29	Adds an interface (<port>, <area>, <property>, or <switchPort>) to a node (<media>, <body>, <context>, or <switch> element) of an NCL application in an open private base. If the application is not in the private base, or the destination node does not exist, or the interface already exists (including properties defined by the system) and the destination node is being presented, ignores the command. If the interface already exists and the destination node is not being presented, first remove it. When an interface is included, only changes on its corresponding event state machine that will occur after the moment of the includion shall be reported, following the rules of Clause 8.
removeInterface (baseId, documentId, nodeId, interfaceId)	0x2A	Removes an interface (<port>, <area>, <property>, or <switchPort>) from a node (<media>, <body>, <context>, or <switch> element) of an NCL

Command string	Command tag	Description
		application in an open private base. The *interfaceID* shall identify a <property> element's name attribute or a <port>, <area>, or <switchPort> element's id attribute. If the application is not in the private base, or the destination node does not exist, or the interface does not exist, or the destination node is being presented, ignores the command.
addLink (baseId, documentId, compositeId, xmlLink)	0x2B	Adds a <link> element to a composite node (<body>, <context>, or <switch> element) of an NCL application in an open private base. If the application is not in the private base, or the destination composite node does not exist, or the link is being computed (has been already triggered), ignores the command. If the <link> element already exists and it is not being computed, first remove it.
removeLink (baseId, documentId, compositeId, linkId)	0x2C	Removes a <link> element from a composite node (<body>, <context>, or <switch> element) of an NCL application in an open private base. If the application is not in the private base, or the destination composite node does not exist, or the <link> element does not exist, or the <link> element is being computed (has been already triggered), ignores the command.

The identifiers used in the commands shall be in agreement with Table 9.3.

Table 9.3 – Identifiers used in Editing Commands

Identifiers	Definition
baseId	The identifier_of_a_tuned_TV_channel (set of services), or the identifier_of_a_tuned_TV_channel.identifier_of_one_of_its_services as specified by the (broadcast, or broadband, or integrated) DTV system, or the identifier_of_a_Web_service. When the parameter is specified as "null", it shall assumed the tuned TV channel or Web service identifier through which the *nclEditingCommand* was issued. When the *baseId* parameter of an *nclEditingCommand* coming from an NCLua object running in a certain private base is specified as "null", it shall assumed the same *baseId* value of this private base.
documentId	The *id* attribute of an <ncl> element of an NCL document
nptBaseId	The identifier of an NPT time base
nptTrigger	A value of NPT
regionId	The *id* attribute of a <region> element of an NCL document
ruleId	The *id* attribute of a <rule> element of an NCL document

Identifiers	Definition
connectorId	The *id* attribute of a <connector> element of an NCL document
descriptorId	The *id* attribute of a <descriptor> element of an NCL document
descriptorSwitchId	The *id* attribute of a <descriptorSwitch> element of an NCL document.
transitionId	The *id* attribute of a <transition> element of an NCL document
regionBaseId	The *id* attribute of a <regionBase> element of an NCL document
ruleBaseId	The *id* attribute of a <ruleBase> element of an NCL document
connectorBaseId	The *id* attribute of a <connectorBase> element of an NCL document.
descriptorBaseId	The *id* attribute of a <descriptorBase> element of an NCL document
transitionBaseId	The *id* attribute of a <transitionBase> element of an NCL document
docBaseId	The *id* attribute of a <regionBase>, <ruleBase>, <connectorBase>, <descriptorBase>, or <transitionBase> element of an NCL document
documentURI	The *documentURI* attribute of an <importBase> element or an <importNCL> element of an NCL document
importedDocumentBaseId	The *id* attribute of a <importedDocumentBase> element of an NCL document
compositeID	The *id* attribute of a <body>, <context> or <switch> element of an NCL document. If the parameter is specified as "null", the <body> element shall be assumed as the composite to be edited.
nodeId	The *id* attribute of a <body>, <context>, <switch> or <media> element of an NCL document
interfaceId	The *id* attribute of a <port>, <area>, <property> or <switchPort> element of an NCL document
linkId	The *id* attribute of a <link> element of an NCL document
propertyId	The *id* attribute of a <property> or <switchPort> element of an NCL document

9.2 Command parameters XML schemas

NCL entities used in Editing Commands shall be a document in conformance with the NCL 3.1 Command profile defined by the XML Schema that is found in the electronic attachment **NCL31EdCommand.xsd** to this Draft Recommendation.

Note that different from NCL documents, several <ncl> elements may be the root element in the XML command parameters.

9.3 NCL Editing Commands in Ginga-NCL

Some constraints are defined by Ginga-NCL considering private bases.

Ginga associates at least one private base, the *default private base*, with each IBB (Integrated Broadband Broadcast) Service. An IBB Service may be a broadcast TV channel, or an IPTV channel, or a Web service that may use resources from several broadcast and broadband networks. When an IBB service is tuned, its corresponding default private base is opened and activated by the Private Base Manager. Other private bases can then be opened (or created), but at most one associated with each service of an IBB Service (for example, a service of a broadcast TV Channel). When an IBB Service has just one service it shall have just one private base associated with it, the default private base.

NCL Editing Commands that manipulates private bases (the first subset of commands) created for an IBB Service shall be considered only if they come from the tuned IBB Service. The other NCL Editing Commands targeting a private base of an IBB Service shall be considered only if they come from the tuned IBB service, or result from running NCL applications include in the private bases associated with this IBB Service.

NOTE. The private base associated with an IBB Service shall have the identifier (*baseId* parameter) equal to IBB Service identifier (identifier_of_a_tuned_TV_channel). The possible private base associated with an IBB Service's service shall have the identifier (*baseId* parameter) equal to the "vale of the identifier one of its services" prefixed by "value of the IBB Service identifier" (identifier_of_a_tuned_TV_channel.identifier_of_one_of_its_services).

NCL resident applications are managed in a specific private base. Resident application can only issue NCL Editing Commands targeting targeting this specific private base.

The number of private bases that may be kept open is a specific middleware implementation decision.

In brief: NCL Editing Commands sent from an IBB Service can control only the private bases created by previous NCL Editing Commands sent through the open IBB Service and the default private base associated with the IBB Service. In other words, NCL Editing Commands coming from an IBB Service have no effect on private bases associated with other IBB Services. For example, NCL Editing Commands coming from a tuned broadcast TV channel have no effect on private bases associated to other TV channels.

NCLua events (NCL Editing Command events) generated by NCLua objects running in private bases may control just these private bases.

In Ginga-NCL, event descriptors (defined in 9.1) can be transported using any protocol, in special those for pushed data transmission.

In environments that adopt DSM-CC for digital media transport, Ginga-NCL defines how this can be done. In this case, NCL Editing Commands are transported in DSM-CC stream-event descriptors. As specified on [ISO/IEC 13818-6], a DSM-CC stream-event descriptor has a very similar structure to the event descriptor presented in Table 9.1 (see Table 9.4).

Table 9.4 - Editing command stream event descriptor

Syntax	Number of bits
StreamEventDescriptor ( ) {	
descriptorTag	8
descriptorLength	8
eventId	16
reserved	31
eventNPT	33
privateDataLength	8
commandTag	8
sequenceNumber	7
finalFlag	1
privateDataPayload	8 to 2008
FCS	8

Syntax	Number of bits
}	

Several alternatives have been defined by Ginga-NCL to transport unsolicited NCL Editing Commands parameters. All alternatives are optional, but if one of them is chosen, it shall be in agreement with this Draft Recommendation, as stated in Clauses 9.3.1 and 9.3.2.

9.3.1 DSM-CC transport of Editing Command parameters using object carousels

The DSM-CC object carousel protocol allows the cyclical transmission of stream event objects and file systems. Stream event objects are used to map event names into event ids defined in event descriptors. The Private Base Manager should register itself as a listener of event descriptors it handles using event names; in the case of Editing Commands the name "*nclEditingCommand*".

Besides stream event objects the DSM-CC object carousel protocol can also be used to transport files organized in directories. A DSM-CC demultiplexer is responsible for mounting the file system at the receiver device. XML-based *command parameters* specified as XML documents (NCL documents or NCL entities to be added) can thus be organized in file system structures to be transported in these carousels, as an alternative to the direct transportation in the payload of stream event descriptors. A DSM-CC carousel generator is used to join the file systems and stream event objects into data elementary streams.

Thus, when an NCL Editing Command needs to be sent, a DSM-CC stream event object shall be created, mapping the string "*nclEditingCommand*" into a selected event id, and shall be put in a DSM-CC object carousel sent in a elementary stream of type = "0x0B". If DSM-CC stream event descriptors are used, one or more of these descriptors, with a previous selected event 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 may be postponed to be executed at a specific time. The Private Base Manager shall register itself as an "*nclEditingCommand*" listener in order to be notified when this kind of stream event arrives.

The *commandTag* of the received stream event descriptor 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 event descriptor payload. Otherwise, the *privateDataPayload* field carries a set of reference pairs. In this case, the XML specification shall be placed in the same object carousel that carries the stream event object. The *uri* parameter of the first reference pair shall have the schema (optional) and the absolute path of the XML specification (the path in the data server). The corresponding *id* parameter in the pair shall refer to the XML specification IOR (carouselId, moduleId, objectKey; see [ISO/IEC 13818-6]) in the object carousel. If other file systems need to be transmitted using other object carousels to complete the Editing Command with media contents (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 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) of any root child file or child directory in the object carousel (the carousel service gateway).

9.3.2 Transport of editing commands parameters using specific Ginga-NCL structures

Three data structure types are defined to support the transmission of NCL Editing Command parameters: maps, metadata and data files.

For map structures, the *mappingType* field identifies the map type. If the *mappingType* is equal to "0x01" ("events"), an event-map is characterized. In this case, a list of event identifiers comes after the *mappingType* field, as defined in Table 9.5. Other *mappingType* values may also be defined.

Table 9.5 – List of event identifiers defined by the mapping structure

Syntax	Number of bits
mappingStructure ( ) {	
mappingType	8
for (i=1; i<N; i++){	
eventId	8
eventNameLength	8
eventName	8 to 255
}	
}	

Maps of type "events" (*event maps*) are used to map event names into *eventIds* of event descriptors (see Table 9.1). Event maps are used to inform which events shall be received. Event names allow specifying types of events, offering a higher abstraction level for middleware applications. The Private Base Manager, as well as NCL imperative and declarative media objects, should register themselves as listeners of events they handle, using event names.

When an NCL Editing Command needs to be sent, an event map shall be created, mapping the string "*nclEditingCommand*" into a selected event descriptor id (see Table 9.1). One or more event descriptors with the previous selected *eventId* are then created and sent (for example, it can be sent in an MPEG-2 TS elementary stream, or using some protocol for pushed data transmission). These event descriptors may have their time reference set to zero, but may be postponed to be executed at a specific time. The Private Base Manager shall register itself as an "*nclEditingCommand*" listener in order to be notified when this type of event arrives.

Each data file structure is indeed a file content that composes an NCL application or an NCL entity specification: the XML specification file or its media content files (video, audio, text, image, ncl, lua, etc.).

A metadata structure is an XML document, as defined by the schema in the electronic attachment file **NCLSectionMetadataFile.xsd**. Note that the schema defines, for each pushed file, an association between its location in a transport system (transport system identification (*component_tag* attribute) and the file identification in the transport system (*structureId* attribute)), and its Universal Resource Identifier (*uri* attribute).

For each NCL Document file or other XML Document files used in *addDocument* or *addNode* Editing Command parameters, at least one metadata structure shall be defined. Only one NCL application file or XML document file representing an NCL node to be inserted may be defined in a metadata structure. More precisely, there can be only one <pushedRoot> element in a metadata XML document. However, an NCL application (and its content files) or an XML document (and its content files) may extend for more than one metadata structure. Moreover, there may also be a metadata structure without any NCL application or XML document described in its <pushedRoot> and <pushedData> elements.

Some alternatives have been defined by Ginga-NCL to transport these three aforementioned data structures. All alternatives are optional, but if one of them is chosen, it shall be in agreement with this Recommendation, as stated in 9.3.2.1.

9.3.2.1 Transporting in unsolicited NCL Sections

The use of NCL Sections may allow the transmission of the three data structure types: maps, metadata and data files. Every NCL Section contains data of a single structure. However, one structure may extend through several Sections. Every data structure can be transmitted in any order and how many times it is necessary. All NCL Section transmitted in sequence compound an NCL Section stream.

NCL Sections have a header and a payload. The first byte of an NCL Section payload identifies the structure type (0x01 for metadata; 0x02 for data files, and 0x03 for event-map). The second payload byte carries the unique identifier of the structure (*structureId*).

The NCL Section stream and the structure identifier are those that are associated by the metadata structure to a file locator (URL), through the *component_tag* and *structureId* attributes of the <pushedRoot> and <pushedData> elements.

After the second byte comes a serialized data structure that can be a mappingStructure (as depicted by Table 9.3), or a metadata structure (an XML document), or a data file structure (a serialized file content). The NCL Section demultiplexer is responsible for mounting the application's structure at the receiver device.

In the same NCL Section stream that carries the XML specification (the NCL Document file or other XML Document file used in NCL Editing Commands), an event-map file should be transmitted in order to map the name "*nclEditingCommand*" to the *eventId* of the event descriptor, which shall carry an NCL Editing Command, as described in 9.1. The *privateDataPayload* of the event descriptor shall carry a set of {uri, id} reference pairs. The *uri* parameters are always "null". In the case of *addDocument* and *addNode* commands, the *id* parameter of the first pair shall identify the NCLSection stream ("component_tag") and its metadata structure ("structureId") that carries the absolute path of the NCL document or the NCL node specification (the path in the data server) and the corresponding related structure ("structureId") transported in NCL Sections of the same NCL Section stream. If other additional metadata structures are used in order to complete the *addDocument* or *addNode* command, other {uri, id} pairs shall be present in the command. In this case, the *uri* parameter shall also be "null" and the corresponding id parameter in the pair shall refer to the component_tag and the corresponding metadata structureId.

NCL Sections can be wrapped in other protocol data format like FLUTE packets, or MPEG-2 specific Section type.

NCL Sections can also transport the aforementioned data structures encapsulated in other data structures. For example, MPEG-2 MPE (Multi-protocol Encapsulation) can be used and be wrapped in MPEG-2 Sections; in this case, NCL Sections are MPEG-2 Datagram Sections.

Instead of transporting metadata structures directly inside NCL Sections, a second alternative procedure is treating metadata structures as command parameters, which are transported in the *privateDataPayload* field of an event descriptor.

In this situation, the set of {uri, id} parameter pairs of *addDocument* and *addNode* command is substituted by metadata structure parameters that define a set of {"uri", "component_tag, structureId"} pairs for each pushed file.

Still another alternative is transporting NCL Sections containing metadata structures as MPEG-2 Metadata Sections, transported in MPEG-2 stream type = "0x16".

10 Lua imperative objects in NCL presentations

The scripting language adopted by Ginga-NCL to implement imperative objects in NCL documents is *Lua* (<media> elements of type "application/x-ginga-NCLua"). In the NCL Recommendation, the support to NCLua objects (<media> element of "application/x-ginga-NCLua" type) is optional. Any imperative scripting language could be used as NCL scripting language. However, in the Ginga-NCL Recommendation, Lua is required as an NCL scripting language. The complete definition of Lua is presented in [b_H.IPTV-MAFR.14].

10.1 Lua language - functions removed from the standard Lua library

The following functions are platform dependent and were removed in the implementation:

1) in module *package*: *loadlib*;
2) in module *os*: *clock, execute, exit, getenv, remove, rename, tmpname* and *setlocale*;
3) in module *debug*: all functions.

10.2 Execution model

The lifecycle of an NCLua object is controlled by the NCL Player. The NCL Player is responsible for triggering the execution of an NCLua object and for mediating the communication between an NCLua object and other nodes in an NCL application, as defined in Clause 8.5.

As with all media object players, once instantiated, the Lua player shall execute an initialization procedure. However, different from other media players, this initialization code is specified by the NCLua object author. This initialization procedure is executed only once, for each instance, and creates functions and objects that may be used during the NCLua object execution and, in particular, registers one (or more) event handler for communication with the NCL formatter.

After the initialization, the execution of the NCLua object becomes event oriented in both directions; i.e., any action commanded by the NCL Player reaches the registered event handlers, and any NCL event state change notification is sent as an event to the NCL Player (as for example, the natural end of a procedure execution). After the initialization, the Lua Player is then ready to perform any *start* operation (see Clause 8.3).

10.3 Additional modules

Besides the Lua standard library, the following modules shall be implemented and automatically loaded:

1) module *canvas*: offers an API to draw graphical primitives and manipulate images;
2) module *event*: allows NCLua applications to communicate with the middleware through events (NCL, pointer and key events);
3) module *settings*: exports a table with variables defined by the NCL document author and reserved environment variables contained in an "application/x-ncl-settings" node;
4) module *persistent*: exports a table with persistent variables, which may be manipulated only by imperative objects.

The definition of the prototypeof each function, in the above modules, uses the following naming schema:

funcname (*arglist* [*optarglist*]) -> *retlist* where *funcname* denotes the name of the function, *parname* denote the list of required parameters, *optarglist* denote the list of optional parameters, and *retlist* denote the list of results returned by the function. The lists *arglist*, *optarglist*, and *retlist* are variable size lists of the form

$$var_{11}, ..., var_{1i}:type_1; ...; var_{n1}, ..., var_{nj}:type_n$$

where *var* denotes a variable name and *type* denotes a type name, for some *i*, *j*, and *n* greater than or equal to zero. Thus, for example, the prototype

func (x, y:number; s, t:string; [b:boolean]) -> t:table

denotes a function, called *func*, that expects the required number arguments *x* and *y*, the required string arguments *s* and *t*, and the optional boolean argument *b*, and that returns a table.

The functions of the canvas; event; settings; persistent modules shall use the following policy for error handling (in this order):

1. If the number of parameters of a call is greater than the one expected, the functions shall ignore the additional parameters;
2. If a parameter *P* of a call is omitted, the function shall assume the default value of *P*, as defined in this Draft Recommendation;
3. If the type of a parameter *P* is different from the one expected, the function shall convert *P* to the expected type, if the conversion is possible. Otherwise, the function shall produce an error.
4. If the value of a parameter *P* is invalid, the function shall assume the recommended value for *P*, as defined in this Draft Recommendation. If there is no recommended value, the function must produce an error;
5. If the parameters of a call are valid and the call fails, the function must produce an error.

In items 3 to 5, “produce an error” means:

1. If the function has an error status return, for example, event.post() -, then the function returns the value that indicates the error followed by an error message (a string);
2. If the function has no error status return, then it triggers a Lua exception - error() or lua_error() call – having an error message (a string). If the script does not capture this exception, –pcall() call– the associated media object shall be aborted.

10.3.1 The *canvas* module

10.3.1.1 The canvas object

When an NCLua media object is initialized, the corresponding region of the <media> element (of type “application/x-ginga-NCLua”) is available as the global *canvas* variable for the Lua script. If the <media> element has no associated region defined (*left, right, top,* and *bottom* properties), then the value for *canvas* is set to “nil”.

As an example, assume an NCL document region defined as:

```
<region id="luaRegion" width="300" height="100" top="200" left="20"/>
```

The `canvas` variable in an NCLua media object referring to “luaRegion” is bound to a canvas object of size 300x100, associated with the specified region at position (20,200).

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

A canvas keeps in its state a set of attributes under which the drawing primitives operate. For instance, if its color attribute is blue, a call to `canvas:drawLine()` will draw a blue line on the canvas.

The coordinates are always relative to the top-leftmost point in canvas, i.e., coordinate (0,0).

10.3.1.2 Constructors

From any canvas object, it is possible to create new canvas and combine them through composite operations.

canvas.new (image_path: string) -> canvas: object

Arguments

image_path	Path to the image file

Return values

canvas	Canvas representing the image

Description

Returns a new canvas whose content is the image received as a parameter.If the *image_path* path is invalid, the function shall return nil and an error message.

canvas.new (width, height: number) -> canvas: object

Arguments

width	Canvas width
height	Canvas height

Return values

canvas	New canvas

Description

Returns a new canvas of the given size.

Initially, all pixels shall be transparent.

If *width, height* < 0, the function shall assume 0.

10.3.1.3 Attributes

All attribute methods have the prefix “attr” and are used to get and set attributes (with the exceptions specified).

If a method is invoked without input parameters, the current attribute value is returnedIf a method is invoked with input parameters, these parameters are used as the new attribute values.

canvas:attrSize () -> width, height: number

Arguments

Return values

width	Canvas width
height	Canvas height

Description

Returns the canvas dimensions.

Note that it is not possible to change the dimensions of an existing canvas.

canvas:attrColor (R, G, B, A:number)

Arguments

R	Color red component
G	Color green component
B	Color blue component
A	Color alpha component

Description

Change canvas’ attribute color.

The colors are given in RGBA, where A varies from 0 (full transparency) to 255 (full opacity).

The primitives (see Clause 10.3.1.4) are drawn with the color set to this attribute.

The initial value is black ‘0,0,0,255’.

If *r, g, b, a* < 0, the function should assume 0. If r, g, b, a > 255, the function shall assume 255.

canvas:attrColor (clr_name:string)

Arguments

clr_name	Color name

Change canvas’ attribute color.

The colors are given as a string corresponding to one of the 16 pre-defined NCL colors:

“white”, “aqua”, “lime”, “yellow”, “red”, “fuchsia”, “purple”, “maroon”,

“blue”, “navy”, “teal”, “green”, “olive”, “silver”, “gray”, “black”

The values given have their alpha equal to full opacity (i.e., “A = 255”).

The primitives (see Clause 10.3.1.4) are drawn with the color set in this attribute.

If *clr_name* is different from every valid color, the function shall assume “black”.

canvas:attrColor () -> R, G, B, A:number

Return values

R	Color red component
G	Color green component
B	Color blue component
A	Color alpha component

Description

Retorns the canvas' color.

canvas:attrFont (face:string; size:number; [style:string])

Arguments

face	Font name
size	Font size
style	Font style

Description

Changes canvas' font attribute.

The following fonts shall be available: "Tiresias" and "Verdana".

The size is in pixels, and it represents the maximum height of a line written with the chosen font.

The possible style values are: "bold", "italic", or "bold-italic". F no style is given, no style is used.

Any invalid input value shall raise an error.

The initial font value is undefined.

If *face* is not supported or *face*=nil, the function shall assume "Tiresias" (*default)*. If *size* < 0, the function shall assume 0. If *style* is different from "bold", "italic" and "bold-italic", the function shall assume *style*=nil.

canvas:attrFont () -> face:string; size:number; style:string

Return values

face	Font name
size	Font size
style	Font style

Description

Returns the canvas font.

canvas:attrClip (x, y, width, height:number)

Arguments

x	Clipping area coordinate
y	Clipping area coordinate
width	Clipping area width
height	Clipping area height

Description

Changes the canvas clipping area.

The drawing primitives (see Clause 10.3.1.4) and the method `canvas:compose()` only operate inside this clipping region.

The initial value is the whole canvas.

If *x, y, width, height* < 0, the function shall assume 0. If *x* is greater than the canvas width (canvas.width), the function shall assume canvas.width. If y is greater than the canvas height (canvas.height), the function shall assume canvas.height. If *x*+*width* > canvas.width, the function shall assume *width*=canvas.width–*x.* If *y*+*height* > canvas.height, the function shall assume *height*=canvas.height–*y*.

canvas:attrClip () -> x, y, width, height:number

Return values

x	Clipping area coordinate
y	Clipping area coordinate
width	Clipping area width
height	Clipping area height

Description

Returns the canvas clipping area.

canvas:attrCrop (x, y, w, h:number)

Arguments

x	Crop region coordinate
y	Crop region coordinate
w	Crop region width
h	Crop region height

Description

Sets the canvas crop region used when the canvas is composed.

The initial *crop* region is the whole canvas.

If *x, y, width, height* < 0, the function shall assume 0. If *x* is greater than the canvas width (canvas.width), the function shall assume canvas.width. If y is greater than the canvas height (canvas.height), the function shall assume canvas.height. If *x*+*width* > canvas.width, the function shall assume *width*=canvas.width–*x*. If *y*+*height* > canvas.height, the function shall assume *height*=canvas.height–*y*.

canvas:attrCrop () -> x, y, w, h:number

Return values

x	Crop region coordinate
y	Crop region coordinate
w	Crop region width
h	Crop region height

Description

Returns the canvas *crop* region.

canvas:attrFlip (horiz, vert:boolean)

Arguments

horiz	If canvas should be flipped horizontally
vert	If canvas should be flipped vertically

Description

Sets the canvas flipping mode used when the canvas is composed.

canvas:attrFlip () -> horiz, vert:boolean

Return values

horiz	If canvas is flipped horizontally
vert	If canvas is flipped vertically

Description

Returns the current canvas' flipping setup.

canvas:attrOpacity (opacity:number)

Argument

opacity	Canvas opacity

Description

Sets the canvas opacity used when the canvas is composed.

The opacity values varies between 0 (full transparency) to 255 (full opacity).

If *opacity* < 0, the function shall assume 0. If *opacity* > 255, the function shall assume 255.

canvas:attrOpacity () -> opacity:number

Return value

opacity	Canvas opacity

Description

Returns the current canvas opacity.

canvas:attrRotation (degrees:number)

Argument

degrees	Canvas rotation in degrees.

Description

Sets the canvas rotation attribute used when the canvas is composed.

The rotation value must be multiple of 90^o, and follows the clockwise motion.

canvas:attrRotation () -> degrees:number

Return value

degrees	Canvas rotation in degrees

Description

Returns the current canvas rotation value.

canvas:attrScale (w, h:number)

Arguments

w	Canvas scaling width
h	Canvas scaling height

Description

Sets the canvas scale used when the canvas is composed.

If the width parameter is nil, then the scaling width is computed from the given height by assuming that the aspect ratio is kept. If the height parameter is nil or is omitted, then the scaling height is computed from the given width by assuming that the aspect ratio is kept.

The scaling attribute is independent of the size attribute, which shall remain the same.

If *width*, *height* < 0, the function shall assume 0. If the implicit parameter "canvas" is the main canvas, the function must produce an error. Note that "scale" a canvas means to alter the scale (*default* 1.0) used when this canvas is composed with other canvas objects. Thus, the canvas:attrScale() function does not change the pixel matrix of the canvas. It only specifies the (horizontal and vertical) factors used to interpolate the pixel matrix of this canvas at a moment immediately before the composition operation.

canvas:attrScale () -> w, h:number

Return values

w	Canvas scaling width
h	Canvas scaling height

Description

Returns the current canvas scaling values.

10.3.1.4 Primitives

The following methods take into account the canvas attributes defined in the previous clause.

NOTE. In all primitives, the line width shall be assumed as 1 pixel.

canvas:drawLine (x1, y1, x2, y2:number)

Arguments

x1	Line extremity 1st coordinate
y1	Line extremity 1st coordinate
x2	Line extremity 2nd coordinate
y2	Line extremity 2nd coordinate

Description

Draws a line with its extremities in coordinates (x1,y1) and (x2,y2).

If the draw operation exceeds the canvas size, the function shall draw only the points that are within the canvas.

canvas:drawRect (mode:string; x, y, width, height:number)

Arguments

mode	Drawing mode
x	Rectangle coordinate
y	Rectangle coordinate
width	Rectangle width
height	Rectangle height

Description

Method for rectangle drawing and filling.

The parameter mode may receive “frame” or “fill” values, for drawing the rectangle without filling it or filling it, respectively.

If *mode* is different from “frame” and “fill”, the function shall assume “fill”. If *width, height* < 0, the function shall assume 0. If the draw operation exceeds the canvas size, the function shall draw only the points that are within the canvas.

canvas:drawRoundRect (mode:string; x, y, width, height, arcWidth, arcHeight:number)

Arguments

mode	Drawing mode
x	Rectangle coordinate
y	Rectangle coordinate
width	Rectangle width
height	Rectangle height
arcWidth	Rounded edge arc width
arcHeight	Rounded edge arc height

Description

Function for rounded rectangle drawing and filling.

The parameter *mode* may be *“frame”* to draw the rectangle frame or *“fill”* to fill it.

If *arc_width, arc_height* < 0, the function shall assume 0. The handling of the other parameters is similar to the canvas:drawRect() handling.

canvas:drawPolygon (mode:string) -> drawer:function

Arguments

mode	Drawing mode

Return values

f	Drawing function

Description

Method for polygon drawing and filling.

The parameter mode may receive the “open” value, to draw the polygon not linking the last point to the first; the “close” value, to to draw the polygon linking the last point to the first; or the “fill” value, to draw the polygon linking the last point to the first and painting the region inside.

The function canvas:drawPolygon returns an anonymous function “drawer” with the signature:

function (x, y) end

The returned function, receives the next polygon vertex coordinates and returns itself as the result. This recurrent procedure allows the idiom:

canvas:drawPolygon('fill')(1,1)(10,1)(10,10)(1,10)()

When the function “drawer” receives *nil* as input, it completes the chained operation. Any subsequent call shall raise an error.

If *mode* is different from “open”, “close” and “fill”, the function shall assume “fill”. If the draw operation exceeds the canvas size, the function shall draw only the points that are within the canvas. If the drawer() function, returned by canvas:drawPolygon(), is called more than once with the “nil” parameter, the function shall produce an error.

canvas:drawEllipse (mode:string; xc, yc, width, height, ang_start, ang_end:number)

Arguments

mode	Drawing mode
xc	Ellipse center
yc	Ellipse center
width	Ellipse width
height	Ellipse height
ang_start	Starting angle
ang_end	Ending angle

Description

Draws an ellipse and other similar primitives as circle, arcs and sectors.

The parameter mode may receive “arc” to only draw the circunference or “fill” for internal painting.

The angle units shall be assumed as degrees. The 0 degree angle is in the higher Y coordinate of the ellipse and the angle progression follows the clockwise motion.

If *mode* is different from “arc” and “fill”, the function shall assume “fill”. Se *width, height* < 0, the function shall assume 0. If the draw operation exceeds the canvas size, the function shall draw only the points that are within the canvas. If ang_start or ang_end are *nil*, an error condition shall be reported.

canvas:drawText (x, y: number; text: string)

Arguments

x	Text coordinate
y	Text coordinate
text	Text do be drawn

Description

Draws the given text at coordinate (x,y) in the canvas, using the font set by `canvas:attrFont()`.

If the draw operation exceeds the canvas size, the function shall draw only the points that are within the canvas.

10.3.1.5 Miscellaneous

canvas:clear ([x, y, w, h:number])

Arguments

x	Clear area coordinate
y	Clear area coordinate
w	Clear area width
h	Clear area height

Description

Clears the canvas with the color set by *canvas:attrColor*.

If the area parameters are not given, the whole canvas surface should be cleared.

If *width, height* < 0, the function should assume 0. If the draw operation exceeds the canvas size, the function shall clear only the area that is within the canvas.

canvas:flush ()

Description

Flushes the canvas after a set of drawing and composite operations.

It is enough to call this method only once, after a sequence of operations.

canvas:compose (x, y:number; src:canvas; [src_x, src_y, src_width, src_height:number])

Arguments

x	Position of the composition
y	Position of the composition
src	Canvas to compose with
src_x	Position in the canvas src
src_y	Position in the canvas src
src_width	Composition width in the canvas src
src_height	Composition height in the canvas src

Description

Composes the canvas *src* on the current canvas (the implicit first argument) at position (x,y).

The other parameters are optional and indicate which region in the canvas src is used to compose with. When absent the whole canvas is used.

This operation calls *src:flush()* automatically before the composition.

The composition operation satisfies the following equation:

$$\alpha_R = \alpha_A + \alpha_B \cdot (1 - \alpha_A)$$

$$c_R = [c_A\alpha_A + c_B\alpha_B \cdot (1 - \alpha_A)] \div \alpha_R$$

in which:

c_R = resulting color normalized between 0 and 1

α_R = resulting alfa component normalized between 0 and 1

c_A = color of the source canvas (src) normalized between 0 and 1

α_A = alpha component of the source canvas (src) normalized between 0 and 1

c_B = color of the destination canvas (canvas) normalized between 0 and 1

α_B = alpha component of the destination canvas (canvas) normalized between 0 and 1

After the operation, the destination canvas has the resulting content and the canvas src remains intact.

If *src_width, src_height* < 0, the function shall assume 0. If the draw operation exceeds the canvas size, the function shall draw only the points that are within the canvas.

canvas:pixel (x, y, R, G, B, A:number)

Arguments

x	Pixel position
y	Pixel position
R	Color red component
G	Color green component
B	Color blue component
A	Color alpha component

Description

Changes the color of a given pixel.

If *r, g, b, a* < 0, the function shall assume 0. If *r, g, b, a* > 255, the function shall assume 255. If the draw operation exceeds the canvas size, the function shall draw only the points that are within the canvas.

canvas:pixel (x, y:number) -> R, G, B, A:number

Arguments

x	Pixel position
y	Pixel position

Return values

R	Color red component
G	Color green component
B	Color blue component
A	Color alpha component

Description

Returns the color of a given pixel.

canvas:measureText (text:string) -> dx, dy: number

Arguments

text	Text to be measured

Return values

dx	text width
dy	text height

Description

Returns the dimensions that the given text would have if it were drawed with the font configured by **`canvas:attrFont()`**.

The rendered text size depends only on the used text font.

10.3.2 The *event* module

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

An application may also use this mechanism internally, using the "user" event class.

The typical use of NCLua application is to handle events: NCL events (see Clause 7.2.12) or events coming from user interactions (for example, through the remote control).

During its initiation, before becoming event oriented, a Lua script may register an event handler function. After the initialization any action performed by the script will be in response to an event notified to the application, i.e., to the event handler function, if any.

```
=== example.lua ===
...                 -- initializing code
function handler (evt)
  ...               -- handler code
end
event.register(handler)   -- register as an event listener
=== end ===
```

Among the event types that may be received by the handler function are all those generated by the NCL Player. As aforementioned, Lua scripts are also capable of generating events, through a call to the `event.post(evt)` function.

10.3.2.1 Functions

event.post ([dst:string]; evt:event) -> sent:boolean; err_msg:string

Arguments

dst	Event destination
evt	Event to be posted

Return values

sent	If the event was successfully sent
err_msg	Error message in case of errors

Description

Posts the given event.

The parameter "dst" is the event destination and may assume the values "in" (send to itself) and "out" (send to the NCL Player). The default value is 'out'.

If *dst* is different from "in" and "out", the function shall assume "out". If *evt.class* is a known class and *evt* does not have the required fields of this class, or the values of some fields are invalid, the function shall produce an error.

event.timer (time:number, f:function) -> cancel:function

Arguments

time	Time in milliseconds
f	Callback function

Return value

cancel	Function to cancel the timer

Description

Creates a timer that expires after a timeout (in milliseconds) and then call the callback function f.

The signature of `f` is simple, no parameters are received or returned:

```
function f () end
```

The value of 0 milliseconds is valid. In this case, *event.timer()* shall return immediately and `f` shall be called as soon as possible.

If *time* < 0, the function should assume 0.

event.register ([pos:number]; f:function; [class:string]; […:*any*])

Arguments

pos	Register position
f	Callback function
class	Class filter
…	Class dependent filter

Description

Registers the given function as an event listener, i.e., whenever an event happens, `f` is called (the function `f` is an event handler).

The optional parameter *pos* indicates the position where *f* is registered. If *pos* is not given, the function is registered in the last position. The initial position is 1.

The optional parameter *class* indicates the class of events the function shall receive. If *class* is given, other class dependent filters may be defined. In this case, a *nil* value in any position indicates that the parameter shall not be filtered.

The signature for `f` is:

```
    function f (evt) end -> handled: boolean
```

Where `evt` is the event that triggers the function.

The function may return "true", to signalize that the event was handled and, therefore, should not be sent to other handlers.

It is recommended that the function, defined by the application, returns fast, since while it is running no other event may be processed.

The NCL Player shall notify the listeners in the order they were registered and if any of them returns true, the formatter shall not notify the remaining listeners.

If *pos* < *0* or *pos* is greater than the queue of registered handlers, the function registers *f* at the end of the queue. 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.

event.unregister (f:function)

Arguments

f	The handler function to be unregistered

Description

Unregisters the given function as a listener, i.e., new events will no longer be notified to `f`.

If `f` is not registered, the call shall be ignored.

event.uptime () -> ms:number

Return values

ms	Time in milliseconds

Description

Returns the number of milliseconds elapsed since the beginning of the NCLua execution.

10.3.2.2 Event classes

The function `event.post()` and the registered handler in `event.register()` receive events as parameters.

An event is described by a common Lua table, where the class field is mandatory and identifies the event class.

The following event classes are defined:

`key` class:

evt = { class="key", type:string, key:string}

* type may be "press" or "release".

* key is the value of the pressed or released key; the "event.keys" table holds all key codes available in the NCL.

EXAMPLE evt = { class="key", type="press", key="0"}

NOTE. The class dependent filters of the "key" class are *type* and *key*, in this order.
The NCLua *script* can post events of this class

`pointer` class:

evt = { class="pointer", type:string, x, y:number }

* type may be "press", "release", or "move"

* x and y refer to the coordinates of the pointer event occurrence

EXAMPLE evt = { class="pointer", type="press", x=20, y=50}

NOTE. The class dependent filter of the "pointer" class is *type*.

`ncl` class:

Relationships among NCL media nodes are based on events. Lua has access to these events through the `ncl Class`.

Events may transit in two directions: from the NCL Player to the NCLua Player and from the NCLua Player to the NCL Player. The NCL Player may send action events to change the state of the NCLua player, which, in turn, may trigger transition events to signal state changes.

In events of the "ncl" class , the *type* field shall assume one of the following three values: "presentation", "selection" or "attribution".

Events may be directed to specific anchors or to the *whole content anchor*. If the event is directed to the *whole content anchor*, the *label* field is equal to the empty string "".

In the case of an event generated by the NCL Player the *action* field shall have one of the following values: “start”, “stop”, “abort”, “pause” or “resume”.

Type ‘presentation’:

```
evt = { class='ncl', type='presentation', label='?', action='?'}
```

Type ‘attribution’:

```
evt = { class='ncl', type='attribution', name='?', action='?', value=’?’ }
```

For events generated by the Lua player, the *action* field shall assume one of the following values: “start”, “stop”, “abort”, “pause, or “resume”, depending on the *type* field

Type ‘presentation’:

```
evt = { class='ncl', type='presentation', label='?',
              action='start'/'stop'/'abort'/'pause'/'resume'}
```

Type ‘selection’:

```
evt = { class='ncl', type='selection', label='?', action='start'/'stop' }
```

Type ‘attribution’:

```
evt = {class='ncl', type='attribution', name='?',
              action='start'/'stop'/'abort'/'pause'/'resume', value=’?’}
```

NOTE. The class dependent filter of the “ncl”class are *type*, *label* or *name*, and *action*, in this order.

edit class:

This class reproduces the Editing Commands for the Private Base manager (see Clause 9). However, there is an important difference between Editing Commands coming from systems external to private bases, and the Editing Commands performed by Lua scripts (NCLua objects). The first ones may alter not only the NCL running application, but also the NCL document specification; i.e., in the end of the process a new NCL document is generated incorporating all editing results. On the other hand, Editing Commands coming from NCLua media objects may only alter the NCL running application. The original document is preserved.

Just like in other event classes, an editting command is represented by a Lua table. All events shall contain the *command* field: a string with the command name. The other fields depend on the command type (see Table 9.1). The only difference is with regard to the field that defines the reference pairs {uri,ior}, called *data* in the edit class. This field's values may be not only the reference pairs mentioned in Table 9.1, but also XML strings with the content to be added.

Example:

```
evt = {
    command = 'addNode',
    compositeId = 'someId',
    data = '<media>...',
}
```

The *baseId* e *documentId* fields are optional (when applicable) and they assume by default the base and document identifiers where the NCLua object is in execution.

The event describing the editting command may also receive a time reference as an optional parameter (optional parameters are indicated in the function signatures as arguments between brackets). This optional parameter may be used to specify the exact moment when the Editing Command shall be executed. If this parameter is not provided in the function call, the Editing Command shall be executed immediately. When provided, this parameter may have two different types of values, with two different meanings. If it is a number value, it defines the amount of time, in seconds, for how long the command shall be postponed. However, this parameter may also specify the exact moment, in absolute values, the command shall be executed. In this case, this parameter shall be a table value with the following fields: year (four digits), month (1-12), day (1-31), hour (0-23), min (0-59), sec (0-59), and isdst (a daylight saving flag, a boolean).

If the *data* field is not a valid XML, the event shall be ignored.

tcp class:

In order to send or receive a tcp data, a connection shall be firstly established through posting an event in the form:

```
evt = { class='tcp', type='connect', host=addr, port=number,
              [timeout=number] }
```

The connection result is returned in a pre-registered event handler for the class. The returned event is in the form:

```
evt = { class='tcp', type='connect', host=addr, port=number,
              connection=identifier, error=err_msg}
```

The *error* and *connection* fields are mutually exclusive. If there is a communication error, an error message is returned in the error field. If, however, the connection is successfully established, the connection identifier is returned in the *connection* field.

An NCLua application sends data, using TCP protocol, by posting events of the form:

```
evt = { class='tcp', type='data', connection=identifier,
                value=string, [timeout=number] }
```

Similarly, an NCLua application receives data transported by TCP protocol y receiving events of the form:

```
evt = { class='tcp', type='data', connection=identifier,
                value=string, error=msg}
```

The *error* and *value* fields are mutually exclusive. If there is a communication error, an error message is returned in the error field. Otherwise, the message is passed in the *value* field.

In order to close the connection, an event of the following form shall be posted:

```
evt = { class='tcp', type='disconnect', connection=identifier }
```

NOTE. A specific Ginga-NCL implementation should handle issues like authentication, connection timeout/retry, whether a connection should keep open, etc.

The class dependent filter of the “tcp” class is *connection*.

udp class:

In order to send or receive a tcp data, an association shall be firstly established through posting an event in the form:

```
evt = { class='udp', type='connect', host=addr, port=number,
                [timeout=number] }
```

The connection result is returned in a pre-registered event handler for the class. The returned event is in the form:

```
evt = { class='udp', type='connect', host=addr, port=number,
                association=identifier, error=err_msg}
```

The *error* and *association* fields are mutually exclusive. If there is a communication error, an error message is returned in the error field. If, however, the association is successfully established, the association identifier is returned in the *association* field.

An NCLua application sends data, using UDP protocol, by posting events of the form:

```
evt = { class='udp', type='data', association=identifier,
                value=string, [timeout=number] }
```

Similarly, an NCLua application receives data transported by UDP protocol y receiving events of the form:

```
evt = { class='udp', type='data', association=identifier,
                value=string, error=msg}
```

The *error* and *value* fields are mutually exclusive. If there is a communication error, an error message is returned in the error field. Otherwise, the message is passed in the *value* field.

In order to releas the association, an event of the following form shall be posted:

```
evt = { class='tcp', type='disconnect', association=identifier }
```

NOTE. A specific Ginga-NCL implementation should handle issues like authentication, association timeout/retry, whether an association should keep open, etc.

The class dependent filter of the “udp” class is *association*.

http class:

An NCLua application sends data, using HTTP protocol, by posting events in the form:

```
evt = { class='http', host=addr, port=number, path=string (i.e.,
file_path/#fragment_identifier), type=method (i.e., 'get', 'post', etc.)
value=string, [timeout=number] }
```

Similarly, an NCLua application receives data transported by http protocol using events in the form:

```
evt = { class='http', host=addr, port=number, path=string, type=method,
value=string, error=msg}
```

The *error* and *value* fields are mutually exclusive. When there is a communication error, a message is returned in the error field. When the communication is succeeded, the message is passed in the *value* field.

NOTE. An specific Ginga-NCL implementation should handle issues like authentication, connection timeout/retry, whether a connection should keep open, etc.

The class dependent filter of the “http” class, are *host,* and *port,* in this order.

sms class:

The behaviour for sending and receiveing data using SMS is very similar to the one of the tcp class. The “sms” class is optional in a Ginga_NCL conformant implementation.

An NCLua application sends data, using SMS, through posting events in the form:

```
evt = { class='sms', type='send', to='string', value=string [, id:string]}
```

The *to* field contains the destination number (phone number or large account number). If they are not specified, region and country code prefixes will receive the respective region and country codes from where the message is being sent.

The *value* field contains the message content.

The *id* field can be used to identify the SMS that will be dispatched. The application is responsible for defining the *id* value and for guaranteeing its uniqueness.

A confirmation event must be sent back to the NCLua application, following the format:

```
evt = { class='sms', type='send', to:string, sent:boolean [,error:string] [,
id:string] }
```

In the confirmation message the *to* field shall have the same value as in the original event posted by the NCLua application. The *sent* field notify if the SMS was dispatched by the device (true) or not. The *error* field is optional. If the *sent* field value is false, it may contain a detailed error message. If the original SMS is posted with the *id* field defined, the confirmation event shall arrive with the same id value. Thus, the NCLua application will be able to make an association between both events, and deal with multiple SMS messages being dispatched simultaneously.

Similarly, an NCLua application registers itself to receive SMS messages by posting events in the form:

```
evt = { class='sms', type ='register', port:number }
```

The *port* field shall receive a valid TCP port number. For compliance with the GSM Standards

(3GPP TS 23.040 V6.8.1, of 2006-10), this value should be in the interval [16000,16999].

Events received by the handler have the following format:

```
evt = { class='sms', type='receive', from:string, port:number, value:string }
```

The *port* field is defined as in the *type* = "register". The *from* field contains the source message number (phone number or large account number). Region and country code prefixes may be omitted if they are equal to the receiver ones. The *value* field contains the message content.

At any moment, the application can request to stop receiving SMS messages in a given port, posting the event:

```
evt = { class='sms', type='unregister', port:number }
```

The *port* field is defined as in the type = "register".

At the moment the NCLua media presentation stops, the Ginga-NCL implementation shall ensure that all ports will be unregistered.

NOTE. An specific Ginga-NCL implementation should handle issues like authentication, etc.

The class dependent filter of the "sms" class, are *from* and *port*, in this order.

The purpose of the port number is to avoid conflicts between common SMS messages received by a user, and SMS messages that should be handled only by the application.

A Ginga-NCL implementation shall immediately return *false* in every call to *event.post()* that uses an event class that is not supported. The NCLua application must capture this error condition in order to verify if the SMS dispatch failed.

`si` class:

The `si` event class provides access to a set of information multiplexed in a transport stream and periodically transmitted.

The information acquisition process shall be performed in two steps:

1) A request is made calling the asynchronous `event.post()` function;
2) An event is received in return, to be delivered to the registered-event handlers of an NCLua script, whose data field contains a set of subfields and is represented by a Lua table. The set of subfields depends on requested information.

NOTE - In the si class, the class dependent filter could only be *type*.

Three event types are defined by the following tables:

type = 'services'

The table of 'services' event type is made up by a set of vectors, each one with information related with a multiplexed service of the tuned transport stream.

Each request for a table of 'services' event type shall be carried out through the following call:

event.post('out', { class='si', type='services'[, index=N][, fields={field_1, field_2,…, field_j}]}),

where:

i) the *index* field defines the service index, when specified; if not specified, all services of the tuned transport stream shall be present in the returned event;

ii) the *fields* list may have as a value any subset of subfields defined for the *data* table of the returned event (thus, field_i represents one of the subfields of the data table). If the *fields* list is not specified, all subfields of the data table shall be filled.

The returned event is created after all requested information is processed by the middleware (information that is not received within a maximum interval shall be returned as ‘nil’).

NOTE. In order to compute the values of the data-table subfields to be returned in events of services type, SI tables should be used as a basis, as well as the descriptors associated with the service [i].

Some information from SI may be specific for a country, service provider or system used. Therefore, data-table subfields are left to be defined for each case.

type = ‘epg’

The table of the ‘epg’ event type is made up by a set of vectors. Each vector contains information about an event of the content being transmitted.

Each request for a table of ‘epg’ event type shall be carried out through one of the following possible calls:

1) event.post('out', { class='si', type='epg', stage=’current’[, fields={field_1, field_2,…, field_j}]})

   where the *fields* list may have as a value any subset of subfields defined for the *data* table of the returned event (thus, field_i represents one of the subfields of the data table). If the *fields* list is not specified, all subfields of the data table shall be filled.

   Description: returns information regarding to the current content (from now on called TV-event in order to differentiate from the NCL and Lua events) being transmitted.

2) event.post('out', {class='si', type='epg', stage='next'[, eventId=<number>][, fields={field_1, field_2,…, field_j}]})

   where:

   i) the *eventId* field, when specified, identifies the TV-event immediately before the TV-event whose information is required. When not specified, the requested information is for the event that immediately follows the current TV event.

   ii) the *fields* list may have as a value any subset of subfields defined for the *data* table of the returned event (thus, field_i represents one of the subfields of the data table). If the *fields* list is not specified, all subfields of the data table shall be filled.

   Description: returns information regarding to the TV-event immediately after the TV-event defined in *eventId,* or information regarding to the TV-event immediately after the current TV-event, when *eventId* is not specified.

3) event.post('out', {class='si', type='epg', stage=’schedule’, startTime=<date>, endTime=<date>[, fields={field_1, field_2,…, field_j}]})

   where the *fields* list may have as a value any subset of subfields defined for the *data* table of the returned event (thus, field_i represents one of the subfields of the data table). If the *fields* list is not specified, all subfields of the data table shall be filled.

   Description: returns information regarding to TV-events within the time interval defined by the startTime and endTime fields, which have tables in the <date> format as values.

The returned event is created after all request information is processed by the middleware (information that is not broadcasted within a maximum interval shall be returned as ‘nil’).

NOTE. In order to compute the values of the data-table subfields to be returned in events of epg type, SI tables should be used as a basis, as well the descriptors associated with the TV-event [i].

Some information from SI may be specific for a country, service provider or system used.

Therefore, data-table subfields are left to be defined for each case.

type=’time’

The table of the ‘time’ event type contains information about the current UTC (Universal Time Coordinated) date and time, but in the official country time zone in which the receptor is located.

Each request for a table of ‘time’ event type shall be carried out through the following call:

```
     event.post('out', { class='si', type=’time’})
```

The returned event is created after all request information is processed by the middleware (information that is not broadcasted within a maximum interval shall be returned as ‘nil’). The data table is returned as follows:

```
evt = {
    class = 'si',
    type = 'time',
    data = {
                year             = <number>,
                month          = <number>,
                day              = <number>,
                hours            = <number>,
                minutes        = <number>,
                seconds       = <number>
    }
```

NOTE. In order to compute the values of the data-table subfields to be returned in events of time type, the appropriate SI table should be used as a basis.

The SI table used is left to be defined for each case, since some information from SI may be specific for a country, service provider or system used.

metadata class:

The `metadata` event class provides access to information about content, users, systems, providers, etc., as defined in the high-level specification of metadata for IPTV services [ITU-T H.750].

The information acquisition process shall be performed in two steps:

1) A request is made calling the asynchronous `event.post()` function;
2) An event is received in return, to be delivered to the registered-event handlers of an NCLua script, whose data field contains a set of subfields and is represented by a Lua table. The set of subfields depends on the requested information.

In the metadata class, no fields are defined (with the exception of the class field), they are left to be specified by vendors, operators and providers, for example.

user class:

By using the user class, applications may create their own events.

This class defines no fields (with the exception of the class field).

NOTE. The “user” class has no class dependent filters.

10.3.3 The *settings* module

The settings module exports a global table, called *settings*, which contains the reserved environment variables and the variables defined by the NCL document author, as defined in the “application/x-ncl-settings” node.

The table fields that represent variable in the settings node are read-only, i.e., the NCLua script cannot change their values. If the script tries to change the value of a read-only variable, then an error shall be raised. Properties of the “application/x-ncl-settings” node may only be changed by using NCL links.

The settings table contains subtables, corresponding to each “application/x-ncl-settings” node’s group. For instance, in an NCLua object, the settings node’s variable “system.CPU” is referred to as settings.system.CPU.

Examples:

```
lang = settings.system.language
age = settings.user.age
val = settings.default.selBorderColor
settings.service.myVar = 10
settings.user.age = 18 --> ERROR!
```

10.3.4 The *persistent* module

NCLua applications may save data in a restricted middleware area and recover it between executions. NCLua player allows an NCLua application to persist a value to be used by itself or by another imperative object. In order to do that the NCL player defines a reserved area, inaccessible to non-imperative NCL media objects. This area is split into the groups “service”, “channel” and “shared”, with same semantics, concerning persistent duration, of the homonym groups of the NCL settings node. There are no predefined or reserved variables in these groups, and imperative objects are allowed to change variable’s values directly. Other imperative languages should offer an API to access this same area.

In this module, Lua offers an API to export the *persistent* table with the variables defined in the reserved area.

The use of the *persistent* table is very similar to the *settings* table, except that, in this case, imperative codes may change field values.

Examples of use:

```
persistent.service.total = 10
color = persistent.shared.color
```

11 Security API

11.1 Security control in Ginga-NCL

Usually, in NCL applications security issues are not handled by their authors. It is assumed that media players provide support to protocols required for secure communication (e.g., via HTTPS / TLS).

It is also assumed that permission to access resources in specific network domains is given by application authentication, and that Ginga-NCL and its media players are responsible for this task, based on permission information received when an NCL application or an NCL node is added to an NCL private base.

In order to apply other specific security mechanisms, NCL application authors must use the NCLua security API.

12.2 The NCLua Security API

The *Security Extensions of NCLua API* provides basic functions such as the generation and verification of digital signatures, message digest generation, and data encryption.

12.2.1 NCLua event classes for security control

Operations for security management are asynchronous and work by sending and receiving events. Each operation is performed by sending an event that is associated with an arbitrary identifier (request_id). The identifier will be used by registered event handlers for retrieving the result of each call. The following additional event classes shall be implemented: *signature*, *digest* and *cipher*.

`signature` class:

The `signature` event class provides the ability to generate and verify data signed by digital signatures. One can determine which algorithm should be used in each call for generation or verification. The signature verification shall be performed from a key contained in an X.509 v3 certificate [X.509], an ITU-T standard for public key infrastructure (PKI).

To sign data with a private key, in the format of a specified algorithm, an event of type “sign” must be posted, following the form described below:

evt = {class = “signature”, type = “sign”, algorithm = *string*, request_id = *identifier*, private_key = *string*, data = *string*}

The signed information is returned on a pre-registered event handler for the class. The returned event has the following form:

evt = {class = “signature”, type = “sign”, algorithm = *string*, request_id = *identifier*, signed_data = *string*, error = *err_msg*}

The *signed_data* and *error* fields are mutually exclusive. If there is an error in the generation process of the signed information, an error message must be returned in the *error* field. Otherwise, the generated value is returned in the *signed_data* field (encoded in Base64).

To verify that data has been signed with a particular key, an event of *“verify”* type must be used. A certificate can be used (coded in PEM format, in Base64, according to RFC 1421) to recover the public key to be verified. Events of *“verify”* type have the following form:

evt = {class = “signature”, type = “verify”, algorithm = *string*, request_id = *identifier*, data = *string*, signature = *string*, public_key = *string*}

Similarly, an NCLua application receives the return of the signature verification by means of events of the form

evt = {class = “signature”, type = “verify”, algorithm = *string*, request_id = *identifier*, result = *boolean*}

The *result* field indicates the result of the operation. The possible values are: valid signature (1) and invalid signature (0).

The signature class shall support the algorithms listed in Table 11.1.

Table 11.1 – Algorithms supported by the *signature* class

Mnemonic	Meaning

SHA1withRSA	Algorithm SHA 1 [SHS] for resume generation with RSA public key algorithm, as defined by [PKCS#1].
SHA256withRSA	Algorithm SHA 256 [SHS] for resume generation with RSA public key algorithm, as defined by [PKCS#1].
SHA384withRSA	Algorithm SHA 384 [SHS] for resume generation with RSA public key algorithm, as defined by [PKCS#1].
SHA512withRSA	Algorithm SHA 512 [SHS] for resume generation with RSA public key algorithm, as defined by [PKCS#1].
SHA1withDSA	Algorithm SHA 1 [SHS] for resume generation with DAS public key algorithm, as defined by [DSS].

Additionally, it is recommended that the “signature” class supports the algorithms listed in Table 11.2.

Table 11.2 – Optional algorithms for the *signature* class

Mnemonic	**Meaning**
SHA1withECDSA	Algorithm SHA 1 [SHS] for resume generation with ECDSA public key algorithm, as defined by [ANSI X9.62].
SHA256withECDSA	Algorithm SHA 256 [SHS] for resume generation with ECDSA public key algorithm, as defined by [ANSI X9.62].
SHA384withECDSA	Algorithm SHA 384 [SHS] for resume generation with ECDSA public key algorithm, as defined by [ANSI X9.62].
SHA512withECDSA	Algorithm SHA 512 [SHS] for resume generation with ECDSA public key algorithm, as defined by [ANSI X9.62].

`digest` class:

The `digest` event class provides functionality for checking the data integrity, through the generation of message digests. The message digests are generated from asynchronous requests that identify which algorithm should be used for hashing.

To generate a message digest for a set of data using a specified algorithm, an event of type “generate” must be posted, following the form:

evt = {class = “digest”, type = “generate”, algorithm = *string*, request_id = *identifier*, data = *string*}

The result of the generation of message digest is returned in a pre-registered event handler for this class. The returned event is of the form:

evt = {class = “digest”, type = “generate”, algorithm = *string*, request_id = *identifier*, message_digest = *string*, error = *err_msg*}

The *message_digest* and *error* fields are mutually exclusive. If there is an error in the generation of the message digest, an error message should be returned in the *error* field. Otherwise, the digest value is returned in the *message_digest* field.

The “digest” class shall support the algorithms listed in Table 11.3.

Table 11.3 – Algorithms supported by the *digest* class

Mnemonic	**Meaning**
sha1	Algorithm SHA 1 [SHS] for resume generation.
sha256	Algorithm SHA 256 [SHS] for resume generation.
sha384	Algorithm SHA 384 [SHS] for resume generation.
sha512	Algorithm SHA 512 [SHS] for resume generation.

`cipher` class

The `cipher` event class is used to encrypt and decrypt data from symmetric and asymmetric key algorithms that are identified on each request. If a symmetric encryption algorithm is specified, the key used will be considered symmetric. Otherwise, it will be considered a private key (which can be retrieved from a certificate containing an asymmetric key).

An NCLua application encrypts data (using a valid key) by sending events of "encrypt" type that follow the form described below:

evt = {class = "cipher", type = "encrypt", algorithm = *string*, padding = *boolean*,
request_id = *identifier*, data = *string*, key = *string*}

The *padding* field indicates that standard filling must not be used (PKCS5Padding [PKCS # 5]) when encrypting. The default value of this field is *false*.

An NCLua application receives the encrypted data through events of the form:

evt = {class = "cipher", type = "encrypt", algorithm = *string*, request_id = *identifier*,
encrypted_data = *string*, error = *err_msg*}

The *encrypted_data* and *error* fields are mutually exclusive. If there is an error in data encryption, an error message must be returned in the *error* field. Otherwise, the generated value is returned in the *encrypted_data* field.

To decrypt data from a valid key and from a specified algorithm, an event of "decrypt" type must be posted, following the form:

evt = {class = "cipher", type = "decrypt", algorithm = *string*, padding = *boolean*,
request_id = *identifier*, data = *string*, key = *string*}

The application receives decrypted data through events of the form:

evt = {class = "cipher", type = "decrypt", algorithm = *string*, request_id = *identifier*,
decrypted_data = *string*, error = *err_msg*}

The *decrypted_data* and *error* fields are mutually exclusive. If there is an error in the decrypting data process, an error message must be returned in the *error* field. Otherwise, the generated value is returned in the *decrypted_data* field.

The "cipher" class shall support the algorithms listed in Table 11.4, with their respective modes of operation:

Table 11.4 – Algorithms supported by the *cipher* class

Mnemonic	**Meaning**
des-ede-cbc	Triple DES/EDE with two keys in CBC [ISO 18033-3] mode.
des-ede	Triple DES/EDE with two keys in ECB [ISO 18033-3] mode.
des-ede-cfb	Triple DES/EDE with two keys in CFB [ISO 18033-3] mode.
des-ede-ofb	Triple DES/EDE with two Keys in OFB [ISO 18033-3] mode.
des-ede3-cbc	Triple DES/EDE with three keys in CBC [ISO 18033-3] mode.
des-ede3	Triple DES/EDE with three keys in ECB [ISO 18033-3] mode.
des3	Alternative for des-ede3-cbc [ISO 18033-3].
des-ede3-cfb	Triple DES/EDE with three keys in CFB [ISO 18033-3] mode.
des-ede3-ofb	Triple DES/EDE with three keys in OFB [ISO 18033-3] mode.
aes-128-cbc aes-192-cbc aes-256-cbc	AES with 128/192/256 bits in CBC [ISO 18033-3] mode.
aes-128 aes-192 aes-256	Alternative for aes-[128|192|256]-cbc [ISO 18033-3]
aes-128-cfb aes-192-cfb aes-256-cfb	AES with 128/192/256 bits inCFB mode of 128 bits [ISO 18033-3].
aes-128-cfb1 aes-192-cfb1 aes-256-cfb1	AES with 128/192/256 bits in CFB mode of 1 bit [ISO 18033-3].
aes-128-cfb8 aes-192-cfb8 aes-256-cfb8	AES with 128/192/256 bits in CFB mode of 8 bits [ISO 18033-3].
aes-128-ecb aes-192-ecb aes-256-ecb	AES with 128/192/256 bits in ECB [ISO 18033-3] mode.
aes-128-ofb aes-192-ofb aes-256-ofb	AES with 128/192/256 bits in OFB [ISO 18033-3] mode.

Annex A

NCL 3.1 module schemas used in the Enhanced DTV profile

(This annex forms an integral part of this Recommendation.)

The following NCL 3.1 module schemas used in the Enhanced DTV profile are available as an electronic attachment to this Recommendation:

– Structure module: NCL31Structure.xsd
– Layout module: NCL31Layout.xsd
– Media module: NCL31Media.xsd
– Context module: NCL31Context.xsd
– MediaContentAnchor module: NCL31MediaContentAnchor.xsd
– CompositeNodeInterface module: NCL31CompositeNodeInterface.xsd
– PropertyAnchor module: NCL31PropertyAnchor.xsd
– SwitchInterface module: NCL31SwitchInterface.xsd
– Descriptor module: NCL31Descriptor.xsd
– Linking module: NCL31Linking.xsd
– ConnectorCommonPart Module: NCL31ConnectorCommonPart.xsd
– ConnectorAssessmentExpression Module: NCL31ConnectorAssessmentExpression.xsd
– ConnectorCausalExpression Module: NCL31ConnectorCausalExpression.xsd
– CausalConnector module: NCL31CausalConnector.xsd
– ConnectorBase module: NCL31ConnectorBase.xsd
– NCL31CausalConnectorFunctionality.xsd
– TestRule module: NCL31TestRule.xsd
– TestRuleUse module: NCL31TestRuleUse.xsd
– ContentControl module: NCL31ContentControl.xsd
– DescriptorControl module: NCL31DescriptorControl.xsd
– Timing module: NCL31Timing.xsd
– Import module: NCL31Import.xsd
– EntityReuse module: NCL31EntityReuse.xsd
– ExtendedEntityReuse module: NCL31ExtendedEntityReuse.xsd
– KeyNavigation module: NCL31KeyNavigation.xsd
– TransitionBase module: NCL31TransitionBase.xsd

- Animation module: NCL31Animation.xsd
- Transition module: NCL31Transition.xsd
- Metainformation module: NCL31Metainformation.xsd

Appendix I

Ginga architecture

Ginga-NCL was originally built as a component of the middleware Ginga [b_ABNT NBR 15606-2], as depicted in Figure I.1-1.

A Ginga implementation should be open, flexible, granular, self-contained and component-based. However, this Draft Recommendation does not specify any Ginga implementation in a compliant receiver. The architecture presented in this Draft Recommendation only helps to present the requirements and recommendations of a Ginga implementation. A receiver manufacturer may implement all subsystems and their modules as a single subsystem; alternatively, all modules may be implemented as distinct components with well-defined interfaces.

Ginga-NCL is the logical subsystem of the Ginga System that processes NCL documents. Ginga allows some extensions, but they are optional. To be Ginga complaint, the Ginga-NCL subsystem is required. This avoids the threat of market fragmentation and ensures that Ginga always offer backward-compatible profiles.

Media players serve application needs for decoding and presentation content types such as PNG, JPEG, MPEG, and other formats. The Ginga Common Core is composed of media players, procedures to obtain contents transported in the several networks accessed by a receiver, the conceptual display graphical model defined by the receiver platform, and other functions. The Ginga Common Core is also responsible to gather metadata information and to provide this information through NCL *settings* media object (see 7.2.3).

It is also recommended that the Ginga Common Core provides an API to communicate with DRM system; pulls together context information (like user profiles and receiver profiles available on a local or removable storage device) and provides context awareness through NCL *settings* media object (see Clause 7.2.3); and supports software version management (update) of Ginga's components.

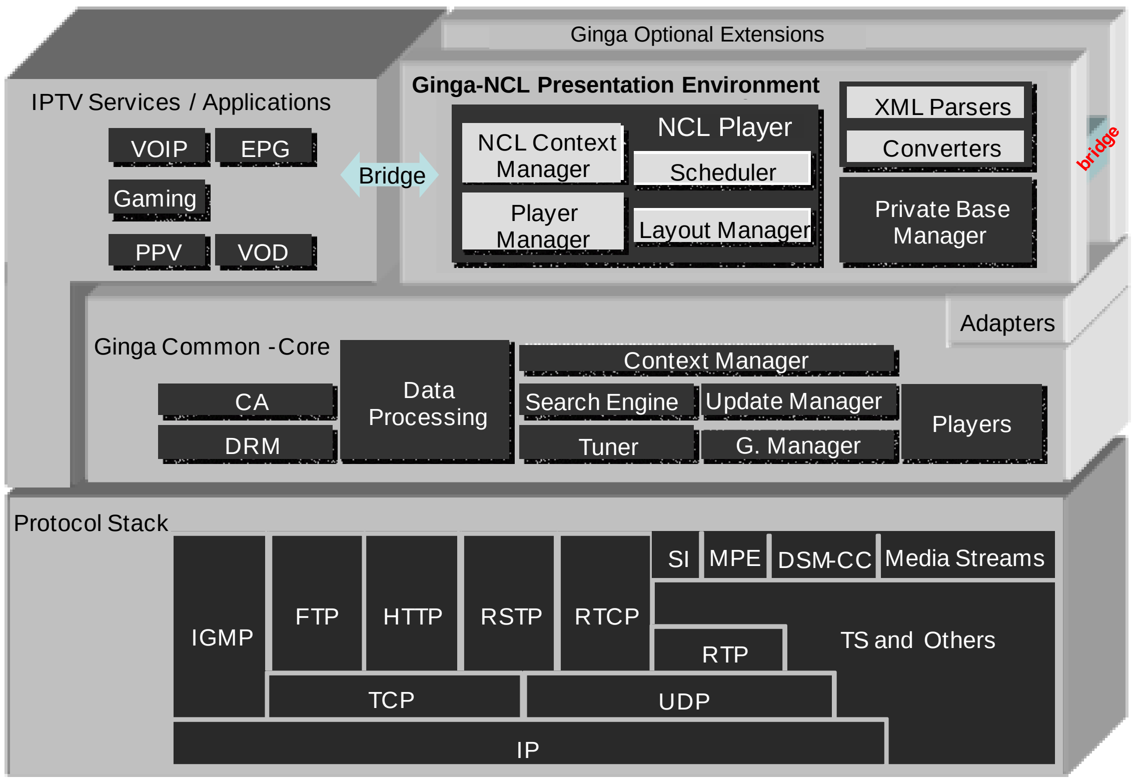

Figure I.1-1: Ginga architecture

The core of Ginga-NCL Presentation Environment is the NCL Player. This component is in charge of receiving and controlling multimedia applications written in NCL. Applications are delivered to the NCL Player by the Ginga Common Core subsystem. Upon receiving an application, the NCL Player requests the XML Parser and Converter component to translate the NCL application to the Ginga-NCL internal data structures necessary for controlling the application presentation. From then on, the Scheduler component is started in order to orchestrate the NCL document presentation. The pre-fetching of media object's contents, the evaluation of link conditions and the scheduling of corresponding link's actions that guide the presentation flow are some tasks performed by the Scheduler component. In addition, the Scheduler component is responsible for command the Player Manager component to instantiate an appropriate Player, according to the media content type to be exhibited in a given moment in time. Media contents are acquired through the Protocol Stack, and can come from different communication (broadband and broadcast) networks.

One important player, part of Ginga-NCL, is the Lua Engine, responsible for the execution of NCLua objects, i.e., media objects with Lua code [b_H.IPTV-MAFR.14].

In Ginga-NCL, a generic API (see 8.1) is defined to establish the necessary communication between Players components and the Presentation Engine (Scheduler component). Thanks to this API, the Ginga-NCL Presentation Environment and the Ginga Common Core are strongly coupled but independent subsystems. Ginga Common Core may be substituted for other third part implementations, allowing Ginga-NCL to be integrated in other DTV middleware specifications, extending their functionalities with NCL facilities.

Players are responsible for notifying the NCL Player about presentation, selection and attribution events (see Clause 7.12) defined in NCL applications. Players that do not follow the generic API (see Clause 8.1) are required to use the services provided by Adapters. Any declarative or imperative language engine can be adapted to act as a Ginga-NCL's media player, e.g., XHTML browsers or a Java engine.

In Ginga-NCL, a declarative application can be generated or modified on-the-fly, using Ginga-NCL Editing Commands (see Clause 9).

The Presentation Engine deals with NCL applications collected inside a data structure known as private base. A Private Base Manager component is in charge of receiving NCL Editing Commands and maintaining the NCL documents being presented.

Ginga-NCL Presentation Engine supports multiple presentation devices through its Layout Manager module. This component is responsible for mapping all presentation regions (see Clause 7.2.5) defined in an NCL application to canvas on receiver's exhibition devices.

In particular, Ginga-NCL provides declarative support to IPTV specific services, such as VoD (Video on Demand, datacasting, etc. Thus, a VoD service may, for example, play an NCL application besides the main audiovisual stream. Moreover, an IPTV service itself can be an NCL application.

Appendix II

An NCL example

(This appendix does not form an integral part of this Recommendation.)

An example NCL application is available as an electronic attachment to this Recommendation. The example explores many NCL functionalities, including NCLua objects. This is intended to illustrate how NCL applications are usually structured. The example is composed of the following files:

– main.ncl

– causalConnBase.ncl

– counter.lua

Media objects used in this example are not included in the attachment but can be freely obtained from http://club.ncl.org.br/node/48.

Bibliography

[b_H.IPTV-MAFR.14] Draft Recommendation ITU-T H.IPTV-MAFR.14 *Lua script language for IPTV.*

[b_H.IPTV-SDC] Draft Recommendation ITU-T H.IPTV-SDC Mechanisms for service discovery up to consumption for IPTV.

[b_ABNT NBR 15606-2] ABNT NBR 15606-2 (2007), 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.

[b_NCM Core] 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. Also available in http://www.ncl.org.br.

[b_NCL DTV] Soares L.F.G; Rodrigues R.F. *Nested Context Language 3.0: Part 8 – NCL (Nested Context Language) Digital TV Profiles*, Technical Report, Departamento de Informática PUC-Rio, No. 35/06, October 2006, ISSN: 0103-9741. Also available in http://www.ncl.org.br.

[b_NCL Live E.C.] 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, Departamento de Informática PUC-Rio, No. 36/06, December 2006, ISSN: 0103-9741. Also available at http://www.ncl.org.br.

[b_NCL Imp. Obj.] 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, Departamento de Informática PUC-Rio, No. 02/08. Rio de Janeiro. January 2008. ISSN 0103-9741. Also available at http://www.ncl.org.br

[b_NCL Decl. Obj.] 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, Departamento de Informática PUC-Rio, No. 02/09. Rio de Janeiro. January 2009. ISSN 0103-9741. Also available at http://www.ncl.org.br

[b_NCL Multi. Dev.] Soares L.F.G.; Batista C.E.F. *Nested Context Language 3.1 Part 12 – Support to Multiple Exhibition Devices*. Technical Report, Departamento de Informática PUC-Rio, No. 12/13. Rio de Janeiro. September 2013. ISSN 0103-9741. Also available at http://www.ncl.org.br

[b_W3C CSS2] W3C Recommendation CSS2 (1998) - Cascading Style Sheets, level 2.

[b_W3C XMLNAMES1] W3C Recommendation. XML_Names 1.0 (1999) – Namespaces in XML.

[b_W3C RDF]	W3C Recommendation. RDF (1999) - Resource Description Framework (RDF) Model and Syntax Specification.
[b_W3C XHTML]	W3C Recommendation. XHTML 1.0 (2002) - Extensible HyperText Markup Language
[b_W3C SMIL 2.1]	W3C Recommendation. SMIL 2.1 (2005) - Synchronized Multimedia Integration Language – SMIL 2.1 Specification