International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

H.761

(06/2011)

SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS

IPTV multimedia services and applications for IPTV – IPTV multimedia application frameworks

Nested context language (NCL) and Ginga-NCL

Recommendation ITU-T H.761

ITU-T

ITU International Telecommunication Union

ITU-T H-SERIES RECOMMENDATIONS

AUDIOVISUAL AND MULTIMEDIA SYSTEMS

CHARACTERISTICS OF VISUAL TELEPHONE SYSTEMS	H.100–H.199
INFRASTRUCTURE OF AUDIOVISUAL SERVICES	
General	H.200–H.219
Transmission multiplexing and synchronization	H.220–H.229
Systems aspects	H.230–H.239
Communication procedures	H.240–H.259
Coding of moving video	H.260–H.279
Related systems aspects	H.280–H.299
Systems and terminal equipment for audiovisual services	H.300–H.349
Directory services architecture for audiovisual and multimedia services	H.350–H.359
Quality of service architecture for audiovisual and multimedia services	H.360–H.369
Supplementary services for multimedia	H.450–H.499
MOBILITY AND COLLABORATION PROCEDURES	
Overview of Mobility and Collaboration, definitions, protocols and procedures	H.500–H.509
Mobility for H-Series multimedia systems and services	H.510–H.519
Mobile multimedia collaboration applications and services	H.520–H.529
Security for mobile multimedia systems and services	H.530–H.539
Security for mobile multimedia collaboration applications and services	H.540–H.549
Mobility interworking procedures	H.550–H.559
Mobile multimedia collaboration inter-working procedures	H.560–H.569
BROADBAND, TRIPLE-PLAY AND ADVANCED MULTIMEDIA SERVICES	
Broadband multimedia services over VDSL	H.610–H.619
Advanced multimedia services and applications	H.620–H.629
IPTV MULTIMEDIA SERVICES AND APPLICATIONS FOR IPTV	
General aspects	H.700–H.719
IPTV terminal devices	H.720–H.729
IPTV middleware	H.730–H.739
IPTV application event handling	H.740–H.749
IPTV metadata	H.750–H.759
IPTV multimedia application frameworks	**H.760–H.769**
IPTV service discovery up to consumption	H.770–H.779

For further details, please refer to the list of ITU-T Recommendations.

Recommendation ITU-T H.761

Nested context language (NCL) and Ginga-NCL

Summary

Recommendation ITU-T H.761 gives the specification of the nested context language (NCL) and of an NCL presentation engine called Ginga-NCL to provide interoperability and harmonization among IPTV multimedia application frameworks.

NCL is a glue language that holds media objects together in multimedia presentations, no matter which object types they are. 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. A very special NCL object type defined in Ginga-NCL is NCLua, an imperative media-object with Lua code.

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

History

Edition	Recommendation	Approval	Study Group
1.0	ITU-T H.761	2009-04-29	16
2.0	ITU-T H.761 v2	2011-06-13	16

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at http://www.itu.int/ITU-T/ipr/.

Table of Contents

Electronic attachment: NCL 3.0 module schemas

Introduction

Nested context language (NCL) is a declarative XML-based language initially designed for 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 Nipo-Brazilian terrestrial DTV standard (ISDB-T_B).

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 depend on which media players are embedded in the NCL engine (Ginga-NCL).

Ginga-NCL is an NCL presentation engine built as a component of an IPTV middleware. An open source reference implementation of Ginga-NCL is available under the GPLv2 licence. 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 particular NCL object type defined in Ginga-NCL is NCLua, an imperative media-object with Lua code. Because of its simplicity, efficiency and powerful data description syntax, Lua was considered the natural 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 engine 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 engine that integrates NCL and Lua players into a declarative environment. 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 3.0 module schemas used in the Enhanced DTV profile.

3.1.3 application programming interface (API) [ITU-T J.200]: Consists of 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; an XML document instance is an example of a declarative application.

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.

NOTE – In some traditional broadcast services, EPG is defined as an on-screen guide used to display information on scheduled live broadcast television programs, allowing a viewer to navigate, select, and discover programs by time, title, channel, genre, etc. This traditional definition does not cover "catalogues" for on-demand and download services (sometimes called electronic content guide, ECG) and bidirectional interactive services (sometimes called interactive program guide, IPG) for end-user interaction with a server or head-end.

Some EPGs utilize web-pages, or teletext to realize this function.

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 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 [ITU-T H.222.0]: Refers to 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: 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.

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 or execution 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 NCL documents and 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
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 the 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 an external source, 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, that is, media objects with Lua code [b-H.IPTV-MAFR.14]. Lua is the scripting language of NCL.

Ginga-NCL deals with applications collected inside a data structure known as private base. 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 Ginga architecture.

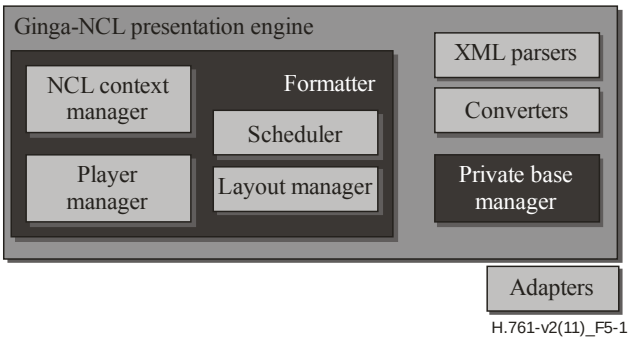

Figure 5-1 – Ginga-NCL presentation environment

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 its media-object content types. Which media objects are supported depends on the media players that are coupled in the NCL formatter. One of these players is the main video and audio decoder/player, usually implemented in hardware in an IPTV receiver. In this way, note that the main video and audio of a service are treated like all other media objects 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.

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. Again, although an HTML-based browser shall be supported by Ginga-NCL, the use of HTML elements to define relationships (including those triggered by external events) is discouraged in authoring NCL documents, for the same reason: structure-based authoring should be emphasized for the well-known reasons largely reported in the literature.

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 of NCL, focal point 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, should ignore all other (unknown) elements and attributes.

NCL edition 3.0 is partitioned into 14 functional areas, which are partitioned again into modules:

1) Structure:
   - Structure Module.
2) Layout:
   - Layout Module.
3) Components:
   - Media Module.
   - Context Module.
4) Interfaces:
   - MediaContentAnchor Module.
   - CompositeNodeInterface Module.
   - PropertyAnchor Module.
   - SwitchInterface Module.

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

7.1 Identifiers for NCL 3.0 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 3.0 modules, elements and attributes are contained within the following namespace: http://www.ncl.org.br/NCL3.0/.

Each NCL module has a unique identifier associated with it. The identifiers for NCL 3.0 modules shall comply with Table 7-1.

Modules may also be identified collectively. The following module collections are defined:

– modules used by the NCL 3.0 Language profile:
http://www.ncl.org.br/NCL3.0/LanguageProfile

– modules used by the NCL 3.0 Causal Connector profile:
http://www.ncl.org.br/NCL3.0/CausalConnectorProfile

– modules used by the NCL 3.0 Enhanced DTV profile:
http://www.ncl.org.br/NCL3.0/EDTVProfile

Table 7-1 – The NCL 3.0 module identifiers

Modules	Identifiers
Animation	http://www.ncl.org.br/NCL3.0/Animation
CompositeNodeInterface	http://www.ncl.org.br/NCL3.0/CompositeNodeInterface
CausalConnector	http://www.ncl.org.br/NCL3.0/CausalConnector
CausalConnectorFunctionality	http://www.ncl.org.br/NCL3.0/CausalConnectorFunctionality
ConnectorCausalExpression	http://www.ncl.org.br/NCL3.0/ConnectorCausalExpression
ConnectorAssessmentExpression	http://www.ncl.org.br/NCL3.0/ConnectorAssessmentExpression
ConnectorBase	http://www.ncl.org.br/NCL3.0/ConnectorBase
ConnectorCommonPart	http://www.ncl.org.br/NCL3.0/ConnectorCommonPart
ContentControl	http://www.ncl.org.br/NCL3.0/ContentControl
Context	http://www.ncl.org.br/NCL3.0/Context
Descriptor	http://www.ncl.org.br/NCL3.0/Descriptor
DescriptorControl	http://www.ncl.org.br/NCL3.0/DescriptorControl
EntityReuse	http://www.ncl.org.br/NCL3.0/EntityReuse
ExtendedEntityReuse	http://www.ncl.org.br/NCL3.0/ExtendedEntityReuse
Import	http://www.ncl.org.br/NCL3.0/Import
Layout	http://www.ncl.org.br/NCL3.0/Layout
Linking	http://www.ncl.org.br/NCL3.0/Linking
Media	http://www.ncl.org.br/NCL3.0/Media
MediaContentAnchor	http://www.ncl.org.br/NCL3.0/MediaContentAnchor
KeyNavigation	http://www.ncl.org.br/NCL3.0/KeyNavigation
PropertyAnchor	http://www.ncl.org.br/NCL3.0/PropertyAnchor
Structure	http://www.ncl.org.br/NCL3.0/Structure
SwitchInterface	http://www.ncl.org.br/NCL3.0/SwitchInterface
TestRule	http://www.ncl.org.br/NCL3.0/TestRule
TestRuleUse	http://www.ncl.org.br/NCL3.0/TestRuleUse
Timing	http://www.ncl.org.br/NCL3.0/Timing
TransitionBase	http://www.ncl.org.br/NCL3.0/TransitionBase
Transition	http://www.ncl.org.br/NCL3.0/Transition
Metainformation	http://www.ncl.org.br/NCL3.0/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. The identifiers of these SMIL 2.0 modules are shown in Table 7-2.

Table 7-2 – The SMIL 2.0 module identifiers used in NCL profiles

Modules	Identifiers
BasicTransitions	http://www.w3.org/2001/SMIL20/BasicTransitions
TransitionModifiers	http://www.w3.org/2001/SMIL20/TransitionsModifiers
Metainformation	http://www.w3.org/2001/SMIL20/Metainformation

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="ISO-8859-1"?>
<ncl id="any string" xmlns="http://www.ncl.org.br/NCL3.0/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 as a decimal number character string 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 a document based on the revised specification, in relation to error corrections, operational reasons, or the addition of a new concise syntax notation ("syntax sugar") that can be translated at compile time to the old one, the new version of NCL shall be released with the minor number updated. If receivers that conform to older versions cannot receive a document based on the revised specifications, 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 CausalConnectorProfile.

7.2 NCL modules

7.2.1 General remarks

The main definitions made by the NCL 3.0 modules that are present in the NCL 3.0 Enhanced DTV profile are given in clauses 7.2.2 to 7.2.15.

The complete definition of these NCL 3.0 modules, using XML schemas, is presented in Annex A. Any ambiguity found in this text can be clarified by consulting the XML schemas.

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 ("). When a value is a string, it may be any string that matches the NCName type [Namespaces in XML:1999]. That is, the value

may be any string that begins with a letter or an underscore and that only contains letters, digits, "." and "_". 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. 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. It defines the root element, called <ncl>, 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 7.2.4) contain other NCM nodes and links.

Almost all NCL elements may have the *id* attribute. This attribute may receive as its value any string that matches the NCName type definition [Namespaces in XML:1999]. That is, 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. 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.

The *title* attribute of <ncl> offers advisory information about the element. Values of the *title* attribute may be rendered by user agents in a variety of ways, which are outside the scope of this Recommendation.

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

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

Documents with *xmlns* attributes different from the two previously mentioned values shall be ignored by an implementation in conformance with this Recommendation.

Child elements of <head> and <body> are defined in other NCL modules. The order in which the <head> child elements may be declared is: importedDocumentBase?, ruleBase?, transitionBase?, regionBase*, descriptorBase?, connectorBase?, meta*, metadata*. However, this order is not a requirement.

The elements of this module, their child elements, and their attributes shall comply with Table 7-3.

Table 7-3 – Extended Structure module

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

7.2.3 Layout functionality

The Layout functionality has a single module, named 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 in <media>, <body>, and <context> elements (see clause 7.2.4).

In short, 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 7-6 (see clause 7.2.4). When the attribute is not specified, the presentation shall take place in the same device that runs the NCL formatter.

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

NOTE 2 – 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. If the attribute is not specified, the exhibition will take place only on the passive class 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 formatter).

In an implementation in conformance with Ginga-NCL specification, a first nesting level 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.

A <region> can also define the following attributes: *title*, *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 non-negative "percentage" values, or integer pixel units. For pixel values, the author may omit the "px" unit qualifier (e.g., "100"). For

percentage values, on the other hand, the "%" symbol shall be indicated (e.g., "50%"). The percentage is always relative to the parent's width, in the case of *right*, *left* and *width* definitions, and parent's height, in the case of *bottom*, *top* and *height* definitions.

The *top* and *left* attributes are the primary region positioning attributes. They place the left-top corner of the region 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 state the distance between the region's right-bottom corner and the right-bottom corner of the parent region (or the device right-bottom edge in the case of the outermost region); see Figure 7-1.

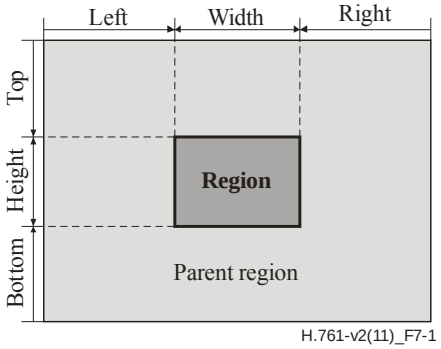

Figure 7-1 – Region positioning attributes

Regarding region sizes, when they are 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 intrinsic size of a region is equal to the size of the logical parent's geometry. This means that if a nested region does not 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 does not specify any positioning or size values, it will be assumed to be the whole device presentation area.

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. When any of these attributes is not specified and cannot have its value computed from the other attributes, the value shall be inherited from the corresponding parent absolute value. Another restriction is that child regions cannot lay outside the area established by their parent regions. When some portion of the child region lies outside of its parent region, the child region shall be ignored (considered as if it were not specified).

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

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 7-4.

Table 7-4 – Extended Layout module

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

7.2.4 Components functionality

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

The Media module defines basic media object types. For defining media objects, this module defines the <media> element. Each media object has two main attributes, besides its *id* attribute: *src*, which defines a 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 7-5 shall be supported.

Table 7-5 – Allowed URIs

Scheme	Scheme-specific-part	Use
file:	///file_path/#fragment_identifier	For local files
http:	//server_identifier/file_path/#fragment_identifier	For remote files downloaded using the http protocol
https:	//server_identifier/file_path/#fragment_identifier	For remote files downloaded using the https protocol
rtsp:	//server_identifier/file_path/#fragment_identifier	For streams downloaded using the rtsp protocol
rtp:	//server_identifier/file_path/#fragment_identifier	For streams using the rtp protocol
ncl-mirror:	//media_element_identifier	for a content flow identical to the one in presentation by another media element
ts:	//program_number.component_tag	For elementary streams contained in the tuned 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 URI path, as well.

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 may not cause tuning to occur. References that imply tuning to access a resource shall behave as if the resource were unavailable.

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. As a consequence, the content of each object can have its presentation started at different moments in time, depending on the time the media objects were started, and their presentation 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. It should be stressed that the mirroring relation is reflexive, symmetric and transitive.

As any other media object's content, if more than one media object having the same *src* attribute with a value referring to a content transported in the transport stream (TS) are started, more than one presentation shall be started. Moreover, as usual, these objects can have different presentation regions that can be re-dimensioned by an application. For media objects with the *src* attribute whose value identifies the "ts" scheme, the specific part of the scheme, that is, the program_number. component_tag, can be substituted by the following reserved words:

– video: Corresponding to the primary video ES being presented on the video plan.
– audio: Corresponding to the primary audio ES.
– text: Corresponding to the primary text ES.
– video(i): Corresponding to the ith smaller video ES *component_tag* listed in the PMT of the tuned services.
– audio(i): Corresponding to the ith smaller audio ES *component_tag* listed in the PMT of the tuned services.
– text(i): Corresponding to the ith smaller text ES *component_tag* listed in the PMT of the tuned services.

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

The *type* attribute allowed values shall follow MIME Media Types format (or, more simply, mimetypes). A mimetype 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.). Mimetypes may be registered or informal. Registered mimetypes are controlled by the Internet Assigned Numbers Authority (IANA). Informal mimetypes are not registered with IANA, but are defined by common agreement.

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.

The "application/x-ginga-NCL" type shall be applied to <media> elements with NCL code content (indeed, 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 objects embedded in NCL applications and HTML-based objects embedded in NCL applications shall follow the guidelines established in "Nested Context Language 3.0: Part 11 – Declarative Hypermedia Objects in NCL: Nesting Objects with NCL code in NCL Documents" [b-NCL Decl. Obj.].

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

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

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.

The "application/x-ncl-settings" 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 document processing. Table 7-6 states the already defined variables and their semantics.

The application/x-ncl-time type shall be applied to a special <media> element (it may be only one in an NCL document), whose content is 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.

NOTE 4 – The content of a <media> element of application/x-ncl-time type shall be specified according to 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.

Table 7-6 – 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 all receiver life cycle.	system.language	Audio language.	ISO 639-1 code
	system.caption	Caption language.	ISO 639
	system.subtitle	Subtitle Language.	ISO 639
	system.returnBitRate(i)	Bit rate of network interface (i) in kbit/s.	real
	system.screenSize	Device screen size, in (lines, pixels/line), when a class is not defined.	(integer, integer)
	system.screenGraphicSize	Resolution set for the device's screen graphics plane, in (lines, pixels/line), when a class is not defined.	(integer, 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).	(integer, integer)
	system.screenGraphicSize(i)	Resolution set for the screen graphics plane of the class (i) of devices, in (lines, pixels/line).	(integer, 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).	integer
	system.classType(i)	Type of the class (i).	("passive" | "active")
	system.info(i)	List of class (i)'s media players.	string
	system.classNumber	Number of classes that have been defined.	integer
	system.CPU	CPU performance in MIPS, regarding its capacity to run applications.	real
	system.memory	Minimum memory space in Mbytes provided to applications.	integer
	system.operatingSystem	Type of the operating system.	string to be defined
	system.luaVersion	Version of the Lua engine supported by the receiver.	string

Table 7-6 – Global variables

Group	Variable	Semantics	Possible values
	system.ncl.version	NCL language version.	string
	system.GingaNCL.version	Ginga-NCL environment version.	string
	system.xxx	Any variable with the "system" prefix 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 imperative or declarative procedure; – receiver's native applications may change the variables' values; – they shall persist during all receiver life cycle.	user.age	User age.	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.xxx	Any variable with the "user" prefix 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 for 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).	integer
	si.channelNumber	Number of the tuned channel.	integer
	si.xxx	Any variable with the "si" prefix shall follow the rules specified for the group.	

Table 7-6 – 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.xxx	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 7-6 – 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.	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.
	default.xxx	Any variable with the "default" prefix shall be reserved for future use.	
service – set of variables managed by the NCL formatter; – they may be read and have their values changed by an NCL application of the same service; – 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; variable changes shall be done using NCL commands; – they shall persist at least during the service life cycle.	service.currentFocus	The focusIndex value of the <media> element on focus.	integer
	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
	service.xxx	Any variable with the "service" prefix shall follow the rules specified for the group.	

Table 7-6 – Global variables

Group	Variable	Semantics	Possible values
channel – set of variables managed by the NCL formatter; – 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 of the same channel; variable changes shall 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.	(integer)
	channel.virtualKeyboard	Request of a virtual keyboard for NCL applications.	(true | false)
	channel.keyboardBounds	Virtual keyboard region (left, top, width, height).	(integer, integer, integer, integer)
	channel.xxx	Any variable with the "channel" prefix shall follow the rules specified for the group.	

Table 7-6 – Global variables

Group	Variable	Semantics	Possible values
shared – set of variables managed by the NCL formatter; – 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; variable changes shall be done using NCL commands; – they shall persist at least during the life cycle of the service that has defined them.	shared.xxx	Any variable with the "shared" prefix shall follow the rules specified for the group.	

Table 7-7 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 (except for <media> elements with no *src* attribute defined) and should be used to guide the player's (presentation tool) choice by the formatter. When the *type* attribute is not specified, the formatter should use the content extension specification in the *src* attribute to 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 7-7 – 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)*

If the number of media objects of a certain type exceeds 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.

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> and <media> element within a document.

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

NOTE 4 – A <media> element of application/x-ginga-NCL type may not have the *instance* and *refer* attributes.

The elements of these two modules, their child elements, and their attributes shall comply with Tables 7-8 and 7-9.

Table 7-8 – Extended Media module

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

Table 7-9 – Extended Context module

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

7.2.5 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 for media nodes (<media> elements);

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

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

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

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); the definition of content anchors representing temporal portions, through *begin* and *end* attributes; and the definition of content anchors representing temporal and spatial portions through *coords*, *begin* and *end* attributes. In addition, the <area> element allows for the definition of textual anchors, through the *beginText*,*beginPosition* and *endText*, *endPosition* attributes that define the string and the string's occurrence in the text, respectively. Besides, the <area> element may also define a content anchor based on the number of audio samples or video frames, through *first* and *last* attributes, which shall indicate the initial and final sample/frame.

NOTE 1 – 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.

NOTE 2 – 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* attribute of the <media> element that contains the <area> element.

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 the case of a <media> element of the application/x-ncl-time type, the *begin* and *end* attributes shall be defined and shall assume an absolute value of the Universal Time Coordinated (UTC).

NOTE 3 – Except for the <media> element of the application/x-ncl-time type, the *begin* and *end* attributes shall be specified according to one of the following syntaxes: 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; ii) Seconds"s", where Seconds is a positive real number.

NOTE 4 – For the <media> element of the application/x-ncl-time type, the *begin* and *end* attributes shall be specified according to 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 one corresponding to the UTC.

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.

The <area> element may also define a content anchor based on the *label* attribute, which specifies a string that should be used by the media-object player to identify a content region. Moreover, 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 <media> element of a declarative *type* (application/x-???) shall be used to specify a declarative hypermedia object in an NCL application. In this case, the object's content (located through the *src* attribute) shall be a declarative code span to be executed. As an example, Ginga-NCL 3.0 allows for the application/x-ginga-NCL type, for defining NCL applications (file extension .ncl) nested in an NCL parent application.

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 (occurrences in time), 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 "(chainId, beginOffset, endOffset)". The *chainId* parameter identifies one of the chains defined by the declarative hypermedia object. The *beginOffset* and *endOffset* parameters define the begin time and the end time of the content anchor, with regards the chain beginning time. When a declarative hypermedia object defines just one temporal chain, the *chainId* parameter may be omitted. The *beginOffset* and *endOffset* may also be omitted when they assume their default values: 0s and the chain end time, respectively.

For a declarative hypermedia object with NCL code (<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, 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 (<media type="application/x-ginga-NCL" …>), 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>'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. Thus, 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.

In a media object of "application/x-ginga-NCLua", 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 (<media type="application/x-ginga-NCLua" …>, for example), 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 7-10.

Table 7-10 – Extended MediaContentAnchor module

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

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 7-11.

Table 7-11 – Extended CompositeNodeInterface module

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

The PropertyAnchor module defines an element named <property>, which may be used for defining a node 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. 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.

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 define the interfaces.

The <body>, <context>, and <media> elements 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: top, left, bottom, right, width, height, zIndex, plan (defining in which plan 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. These properties assume as their initial values those defined in homonym attributes of their node-associated descriptor and region (see clauses 7.2.3 and 7.2.6).

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.

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 "null" 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.

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

NOTE 6 – 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 exhibited but hidden. In particular, when a document has its <body> element with its *visible* property set to "false" and its presentation event in the *paused* state, the document is said to be in stand-by. When an application is in stand-by, the service's main video shall be dimensioned to 100% of the screen, and the main audio shall be set to 100% of volume.

It should be remarked that an object with a *visible* property equal to "false", that is, an object exhibited as hidden, may not transit selection event machines defined by its content anchors to the "occurring" state (see 7.2.8) while the *visible* property value persists as "false".

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 top, left, bottom, right, width, height, zIndex, 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.

Every <property> element has a Boolean attribute named *externable* that shall be set to "true" by default. If a property is defined in a <descriptor> or <region> element, its *externable* attribute shall be set to "false" 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". 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 possible values for the reserved property names shall comply with Table 7-12.

Properties that have reserved colour string names as values ("white", "black", "silver", "gray", "red", "maroon", "fuchsia", "purple", "lime", "green", "yellow", "olive", "blue", "navy", "aqua", or "teal") follow the CSS1 colour standard, as defined in Table 7-13.

The <property> element and its attributes shall comply with Table 7-14.

Table 7-12 – Reserved parameter/attribute and possible values

Parameter/attribute name	Value	Default
top, left, bottom, right, width, height	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 non-negative integer, in the case of width and height).	If any of these properties are not defined and cannot be inferred from the NCL rules, they shall assume value "0"
location	Two numbers separated by a comma, each one following the value rule specified for left and top parameters, respectively.	See first row
size	Two values separated by a comma. Each value shall follow the same rule specified for width and height parameters, respectively.	See first row
bounds	Four values separated by commas. Each value shall follow the same rule specified for left, top, width and height parameters, respectively.	See first row
plan	"background", "video" and "graphic", following the plan definition of the DTV system.	"video", for media with *src* attribute referring to a TS's PES, "graphics", for all other cases.
baseDeviceRegion		
deviceClass		0
explicitDur	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 "nill" value.	For continuous media, the default value shall be set to the natural content presentation duration, otherwise it must be set to nill

Table 7-12 – Reserved parameter/attribute and possible values

Parameter/attribute name	Value	Default
background	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	"true" or "false".	true
transparency	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	An RGB 888 value.	nill
fit	"fill", "hidden", "meet", "meetBest", "slice". "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 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.	fill

Table 7-12 – Reserved parameter/attribute and possible values

Parameter/attribute name	Value	Default
scroll	"none", "horizontal", "vertical", "both", or "automatic".	none
style	The locator of a stylesheet file.	nill
soundLevel, trebleLevel, bassLevel	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	A real number in the range [–1,1].	0
zIndex	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	"white", "black", "silver", "gray", "red", "maroon", "fuchsia", "purple", "lime", "green", "yellow", "olive", "blue", "navy", "aqua", or "teal".	white
textAlign		left
fontFamily	A prioritized list of font family names and/or generic family names.	DTV system dependent
fontStyle	Sets the style of the font ("normal", or "italic").	normal
fontSize	The size of a font.	DTV system dependent
fontVariant	Displays text in a "small-caps" font or a "normal" font.	normal
fontWeight	Sets the weight of a font ("normal", or "bold").	normal
player		
reusePlayer	Boolean value: "false", "true".	false
playerLife	"keep", "close".	close
moveLeft, moveRight, moveUp, moveDown, focusIndex	Positive integer.	nill
focusBorderColor;	"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	"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	An integer value specifying the attribute in pixels.	DTV system dependent: the value defined by the *default.focusBorderWidth*
focusBorderTransparency	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*

Table 7-12 – Reserved parameter/attribute and possible values

Parameter/attribute name	Value	Default
focusSrc, focusSelSrc	String: an URI.	nill
freeze	"true", "false".	"false"
transIn, transOut	A semicolon-separated list of <transition> element identifiers defined in the <transitionBase> element.	Empty string

Table 7-13 – 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%

Table 7-14 – Extended PropertyAnchor module

Elements	Attributes	Content
property	name, value, externable	Empty

The SwitchInterface module allows for the creation of <switch> element interfaces (see clause 7.2.4), 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).

It is important to remark that every element representing an object interface (<area>, <port>, <property>, and <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 none of the rules bind to the components defined by their child <mapping> elements is evaluated as true, 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 7-15.

Table 7-15 – Extended SwitchInterface module

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

7.2.6 Presentation Specification functionality

The Presentation Specification functionality has a single module named 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*.

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 document head, 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: *explicitDur* and *freeze,* defined by the Timing module (see clause 7.2.10); an attribute named *player*; an attribute named *region*, which refers to a region defined by elements of the Layout module (see clause 7.2.3); and key-navigation attributes: *moveLeft*, *moveRight*, *moveUp*; *moveDown*, *focusIndex*, *focusBorderColor*; *focusBorderWidth*; *focusBorderTransparency*, *focusSrc*, *selBorderColor*, and *focusSelSrc*, defined by the KeyNavigation module (see clause 7.2.12); and transition attributes: *transIn* and *transOut* (see clause 7.2.14).

NOTE – A <descriptor> element of a <media> element of application/x-ginga-NCL type shall not have the *player* attribute. In this case, an NCL player in a specific exhibition device needs to be defined.

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 values, such as *plan*; *rgbChromakey*; *background*; *visible*; *fit*; *scroll*; *transparency*; *style*; and also specific attributes for audio objects, such as *soundLevel*, *balanceLevel*, *trebleLevel* and *bassLevel*. 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*).

Besides all the aforementioned attributes, the <descriptor> element may also have attributes defined in the Transition effects functionality (see clause 7.2.14).

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 7.2.9).

The elements of the Descriptor module, their child elements, and their attributes shall comply with Table 7-16.

Table 7-16 – Extended Descriptor module

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*	Empty
descriptorBase	*id*	(importBase|descriptor|descriptorSwitch)+

It must be stressed that <descriptor> and <region> elements are just "syntactic sugar" for the NCL language 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 Linking functionality

The Linking functionality defines the Linking module, responsible for defining links using connectors. 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 7.2.11); or simply *connector_id*, for connectors defined in the document itself.

A <link> element must be ignored if the *xconnector* attribute is not defined, or if the *xconnector* attribute refers to an inexistent hypermedia connector.

The <link> element also contains child elements called <bind> elements, which allow associating nodes with connector roles (see clause 7.2.8). 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, that is, 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 7.2.5, except for media objects with imperative code content, as explained in clause 8.3.1.

If the connector element defines parameters (see clause 7.2.8), 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.

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

The elements of the linking module, their attributes, and their child elements shall comply with Table 7-17.

Table 7-17 – Extended Linking module

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

7.2.8 Connectors functionality

The NCL 3.0 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. These modules are the core by themselves of an XML application language (indeed other NCL 3.0 profiles) for the definition of connectors, which may be used to specify spatiotemporal synchronization relations, treating reference (user interaction) relations as a particular case of temporal synchronization relations.

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

A <causalConnector> element represents a causal relation that may be used for creating <link> elements in documents. In a causal relation, a condition shall be satisfied in order to trigger an action.

A <causalConnector> specifies a relation independently of relationships, that is, 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 <causalConnector> specifies, through its child elements, a set of interface points, called *roles*. A <link> element refers to a <causalConnector> 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 3.0 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, specified in NCL by the <area> element, or by the media node itself (whole content presentation), being presented and visible;

– 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), which shall be declared in a <property> child element of the node; and

– composition event, which is defined by the presentation of the structure of a composite node (represented by a <body>, <context>, or <switch> element). Composition events are used to present the composite map (composite organization). This functionality is optional.

Each event defines a state machine that should be maintained by the NCL formatter (see Figure 7-2). Moreover, every event has an associated attribute, named *occurrences*, which counts how many times the event transits from occurring to sleeping state during a document presentation. Events of presentation and attribution types have also an attribute named *repetitions*, which counts how many times the event shall be automatically restarted (transited from sleeping to occurring states) by the formatter. This attribute may contain the "indefinite" value, leading to an endless loop of the event occurrences, until some external interruption.

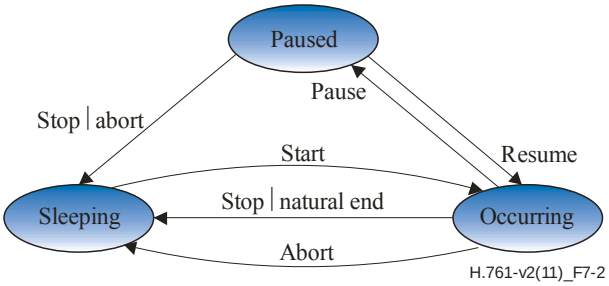

Figure 7-2 – Event state machine

Transition names for the event state machine shall comply with Table 7-18.

Table 7-18 – 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. In both cases, the *occurrences* attribute is incremented, and the *repetitions* attribute is decremented by one. If after being decremented, the *repetitions* attribute value is greater than zero, the event is automatically restarted (set again to the occurring state). When the presentation of an event is abruptly interrupted, through an abort presentation command, the event also goes to the sleeping state, but without incrementing the *occurrences* attribute and setting the *repetitions* attribute value to zero. 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), 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 context 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.

A composition event (associated to a composite node represented by a <body>, <context> or <switch> element) stays in the occurring state while the composition map is being presented.

Relations are defined based on event states, changes on the event state machines, on event attribute values, and on node (<media>, <body>, <context> or <switch> element) property values. The CausalConnectorFunctionality module allows only 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 7-19. 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 7.2.12, 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 7-19 – 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, otherwise the link shall be ignored. If minimal and maximal cardinalities are not informed, "1" shall be assumed as the default value for both parameters. When the maximal cardinality value is greater than one, several participants may play the same role, 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 7-20 the possible values for the *qualifier* attribute are: "or" and "and". If the qualifier establishes an "or" logical operator, the link action will be triggered whenever any condition occurs. If the qualifier establishes an "and" logical operator, the link action will be triggered after all the simple conditions occur. If not specified, the default value "or" shall be assumed.

Table 7-20 – 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 are satisfied.

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 7-21. 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. If this value cannot be retrieved, no attribution shall be made.

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, a qualifier shall be specified. Table 7-22 presents possible qualifier values. If the qualifier value is not specified, the default value "par" shall be assumed.

Table 7-21 – 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*

Table 7-22 – Action qualifier values

Role element	Qualifier	Semantics
simpleAction	*par*	All actions shall be executed in parallel
simpleAction	*seq*	All actions shall be executed in the bind sequence

A *delay* attribute may also be defined for a <simpleAction> specifying that the action shall be triggered only after having waited for the specified time. Besides, the <simpleAction> may also define a *repeat* attribute to be assigned to the *repetitions* attribute of the event, and a *repeatDelay* to be awaited before repeating the action.

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 has an *operator* attribute ("par" or "seq") relating its child elements: <simpleAction> and <compoundAction>. Parallel ("par") and sequential ("seq") compound actions specify that the execution of actions shall be performed in any order or in a specific order, respectively. A *delay* attribute may also be defined specifying that the compound action shall be applied after the specified delay.

NOTE 6 – When the sequential operator is used, actions shall be triggered in the specified order. However, an action does not need to wait for the previous one to be finished in order to be triggered.

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", the *attributeType* attribute specifies the event attribute ("occurrences" or "repetition") or the event state ("state"); if the *eventType* value is "selection", the *attributeType* attribute is optional and, if present, it may have the value "occurrences" (default) or "state"; if the *eventType* is "attribution" the *attributeType* is optional and may have the value "nodeProperty" (default), "occurrences", "repetition" or "state". In the first case, the event represents a node property to be evaluated; in the other ones the event represents the evaluation of the corresponding attribution event property or the 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 or event attribute.

The <assessmentStatement> element has a *comparator* attribute that compares the values inferred from its child elements (<attributeAssessment> element and <valueAssessment> element). In the case of <attributeAssessment>: a node property value [*eventType* = "attribution" and the *attributeType* = "nodeProperty"]; an event attribute value [*eventType* = ("presentation", "attribution" or "selection") and the *attributeType* = ("occurrences", or "repetition")]; or an event state [*eventType* = ("presentation", "attribution" or "selection") and the *attributeType* = "state"]. In the case of <valueAssessment>: a value of its *value* attribute. The *comparator* attribute shall have one of the values: "eq", "ne", "gt", "lt", "gte", or "lte". If a value different from these is specified, the <assessmentStatement> element shall be ignored.

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

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 7-23.

Table 7-23 – Extended CausalConnectorFunctionality module

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

The ConnectorBase module defines an element named <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 define connectors, store them in libraries (connector bases) that may be imported, and make them available to others for creating links.

The element of the ConnectorBase module, its attributes, and its child elements shall comply with Table 7-24.

Table 7-24 – Extended ConnectorBase module

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

7.2.9 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, named TestRule, TestRuleUse, ContentControl and DescriptorControl.

The TestRule module allows for the definition of rules that, when satisfied, select alternatives for document presentation. The specification of rules in NCL 3.0 was 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. 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". If a value different from these is specified, the <rule> element shall be ignored.

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 7-25.

Table 7-25 – Extended TestRule module

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

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 7-26.

Table 7-26 – Extended TestRuleUse module

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

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 or are test rules specifically defined and embedded in an NCL formatter implementation. 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. 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 did not exist.

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 7.2.11).

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

The ContentControl module elements, their attributes and child elements shall comply with Table 7-27.

Table 7-27 – Extended ContentControl module

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

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, or test rules specifically defined and embedded in an NCL formatter implementation. 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 7-28.

Table 7-28 – Extended DescriptorControl module

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

During a document presentation, from the moment a <switch> is evaluated, 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. During a document presentation, from the moment 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, that is, while any presentation event associated with the <media> element is in the "occurring" or "paused" state.

NOTE – NCL formatters should delay the switch evaluation to the moment that a link anchoring in the switch needs to be evaluated. The descriptorSwitch evaluation should be delayed until the object referring the descriptorSwitch needs to be prepared to be presented.

7.2.10 Timing functionality

The Timing functionality defines the Timing module. The Timing module allows for the definition of temporal attributes for document components. Basically, this 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, that is, until its end is determined by an external event (for example, coming from a <link> evaluation), or by the *explicitDur* value for that object.

The *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.

7.2.11 Reuse functionality

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

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

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 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. Although NCL defines its layout model, nothing prevents an NCL document from using other layout models, since they define regions where objects may be presented, as for example SMIL 2.1 [b-W3C SMIL 2.1] layout models. On importing a <regionBase>, an optional attribute named 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 (the document that 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.

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. The bases will be treated as if each one were imported by an <importBase> element. 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 operation has also the transitive property, that is, if *documentA* imports *documentB* that imports *documentC*, then *documentA* imports *documentC*. However, the *alias* defined for *documentC* inside *documentB* shall not be considered by *documentA*. 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 7-29.

Table 7-29 – Extended Import module

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

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; that is, 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 *D* import.

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 will represent the same node previously defined in the document <body> itself or in an external imported <body>. This 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 a <body> element that will represent the same context, which is previously defined in the document <body> itself or in an external imported <body>. This 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. All other attributes and child elements, if they are 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, *instance*, may be defined. 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. The *instance* attribute is defined in the ExtendedEntityReuse module and has "new" as its default string value.

The referred element and the element that refers to it shall be considered the same, regarding its data specification. In other words it means that a single NCM node (see [b-NCM Core]) can be represented by more than one NCL element. As nodes contained in an NCM composite node define a set, an NCM node may be represented by no more than one NCL element inside a composition. This means that the *id* attribute of an NCL element representing an NCM node is not only a unique identifier for the element, but also the unique identifier for the NCM node in the composition.

EXAMPLE – Consider the NCL element (*node1*) that defines an NCM node. The NCL elements that refer to it (*node1ReuseA, node1ReuseB*) represent the same NCM node. In other words, the single NCM node is represented by more than one NCL element (*node1, node1ReuseA,* and *node1ReuseB*). Moreover, since nodes contained in an NCM composite node define a set, the NCL elements *node1, node1ReuseA,* and *node1ReuseB* shall each be declared inside a different composition.

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

– Assume the set of <media> elements composed of the referred <media> element and all the referring <media> elements. If any element of the subset formed by the referred <media> element and all other <media> elements having the *instance* attribute equal to "instSame" or "gradSame" is scheduled to be presented, all other elements in this subset, which are not child descendants of a <switch> element, are also assumed as scheduled for presenting, and more than that, when they are being presented, they shall be represented by the same presentation instance. 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.
– If the *instance* attribute is equal to "instSame", all scheduled nodes of the subset shall be presented at the same time through a unique instance (start instruction applied on all subset elements).
– If the *instance* attribute is equal to "gradSame", all scheduled nodes of the subset shall be presented through a unique instance, but now gradually, as start instructions are applied, coming from a link, etc.
– The common instance 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.

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. When they are individually 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> element have the same behaviour regarding reuse, including the <media> element of application/x-ginga-settings (or application/x-ncl-settings) type.

7.2.12 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.

The *focusIndex* attribute specifies an index for the <media> element to which the focus may be applied, when this element is in exhibition. The *focusIndex* may be defined using a <property> or a <descriptor> element. When this property is not defined, the object is considered as if no focus could be set. 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 index value. Values of *focusIndex* attribute shall be unique in an NCL document. Otherwise, the repeated attributes will be ignored if at a certain moment there is more than one <media> element to gain the focus. Moreover, when a <media> element refers to another <media> element (using the *refer* attribute specified in clause 7.2.11), it shall ignore the *focusIndex* associated with the referred <media> element.

The *moveUp* attribute specifies a value equal to the *focusIndex* value associated to an element to which the focus should be applied when the "up arrow key" is pressed. The *moveDown* attribute specifies a value equal to the *focusIndex* value associated to an element to which the focus should be applied when the "down arrow key" is pressed. The *moveRight* attribute specifies a value equal to the *focusIndex* value associated to an element to which the focus should be applied when the "right arrow key" is pressed. The *moveLeft* attribute specifies a value equal to the *focusIndex* value associated to an element to which the focus should be applied when the "left arrow key" is pressed.

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.

The *focusSrc* attribute 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* attribute defines the decorative colour.

In an implementation in conformance with Ginga-NCL specification, the *focusBorderColor* attribute 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* attribute 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), and the *focusBorderTransparency* attribute 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 the *focusBorderColor,* the *focusBorderWidth,* or the *focusBorderTransparency* are not defined, default values shall be assumed. These values are specified in the properties of the <media> element of application/x-ncl-settings type: *default.focusBorderColor, default.focusBorderWidth, default.focusTransparency*, respectively.

When an element on focus is selected by pressing the activation (select or enter) key, the *focusSelSrc* attribute can specify an alternative media source to be presented, instead of the current presentation. This attribute 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* attribute (default value specified by the

default.selBorderColor of the <media> element of application/x-ncl-settings type), the width of the decorative border defined by the *focusBorderWidth* attribute, and the decorative colour transparency defined by the *focusBorderTransparency* attribute.

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 "Nested Context Language 3.0: Part 12 – Support to Multiple Exhibition Devices".

NOTE – The focus control may also be passed 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.

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

7.2.14 Transition Effects functionality

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

The TransitionBase module is defined by NCL 3.0 and consists of 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 7-30.

Table 7-30 – Extended TransitionBase module

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

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

In NCL 3.0 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.

Seven <transition> element attributes 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 is used to specify the general transition. If the named type is not supported by the NCL formatter, the transition is ignored. Note that this is not an error condition, since implementations are free to ignore transitions.

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 7-31 shall be supported. The other subtypes defined in SMIL specifications [b-W3C SMIL 2.1] are optional

Table 7-31 – 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 default duration is 1 second.

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, we may want to begin a crossfade with the destination image being already 40% faded in. In this case, *startProgress* would be 0.4. The default value is 0.0.

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 default value is 1.0.

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. Transitions that do not have a reverse interpretation should have the *direction* attribute ignored and the default value of "forward" assumed.

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. 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. The default value is "black".

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. 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 *transIn* and *transOut* attributes are added to <descriptor> elements. The default value of both attributes is an empty string, which indicates that no transition shall be performed. The properties may also be defined using <property> elements.

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 should be performed if the user-agent has implemented this transition. If this transition is not available, then the second transition in the list should be performed, and so on. If the value of the *transIn* attribute or the *transOut* attribute does not correspond to the value of the XML identifier of any one of the transition elements previously defined, then this is an error. In the case of this error, the value of the attribute should be considered to be the empty string and therefore no transition should be performed.

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 *horRepeat* attribute specifies how many times to perform the transition pattern along the horizontal axis. The default value is 1 (the pattern occurs once horizontally). The *vertRepeat* attribute specifies how many times to perform the transition pattern along the vertical axis. The default value is 1 (the pattern occurs once vertically). 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. The default value is 0. 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 default value for this attribute is "black".

The element of the Extended Transition Module, its child elements, and its attributes shall comply with Table 7-32.

Table 7-32 – Extended Transition module

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

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].

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 7-33.

Table 7-33 – Extended Metainformation module

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

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 3.0 Full profile, also called *NCL 3.0* Language profile, is the "complete profile" of the NCL 3.0 language. It comprises all NCL modules (including those discussed in clause 7.2) and provides all facilities for declarative authoring of NCL documents.

The profiles defined for IPTV are:

a) NCL 3.0 Enhanced DTV profile: includes the Structure, Layout, Media, Context, MediaContentAnchor, CompositeNodeInterface, PropertyAnchor, SwitchInterface, Descriptor, Linking, CausalConnectorFunctionality, ConnectorBase, TestRule, TestRuleUse, ContentControl, DescriptorControl, Timing, Import, EntityReuse, ExtendedEntityReuse KeyNavigation, Animation, TransitionBase, Transition and Metainformation modules of NCL 3.0. The tables in clause 7.2 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 NCL30EDTV.xsd of this Recommendation).

b) NCL 3.0 CausalConnector profile: allows for the creation of simple hypermedia connectors. This profile includes the Structure, CausalConnectorFunctionality, and ConnectorBase modules. In the profile, the <body> element of the Structure module is not used (see XML schemas in the electronic attachment CausalConnector.xsd of this Recommendation).

8 Media objects in NCL presentations

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

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

Figure 8-1 illustrates a modular organization for an NCL presentation environment. Note that media players are plug-in modules of the presentation engine. Since it can be interesting to use already existing media players that can have proprietary interfaces that are not compatible with the one required by the presentation engine, it will be necessary to develop modules to make the necessary adaptations. In this case, the media player will be constituted of an adapter, besides the player itself.

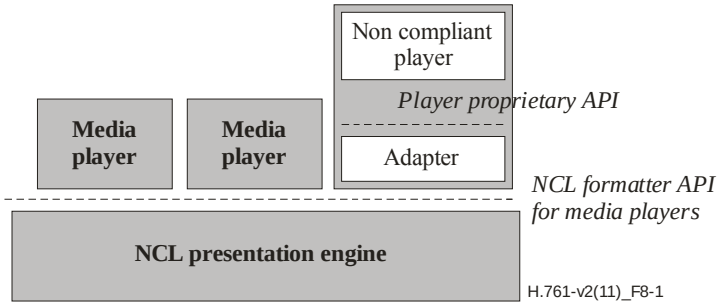

Figure 8-1 – APIs for integrating media players with an NCL presentation engine implementation

As the Ginga-NCL architecture and implementation is a choice of each receiver developer, the following clauses do not intend to standardize the syntax of the presentation engine API. The goal is just to define the expected behaviour of a media player when controlling objects that take part in an NCL document.

8.1 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 and text/html type) and different from any media object containing imperative or functional 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. This identification is called in this clause *representationObjectId*.

8.1.1 *start* instruction for presentation events

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

The *start* instruction issued by a formatter shall inform the following parameters to the media player: the locator of the content of the media object to be controlled, a list of all properties associated with the media object, the media object identification during execution (*representationObjectId*), 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 default content anchor), the presentation event that needs to be started (called here main event), an optional offset-time and an optional delay-time.

NOTE – The presentation properties that are not required by a DTV system that conforms with this Recommendation must be ignored by the media player.

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

The descriptors shall be chosen by the formatter following the directives specified in the NCL document. If the *start* instruction results from a link action that has a descriptor explicitly declared in its <bind> element (*descriptor* attribute of the children <bind> element of the <link> element), the resulting descriptor shall merge the attributes of the bind descriptor with the attributes of the descriptor specified in the corresponding <media> element, if this attribute was specified. For the common attributes, the <bind> descriptor information shall superpose the <media> descriptor data. If the <bind> element does not contain an explicit descriptor, the evaluated descriptor shall be the <media> descriptor, if this attribute was specified. Otherwise, a default descriptor for that *type* of <media> shall be chosen by the formatter. Based on this procedure, the list of <descriptor> elements' *id* that are associated with the media object is computed, and, unifying the properties defined by the resulting descriptor with those properties explicitly defined by the <media> element, the list of properties associated with the media object is evaluated.

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

The offset-time parameter is optional, it has "zero" as its default value, and is meaningful only for continuous media or static media with explicit duration. In this case, this parameter defines a time offset from the beginning (beginning-time) of the main event, from which the presentation of the main event shall be immediately started (i.e., it commands the player to jump to the beginning-time + offset-time). Obviously, the offset-time value shall be lower than the main event duration; otherwise, that *start* instruction shall be ignored. If the offset-time is greater than zero, the media player shall put the main event in the *occurring* state, but the event *starts* transition shall not be notified. If the offset-time is zero, the media player shall put the main event in the *occurring* state and notify the *starts* transition occurrence.

The events that would have their end-times before the beginning-time of the main event, and the events that would have their beginning times after the end-time of the main event do not need to be monitored by the media player (the formatter should do this verification when building the monitored event list).

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

Monitored events that would have their end times after the main event beginning-time, but before the start time (beginning-time + offset-time) shall have their *occurrences* attribute incremented, but the *starts* and *stops* transitions shall not be notified.

The delay-time is also an optional parameter and its default value is "zero" too. If greater than zero, this parameter contains a time to be waited by the media player before starting the presentation.

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

8.1.2 *stop* instruction for presentation events

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

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

NOTE 1 – The *stop* instruction shall transit the monitored events to the *sleeping* state irrespective of whether a transition effect is being applied to the media object. In other words, the transition effect shall also be stopped. Transition effects are never applied after an object is subjected to a *stop* instruction.

NOTE 2 – When all media objects referring to the elementary stream that carries the service main video are in the sleeping state, the main video shall be dimensioned to 100% of the screen. The main video can be redimensioned only using a media object (referring to the main video) in presentation. The same happens with the main audio. When all media objects referring to the elementary stream that carries the service main audio are in the sleeping state, the main audio shall be presented with 100% of its volume.

8.1.3 *abort* instruction for presentation events

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

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

8.1.4 *pause* instruction for presentation events

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

If the object is not being presented (the main event, passed as a parameter when the media object was started, is not in the *occurring* state) and the media player is not waiting for the start delay, the instruction shall be ignored. If the object is being presented, the main event and all monitored events in the *occurring* state shall transit to the *paused* state and their *pauses* transitions shall be notified. The object presentation shall be paused and the pause elapsed time shall not be considered as part of the object duration. As an example, if an object has an explicit duration of 30 s and, after 25 s it is paused, even if the object stays paused for 7 min, after resuming the object main event shall stay occurring for 5 s. If the main event is still not occurring because the media player is waiting for the start delay, the media object shall wait for a resume instruction to continue waiting for the remaining start delay.

8.1.5 *resume* instruction for presentation events

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

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

8.1.6 *start* instruction for attribution events

The *start* instruction may be applied to an object property independently of whether the object is being presented or not (in the latter case, although the object is not being presented, its media player shall be already instantiated). The *start* instruction needs to identify the media object being controlled (*representationObjectId*) and a monitored attribution event, and needs to determine a value to be assigned to the property wrapped by the event, the duration of the attribution process, and 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.

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 *start* instruction.

8.1.7 *stop, abort, pause and resume* instructions for attribution events

The *stop, abort, pause and resume* instructions need to identify a media object already being controlled (*representationObjectId*) and an attribution event being monitored.

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

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

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

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

8.1.8 *addEvent* instruction

The *addEvent* instruction is issued in the case of receiving an *addInterface* NCL editing command (see clause 9). The instruction needs to identify a media object already being controlled and a new event that shall be included to be monitored. All rules applied to the intersection of monitored events with the main event shall be applied to the new event. If the new event start time is 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.

8.1.9 *removeEvent* instruction

The *removeEvent* instruction is issued in the case of receiving a *removeInterface* NCL editing command. The instruction needs to identify a media object already being controlled and a monitored event that should not be controlled any more. The event state shall be put in the *sleeping* state without generating any transition.

8.1.10 Natural end of a 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 and without incrementing the *occurrences* attribute. It is important to remark that if the main event corresponds to an object internal temporal anchor, when this anchor presentation finishes, the whole media object presentation shall finish.

8.2 Expected behaviour 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.

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 document. Thus, a two-way synchronization can be established between a declarative code and the remainder of the NCL document.

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 formatter. 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.

8.2.1 *start* instruction for presentation events

The *start* instruction issued by a formatter 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*), 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), the *clip* or *label* content anchor, or by default the *whole content anchor*, identifying the associated temporal chain sections to be started (called here main event), an optional offset-time and an optional delay-time. From the locator (*src* attribute of the media object), the declarative hypermedia object player tries to locate the temporal chain section and start its execution. If the content cannot be located, the player shall finish the starting operation, without performing any action.

From these *start* instruction parameters, the formatter shall follow the same procedure defined for basic media objects, defined in clause 8.1.1, with the exception presented in the next three paragraphs.

If a declarative hypermedia object player receives a *start* instruction for a temporal chain already being presented (paused or not), it shall ignore the instruction and keep on controlling the ongoing presentation. However, different from what is performed on other <media> elements, if the start instruction is for a temporal chain that is not being presented, the instruction must be executed even if another temporal chain is being presented (paused or occurring). As a consequence, different from what happens for other <media> elements, a <simpleAction> element with an *actionType* attribute equal to "stop", "pause", "resume" or "abort" shall be bound through a link to a 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 assumed, as usual, meaning that the presentation of every chain shall be started in parallel.

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

8.2.2 *stop* instruction for presentation events

The *stop* instruction needs to identify a temporal chain already being controlled (or by default, all of them). To identify the temporal chain means to identify the corresponding declarative hypermedia object under control (*representationObjectId*), and a <media> element's interface.

The *stop* instruction issued by an NCL formatter 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) and the declarative hypermedia object player is not waiting to exhibit that temporal chain due to a delayed *start* instruction. 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* state or in the *paused* state with end time equal or previous to the main event end time shall transit to the *sleeping* state, and their *stops* transitions shall be notified. Monitored events in the *occurring* state or in the *paused* state with end time posterior to the main event end time shall be put in the *sleeping* state, but their *stops* transitions shall not be notified and their *occurrences* attribute shall not be incremented. The temporal chain

presentation shall be stopped. If the *repetitions* event attribute is greater than zero, it shall be decremented by one and the main event presentation shall restart after the repeat delay time (the repeat delay shall have been passed to the media player as the start delay parameter). If the temporal chain associated with the specified interface is waiting to be presented after a delayed *start* instruction and a *stop* instruction is issued, the previous *start* instruction shall be removed.

Unlike other basic <media> elements, if any content anchor is stopped and all other presentation events are in the *sleeping* state, the *whole content anchor* shall be put in the *sleeping* state. If a content anchor is stopped and at least 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* instruction is applied to a declarative hypermedia object without specifying the node's interface, the *whole content anchor* is assumed. In this case, *stop* instructions shall be issued for all temporal chains.

8.2.3 *abort* instruction for presentation events

The *abort* instruction needs to identify a temporal chain already being controlled (or, by default, all of them). To *identify the temporal chain* means to identify the corresponding declarative hypermedia object under control (*representationObjectId*), and a <media> element's interface.

The *abort* instruction issued by an NCL formatter shall be ignored by a declarative hypermedia object player if the temporal chain associated with the specified interface is not being presented (i.e., if none of the events in the object list of events is in the *occurring* or *paused* state) and the declarative hypermedia object player is not waiting to exhibit that temporal chain due to a delayed *start* instruction. 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* state 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. If the *repetitions* event attribute is greater than zero, it shall be set to zero and the temporal chain presentation shall not restart. If the temporal chain associated with the specified interface is waiting to be presented after a delayed *start* instruction and an *abort* instruction is issued, the previous *start* instruction shall be removed.

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

8.2.4 *pause* instruction for presentation events

The *pause* instruction needs to identify a temporal chain already being controlled (or, by default, all of them). To *identify the temporal chain* means to identify the corresponding declarative hypermedia object under control (*representationObjectId*), and a <media> element's interface.

The *pause* instruction issued by an NCL formatter shall be ignored by a declarative hypermedia object player if the temporal chain associated with the specified interface is not being presented (i.e., if none of the events in the object list of events is in the *occurring* or *paused* state) and the declarative hypermedia object player is not waiting to exhibit that temporal chain due to a delayed *start* instruction. 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* state 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 its duration.

If the temporal chain associated with the specified interface is waiting to be presented after a delayed *start* instruction and a *pause* instruction is issued, the temporal chain shall wait for a resume instruction to continue waiting for the remaining start delay.

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

8.2.5 *resume* instruction for presentation events

The *resume* instruction needs to identify a temporal chain already being controlled (or, by default, all of them). To *identify the temporal chain* means to identify the corresponding declarative hypermedia object under control (*representationObjectId*), and a <media> element's interface.

The *resume* instruction issued by an NCL formatter shall be ignored by a declarative hypermedia object player if the temporal chain associated with the specified interface is not paused, or the declarative hypermedia object player is not waiting to exhibit that temporal chain due to a delayed *start* instruction. If the declarative hypermedia object player is paused waiting for the start delay, it shall resume the wait from the instant it was paused. If the temporal chain 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.

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

8.2.6 *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. Events chain of the same temporal chain in the *occurring* state with end time posterior to the main event end time shall be put in the *sleeping* state, but without generating the *stops* transition and without incrementing the *occurrences* attribute.

In the case of a natural end of a main event, if the *repetitions* event attribute is greater than zero, it shall be decremented by one and the main event presentation shall restart after the repeat delay time (the repeat delay shall have been passed to the media player as the start delay parameter).

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

8.2.7 *start, stop, abort, pause and resume* instructions for attribution events

All instructions for attribution events have the same effect on the corresponding property attribution as they have on any property attribution of any type of NCL object, as specified in clauses 8.1.6 and 8.1.7.

8.3 Expected behaviour of imperative-object players in NCL applications

In an implementation in conformance with Ginga-NCL specification, EDTV profile of NCL 3.0, support of the application/x-ginga-NCLua type to be associated with the <media> element is required for Lua imperative code content (file extension .lua). It is also possible to support other imperative object types.

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

As stated in clause 7.2.5, 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, call-backs in the imperative code span shall be triggered. The way these call-backs 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* and it is declared by default in NCL documents. This content anchor, however, has a special meaning. It represents the execution of any code span inside the imperative code object. Another content anchor is also defined by default, called *main* content anchor. 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.

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

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

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

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

NCL formatter. If the <property> element is mapped to a code span, it shall be called and the output value resulting from its execution shall be returned by the imperative player to the NCL formatter.

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

As with all media object players, once instantiated, the imperative object player shall execute an initialization procedure. However, different from other media players, this initialization code is specified by the author of the imperative code. This initialization procedure is executed only once for each instance; 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 formatter. Note that at least the code span associated with the main content anchor shall be created during the initialization procedure.

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

8.3.1 *start* instruction for presentation events

The *start* instruction issued by a formatter 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, the media object identification during execution (*representationObjectId*), a list of events (defined by the <media> element's <area> and <property> child elements, and by the default content anchors) that need to be monitored by the imperative object player, the content anchor *label,* or by default the *main* content anchor, identifying the associated imperative code to be started, and an optional delay time. From the *src* attribute 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.

From these *start* instruction parameter, the formatter shall follow the same procedure defined for basic media objects, defined in clause 8.1.1, with the exception presented in the next two paragraphs.

Unlike what is performed on basic <media> elements, if an imperative object player receives a *start* instruction for an event associated with a content anchor, and this event is in the *sleeping* state, it shall start the execution of the imperative code associated with the element, even though other portion of the object's imperative code is in execution (paused or not). However, if the event associated with the target content anchor is in the *occurring* or *paused* state, the *start* instruction shall be ignored by the imperative code player that controls the ongoing execution. As a consequence, differently from what happens for other <media> elements, a <simpleAction> element with an *actionType* attribute equal to "stop", "pause", "resume" or "abort" shall be bound through a link to an imperative node interface, which shall not be ignored when the action is applied.

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

span to command any event state change, and to inform if the associated transition shall be notified, if the code span execution starts another code span associated with a content anchor.

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

8.3.2 *stop* instruction for presentation events

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 object in execution and a <media> element's interface.

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

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

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

8.3.3 *abort* instruction for presentation events

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

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

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

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

8.3.4 *pause* instruction for presentation events

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

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

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

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

8.3.5 *resume* instruction for presentation events

The *resume* instruction needs to identify an imperative code span already being controlled. To *identify the imperative code span* means to identify the corresponding media object in execution and a <media> element's interface.

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

For the same reason discussed in the *start* instruction, except for the event associated with the *whole content anchor*, it is the responsibility of the paused code span, before it 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.

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

8.3.6 Natural end of a code execution for presentation events

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

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

8.3.7 *start* instruction for attribution events

The *start* instruction issued by an NCL formatter may be applied to an imperative object's property independently from whether the object is 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 *start* instruction needs to identify the imperative object (representationObjectId), a monitored attribution event, and, if it is the case, a value to be passed to the imperative code wrapped by the event. When setting a value to an attribute, the imperative object player shall set the event state machine to the *occurring* state, and after finishing the attribution, again to the *sleeping* state, generating the *starts* transition and afterwards the *stops* transition.

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

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

8.3.8 *stop, abort, pause* and *resume* instructions for attribution events

With the exception of the *start* instruction discussed in the previous clause, all other instructions have the same effect on the corresponding property attribution as they have on any property attribution of any type of object, as specified in clause 8.1.7.

8.4 Expected behaviour of media players after instructions applied to composite objects

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

8.4.1 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 clause 7.2.8. Analogously, an <attributeAssessment> with *eventType* attribute value equal to "presentation" and *attributeType* equal to "state", "occurrences" or "repetitions" may be bound by a link to a composite node (represented by a <context>, <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 instruction shall also be reflected to the event state machines of the composite child nodes, as explained in the following clauses.

8.4.2 Starting a context presentation

If a <context> or <body> element participates in an action role whose action type is "start", when the action is triggered without referring to any specific interface, the *start* instruction shall also be applied to all presentation events mapped by the <context> or <body> element's ports.

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

8.4.3 Stopping a context presentation

If a <context> or <body> element participates in an action role whose action type is "stop", when the action is triggered without referring to any specific interface, the *stop* instruction shall also be applied to all presentation events of the composite child nodes.

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

8.4.4 Aborting a context presentation

If a <context> or <body> element participates in an action role whose action type is "abort", when the action is triggered without referring to any specific interface, the *abort* instruction shall also be applied to all presentation events of the composite child nodes.

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

8.4.5 Pausing a context presentation

If a <context> or <body> element participates in an action role whose action type is "pause", when the action is triggered without referring to any specific interface, the *pause* instruction shall also be applied to all presentation events of the composite child nodes that are in the occurring state.

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

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

8.4.6 Resuming a context presentation

If a <context> or <body> element participates in an action role whose action type is "resume", when the action is triggered without referring to any specific interface, the *resume* instruction shall also be applied to all presentation events of the composite child nodes that are in the paused state, except those that were already paused before the composite has been paused.

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

8.5 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 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 instructions 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 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 formatter and its private base manager module.

The NCL formatter is in charge of receiving an NCL document and controlling its presentation, trying to guarantee that the specified relationships among media objects are respected. The formatter deals with NCL documents that are collected inside a data structure known as *private base*. NCL documents 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 documents (documents being presented).

Editing commands are wrapped in a structure called event descriptors. Event descriptors have a structure composed basically of an id (identification), a time reference and a private data field. The identification uniquely identifies editing command events. The time reference indicates the exact moment to trigger the event. 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 9-1. 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 (that is, it cannot be multiplexed with other editing commands with the same *commandTag*), with the *finalFlag* equal to 1, except for the last descriptor, which 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 document manipulation in a private base (to add, remove, and save a document in an open private base, and to start, pause, resume, and stop document 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 identical to the syntax notation used in the NCL 3.0 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 formatter, 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 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 located 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) in parenthesis. 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)	0x00	Opens an existing private base located using the location parameter. If the private base does not exist or the location parameter is not communicated, a new base is created with the baseId identifier. The location parameter shall specify the storage device in the receiver environment and the path for opening the base.
activateBase (baseId)	0x01	Turns on an open private base. All applications are then available to be started.
deactivateBase (baseId)	0x02	Turns off an open private base. All applications shall be stopped.
saveBase (baseId, location)	0x03	Saves all private base content into a persistent storage device (if available). The location parameter shall specify the device and the path for saving the base.
closeBase (baseId)	0x04	Closes the open private base and disposes of all the private base content.
addDocument (baseId, {uri, id}+)	0x05	Adds an NCL document to an open private base. The NCL document's files can be: i) sent in the data-cast 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 for the middleware to map any file reference present in the NCL document 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, in case the NCL document is not received as a pushed file.

Table 9-2 – Editing commands for Ginga's private base manager

Command string	Command tag	Description
removeDocument (baseId, documentId)	0x06	Removes an NCL document from an open private base.
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 document with regard 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 document shall be started immediately from its beginning time+offset+(NPT – nptTrigger) seconds. NOTE 1 – Only in this case, the offset parameter value may be a negative time value, but offset+(NPT – nptTrigger) seconds shall be a positive time value.
		iii) If nptTrigger is equal to 0, the document shall start its presentation immediately from its beginning time+offset. NOTE 2 – If the refDocumentId or refNodeId is specified as "null", the NPT value is assumed as "do it now", independently from the value specified in the event descriptor. NOTE 3 – If the interfaceId parameter is specified as "null", all <port> element of the <body> element shall be triggered (started). NOTE 4 – If the offset parameter is specified as "null", it shall assume the "0" as value.
stopDocument (baseId, documentId)	0x08	Stops the presentation of an NCL document in an active private base. All document events that are occurring shall be stopped.
pauseDocument (baseId, documentId)	0x09	Pauses the presentation of an NCL document in an active private base. All document events that are occurring shall be paused.
resumeDocument (baseId, documentId)	0x0A	Resumes the presentation of an NCL document in an active private base. All previously document events that were paused by the pauseDocument editing command shall be resumed.

Table 9-2 – Editing commands for Ginga's private base manager

Command string	Command tag	Description
saveDocument (baseId, documentId, location)	0x2E	Saves an NCL document 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 document. If the NCL document to be saved is running in the open private base, its presentation is first stopped (all document events that are occurring shall be stopped).
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="null") of an NCL document in an open private base.
removeRegion (baseId, documentId, regionId)	0x0C	Removes a <region> element from a <regionBase> of an NCL document in an open private base.
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.
removeRegionBase (baseId, documentId, regionBaseId)	0x0E	Removes a <regionBase> element from the <head> element of an NCL document in an open private base.
addRule (baseId, documentId, xmlRule)	0x0F	Adds a <rule> element to the <ruleBase> of an NCL document in an open private base.
removeRule (baseId, documentId, ruleId)	0x10	Removes a <rule> element from the <ruleBase> of an NCL document in an open private base.
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.
removeRuleBase (baseId, documentId, ruleBaseId)	0x12	Removes a <ruleBase> element from the <head> element of an NCL document in an open private base.
addConnector (baseId, documentId, xmlConnector)	0x13	Adds a <connector> element to the <connectorBase> of an NCL document in an open private base.
removeConnector (baseId, documentId, connectorId)	0x14	Removes a <connector> element from the <connectorBase> of an NCL document in an open private base.
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.
removeConnectorBase (baseId, documentId, connectorBaseId)	0x16	Removes a <connectorBase> element from the <head> element of an NCL document in an open private base.

Table 9-2 – Editing commands for Ginga's private base manager

Command string	Command tag	Description
addDescriptor (baseId, documentId, xmlDescriptor)	0x17	Adds a <descriptor> element to the <descriptorBase> of an NCL document in an open private base.
removeDescriptor (baseId, documentId, descriptorId)	0x18	Removes a <descriptor> element from the <descriptorBase> of an NCL document in an open private base.
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.
removeDescriptorSwitch (baseId, documentId, descriptorSwitchId)	0x1A	Removes a <descriptorSwitch> element from the <descriptorBase> of an NCL document in an open private base.
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 separate file system, the xmlDescriptorBase parameter is just a reference to this content.
removeDescriptorBase (baseId, documentId, descriptorBaseId)	0x1C	Removes a <descriptorBase> element from the <head> element of an NCL document in an open private base.
addTransition (baseId, documentId, xmlTransition)	0x1D	Adds a <transition> element to the <transitionBase> of an NCL document in an open private base.
removeTransition (baseId, documentId, transitionId)	0x1E	Removes a <transition> element from the <transitionBase> of an NCL document in an open private base.
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 separate file system, the xmlTransitionBase parameter is just a reference to this content.
removeTransitionBase (baseId, documentId, transitionBaseId)	0x20	Removes a <transitionBase> element from the <head> element of an NCL document in an open private base.
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.
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.

Table 9-2 – Editing commands for Ginga's private base manager

Command string	Command tag	Description
addImportedDocumentBase (baseId, documentId, xmlImportedDocumentBase)	0x23	Adds an <importedDocumentBase> element to the <head> element of an NCL document in an open private base.
removeImportedDocumentBase (baseId, documentId, importedDocumentBaseId)	0x24	Removes an <importedDocumentBase> element from the <head> element of an NCL document in an open private base.
addImportNCL (baseId, documentId, xmlImportNCL)	0x25	Adds a <importNCL> element to the <importedDocumentBase > element of an NCL document in an open private base.
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.
addNode (baseId, documentId, compositeId, {uri, id}+)	0x27	Adds a node (<media>, <context>, or <switch> element) to a composite node (<body>, <context>, or <switch> element) of an NCL document 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 files; the {uri,id} pair 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 for the middleware to map 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.
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 document in an open private base.
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 document in an open private base.

Table 9-2 – Editing commands for Ginga's private base manager

Command string	Command tag	Description
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 document 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.
addLink (baseId, documentId, compositeId, xmlLink)	0x2B	Adds a <link> element to a composite node (<body>, <context>, or <switch> element) of an NCL document in an open private base.
removeLink (baseId, documentId, compositeId, linkId)	0x2C	Removes a <link> element from a composite node (<body>, <context>, or <switch> element) of an NCL document in an open private base.
setPropertyValue (baseId, documentId, nodeId, propertyId, value)	0x2D	Sets the value for a property. The propertyId shall identify a <property> element's name attribute or a <switchPort> element's id attribute. The <property> or <switchPort> shall belong to a node (<body>, <context>, <switch> or <media> element) of an NCL document in an open private base identified by the parameters.

The identifiers used in the commands shall comply 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 IPTV system. When the parameter is specified as "null", it shall assume the tuned TV channel 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 assume 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.
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.

Identifiers	Definition
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 3.0 Command profile defined by the XML Schema that is found in the electronic attachment NCL30EdCommand.xsd to this Recommendation.

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

9.3 NCL editing commands in Ginga-NCL

As stated in the scope of this Recommendation, NCL can be used in other declarative environments besides Ginga-NCL. When Ginga-NCL is used, some constraints are defined.

Ginga associates at least one private base with each TV channel (set of services) – the TV channel's default private base. When a TV channel is tuned, its corresponding default private base is opened and activated by the Private Base Manager; other private bases shall be deactivated. Other private bases can be opened (or created), but at most one associated with each service of a TV channel. When a TV channel has just one service, it shall have just one private base associated with the TV channel, namely the TV channel's default private base.

NOTE 1 – The identifier (baseId parameter) of the private base associated with the TV channel shall be equal to the TV channel identifier. The identifier (baseId parameter) of the possible private base associated with a TV channel's service shall be equal to the "value of the channel identifier.value of the service identifier".

NCL resident applications are managed in a specific private base.

For security reasons, only one private base may be active at a time, among those controlled through the tuned TVchannel. The simplest and most restricted way to manage private bases is to have only one private base open at a time, among those controlled through the tuned TV channel. However, the number of private bases that may be kept open is a specific middleware implementation decision.

NOTE 2 – Only private bases created by NCL editing commands sent through the tuned TV channel and the default private base associated with this TV tuned TV channel can be controlled by NCL Editing Commands sent through the same tuned TV channel.

NOTE 3 – NCLua events (NCL editing command events) generated by NCLua objects running in private bases controlled by NCL editing commands sent through a tuned TV channel may control just these mentioned private bases.

In Ginga-NCL, event descriptors (defined in clause 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 in [ISO/IEC 13818-6], a DSM-CC stream-event descriptor has a very similar structure as 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
}	

Several alternatives have been defined by Ginga-NCL to transport unsolicited NCL editing command parameters. All alternatives are optional, but if one of them is chosen, it shall comply with clauses 9.3.1 and 9.3.2.

9.3.1 DSM-CC transport of editing commands 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 to event ids defined in event descriptors. The private base manager should register itself as a listener of the event descriptors it handles, by using event names; or the name "*nclEditingCommand*" in case of editing commands.

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 an 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, after the *mappingType* field comes a list of event identifiers 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 objects, should register themselves as listeners of the 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 over 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 comply with clause 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 over several sections. Every data structure can be transmitted in any order and as many times as necessary. All NCL sections 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), 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 clause 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 NCL section 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 formats like FLUTE packets, or MPEG-2 specific section types.

NCL sections can also transport the aforedefined 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 consists in 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 to transport 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 Lua library

The following functions are platform dependent and were removed in the implementation:

1) from module *package*: *loadlib*;
2) from module *os*: *clock, execute, exit, getenv, remove, rename, tmpname* and *setlocale*;
3) from module *debug*: all functions.

10.2 Execution model

The lifecycle of an NCLua object is controlled by the NCL formatter. The formatter 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 document, as defined in clause 8.4.

As with all media object players, once instantiated, the Lua player shall execute an initialization procedure. However, unlike 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(s) for communicating with the NCL formatter.

After the initialization, the execution of the NCLua object becomes event oriented in both directions. That is, any action commanded by the NCL formatter reaches the registered event handlers, and any NCL event state change notification is sent as an event to the NCL formatter (as

for example, the natural end of a procedure execution). The Lua Player is then ready to perform any *start* or *set* instruction (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 each function in the above modules uses the following naming convention:

```
funcname (parnameI: partypeI [; optnameI: opttypeI]) -> retname: rettype
```

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 (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, font, images, etc.

A canvas keeps in its state a set of attributes under which the drawing primitives operate. For instance, if its colour 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 (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: Image path.

Return values

canvas: Canvas representing the image.

Description

Returns a new canvas whose content is the image received as a parameter.

The new canvas shall keep the transparency aspects of the original image.

canvas:new (width, height: number) → canvas: object

Arguments

width: Canvas width.

height: Canvas height.

Return values

canvas: New canvas.

Description

Returns a new canvas with the received size.

Initially, all pixels shall be transparent.

10.3.1.3 Attributes

All attribute methods have the prefix "attr" and are used to get and set attributes (with the exceptions specified).

When a method is invoked without input parameters, the current attribute value is returned. On the other hand, when a method is invoked with input parameters, these parameters must be used as the new attribute values.

canvas:attrSize () → width, height: number

Return values

width: Canvas width.

height: Canvas height.

Description

Returns the canvas dimensions.

It is important to note that it is not possible to change the dimensions of an existing canvas.

canvas:attrColor (R, G, B, A: number)

Arguments

R: Red colour component.

G: Green colour component.

B: Blue colour component.

A: Alpha colour component.

Description

Change the canvas' attribute colour.

The colours 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 colour set to this attribute.

The initial value is '0,0,0,255' (black).

canvas:attrColor (clr_name: string)

Arguments

clr_name: Colour name.

Change the canvas' attribute colour.

The colours are given as a string corresponding to one of the 16 pre-defined NCL colours:

'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 ("A = 255").

The primitives (see clause 10.3.1.4) are drawn with the colour set in this attribute.

The initial value is 'black'.

canvas:attrColor () → R, G, B, A: number

Return values

R: Red colour component.

G: Green colour component.

B: Blue colour component.

A: Alpha colour component.

Description

Returns the canvas' colour.

canvas:attrFont (face: string; size: number; style: string)

Arguments

face: Font name.

size: Font size.

style: Font style.

Description

Changes the 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', 'bold-italic' and 'nil'. A 'nil' value assumes that no style will be used.

Any invalid input value shall raise an error.

The initial font value is undefined.

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.

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

Changes the canvas *crop* region.

Only the set region is affected by operations following graphical compositions.

The initial *crop* region is the whole canvas.

The main canvas cannot have its *crop* region changed, as it is controlled by the NCL formatter.

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.

The main canvas cannot be flipped as it is controlled by the NCL formatter.

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

Changes canvas opacity.

The opacity values varies between 0 (full transparency) to 255 (full opacity).

The main canvas cannot have its value changed as it is controlled by the NCL formatter.

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, which must be a multiple of 90^{o}.

The main canvas cannot have its value changed as it is controlled by the NCL formatter.

canvas:attrRotation () → degrees: number

Return value

degrees: Canvas rotation in degrees.

Description

Returns the current canvas rotation value.

canvas:attrScale (w, h: number|boolean)

Arguments

w: Canvas scaling width.

h: Canvas scaling height.

Description

Scales the canvas to a given width and height.

One of the given values may be *true,* indicating that the aspect ratio must be kept.

The scaling attribute is independent of the size attribute, which shall remain the same.

The main canvas cannot have its value changed as it is controlled by the NCL formatter.

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

All the following methods take the canvas attributes into account.

NOTE – In all primitives, the line width shall be assumed as 1 pixel.

canvas:drawLine (x1, y1, x2, y2: number)

Arguments

x1: Line extremity 1 coordinate.

y1: Line extremity 1 coordinate.

x2: Line extremity 2 coordinate.

y2: Line extremity 2 coordinate.

Description

Draws a line with its extremities in (x1,y1) and (x2,y2).

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 with no fill or filling it, respectively.

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'* in order to draw the rectangle frame or *'fill'* to fill it.

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 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.

canvas:drawEllipse (mode: string; xc, yc, width, height, ang_start, ang_end: number)

Arguments

mode: Drawing mode.

xc: Ellipse centre.

yc: Ellipse centre.

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 circumference 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.

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 (x,y) in the canvas, using the font set by `canvas:attrFont()`.

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 colour set to *attrColor*.

If the area parameters are not given, all the canvas should be cleared.

canvas:flush ()

Description

Flushes the canvas after a set of drawing and composite operations.

It suffices 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 with which to compose.

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 pixel by pixel the canvas src on the current canvas (destination canvas), at position (x,y).

The other parameters are optional and indicate which region in the canvas src is used for the composition. When absent, the whole canvas is used.

This operation calls `src:flush()` automatically before the composition.

The operation satisfies the following equations:

$$Cd = Cs*As + Cd*(255 - As)/255$$

$$Ad = As*As + Ad*(255 - As)/255$$

where:

Cd = Colour of the destination canvas (canvas).

Ad = Alpha of the destination canvas (canvas).

Cs = Colour of the source canvas (src).

As = Alpha of the source canvas (src).

After the operations, the destination canvas has the resulting content and the canvas src remains intact.

canvas:pixel (x, y, R, G, B, A: number)

Arguments

x: Pixel position.

y: Pixel position.

R: Colour red component.

G: Colour green component.

B: Colour blue component.

A: Colour alpha component.

Description

Changes a pixel's colour.

canvas:pixel (x, y: number) → R, G, B, A: number

Arguments

x: Pixel position.

y: Pixel position.

Return values

R: Colour red component.

G: Colour green component.

B: Colour blue component.

A: Colour alpha component.

Description

Returns a pixel's colour.

canvas:measureText (text: string) → dx, dy: number

Arguments

x: Text coordinate.

y: Text coordinate.

text: Text to be measured.

Return values

dx: Text width.

dy: Text height.

Description

Returns the border coordinates for the given text, as if it were drawn at (x,y) with the configured font of **`canvas:attrFont()`**.

10.3.2 The *event* module

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

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 7.2.8) or events coming from user interactions (for example, through the remote control).

During its initiation, before becoming event oriented, a Lua script has to 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, that is, to the event handler function.

```
=== 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 formatter. As mentioned before, a Lua script is also capable of generating events, called "spontaneous", 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 formatter). The default value is 'out'.

event.timer (time: number, f: function) → cancel: function

Arguments

time: Time in milliseconds.

f: Call-back function.

Return value

unreg: Function to cancel the timer.

Description

Creates a timer that expires after a timeout (in milliseconds) and then calls the call-back 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.

event.register ([pos: number]; f: function; [class: string]; […: any])

Arguments

pos: Register position (optional).

f: Call-back function.

class: Class filter (optional).

…: Class dependent filter (optional).

Description

Registers the given function as an event listener, that is, whenever an event happens, *f* is called (function *f* is an event handler).

The parameter *pos* is optional. It indicates the position where *f* is registered. If it is not given, the function is registered in the last position. The initial register position is 1.

The parameter *class* is optional and indicates which class of events the function shall receive. If *class* is specified, other class dependent filters may be defined. 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 no other event may be processed while it is running.

The NCL formatter 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.

event.unregister (f: function)

Arguments

f: Call-back function.

Description

Unregisters the given function as a listener, that is, new events will no longer be notified to *f*.

event.uptime () → ms: number

Return values

ms: Time in milliseconds.

Description

Returns the number of milliseconds elapsed since the beginning of the application.

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}

where:

type may be 'press' or 'release';

key is the key value; the "event.keys" table holds all keycodes available in the NCL.

Example: `evt = { class='key', type='press', key='0'}`

NOTE 1 – In the key class, the class dependent filter could be *type* and *key*, in this order.

`pointer class:`

evt = { class='pointer', type: string, x:number, y:number}

type may be 'press' or '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 2 – In the pointer class, the class dependent filter could only be *type*.

`ncl class:`

Relationships among NCL media nodes are based on events. Lua has access to these events through `ncl Class`.

Events may act in two directions, that is, the formatter may send action events to change the state of the Lua player, which in turn may trigger transition events to signal state changes.

In events, the *type* field shall assume one of the three values: 'presentation', 'selection' or 'attribution'.

Events may be directed to specific anchors or to the whole node, this is identified by the *label* field, which assumes the whole node when absent.

In the case of an event generated by the formatter, 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='stop' }
```

Type 'attribution':
```
evt = {class='ncl', type='attribution', name='?',
         action='start'/'stop'/'abort'/'pause'/'resume', value='?'}
```

NOTE 3 – In the ncl class, the class dependent filter could be *type*, *label*, 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 document presentation, but also the NCL document specification. That is, at 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 document presentation. The original document is preserved throughout the editing process.

Just like in other event classes, an editing 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 unique difference regards the field that defines the reference pairs {uri,ior}, named *data* field 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* and *documentId* fields are optional (when applicable) and they assume by default the base and document identifiers where the NCLua object is being executed.

The event describing the editing command may also receive a time reference as an optional parameter (optional parameters are indicated in the function signatures as arguments between square 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, at which 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).

tcp class:

In order to send or receive tcp data, a connection shall be firstly established by 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. When there is a communication error, a message is returned in the error field. If the communication is successful, the connection identifier is returned in the *connection* field.

An NCLua application sends data, using the tcp protocol, through posting events in the form:

```
evt = { class='tcp', type='data', connection=identifier,
              value=string, [timeout=number] }
```

Similarly, an NCLua application receives data transported by the tcp protocol, by using events in 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, a message is returned in the error field. If the communication is successful, the message is passed in the *value* field.

In order to close the connection, the following event shall be posted:

```
evt = { class='tcp', type='disconnect', connection=identifier }
```

NOTE 4 – A specific middleware implementation should handle issues like authentication, connection timeout/retry, whether a connection should be kept open, etc.

NOTE 5 – In the tcp class, the class dependent filter can only be *connection*.

http class:

An NCLua application sends data, using the http protocol, by posting events in the form:

```
evt = { class='http', host=addr, port=number,
              value=string, [timeout=number] }
```

Similarly, an NCLua application receives data transported by the http protocol, by using events in the form:

```
evt = { class='http', host=addr, port=number, value=string, error=msg}
```

The *error* and *value* fields are mutually exclusive. If there is a communication error, a message is returned in the error field. If the communication is successful, the message is passed in the *value* field.

NOTE 6 – A specific middleware implementation should handle issues like authentication, connection timeout/retry, whether a connection should be kept open, etc.

NOTE 7 – In the http class, the class dependent filter can only be *host, port*.

rtp class:

An NCLua application sends data, using the rtp protocol, by posting events in the form:

```
evt = { class='rtp', host=addr, port=number, value=string, timeout=number }
```

Similarly, an NCLua application receives data transported by the rtp protocol, by using events in the form:

```
evt = { class='rtp', host=addr, port=number, value=string, error=msg}
```

The *error* and *value* fields are mutually exclusive. If there is a communication error, a message is returned in the error field. If the communication is successful, the message is passed in the *value* field.

NOTE 8 – A specific middleware implementation should handle issues like authentication, connection timeout/retry, whether a connection should be kept open, etc.

NOTE 9 – In the rtp class, the class dependent filter can only be *host, port*.

sms class:

The behaviour for sending and receiving data using SMS is very similar to that of the *tcp* class. The *sms* class is optional in a Ginga_NCL conformant implementation.

An NCLua application sends data, using SMS, by 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 guarantee its unicity.

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 notifies 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 (in particular, [b-3GPP TS 23.040]), 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 by 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 middleware implementation shall ensure that all ports will be unregistered.

NOTE 10 – A specific middleware implementation should handle issues like authentication, etc.

NOTE 11 – In the *sms* class, the class dependent filter could only be *from* and *port*, in this order.

NOTE 12 – The purpose of the port number is to avoid conflicts between common SMS messages received by a user, and SMS messages that are to be handled only by the application.

NOTE 13 – 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 14 – In the si class, the class-dependent filter can only be *type*.

Three event types are defined as follows:

type = 'services'

The table of 'services' event type is made up of a set of vectors, each one with information related to 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 15 – 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 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 relative 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:

   a) 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 about the event that immediately follows the current TV event;

   b) 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 the TV event immediately after the TV event defined in *eventId*, or information relative to the TV event immediately after the current TV event, if *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 relative 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 16 – 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 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 the requested information has been 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 17 – 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.

NOTE 18 – In the metadata class, the class dependent filter could be *type*, if this field is defined.

user class:

By using the class *user*, applications may extend their functionalities, create their own events.

In this class, no fields are defined (with the exception of the class field).

NOTE 19 – In the *user* class, the class dependent filter could be *type*, if this field is defined.

10.3.3 The *settings* module

Exports the *settings* table with the reserved environment variables and the variables defined by the NCL document author, as defined in the application/x-ncl-settings node.

It is not allowed that values be set to fields representing variables in the settings node. An error shall be raised in this case. Properties of the application/x-ncl-settings node may only be changed through using NCL links.

The settings table splits its groups into several 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 of use:

```
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. Lua player allows an NCLua application to persist a value to be used by itself or by another imperative object. In order to do so, it defines a reserved area, inaccessible to non-imperative NCL media objects. This area is split into the groups "service", "channel" and "shared", with the same semantics 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 the 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
```

Annex A

NCL 3.0 module schemas used in the Enhanced DTV profile

(This annex forms an integral part of this Recommendation.)

The following NCL 3.0 module schemas used in the Enhanced DTV profile (NCL30EDTV.xsd) are available as an electronic attachment to this Recommendation:

- Animation module: NCL30Animation.xsd
- CausalConnector module: NCL30CausalConnector.xsd
- CausalConnectorFunctionality: NCL30CausalConnectorFunctionality.xsd
- CompositeNodeInterface module: NCL30CompositeNodeInterface.xsd
- ConnectorAssessmentExpression Module: NCL30ConnectorAssessmentExpression.xsd
- ConnectorBase module: NCL30ConnectorBase.xsd
- ConnectorCausalExpression Module: NCL30ConnectorCausalExpression.xsd
- ConnectorCommonPart Module: NCL30ConnectorCommonPart.xsd
- ContentControl module: NCL30ContentControl.xsd
- Context module: NCL30Context.xsd
- DescriptorControl module: NCL30DescriptorControl.xsd
- Descriptor module: NCL30Descriptor.xsd
- EntityReuse module: NCL30EntityReuse.xsd
- ExtendedEntityReuse module: NCL30ExtendedEntityReuse.xsd
- Import module: NCL30Import.xsd
- KeyNavigation module: NCL30KeyNavigation.xsd
- Layout module: NCL30Layout.xsd
- Linking module: NCL30Linking.xsd
- MediaContentAnchor module: NCL30MediaContentAnchor.xsd
- Media module: NCL30Media.xsd
- Metainformation module: NCL30Metainformation.xsd
- PropertyAnchor module: NCL30PropertyAnchor.xsd
- Structure module: NCL30Structure.xsd
- SwitchInterface module: NCL30SwitchInterface.xsd
- TestRule module: NCL30TestRule.xsd
- TestRuleUse module: NCL30TestRuleUse.xsd
- Timing module: NCL30Timing.xsd
- Transition module: NCL30Transition.xsd
- TransitionBase module: NCL30TransitionBase.xsd

Appendix I

Ginga architecture

(This appendix does not form an integral part of this Recommendation.)

Ginga-NCL was originally built as a component of the middleware Ginga [b-ABNT NBR 15606-2], as depicted in Figure I.1.

The universe of Ginga applications can be partitioned into a set of declarative applications and a set of imperative applications. A declarative application is an application whose initial entity is of a declarative content type. An imperative application is an application whose initial entity is of an imperative content type. A purely declarative application is one in which every entity is of a declarative content type. A purely imperative application is one in which every entity is of an imperative content type. A hybrid application is one whose entity set contains entities of both declarative and imperative content types. A Ginga application need not be purely declarative nor imperative. In particular, NCL declarative applications often make use of Lua script content, which is imperative in nature. Therefore, either type of Ginga application may make use of facilities of both the declarative and imperative application environments.

A Ginga implementation is recommended to be open, flexible, granular, self-contained and component-based. However, this Recommendation does not specify any Ginga implementation in a compliant receiver. The architecture presented in this Recommendation only helps to present the requirements and recommendations for 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 a logical subsystem of the Ginga system that processes NCL documents. A key component of Ginga-NCL is the declarative content decoding engine (NCL formatter or NCL user agent). Another important module is the Lua engine, which is responsible for interpreting Lua scripts [b-H.IPTV-MAFR.14].

Ginga-Imp is a logical subsystem of the Ginga system that processes imperative applications. A key component of the imperative application environment is the imperative content execution engine. Ginga-J is a particular case of Ginga-Imp that processes applications coded in Java.

It is important to note that a Ginga-Imp-only implementation shall not claim any kind of Ginga conformance. This avoids the threat of market fragmentation and ensures that Ginga will always offer backward-compatible profiles. A Ginga-NCL-only implementation is not prohibited. In other words, to be Ginga compliant, the Ginga-NCL subsystem is required and the Ginga-Imp subsystem is optional.

Common content decoders serve both imperative and declarative application needs for the decoding and presentation of common content types such as PNG, JPEG, MPEG and other formats. The Ginga Common Core is composed of common content decoders, 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 required to support receiver device start-up and initialization function and server-side device start-up and initialization function. It is also required to gather metadata information and to provide this information through the NCL *settings* media object (see clause 7.2.4).

It is also recommended that the Ginga Common Core provide an API to communicate with a DRM system; pull together context information (like user profiles and receiver profiles available on a local or removable storage device) and provide context awareness through the NCL *settings* media

object (see clause 7.2.4); and to support software version management (update) of Ginga's components.

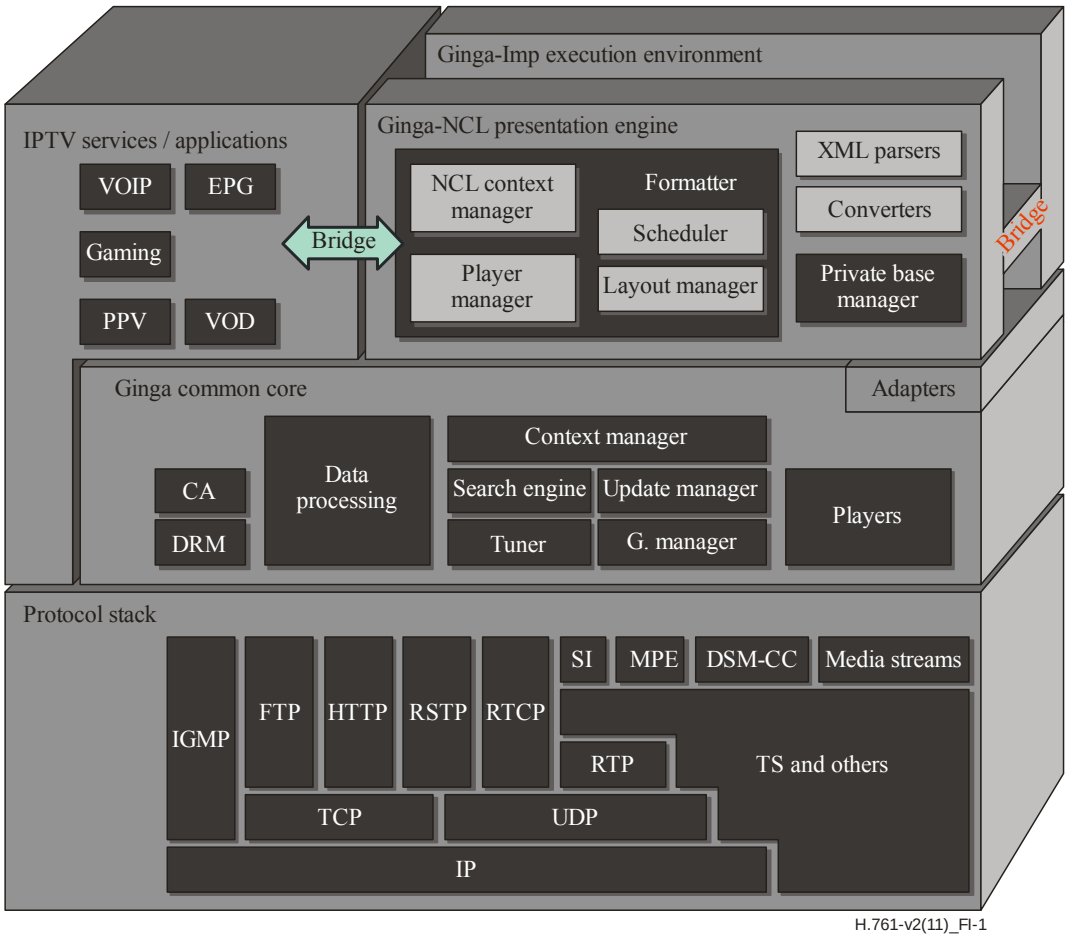

Figure I.1 – Ginga architecture

The core of the Ginga-NCL Presentation Engine is the Formatter. This component is in charge of receiving and controlling multimedia applications written in NCL. Applications are delivered to the Formatter by the Ginga Common Core subsystem. Upon receiving an application, the Formatter 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 commanding the Player Manager component to instantiate an appropriate Player, according to the media content type to be exhibited at a given moment in time. Media contents are acquired through the protocol stack, and can come from different communication networks.

One important player, part of Ginga-NCL, is the Lua Engine, responsible for the execution of NCLua objects, that is, media objects with Lua code.

In Ginga-NCL, a generic API is defined to establish the necessary communication between Players components and the Presentation Engine (Scheduler component). Thanks to this API, the Ginga-NCL Presentation Engine and the Ginga Common Core are strongly coupled but independent subsystems. Ginga Common Core may be substituted for other third party implementations that support IPTV engines, allowing Ginga-NCL to be integrated in other IPTV middleware specifications, extending their functionalities with NCL facilities for supporting NCL applications.

Players are responsible for notifying the Presentation Engine about content anchor events (see clause 8) defined in NCL applications, that is, when a media segment (an anchor) begins and ends its presentation, or when it is selected. Presentation events can be derived from NPT or MPEG-2 timestamps, timers started with images, videos, etc., depending on the media format.

Players that do not follow the generic API are required to use the services provided by adapters. Any user agent or execution engine could be adapted to the Ginga-NCL Players, e.g., XHTML browsers or a Java engine, as mentioned before.

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 document editing commands and maintaining the NCL documents being presented.

The Ginga-NCL Presentation Engine supports multiple presentation devices through its Layout Manager module. This component is responsible for mapping all regions (see clause 7.2.3) defined in an NCL application to a canvas on the receiver's presentation devices.

Ginga-NCL provides declarative support to IPTV specific services, such as VoD, 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.

Bibliography

[b-ITU-T H.770] Recommendation ITU-T H.770 (2009), *Mechanisms for service discovery and selection for IPTV services*.

[b-H.IPTV-MAFR.14] Recommendation ITU-T H.IPTV-MAFR.14 (draft), *Lua script language for IPTV*.

[b-ABNT NBR 15606-2] ABNT NBR 15606-2 (2007), *Digital terrestrial television – Data Coding and transmission specification for digital broadcasting – Part 2: Ginga-NCL for fixed and mobile receivers: XML application language for application coding*.

[b-3GPP TS 23.040] 3GPP TS 23.040 V6.8.1 (2006-10), *Technical realization of the Short Message Service (SMS)*.

[b-NCM Core] Soares L.F.G; Rodrigues R.F (2005), *Nested Context Model 3.0: Part 1 – NCM Core*, Technical Report, Departamento de Informática PUC-Rio, ISSN: 0103-9741. Also available at http://www.ncl.org.br.

[b-NCL DTV] Soares L.F.G; Rodrigues R.F. (2006), *Part 8 – NCL (Nested Context Language) Digital TV Profiles*, Technical Report, Departamento de Informática PUC-Rio, No. 35/06. ISSN: 0103-9741. Also available at http://www.ncl.org.br.

[b-NCL Live E.C.] Soares L.F.G; Rodrigues R.F; Costa, R.R.; Moreno, M.F. (2006), *Part 9 – NCL Live Editing Commands.* Technical Report, Departamento de Informática PUC-Rio, No. 36/06. 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. (2008), *Part 10 – Imperative Objects in NCL: The NCLua Scripting Language*. Technical Report, Departamento de Informática PUC-Rio, No. 02/08. Rio de Janeiro. ISSN 0103-9741. Also available at http://www.ncl.org.br.

[b-NCL Decl. Obj.] Soares L.F.G. (2009), *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. 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 RDF] W3C Recommendation. RDF (1999) – *Resource Description Framework (RDF) Model and Syntax Specification*.

[b-W3C SMIL 2.1] W3C Recommendation. SMIL 2.1 (2005) – *Synchronized Multimedia Integration Language – SMIL 2.1 Specification*.

[b-W3C XHTML] W3C Recommendation. XHTML 1.0 (2000) – *The Extensible HyperText Markup Language*.

[b-W3C XMLNAMES1] W3C Recommendation. XML_Names 1.0 (1999) – *Namespaces in XML*.

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	**Audiovisual and multimedia systems**
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Terminals and subjective and objective assessment methods
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects and next-generation networks
Series Z	Languages and general software aspects for telecommunication systems

Printed in Switzerland
Geneva, 2012