



PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 02/08

Nested Context Language 3.0

**Part 10 – Imperative Objects in NCL:
The NCLua Scripting Language**

**Francisco Figueiredo G. Sant’Anna
Luiz Fernando Gomes Soares
Renato Fontoura de Gusmão Cerqueira**

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

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

RIO DE JANEIRO - BRASIL

Nested Context Language 3.0

Part 10 – Imperative Objects in NCL: The NCLua Objects

Francisco Figueiredo G. Sant’Anna
Luiz Fernando Gomes Soares
Renato Fontoura de Gusmão Cerqueira

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

lfgs@inf.puc-rio.br, francisco@telemidia.puc-rio.br, rcerq@inf.puc-rio.br

Abstract. *This technical report describes how imperative objects may be related with other objects in an NCL document and how imperative object players shall behave. NCL (Nested Context Language) is an XML application language based on the NCM (Nested Context Model) conceptual model for hypermedia document specification, with temporal and spatial synchronization among its media objects. NCLua objects and players are also described as an example. Lua is the main scripting language of NCL and the standard language for the Brazilian DTV System.*

Keywords: *imperative objects, digital TV; middleware; declarative environment; NCL, Lua.*

Resumo. *Este relatório técnico descreve como objetos com código imperativo podem se relacionar com outros objetos em documentos NCL e como exibidores (engines) para esses objetos devem se comportar. NCL é uma aplicação XML baseada no modelo conceitual NCM (Nested Context Model) para a especificação de documentos hipermídia com sincronismo espacial e temporal entre seus objetos. Objetos e exibidores NCLua são também descritos como exemplo. Lua é a principal linguagem de script de NCL, e é linguagem padrão do Sistema Brasileiro de TV Digital.*

Palavras chave: *objetos imperativos; TV digital; middleware; linguagem declarativa; NCL, Lua.*



Nested Context Language 3.0

Part 10 – Imperative Objects in NCL: The NCLua Scripting Language

© Laboratório TeleMídia da PUC-Rio – Todos os direitos reservados

Impresso no Brasil

As informações contidas neste documento são de propriedade do Laboratório TeleMídia (PUC-Rio), sendo proibida a sua divulgação, reprodução ou armazenamento em base de dados ou sistema de recuperação sem permissão prévia e por escrito do Laboratório TeleMídia (PUC-Rio). As informações estão sujeitas a alterações sem notificação prévia.

Os nomes de produtos, serviços ou tecnologias eventualmente mencionadas neste documento são marcas registradas dos respectivos detentores.

Figuras apresentadas, quando obtidas de outros documentos, são sempre referenciadas e são de propriedade dos respectivos autores ou editoras referenciados.

Fazer cópias de qualquer parte deste documento para qualquer finalidade, além do uso pessoal, constitui violação das leis internacionais de direitos autorais.

Laboratório TeleMídia

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

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

22451-900 – Rio de Janeiro – RJ – Brasil

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

Table of Contents

1. Introduction.....	6
2. NCL Historical Evolution.....	7
3. Overview of NCL Elements	10
4. Imperative Objects.....	13
5. Expected Behavior of Imperative Players in NCL Applications.....	15
5.1. Imperative-Object Execution Model	16
5.2. Instructions to Presentation Events.....	16
5.2.1. <i>start</i> instruction.....	16
5.2.2. <i>stop</i> instruction.....	17
5.2.3. <i>abort</i> instruction.....	18
5.2.4. <i>pause</i> instruction.....	19
5.2.5. <i>resume</i> instruction.....	19
5.2.6. <i>Natural end</i> of a code execution.....	20
5.3. Instructions to Attribution Events.....	20
5.3.1. <i>set</i> instruction.....	21
5.3.2. <i>stop, abort, pause and resume</i> instructions	21
6. Recommended APIs	22
7. Final Remarks	23
References.....	24
Appendix A – Lua procedural objects in NCL presentations: The NCLua Scripting Language.....	25
1. Lua language - Optional functions in the Lua library.....	25
2. Additional modules.....	25
2.1. The canvas module	26
2.1.1. The canvas object	26

2.1.2.	Constructors	26
2.1.3.	Attributes	27
2.1.4.	Primitives	32
2.1.5.	Miscellaneous	35
2.2.	The event module	37
2.2.1.	General View	37
2.2.2.	Functions.....	38
2.2.3.	Event classes.....	40
2.3.	The settings module.....	51
2.4.	The persistent module.....	52

Nested Context Language 3.0

Part 10 – Imperative Objects in NCL: The NCLua Scripting Language

Luiz Fernando Gomes Soares
Francisco Figueiredo G. Sant’Anna
Renato Fontoura de Gusmão Cerqueira

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

lfgs@inf.puc-rio.br, francisco@telemidia.puc-rio.br, rcercq@inf.puc-rio.br

***Abstract.** This technical report describes how imperative objects may be related with other objects in NCL documents and how imperative object players shall behave. NCL (Nested Context Language) is an XML application language based on the NCM (Nested Context Model) conceptual model for hypermedia document specification, with temporal and spatial synchronization among its media objects. NCLua objects and players are also described as an example. Lua is the main scripting language of NCL and the standard scripting language for the Brazilian DTV System.*

1. Introduction

Imperative objects may be inserted into NCL documents mainly to bring additional computational capabilities to declarative documents. The way to add an imperative object into an NCL document is to define a <media> element, whose content (located through the *src* attribute) is the imperative code to be executed.

As examples, both EDTV and BDTV profiles of NCL 3.0 allows two media types to be associated with the <media> element: `application/x-ncl-NCLua`, for Lua procedural codes¹ (file extension `.lua`); and `application/x-ginga-NCLet`, for Java (Xlet) codes (file extension `.class` or `.jar`).

This technical report describes how to use imperative objects to add additional computational capabilities to the NCL declarative language. The report is organized as follows. Section 2 gives an historical evolution of the NCL versions. Section 3 presents a brief overview of the NCL 3.0 elements. Section 4 describes how an imperative object may be defined together with its content anchors and properties. Section 5 discusses the expected behavior of imperative object players, and how imperative objects may be related with other objects in an NCL document. Section 6 points out some APIs that shall be offered by an imperative language used as the content of an imperative object. Section 7 presents the final remarks. Appendix A discusses, as an example of imperative objects, NCLua code objects as adopted in Ginga, the middleware of the Brazilian Digital TV System (SBTVD).

¹ Indeed, Lua is a multi-paradigm programming language (an imperative and functional language).

2. NCL Historical Evolution

The first version of NCL [Anto00, AMRS00] was specified through an XML DTD – Document Type Definition [XML98].

The second version of NCL, named NCL 2.0, was specified using XML Schema [SCHE01]. Following recent trends, from version 2.0 on, NCL has been specified in a modular way, allowing the combination of its modules in language profiles.

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

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

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

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

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

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

The NCL 3.0 edition revised some functionalities contained in NCL 2.4. NCL 3.0 is more specific regarding some attribute values. This new version introduced two new functionalities, as well: Key Navigation and Animation functionalities. In addition, NCL 3.0 made depth modifications on the Composite-Node Template functionality and introduces some SMIL based modules to NCL profiles for transition effects in media presentation and for metadata definition. NCL 3.0 also reviewed the hypermedia connector specification in order to have a more concise notation. Relationships among imperative and

declarative objects and other objects are also refined in NCL 3.0, as well as the behavior of imperative and declarative object players. Finally, NCL 3.0 also refined the support to multiple exhibition devices and introduced the support to NCL live editing commands.

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

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

- Part 1 – NCM Core
concerned with the main model entities, which should be present in all NCM implementations².
- Part 2 – NCM Virtual Entities
concerned mainly with the definition of virtual anchors, nodes and links.
- Part 3 – NCM Version Control
concerned with model entities and attributes to support versioning.
- Part 4 – NCM Cooperative Work
concerned with model entities and attributes to support cooperative document handling.

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

- Part 5 – NCL (Nested Context Language) Full Profile
concerned with the definition of an XML application language for authoring and exchanging NCM-based documents, using all NCL modules, including those for the definition and use of templates, and also the definition of constraint connectors, composite-connectors, temporal cost functions, transition effects and metainformation characterization.
- Part 6 – NCL (Nested Context Language) XConnector Profile Family
concerned with the definition of an XML application language for authoring connector bases. One profile is defined for authoring causal connectors, another one for authoring causal and constraint connectors, and a third one for authoring both simple and composite connectors.
- Part 7 – Composite Node Templates
concerned with the definition of the NCL Composite-Node Template functionality, and with the definition of an XML application language (XTemplate) for authoring template bases.
- Part 8 – NCL (Nested Context Language) Digital TV Profiles
concerned with the definition of an XML application language for authoring documents

² It is also possible to have NCM implementations that ignore some of the basic entities, but this is not relevant so as to deserve a minimum-core definition.

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

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

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

3. Overview of NCL Elements

NCL is an XML application that follows the modularization approach. The modularization approach has been used in several W3C language recommendations. A *module* is a collection of semantically-related XML elements, attributes, and attribute's values that represents a unit of functionality. Modules are defined in coherent sets. A *language profile* is a combination of modules. Several NCL profiles have been defined, among them those defined by Parts 5, 6, 7, and 8 of the NCL collection presented in Section 2. Of special interest are the profiles defined for Digital TV, the EDTVProfile (*Enhanced Digital TV Profile*) and the BDTVProfile (*Basic Digital TV Profile*). This section briefly describes the elements that compose these profiles. The complete definition of the NCL 3.0 modules for these profiles, using XML Schemas, is presented in [SoRo06]. Any ambiguity found in this text can be clarified by consulting the XML Schemas.

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

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

The `<body>` element may have `<port>`, `<property>`, `<media>`, `<context>`, `<switch>`, and `<link>` elements as its children. The `<body>` element is treated as an NCM context node. In NCM [SoRo05], the conceptual data model of NCL, a node may be a context, a switch or a media object. Context nodes may contain other NCM nodes and links. Switch nodes contain other NCM nodes. NCM nodes are represented by corresponding NCL elements.

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

The `<context>` element is responsible for the definition of context nodes. An NCM context node is a particular type of NCM composite node and is defined as containing a set of nodes and a set of links [SoRo05]. Like the `<body>` element, a `<context>` element may have `<port>`, `<property>`, `<media>`, `<context>`, `<switch>`, and `<link>` elements as its children.

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

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

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

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

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

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

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

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

Finally, the MetaInformation module is also incorporated, inheriting the same semantics of SMIL MetaInformation module. Meta-information does not contain content information that is used or display during a presentation. Instead, it contains information about content that is used or displayed. The Metainformation module contains two elements that allow describing NCL documents. The <meta> element specifies a single property/value pair. The <metadata> element contains information that is also related to meta-information of the document. It acts as the root element of an RDF tree: RDF element and its sub-elements (for more details, refer to W3C metadata recommendations [RDF99]).

4. Imperative Objects

A `<media>` element of an imperative *type* (`application/x-???`) shall be used to specify an imperative object. In this case, the object's content (located through the `src` attribute) shall be imperative codes to be executed. As an example, the DTV profiles of NCL 3.0 allow the `application/x-ncl-NCLua` type, for Lua procedural codes (file extension `.lua`); and the `application/x-ginga-NCLet` type, for Java (Xlet) codes (file extension `.class` or `.jar`).

Like for any media object, a `<media>` element containing imperative code may define content anchors (through `<area>` elements) and properties (through `<property>` elements). As usual, the *descriptor* attribute of the `<media>` element may refer to a `<descriptor>` element that is responsible for initializing several of the corresponding imperative object's properties necessary for its presentation.

In an imperative object, 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 (a function, a method, etc.). An `<area>` element may also be used just as an interface to be used as conditions of NCL links to trigger actions on other objects, as discussed in Section 5.

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 conventional media content players, imperative-code players shall control event state machines associated with the imperative object's content anchors (see Figure 1). However, different from other media objects, neither the imperative-code player, nor the NCL formatter has the knowledge, by itself, to trigger the event state machine transitions, as discussed in Section 5. Sometimes, the object's imperative code has the responsibility to command the imperative player in the execution of this task.

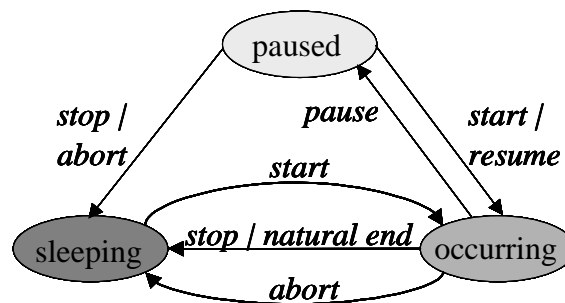


Figure 1 – Event state machine associate with a content anchor or a property

A `<property>` element defined as a child of a `<media>` element representing an imperative code may be mapped to a code span (function, method, etc.), to a code attribute or to an object property (as in any media object, in this last case). The *name* attribute of the

<property> element shall be used to identify the imperative-code span, the code attribute, or the object property, respectively.

Again, when a <property> element is associated with a code span, neither the imperative-code player, nor the NCL formatter has the knowledge, by itself, to trigger transitions of the corresponding attribution-event state machine (see Figure 1). As discussed in Section 5, the object's imperative code has also in this case the responsibility to command the imperative-code player in the execution of this task.

As usual, <area> and <property> elements may be used as interface points of <link> elements as well as the two default content anchors, which establishes a two-way bridge between the declarative and imperative environment controlled by the NCL formatter, as discussed in the next section.

5. Expected Behavior of Imperative Players in NCL Applications

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

Analogous to conventional media content players, 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 has not sufficient information to control by itself all event state machines, and shall rely on the imperative application content to command these controls.

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

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

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

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

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

5.1. Imperative-Object Execution Model

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

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

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

5.2. Instructions to Presentation Events

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

5.2.1. *start* instruction

The *start* instruction issued by a formatter shall inform the following parameters to the imperative-object player: the imperative object to be controlled, its associated descriptor, a list of events (defined by the <media> element's <area> and <property> child elements, and by the default content anchors) that need to be monitored by the imperative-object player, the content-anchor *label*, or by default the *main* content anchor, identifying the associated imperative code to be started, and an optional delay-time. From the *src* attribute, the imperative-object player tries to locate the imperative code and start its execution. If the content cannot be located, the player shall finish the starting operation, without performing any action.

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

formatter shall be the <media> descriptor, if this attribute was specified. Otherwise, a default descriptor for that imperative-object *type* of <media> shall be chosen by the formatter.

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

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

Different from what is performed on other <media> elements, if an imperative-object player receives a *start* instruction for an event associated with a content anchor and this event is in the *sleeping* state, it shall start the execution of the imperative code associated with the element, even though other portion of the object’s imperative code is being in execution (paused or not). However, if the event associated with the target content anchor is in the *occurring* or *paused* state, the *start* instruction shall be ignored by the imperative-code player that keeps on controlling the ongoing execution. As a consequence, 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 NCLua node interface, which shall not be ignored when the action is applied.

Since neither the formatter nor the imperative-code player has any other knowledge about the imperative-object’s content anchors, except their *id*, and *label* attributes, they do not know which other content anchors shall have their associated event put in the occurring state, when a content anchor is started or is being in execution. Therefore, except for the event associated with the *whole content anchor*, it is 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 with 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 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.

Different 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.

5.2.2. stop instruction

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

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

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

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

5.2.3. abort instruction

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

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

For the same reason discussed in the *start* instruction, except for the event associated with the *whole content anchor*, it is 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 with the event state machine associated to the aborted code, and to inform if a transition associated with a change shall be notified.

Different 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.

5.2.4. *pause* instruction

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

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

For the same reason discussed in the *start* instruction, except for the event associated with the *whole content anchor*, it is 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 with the event state machine associated to the paused code, and to inform if a transition associated with a change shall be notified.

Different 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.

5.2.5. *resume* instruction

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

If the imperative code associated with the object's interface is not paused or the imperative-object player is not paused (waiting for the start delay), the instruction shall be ignored. If the imperative-object player is paused waiting for the start delay, it shall resume the wait from the instant it was paused. If the imperative code associated with the object's interface

is paused, its associated event shall transit to the *occurring* state, and their *resumes* transitions shall be notified.

For the same reason discussed in the *start* instruction, except for the event associated with the *whole content anchor*, it is 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 with the event state machine associated to the paused code, and to inform if a transition associated with a change shall be notified.

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

5.2.6. Natural end of a code execution

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

Different 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.

5.3. Instructions to Attribution Events

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

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

5.3.1. *start* instruction

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

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

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

5.3.2. *stop, abort, pause and resume* instructions

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

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

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

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

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

6. Recommended APIs

Besides its standard library, some APIs shall be supported by an imperative language to be used in imperative objects.

The imperative language shall offer an API that allows an imperative code to query any pre-defined or dynamic properties' values of the NCL *settings* node. However, it must be stressed that it is not allowed to directly set values to these properties. Properties of the *settings* node may only be changed trough using NCL links.

The imperative language shall offer an API that provides a set of methods to support NCL editing commands and commands of the Private Base Manager, as presented in [SRRM06].

7. Final Remarks

In order to offer a scalable hypermedia model, with characteristics that may be progressively incorporated in hypermedia system implementations, NCM was divided in several parts, and also its declarative XML application language: NCL. This technical report deals with how imperative media-objects may be related with other objects in NCL applications and how imperative-object players shall behave (NCLua objects and players are also described as an example in Appendix A), which comprises Part 10 – Imperative Objects in NCL: The NCLua Scripting Language.

Acknowledgements

Many people have contributed to the definition of the imperative objects. Chief among them are Rogério Ferreira Rodrigues and Marcio Ferreira Moreno.

References

- [Anto00] Antonacci M.J. NCL: Uma Linguagem Declarativa para Especificação de Documentos Hipermídia com Sincronização Temporal e Espacial. **Master Dissertation, Departamento de Informática, PUC-Rio**, April 2000.
- [AMRS00] Antonacci M.J., Muchaluat-Saade D.C., Rodrigues R.F., Soares L.F.G. NCL: Uma Linguagem Declarativa para Especificação de Documentos Hipermídia na Web, **VI Simpósio Brasileiro de Sistemas Multimídia e Hipermídia - SBMídia2000**, Natal, Rio Grande do Norte, June 2000.
- [IeFC06] Ierusalimschy, R.; Figueiredo, L.H.; Celes, W. Lua 5.1 Reference Manual, August 2006 (ISBN 85-903798-3-3).
- [ISO98] ISO/IEC 13818-6, Information technology - Generic coding of moving pictures and associated audio information - Part 6: Extensions for DSM-CC. 1998/Cor 2:2002.
- [RDF99] Resource Description Framework (RDF) Model and Syntax Specification, Ora Lassila and Ralph R. Swick. W3C Recommendation, 22 February 1999. Available at <http://www.w3.org/TR/REC-rdf-syntax/>
- [SCHE01] XML Schema Part 0: Primer, **W3C Recommendation**, in <http://www.w3.org/TR/xmlschema-0/>, May 2001.
- [SoRo05] Soares L.F.G; Rodrigues R.F. Nested Context Model 3.0: Part 1 – NCM Core, **Technical Report, Departamento de Informática PUC-Rio**, May 2005, ISSN: 0103-9741.
- [SoRo06] Soares L.F.G; Rodrigues R.F. Nested Context Language 3.0: Part 8 – NCL Live Editing Commands, **Technical Report, Departamento de Informática PUC-Rio**, December 2006, ISSN: 0103-9741.
- [SRRM06] Soares L.F.G; Rodrigues R.F.; Costa R.R.; Moreno, M.F. Nested Context Language 3.0: Part 9 – NCL Digital TV Profiles, **Technical Report, Departamento de Informática PUC-Rio**, October 2006, ISSN: 0103-9741.
- [XML98] Bray T., Paoli J., Sperberg-McQueen C.M., Maler E. Extensible Markup Language (XML) 1.0 (Second Edition), **W3C Recommendation**, in <http://www.w3.org/TR/REC-xml>, February 1998.

Appendix A – Lua procedural objects in NCL presentations: The NCLua Scripting Language

1. Lua language - Optional functions in the Lua library

Lua is the scripting language adopted by Ginga-NCL to implement imperative objects in NCL documents (<media> elements of application/x-ncl-NCLua type or application/x-ginga-NCLua type). The complete definition of Lua is presented in [IeFC06].

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

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

2. 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 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 use the following naming convention:

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

2.1. The canvas module

2.1.1. The canvas object

When an NCLua media object is initialized, the corresponding region of the <media> element (of type application/x-ncl-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 a 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 keep in its state a set of attributes under which the drawing primitives operate. For instance, if its color attribute is blue, a call to `canvas:drawLine()` will draw a blue line on the canvas.

The coordinates are always relative to the top/leftmost point in canvas (0,0).

2.1.2. 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.

2.1.3. 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

Arguments

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 Color red component

G Color green component

B Color blue component

A Color alpha component

Description

Change canvas' attribute color.

The colors are given in RGBA, where A varies from 0 (full transparency) to 255 (full opacity).

The primitives (see 10.3.3.4) are drawn with the color set to this attribute.

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

canvas:attrColor (clr_name: string)*Arguments*

clr_name	Color name
----------	------------

Change canvas' attribute color.

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

'white', 'aqua', 'lime', 'yellow', 'red', 'fuchsia', 'purple', 'maroon',
'blue', 'navy', 'teal', 'green', 'olive', 'silver', 'gray', 'black'

The values given have their alpha equal to full opacity ("A = 255").

The primitives (see 10.3.3.4) are drawn with the color set in this attribute.

The initial value is 'black'.

canvas:attrColor () -> R, G, B, A: number*Return values*

R Color red component

G Color green component

B Color blue component

A Color alpha component

Description

Returns the canvas' color.

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

face Font name

size Font size

style Font style

Description

Changes canvas' font attribute.

The following fonts shall be available: 'Tiresias' and 'Verdana'.

The size is in pixels, and it represents the maximum height of a line written with the chosen font.

The possible style values are: 'bold', 'italic', '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 10.2.3.4) 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 that must be multiple of 90°.

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)*Arguments*

w Canvas scaling height

h Canvas scaling width

Description

Scales the canvas by 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.

2.1.4. Primitives

All the following methods take the canvas' attributes into account.

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 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 center
yc	Ellipse center
width	Ellipse width
height	Ellipse height
ang_start	Starting angle
ang_end	Ending angle

Description

Draws an ellipse and other similar primitives as circle, arcs and sectors.

The parameter mode may receive 'arc' to only draw the circumference or 'fill' for internal painting.

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()`.

2.1.5. Miscellaneous

canvas:clear ([x, y, w, h: number])

Arguments

- x Clear area coordinate
- y Clear area coordinate
- w Clear area width
- h Clear area height

Description

Clears the canvas with the color set 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's enough to call this method only once, after a sequence of operations.

canvas:compose (x, y: number; src: canvas; [src_x, src_y, src_width, src_height: number])

Arguments

- x Position of the composition
- y Position of the composition
- src Canvas to compose with
- src_x Position in the canvas src
- src_y Position in the canvas src
- src_width Composition width in the canvas src
- src_height Composition height in the canvas src

Description

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

The other parameters are optionals and indicate which region in the canvas src is used to compose with. When absent the whole canvas is used.

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

The operation satisfies the following equation:

$$C_d = C_s * A_s + C_d * (255 - A_s) / 255$$

$$A_d = A_s * A_s + A_d * (255 - A_s) / 255$$

where:

C_d = color of the destination canvas (canvas)

A_d = alpha of the destination canvas (canvas)

C_s = color of the source canvas (src)

A_s = 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 Color red component
- G Color green component
- B Color blue component
- A Color alpha component

Description

Changes the pixel color.

canvas:pixel (x, y: number) -> R, G, B, A: number*Arguments*

- x Pixel position
- y Pixel position

Return values

- R Color red component
- G Color green component

B Color blue component

A Color alpha component

Description

Returns the pixel color.

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()`.

2.2. The event module

2.2.1. General View

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 Section 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 aforementioned, a Lua script is also capable of generating events, called “spontaneous”, through a call to the `event.post(evt)` function.

2.2.2. 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 Callback function

Return value

unreg Function to cancel the timer

Description

Creates a timer that expires after a timeout (in milliseconds) and then call the callback function f.

The signature of f is simple, no parameters are received or returned:

function f () end

The value of 0 milliseconds is valid. In this case, `event.timer()` shall return immediately and `f` shall be called as soon as possible.

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

pos	Register position (optional)
f	Callback 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 (the 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 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 while it is running no other event may be processed.

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	Callback 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.

2.2.3. Event classes

The function `event.post()` and the registered handler in `event.register()` receive events as parameters.

An event is described by a common Lua table, where the `class` field is mandatory and identifies the event class.

The following event classes are defined:

key class:

```
evt = { class='key', type: string, key: string }
```

* `type` may be 'press' or 'release'.

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

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

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

ncl class:

Relations 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 its 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 `label` field, that 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' and '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', and 'resume'

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='?' value='?' }
```

NOTE 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 Section 9). However, there is an important difference between editing commands coming from DSM-CC stream events (see Section 9), and the editing commands performed by Lua scripts (NCLua objects). The first ones alter not only the NCL document presentation, but also the NCL document specification. That is, in the end of the process a new NCL document is generated incorporating all editing results. On the other hand, editing commands coming from NCLua media objects only alter the NCL document presentation. The original document is preserved during all 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 56 in Section 9). The unique difference is with regards to the field that defines the {uri,id} reference pairs, named *data* field in the edit class. This field's values may be not only the reference pairs mentioned in Table 56, but also XML strings with the content to be added.

Exemple:

```
evt = {  
    command = 'addNode',  
    compositeId = 'someId',  
    data = '<media>...',  
}
```

The *baseId* e *documentId* fields are optional (when applicable) and they assume by default the base and document identifiers where the NCLua object is in execution.

The event describing the editing command may also receive a time reference as an optional parameter (optional parameters are indicated in the function signatures as arguments between brackets). This optional parameter may be used to specify the exact moment when the editing command shall be executed. If this parameter is not provided in the function call, the editing command shall be executed immediately. When provided, this parameter may have two different types of values, with two different meanings. If it is a number value, it defines the amount of time, in seconds, for how long the command shall be postponed.

However, this parameter may also specify the exact moment, in absolute values, the command shall be executed. In this case, this parameter shall be a table value with the following fields: year (four digits), month (1-12), day (1-31), hour (0-23), min (0-59), sec (0-61), and isdst (a daylight saving flag, a boolean).

tcp class:

The use of the return channel is done through this class of events..

In order to send or receive a tcp data, a connection shall be firstly established trough 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. When the communication is succeeded, the connection identifier is returned in the *connection* field.

An NCLua application sends data, using a return channel, through posting events in the form:

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

Similarly, an NCLua application receives data transported in a return channel using events in the form:

```
evt = { class='tcp', type='data', connection=identifier, value=string, error=msg }
```

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

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

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

NOTE An specific middleware implementation should handle issues like authentication.

NOTE In the tcp class, the class dependent filter could only be *connection*.

sms class:

The behaviour for sending and receiveing data using SMS is very similar to the one of the tcp class. The *sms* class is optional in the Ginga implementation for full-seg receivers.

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

```
evt = { class='sms', to='phone number', value=string }
```

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

```
evt = { class='sms', from='phone number', value=string }
```

NOTE An specific middleware implementation should handle issues like authentication, etc.

NOTE In the sms class, the class dependent filter could only be *from*.

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, to be delivered to the registered-event handlers of an NCLua script, whose data field contains a set of subfields and is represented by a Lua table. The set of subfields depends on requested information.

NOTE In the **si** class, the class dependent filter could only be *type*.

Four event types are defined by the following tables:

type = 'services'

The table of 'services' event type is made up by a set of vectors, each one with information related with a multiplexed service of the tuned transport stream.

Each request for a table of 'services' event type shall be carried out through the following call:

```
event.post('out', { class='si', type='services'[, index=N][, fields={field_1, field_2,..., field_j}]}),
```

where:

- i) the *index* field defines the service index, when specified; if not specified, all services of the tuned transport stream shall be present in the returned event;
- ii) the *fields* table 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, as defined in what follows). If the *fields* list is not specified, all subfields of the data table shall be filled.

The returned event is created after all request information is processed by the middleware (information that is not broadcasted within the maximum interval specified by Table 6 of ABNT NBR 15603-2:2007 shall be returned as 'nil'). The data table is returned as follows:

```
evt = {  
  
  class = 'si',  
  
  type = 'services',  
  
  data = {  
  
    [i] = { -- each service for each i  
  
      id                = <number>,  
  
      isAvailable       = <boolean>,  
  
      isPartialReception = <boolean>,  
  
      parentalControlRating = <number>,  
  
      runningStatus     = <number>,  
  
    }  
  
  }  
}
```

```

serviceType          = <number>,
providerName         = <string>,
serviceName          = <string>,
stream = {
    [j] = {
        pid = <number>,
        componentTag = <number>,
        type = <number>,
        regionSpecType = <number>,
        regionSpec    = <string>,
    }
}
}
}
}

```

NOTE In Ginga implementation in conformance with ABNT NBR 15606-2:2007, 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 descriptors associated with the service [i].

The values of the *id* and *runningStatus* data-table subfields should be computed according to the values of *service_id* and *running_status* fields, respectively, of the SDT table (see Table 13 of ABNT NBR 15603-2:2007) that describes the service [i].

The values of the *providerName* and *serviceName* data-table subfields should be computed according to the values of *service_name* and *service_provider_name* fields, respectively, of the *service_descriptor* (see ABNT NBR 15603-2:2007) that describes the service [i].

The value of the *parentalControlRating* data-table subfield should be computed according to the value of the rating field of the *parental_rating_descriptor* that has the *country_code* field with the equivalent country value that has the *user.location* variable of the Settings node.

The value of the *isAvaivable* data-table subfield should be computed according to the value of the *country_code* field (with the available set of countries) of the *country_availability_descriptor* (see Section 8.3.6 of ABNT NBR 15603-2:2007) related with service [i]. The “true” value shall be assigned only if the *country_code* field has a country value equivalent to the value of the *user.location* variable of the Settings node.

The value of the *isPartialReception* data-table subfield should be computed according to the value of *service_id* field of the *partial_reception_descriptor* (see Section 8.3.32 of ABNT NBR 15603-2:2007).

The semantics of the *serviceType* data-table subfield should be defined by Table H.2 (see ABNT NBR 15603-2:2007).

The semantics of the *runningStatus* data-table subfield should be defined by Table 14 of ABNT NBR 15603-2:2007).

The value of the *pid* stream-table subfield should have the same value of the *pid* field of the elementary stream [i] header (see ISO/IEC 13818-1).

The value of the *componentTag* stream-table subfield should be computed according to the value of *component_tag* field of the *stream_identifier_descriptor* (See Section 8.3.16 of ABNT NBR 15603-2:2007) related with the elementary stream [i].

The semantics of the *type* stream-table subfield should be defined according to Table 2-34 of the ITU-T Rec. H.222.0 | ISO/IEC 13818-1: 2008, related with the elementary stream [i].

The coding method for the *regionSpec* stream-table subfield should be defined by *regionSpecType* stream-table subfield, according to the semantics defined in Table 53 of ABNT NBR 15603-2:2007.

The value of the *regionSpec* stream-table subfield should define the region for which the elementary stream [i] is designated.

The *regionSpec* and *regionSpecType* stream-table subfields should also be computed based on the *target_region_descriptor* (See ABNT NBR 15603-2:2007).

type = 'mosaic'

The table of the 'mosaic' event type is made up by a set of information for building the mosaic, and is provided in a matrix format.

Each request for a table of 'mosaic' event type shall be carried out through the following call:

```
event.post('out', { class='si', type='mosaic', 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, as defined in what follows). If the *fields* list is not specified, all subfields of the data table shall be filled.

The returned event is created after all request information is processed by the middleware (information that is not broadcasted within the maximum interval specified by Table 6 of ABNT NBR 15603-2:2007 shall be returned as 'nil'). The data table is returned as follows:

```
evt = {  
  class = 'si',  
  type = 'mosaic',  
  data = {  
    [i] = {  
      [j] = {  
        logicalId    = <number>,  
        presentationInfo = <number>,  
        id           = <number>,  
        linkageInfo  = <number>,  
        bouquetId   = <number>,  
        networkId   = <number>,  
        tsId        = <number>,  
        serviceId   = <number>,  
        eventId     = <number>,  
      }  
    }  
  }  
}
```

```
}  
  
}
```

NOTE In Ginga implementation in conformance with ABNT NBR 15606-2:2007, in order to compute the values of the data-table subfields to be returned in events of mosaic type, SI tables should be used as a basis, as well as descriptors associated with the mosaic.

The maximum values for [i] and [j], as well as the values of the logicalId, presentationInfo, id, linkageInfo, bouquetId, networkId, tsId, serviceId and eventId data-table subfields should be computed according to the values of number_of_horizontal_elementary_cells, number_of_vertical_elementary_cells, logical_cell_id, logical_cell_presentation_info, id, cell_linkage_info, bouquet_id, original_network_id, transport_stream_id, service_id and event_id fields, respectively of the mosaic_descriptor (See Section 8.3.9 of ABNT NBR 15603-2:2007)

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, as defined in what follows). If the *fields* list is not specified, all subfields of the data table shall be filled.

Description: returns information regarding to the current event of the content being transmitted.

```
2) event.post('out', { class='si', type='epg', stage='next'[, eventId=<number>][,  
fields={field_1, field_2,..., field_j}]})
```

where:

i) the eventId field, when specified, identifies the event immediately before the event whose information is required. When not specified, the requested information is for the event that immediately follows the current 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, as defined in what follows). If the *fields* list is not specified, all subfields of the data table shall be filled.

Description: returns information regarding to the event immediately after the event defined in *eventId*, or information regarding to the event immediately after the current event, when *eventId* is not specified.

```
3) event.post('out', { class='si', type='epg', stage='schedule', startTime=<date>,  
endTime=<date>[, fields={field_1, field_2,..., field_j}]})
```

where the *fields* list may have as a value any subset of subfields defined for the *data* table of the returned event (thus, field_i represents one of the subfields of the data table, as defined in what follows). If the *fields* list is not specified, all subfields of the data table shall be filled.

Description: returns information regarding to 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 the maximum interval specified by Table 6 of ABNT NBR 15603-2:2007 shall be returned as 'nil'). The data table is returned as follows:

```
evt = {  
  
  class = 'si',  
  
  type = 'epg',  
  
  data = {  
  
    [i] - {  
  
      startTime      = <date>,  
  
      endTime        = <date>,  
  
      runningStatus  = <number>,  
  
      name           = <string>,  
  
      originalNetworkId = <number>,  
  
      shortDescription = <string>,  
  
      extendedDescription = <string>,  
  
      copyrightId     = <number>,  
  
      copyrightInfo   = <string>,  
  
      parentalRating  = <number>,  
  
      parentalRatingDescription = <string>,  
  
      audioLanguageCode = <string>,  
  
      audioLanguageCode2 = <string>,  
  
      dataContentLanguageCode = <string>,  
  
      dataContentText = <string>,  
  
      hasInteractivity = <boolean>,  
  
      logoURI         = <string>,  
  
      contentDescription = {  
  
        [1] = <content_nibble_1>,  
  
        [2] = <content_nibble_2>,  
  
        [3] = <user_nibble_1>,  
  
        [4] = <user_nibble_2> }  
  
      },  
  
    },  
  
  },  
  
}
```

```

linkage = {
    tsId    = <number>,
    networkId = <number>,
    serviceId = <number>,
    type    = <number>,
    data    = <string>,
},

hyperlink = {
    type      = <number>,
    destinationType = <number>,
    tsId      = <number>,
    networkId  = <number>,
    eventId    = <number>,
    componentTag = <number>,
    moduleId   = <number>,
    serviceId  = <number>,
    contentId  = <number>,
    url       = <string>,
},

series = {
    id          = <number>,
    repeatLabel = <number>,
    programPattern = <number>,
    episodeNumber = <number>,
    lastEpisodeNumber = <number>,
    name = <string>,
},

eventGroup = {
    type = <number>,

```


The value of the *extendedDescription* data-table subfield should be computed according to the value of the *text_char* field of the *extended_event_descriptor* (see Section 8.3.7 of ABNT NBR 15603-2:2007).

The values of the *copyrightId* e *copyrightInfo* data-table subfields should be computed according to the values of the *copyright_identifier* and *additional_copyright_info* fields, respectively, of the *copyright_descriptor* (see Table 2-63 of ITU-T Rec. H.222.0 | ISO/IEC 13818-1: 2008).

The semantics of the *parentalRating* data-table subfield should be defined according to Table 33 of ABNT NBR 15603-2:2007. Its value should be computed according to the value of the *country_code* field of the *parental_rating_descriptor* and the environment variable (Settings node) *user.location*.

The semantics of the *parentalRatingDescription* data-table subfield should be defined according to Table 32 of ABNT NBR 15603-2:2007. Its value should be computed according to the value of the *country_code* field of the *parental_rating_descriptor* and the environment variable (Settings node) *user.location*.

The values of the *audioLanguageCode* and *audioLanguageCode2* data-table subfields should be computed according to the values of the *ISO_639_language_code* and *text_char* fields, respectively, of the *data_content_descriptor* (see Table 54 of ABNT NBR 15603-2:2007).

The values of the *dataContentLanguageCode* and *dataContextText* data-table subfields should be computed according to the values of the *ISO_639_language_code* and *text_char* fields, respectively, of the *data_content_descriptor* (see Table 54 of ABNT NBR 15603-2:2007).

The value of the *hasInteractivity* data-table subfield shall have the “true” value when event [i] has an interactive application available.

The value of the *logoURI* data-table subfield should define the logotype location transmitted in a CDT Table (see Section 8.3.44 of ABNT NBR 15603-2:2007).

The subfield values of the *contentDescription* table should be computed according to corresponding fields of the *content_descriptor* (See Section 8.3.5 of ABNT NBR 15603-2:2007).

The values of the *tsId*, *networkId*, *serviceId*, *type* and *data* linkage-table subfields should be computed according to the values of the *transport_stream_id*, *original_network_id*, *original_service_id*, *description_type* and *user_defined* fields, respectively, of the *linkage_descriptor* (see Section 8.3.40 of ABNT NBR 15603-2:2007).

The values of the *type*, *destinationType*, *tsId*, *networkId*, *eventId*, *componentTag*, *moduleId*, *contentId* and *url* hyperlink-table subfields should be computed according to the values of the *hyper_linkage_type*, *link_destination_type*, *transport_stream_id*, *original_network_id*, *event_id*, *component_tag*, *moduleId*, *content_id* and *url_char* fields, respectively, of the *hyperlink_descriptor* (see Section 8.3.29 of ABNT NBR 15603-2:2007).

The values of the *id*, *repeatLabel*, *programPattern*, *episodeNumber*, *lastEpisodeNumber* and *name* series-table subfields should be computed according to the values of the *series_id*, *repeat_label*, *program_pattern*, *episode_number*, *last_episode_number* and *series_name_char* fields, respectively, of the *series_descriptor* (see Section 8.3.33 of ABNT NBR 15603-2:2007).

The values of the *type*, *id*, *tsId*, *networkId* and *serviceId* eventGroup-table subfields should be computed according to the values of the *group_type*, *event_id*, *transport_stream_id*, *original_network_id* and *service_id* fields, respectively, of the *event_group_descriptor* (see Section 8.3.34 of ABNT NBR 15603-2:2007).

The values of the *type*, *id*, *totalBitRate*, *description*, *caUnit.id*, *caUnit.component[k].tag*, *tsId*, *networkId* and *serviceId* componentGroup-table subfields should be computed according to the values of the *component_group_type*, *component_group_id*, *total_bit_rate*, *text_char*, *CA_unit_id* and *component_tag* fields, respectively, of the *component_group_descriptor* (see Section 8.3.37 of ABNT NBR 15603-2:2007).

type='time'

The table of the ‘time’ event type contains information about the current UTC (Universal Time Coordinated) date and time, but in the official country time zone in which the receptor is located.

Each request for a table of ‘time’ event type shall be carried out through the following call:

```
event.post('out', { class='si', type='time' })
```

The returned event is created after all request information is processed by the middleware (information that is not broadcasted within the maximum interval specified by Table 6 of ABNT NBR 15603-2:2007 shall be returned as ‘nil’). The data table is returned as follows:

```

evt = {
    class = 'si',
    type = 'time',
    data = {
        year      = <number>,
        month     = <number>,
        day       = <number>,
        hours     = <number>,
        minutes   = <number>,
        seconds   = <number>
    }
}

```

NOTE In Ginga implementation in conformance with ABNT NBR 15606-2:2007, in order to compute the values of the data-table subfields to be returned in events of time type, the TOT table should be used as a basis, as well as the local_time_offset_descriptor, according to Section 7.2.9 of ABNT NBR 15603-2:2007.

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

2.3. 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 to set values to the 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 trough 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!

2.4. 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 procedural object. In order to do that it defines a reserved area, inaccessible to non-procedural NCL media objects. This area is split into the groups “service”, “channel” and “shared”, with same semantics of the homonym groups of the NCL settings node. There are no predefined or reserved variables in these groups, and procedural objects are allowed to change variable’s values directly. Other procedural languages, in particular Java for NCLet objects (<media> elements of type application/x-ginga-NCLet) 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, procedural codes may change field values.

Examples of use:

```
persistent.service.total = 10
```

```
color = persistent.shared.color
```