Variable Handling in Time-Based XML Declarative Languages

Luiz Fernando G. Soares[1], Rogério Ferreira Rodrigues[1,2],
Renato Cerqueira[1], Simone Diniz Junqueira Barbosa[1]

lfgs@inf.puc-rio.br, rogerio.rodrigues@fast.no, rcerq@inf.puc-rio.br, simone@inf.puc-rio.br

[1]Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225
22451-900 Rio de Janeiro, RJ Brazil
+55 21 3527-1500 ext:4330

[2]FAST, a Microsoft Subsidiary
Av. Rio Branco 1, 1611
20090-003 Rio de Janeiro, RJ Brazil
+55 21 2102-2013

Variable Handling in Time-Based XML Declarative Languages

Luiz Fernando G. Soares[1], Rogério Ferreira Rodrigues[1,2], Renato Cerqueira[1], Simone Diniz Junqueira Barbosa[1]

lfgs@inf.puc-rio.br, rogerio.rodrigues@fast.no, rcerq@inf.puc-rio.br, simone@inf.puc-rio.br

[1]Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225
22451-900 Rio de Janeiro, RJ Brazil
+55 21 3527-1500 ext:4330

[2]FAST, a Microsoft Subsidiary
Av. Rio Branco 1, 1611
20090-003 Rio de Janeiro, RJ Brazil
+55 21 2102-2013

ABSTRACT

This paper focuses on time-based declarative languages. The use of declarative languages has the advantage of their simplicity and their high-level abstraction, usually requiring few or no programming skills. Moreover, in general, declarative languages benefit portability and allow automatic control of application execution temporal flows, without author awareness. However, most time-based declarative languages have limited support for variable definition and manipulation, which causes developers to resort to imperative languages. This paper discusses and proposes an approach for variable handling in XML-based declarative languages used for temporal synchronization among media objects that balances flexibility and simplicity. An important goal is to resort to imperative languages only for those applications that require intensive algorithmic computation. The proposed solution was adopted by the NCL declarative language of the Brazilian DTV System.

Categories and Subject Descriptors

I.7.2 [**Document Preparation**]: Languages and systems, markup languages, multimedia, hypermedia, standards. D.3.2 [**Language Classifications**]: Specialized application languages

General Terms

Standardization, Design.

Keywords

Variable handling, declarative languages, NCL, digital TV, middleware.

1. INTRODUCTION

Declarative languages emphasize the declarative description of an application, rather than its decomposition into an algorithmic implementation, as it is done when using imperative languages. Declarative languages typically define a specific model to design applications in a target domain. Such declarative descriptions are closer to a high-level specification, and thus they are easier to be written and understood than imperative ones, which usually require programming expertise.

This paper is particularly interested in *time-based declarative application languages*, in the context of multimedia document engineering. They define abstractions to specify the spatial and temporal synchronization among components of an application, what makes them especially suitable for multimedia applications. The synchronization targets include traditional media objects such as video, audio (including streaming video and audio), image, text and others, but also media objects that contain imperative code. There are many examples of such languages applied to the multimedia application domain, for instance: NCL (Nested Context Language) [10], the standard declarative language of the Brazilian terrestrial digital television (DTV) System; XHTML-based DTV middleware languages [2, 4]; SMIL (Synchronized Multimedia Integration Language) [14]; SVG (Scalable Vector Graphics) [16] and BIFS-XMT-O (Binary Format for Scenes of eXtensible MPEG-4 Textual) [8].

Variable handling is a key feature to provide a better control of a multimedia presentation. Variables can be *local* to a media object (e.g., determining the object's position) or *global* to the exhibition context (e.g. defining the available network bandwidth). In addition, the multimedia *presentation state* is an important object for guiding the execution control (e.g. defining for a given time moment which media objects are *occurring, paused, etc.*).

Time-based XML declarative languages should support the following features related to variable handling:

- defining local and global variables (the latter with several different types of persistency and access rights);
- adapting content and content presentation based on the values of global variables;
- conditioning navigation to the values of both local and global variables, as evaluated in simple or compound expressions;
- setting values to both local and global variables, depending on the values of other variables, on the navigational flow and on the spatial behavior of an application;
- continuous setting of values (animation) to both local and global variables;
- storing and sharing the current context of an application execution, for its future retrieval and, therefore, for resuming the application at a later time.

This paper presents the Ginga-NCL approach for supporting these variable handling facilities, illustrated by several examples. NCL is the declarative modular language of Ginga [1]. It provides wide expressiveness in the declarative handling of local and global variables[1] and in managing the program presentation state.

This approach is not limited to NCL, however. NCL modules can be used to add NCL facilities to other XML-based languages. Thus, the ideas presented in this paper can also be inherited by other languages through using NCL modules. Alternatively, the ideas can be taken as a design principle to be incorporated into other time-based declarative language models.

The paper is structured as follows. Section 2 discusses related work done for XML application languages. Section 3 presents how local and global variables are declared in NCL. Section 4 describes how these variables can be handled by declarative language elements. Section 5 introduces an overview of the multimedia-document state management performed by Ginga. Section 6 discusses how the scripting language of NCL can manipulate variables in a controlled way, without impairing the application's temporal graph. Finally, Section 7 presents the paper final remarks.

2. RELATED WORK

XHTML-based declarative languages, such as BML [2] and DVB-HTML [4], allow declarative manipulation of formatting variables through style sheets, usually written in CSS [11]. Document elements and attributes can also be manipulated through the document tree, using DOM events [12]. This last case, however, usually needs imperative coding, commonly written in ECMAScript [3]. DOM events and external events[2] can trigger a scripting object to provide temporal and spatial synchronization, content and presentation adaptability, and other facilities that are not otherwise possible in XHTML.

In contrast, languages like BIFS XMT-O [8], SMIL [14], SVG [16] and HTML+Time [13] provide a better support for declarative manipulation of variables. They integrate SMIL timing concepts within their framework and provide support for a subset of SMIL system global variables, as it will be described. SVG and HTML+Time provide support to query and set their document tree nodes and attributes using DOM.

Based on imperative language syntaxes, King et al. [9] propose a set of extensions to XML application languages. They focus on the possibility of specifying attribute values using expressions that can be dynamically evaluated, and on allowing a document author to define events based on predicate expressions. Although the proposal has been discussed in the context of SVG and SMIL, it was not included in these languages' recommendations.

SMIL allows local positioning-variable definitions and their manipulation. Early SMIL versions allow testing global system-defined variable values, within the language element named `<switch>`. The first child element of a `<switch>` (a media object or a composite object) which has its selection predicate evaluated as "true", is selected for activation. From version 2.0 on, it has been allowed to attach selection predicates directly to SMIL elements, no longer needing a `<switch>` element to condition their activation. Until version 2.1, the activation control of a SMIL element was limited to a simple predicate logic evaluation, comparing the predicate value to an absolute value and returning a Boolean value as a result. Moreover, global test variables have been implemented placing most of their initialization and control within the SMIL engine (user agent), rather than within the language. A new proposal for state variable[3] maintenance has been introduced in SMIL 3.0 [15]. In the new version, variables can be tested using a richer arithmetic and logical operator set (=, !=, <=, <, >=,>, `and`, `or`, `not`, +, -, * and `div`, when XPath [17] syntax is used), besides the previously used "=" operator. Variables are tested in expressions defined within a new attribute named `expr`. The new proposal also obliges the expression evaluation each time an element is activated, solving a previous ambiguity of SMIL that also allowed an evaluation during a document parsing. In the new version 3.0, variables can also be defined, referenced and changed through a new `<setvalue>` SMIL element. The value to be set can be a simple absolute value, or evaluated by an expression defined in the element's `expr` attribute. The `<setvalue>` element also defines a `begin` attribute, specifying when the assignment must happen. Continuous attributions (animations), however, are not allowed in `<setvalue>` elements.

Although the new SMIL version recognizes the importance of defining a standard mechanism for saving, restoring and sharing the multimedia presentation state as a whole, it does not define such mechanism. The whole presentation state can only be partially inferred (coarse tuning) by using `<setvalue>` elements appropriately placed inside a document (similar to what is done when one tries to define debugging points in a program).

A fine multimedia presentation state control should allow maintaining the execution history of an application. This is very important when an application needs to migrate from an exhibition context to another one, as for example, from a display with partial screen exhibition to a full screen, from one device to another (e.g. from a TV set to a mobile phone), or even when an application must be stopped and later resumed, from the same interrupted point. These are common situations found in a DTV environment where, in addition, it is very usual to stop a presentation and then to resume it in a future moment in time, preserving all actions previously executed. This situation often happens, for example, when a viewer changes the TV channel and then returns to it, trying to get away from advertisements or simply looking for the beginning of an event in another channel.

To manage the multimedia presentation state control is to manage the application temporal graph. Besides allowing for dynamic navigation (i.e., changes on the graph traversal), saving the graph context is also important, at least the current position in the graph.

3. NCL VARIABLES

NCL allows defining local and global properties both implicitly, by the NCL user agent (the NCL engine/player), and explicitly, by the application author.

To explicitly define a variable, we use an NCL `<property>` element, containing a `name` attribute, to identify the property, and optionally a `value` attribute, to assign an initial value to it. Also, properties defined inside the same parent element must have

[1] In NCL, variables are called *properties*. In the remainder of this document, the terms *variable* and *property* will be used interchangeably when referring to NCL variables.

[2] In DTV systems, it is usual to have DSM-CC stream events triggering the execution of imperative codes.

[3] SMIL uses the term "state variable" instead of "global variable".

different values for the `name` attribute (i.e., the variable name is unique in its scope). For instance, to define the local property named `viewed` to a certain media object, with the initial value of `false`, we would write:

```
<media id="advertGlasses"
       type="image" src="advert.png">
  <property name="viewed" value="false"/>
</media>
```

The `<property>` element can also be used to define a group of properties. For instance, the dimensions of a media object may be defined either as individual `width` and `height` properties, or as a grouped `size` property, whose value has the format `width,height`. When a group of variables is changed, their consistency must be checked only at the end of the process.

Global variables can be defined by the document author or can be reserved environment variables[4]. The global variable *type* defines the variable scope, its persistency and its access rights. It is important to mention that this attribute has no relation with the type of the attribute `value`, which is implicitly obtained from the variable semantics. The Ginga middleware defines the following global variable types [1]:

- **system** and **user** (e.g. `system.language`, `user.location`): variables that are managed by the receiver system. They may be read, but they may not have their values changed by an NCL application, a Lua procedure or an Xlet procedure[5], only by the receiver's native applications. They persist during the receiver's life cycle.
- **default** (e.g. `default.selBorder`, the default color applied to the border of an element in focus when activated): variables managed by the receiver system. They may be read and have their values changed by an NCL application, by the receiver's native applications, by a Lua procedure or by an Xlet procedure. They persist during the receiver's life cycle. However, they are reset to their initial values when a new channel is tuned.
- **service** (e.g. `service.currentFocus`, which indicates the `<media>` element currently on focus): variables managed by the NCL engine. They may be read and have their values changed by an NCL application of the same service, but they may be only read by a Lua procedure or an Xlet procedure of the same service. They persist at least during the service life cycle.
- **si** (eg. si.channelNumber, the number of the current tuning channel) variables that are managed by the receiver middleware. They may be read, but they cannot have their values changed by an NCL application, by a Lua procedure or by an Xlet procedure. They persist at least till the next channel tuning.
- **channel** (in the format `channel.XXX`): variables managed by the NCL engine. They may be read and have their values changed by an NCL application of the same channel, but they may be only read by a Lua procedure or an Xlet procedure of the same channel. They persist at least until the next channel tuning.
- **shared** (in the format `shared.XXX`): variables managed by the NCL engine. They may be read and have their values changed by an NCL application. They may be only read by a Lua or an Xlet procedure. They persist at least during the life cycle of the service that has defined them.

[4] Reserved variables are the ones whose name and semantics are defined in the NCL profile specification [1].

[5] Lua is the Ginga scripting language, while Xlet is the Java code that runs in Ginga.

To declare a global variable, a `<property>` element is used as a child element of the NCL settings node, represented by a `<media>` element of type `"application/x-ginga-settings"`. A settings node, unique in an NCL document, groups all global variables, either explicitly or implicitly declared. For example, let us suppose that we need to check whether interactive advertisements are to be presented or not. If we want to configure this per channel, and assume the value "true" as the default, we might declare a user-defined global variable of the `channel` type called `interactivity`, as follows:

```
<media id="globalVariables"
  type="application/x-ginga-settings">
  <property name="channel.interactivity"
      value="true"/>
</media>
```

As the variable type is "channel", it will persist for every program (service) in exhibition until the next channel change.

Local variables can also be defined by the document author or can be reserved variables. Local variables may be read and have their values changed only by an NCL `<link>` element, as discussed in Section 4. They shall persist at least during the life cycle of the NCL object that has defined them.

There are several reserved names for local variables, as those for an object spatial positioning and duration, and others defining additional presentation characteristics for an object (e.g. sound level). For instance, we might declare and initialize the reserved local `duration` property and the `bounds` reserved-group of properties as follows:

```
<media id="advertGlasses"
  type="image" src="advert.png">
  <property name="duration" value="7s"/>
</media>
<media id="mainVideo"
  type="video" src="film.mpg">
  <property name="bounds" value="0,0,100%,80%"/>
</media>
```

4. DECLARATIVE HANDLING OF VARIABLES IN NCL

Media objects, including NCL imperative objects, are related (synchronized) using NCL `<link>` elements. In order to specify relationships among NCL objects, NCL links are bound to the media object interfaces: `<area>` elements (defining part of the media object content) and `<property>` elements, as previously defined. Section 4.2 deals with variable manipulations by `<link>` elements.

Variables can also be used to select a node content to be presented, or to select certain presentation characteristics, in adaptive applications, as will be seen next.

4.1 Adapting Content and Presentation

NCL allows the definition of *selection rules* based on global variables. These rules can be used to select applications' content and content presentation, from a set of alternatives. Rules can be simple, defined by the `<rule>` element, or compound, defined by the `<compositeRule>` element. Simple rules test global variables against specified values, using comparison operators ("`eq`", "`ne`", "`gt`", "`lt`", "`gte`", "`lte`"). Compound rules test simple or compound rules joined through logical operators "`and`"/"`or`". NCL rules are grouped into a base (`<ruleBase>` element) specified in the document header.

For example, let us define two simple rules to test the user language preference and two simple rules to test the receiver's CPU capability:

```
<ruleBase>
    <rule id="rNPt" var="system.language" comparator="ne"
        value="pt-br"/>
    <rule id="rPt" var="system.language" comparator="eq"
        value="pt-br"/>
    <rule id="rAdvCPU" var="system.CPU"
        comparator="gte" value="333"/>
    <rule id="rBasicCPU" var="systemCPU"
        comparator="lt" value="333"/>
</ruleBase>
```

We could also define compound rules, for instance, to test "channel" variables (advertisement and interactivity), as follows:

```
<ruleBase>
    <compositeRule id="fullAdvert" operator="and">
      <rule id="rule-pub01"
          var="channel.advertisement"
          comparator="eq" value="true" />
      <rule id="rule-int01"
          var="channel.interactivity"
          comparator="eq" value="true" />
    </compositeRule>
    <compositeRule id="noAdvert" operator="and">
      <rule id="rule-pub02"
          var="channel.advertisement"
          comparator="eq" value="false" />
      <rule id="rule-int02"
          var="channel.interactivity"
          comparator="eq" value="false" />
    </compositeRule>
</ruleBase>
```

A rule may be associated with child elements of `<switch>` elements, similar to SMIL. The first child element that has its associated rule evaluated as true is then selected for activation. Rules are dynamically evaluated every time the `<switch>` element becomes active. For instance, to determine the more suitable media object depending on the "`system.language`" property, we might specify the following `<switch>` element:

```
<switch id="form">
  <bindRule rule="rPt" constituent="ptForm"/>
  <bindRule rule="rNPt" constituent="enForm"/>

  <media id="ptForm" descriptor="formDesc"
    src="example/ptForm.html"/>
  <media id="enForm" descriptor="formDesc"
    src="example/enForm.html"/>
</switch>
```

Different from other declarative languages, NCL allows rules to be used also to select the presentation characteristics of a media object (e.g. sound level), through the `<descriptorSwitch>` element. Similar to the `<switch>` element, rules can be associated with `<descriptorSwitch>` child elements (`<descriptor>` elements). The first rule evaluated as true makes the associated `<descriptor>` element the right selection. As usual, rules are dynamically evaluated each time a `<descriptorSwitch>` element becomes active. In NCL, `<descriptor>` elements are used to separate the document content and structure specification from its presentation behavior definition. As it can be observed in the previous code fragment, each media object is associated with a descriptor. This descriptor could actually be a switch of alternative descriptors, where the form media object presentation characteristics are selected based on the platform capability. For instance, to apply a transition effect only if a minimum CPU clock is available, we could specify the descriptor switch as follows:

```
<descriptorBase>
  <descriptorSwitch id="formDesc">
      <bindRule rule="rAdvCPU"
          constituent="advTxtDsc"/>
      <bindRule rule="rBasicCPU"
          constituent="basicTxtDesc"/>
      <descriptor id="advTxtDesc" region="textRegion"
          transIn="transFade"/>
      <descriptor id="basicTxtDesc"
          region="textRegion"/>
  </descriptorSwitch>
</descriptorBase>
```

4.2 Navigation Based on Variables

The major difference between NCL and related declarative languages is its wealth of declarative possibilities in the use of variables to define the temporal behavior of a document presentation. NCL allows the declarative manipulation of its data model to transform its flow control model, and vice-versa, without compromising the temporal consistency of the presentation. Even manipulations using imperative objects are controlled, as discussed in Section 6, different from what happens with common uses of scripting languages.

The NCL spatial and temporal relationships among media objects are based on a causality/constraint paradigm. Relationships are specified using links (represented by `<link>` elements).

Causal relationships specify conditions (link sources) over event states or media object property (variable) values. When conditions are satisfied, actions (link targets) are performed. Actions may change the state of other events or set values to properties.

Let us consider the following scenario: when an advertisement ends, if interactivity is on, then a form should be presented and the main video resized. This specific relationship (defined by a `<link>` element relating several media objects) refers to a more generic relation (defined by an `<xconnector>` element), which in turn specifies the desired behavior. The following code fragment illustrates such a link:

```
<link id="linkForm" xconnector="onEndVarStartSet">
  <bind role="onEnd" component="advertGlasses"/>
  <bind role="testVar" component="settings"
    interface="channel.interactivity">
    <bindParam name="var" value="true"/>
  </bind>
  <bind role="start" component="form"/>
  <bind role="set" component="mainVideo"
      interface="bounds">
    <bindParam name="bounds" value="0,0,50%,50%"/>
  </bind>
</link>
```

This link can be read as: "Following the behavior specified in the connector `onEndVarStartSet`, when the presentation of `advertGlasses` finishes (`onEnd` role), if the value of the global property `channel.interactivity` is `true` (`testVar` role), then do two things: a) start the presentation of the `form` media object (`start` role), and set the value of the `bounds` property of the `mainVideo` media object (`set` role) to `"0,0,50%,50%"` (i.e., reposition and resize it)".

Note the use of bind parameters (`<bindParam>` elements): first for testing the value "`true`" (parameterized as "`var`") against the "`channel.interactivity`" variable value; second for setting a group of values parameterized as "`bounds`" to a group of variables with the same name.

Constraint relationships[6], without any involved causality, can also be specified by using NCL links. Consider, for instance, the following constraint, defining that two media object local variables, in the case "`left`" variables, must have the same value (thus, the objects must always be left aligned).

```
<link id="linkLeftAlign" xconnector="maintainVarEqual">
  <bind role="left1" component="advertGlasses"
    interface="left"/>
  <bind role="left2" component="advertLegend"
    interface="left"/>
</link>
```

4.3 Continuous Variable Settings

NCL animation primitives allow to progressively change variable values, throughout a specified period of time. Basically, NCL allows two new attributes to be associated with assignment actions in causal links: the "`duration`" and "`by`" parameters.

By default, variable set operations are performed instantaneously. However, if the `duration` parameter is specified, the assignment operation will take the specified period of time. In this case, the change from the old value to the new one can be linear or discrete. In the latter case, the step modification is specified in the *by* attribute.

Together, `duration` and `by` attributes offer support to presentation animations, easily and declaratively specified. The following code span specifies a linear vertical movement (changing the `top` property) for a media object. The animation begins as soon as the object presentation starts.

```
<link id="linkChangeTopPosition"
      xconnector="onBeginSetAnim">
  <bind role="onBegin" component="advertGlasses"/>
  <bind role="set" component="advertGlasses"
        interface="top">
    <bindParam name="var" value="50%"/>
    <bindParam name="duration" value="2s"/>
  </bind>
</link>
```

In the next example, when the glasses advertisement ("`advertGlasses`" object) begins, the legend ("`advertLegend`" object) must also be started. The `left` position value of the glasses advertisement must be retrieved and assigned to the `left` position property of the legend. The change from the old value of the legend's `left` position to the new one is done step by step (2 pixels at a time, as indicated by the `by` parameter), and the movement to align the objects will take 2 seconds (`duration` parameter). Note the use of the bind parameter "`var`" specifying that the value must be retrieved from the role named "`getValue`".

[6] For the Brazilian DTV System, the NCL profile only allows causal relationships.

```
<link id="linkAlignTopPosition"
      xconnector="onBeginStartSetAnim">
  <bind role="onBegin" component="advertGlasses"/>
  <bind role="start" component="advertLegend"/>
  <bind role="getValue" component="advertGlasses
        interface="left"/>
  <bind role="set" component="advertLegend"
        interface="left">
    <bindParam name="var" value="$getValue"/>
    <bindParam name="duration" value="2s"/>
    <bindParam name="by" value="2"/>
  </bind>
</link>
```

5. STORING AND RETRIEVING THE PRESENTATION STATE

Almost all multimedia documents, including DTV applications, are composed by a collection of media objects to be autonomously presented. The document presentation can run as many times as necessary, partially or totally. For each runtime, a new activation is done, without any historical relationship with the previous ones.

However, there are many situations where the application's presentation history is important to be maintained, in order to reach the desired results, as for example:

- When viewers are allowed to explicitly pause a DTV application and then resume it at some later time — possibly days or weeks later, and even on a different device;
- When a viewer changes the TV channel, thus starting another application in the new channel, but then changes her mind and returns to the previous channel, resuming the application and inheriting all information previously given, all answers previously provided, all interactions previously carried out, all environment information (global variables) previously set, etc.

In the first example, the application (and all its content) must be resumed from the exact point where it had been stopped, or from a preceding point. In contrast, in the second example, the application must be resumed in a future point of the TV program, with regards to the time it had been stopped. Nevertheless, in both cases, all previous information should be preserved.

In NCL, the state of a multimedia presentation is maintained using a data structure called *temporal graph*. This data structure can be initially created from the application specification, and represents all events that can happen during the application execution (or presentation). Besides predictable events, unpredictable events are also represented, like possible viewer interactions, decision points for content and presentation adaptation, etc.

An NCL event is an occurrence in time that may be instantaneous or have a measurable duration. NCL defines the following basic types of events (which can be extended): *presentation* (corresponding to playing a content, wholly or partly); *selection* (corresponding to a viewer interaction); and *attribution* (corresponding to setting a value to a variable, that is, a `<property>`).

Each event defines a state machine that should be maintained by the NCL user agent. An event can be in one of three states: *sleeping, occurring* or *paused*. The current state of each event state machine should be maintained by the temporal graph.

During an application presentation, all information gathered from viewers and from the system, all viewer answers, all viewer interactions, and all local and global variable settings are collected, updating the temporal graph. Therefore, this data structure represents the current multimedia presentation state, which can be stored and later retrieved, allowing resuming the application from its saved state.

An NCL user agent shall be capable of resuming an application from a previously saved state point. If the application has some object with live content, the temporal graph should jump to the current time point of this object. Therefore, the two examples presented in the previous paragraph bullets are supported.

In the Ginga middleware, the temporal graph of an application is maintained in a conceptual entity called "private base". In the Brazilian DTV system, there is a private base associated with each TV channel. Ginga provides support to several private base commands, as follows:

- `openBase (baseId, location)`: Opens an existing private base to be found with the location parameter. If the private base does not exist or the parameter attribute is not informed, a new base is created with `baseId`.
- `activateBase (baseId)`: Turns on an opened private base.
- `deactivateBase (baseId)`: Turns off an opened private base.
- `saveBase (baseId, location)`: Saves all private base content (the temporal graph) into a persistent storage device (if available). The `location` attribute shall specify the device and the path for saving the base.
- `closeBase (baseId)`: Closes the private base and disposes all private base content.
- `addDocument (baseId, {uri, ior}+)`: Adds an NCL application to a private base. The NCL document is sent in the object carousel as files in the file system; the pair `{uri,ior}` relates a file system path in the datacast provider to its respective location in a carousel.
- `removeDocument (baseId, documentId)`: Removes an NCL document from a private base.
- `startDocument (baseId, documentId, interfaceId, offset)`: Starts playing an NCL document in a private base, beginning the presentation from a specific document interface.
- `stopDocument (baseId, documentId)`: Stops the presentation of an NCL document in a private base. All document events that are occurring shall be stopped.
- `pauseDocument (baseId, documentId)`: Pauses the presentation of an NCL document in a private base. All document events that are occurring shall be paused.
- `resumeDocument (baseId, documentId)`: Resumes the presentation of an NCL document in a private base. All document events that have been paused by a previous `pauseDocument` command shall be resumed.
- `saveDocument (baseId, documented, location)`: Save an NCL document into a persistent storage device (the location attribute specifies the device and the path for saving).

The NCL user agent is in charge of receiving an NCL document and controlling its presentation, trying to guarantee that the specified relationships among media objects are respected. It deals with NCL documents that are maintained inside a private base, which may be started, paused, resumed, stopped, and may refer to each other. A private base can be opened, closed, activated, deactivated, and saved. If a document is paused and the private base is saved, at a later time, when the private base is opened, the document can be resumed from the point it has been paused.

6. STATE MANIPULATION BY NCL IMPERATIVE MEDIA OBJECTS

Despite its declarative nature, NCL recognizes that the richer expressiveness of imperative languages cannot be ignored, mainly for applications that produce many dynamic contents, as a result of complex operations. Therefore, NCL also includes support for imperative objects (media objects with imperative code). In particular, the NCL profile for the Brazilian DTV middleware allows for imperative objects implemented in Lua [6] or Java.

Imperative languages, in particular scripting languages, provide an expressive support for variable handling. However, they have a high cost: they usually require a programming expertise for application development; they occasionally put at risk the application portability; and the temporal graph management and the presentation control are much more difficult to be done as a rule. In addition to these problems, the use of imperative programming languages is more prone to errors committed by the application programmer. As the loss of the execution control can break the presentation temporal consistency, and because this is a very difficult problem to be treated in dynamic presentations, scripting languages are usually limited in their expressiveness scope. They are usually not allowed to change the temporal graph, except by using declarative primitives.

The drawbacks of the imperative paradigm are the advantages of the declarative one that, in addition, allows easier application transformations[7]. However, a poor model for variable handling can push a time-based declarative language away from its goal, requiring scripting language objects to define more complex interactions.

Thus, a declarative language should offer a good balance between flexibility and simplicity. It should provide a variable handling model sufficiently rich to describe a wide range of interactive applications, avoiding, as much as possible, the help of an imperative scripting language. At the same time, the declarative simplicity should not be lost, leaving for imperative objects the more complex manipulations, with the necessary care to avoid any impact in the application's temporal graph. Variables and the presentation state of an NCL application are handled according to this principle.

Thus, recognizing this situation, NCL supports the use of media objects that represent chunks of code in an imperative language, which are called *imperative media objects*. Specifically to its DTV profiles, NCL supports imperative media objects implemented in Java (NCLet object) and Lua (NCLua object). In fact, Lua is the main scripting language used by NCL. Lua is a powerful, fast, lightweight, embeddable scripting language. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping. Lua has been used in many industrial applications, with an emphasis on embedded systems

[7] In the case of XML languages, XSL transformations can be applied to document trees.

and games. In fact, Lua is currently the leading scripting language in games [7].

The NCL/Lua integration provides a set of controlled mechanisms to support the interaction between these two languages, especially for triggering flow control events and for handling NCL variables and the document structure through imperative commands. Although the rest of this section presents details about the use of Lua in NCL documents, most of this information can also be applied to imperative media objects implemented in Java.

6.1 Imperative Media Objects

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

A `<media>` element containing imperative code can define content anchors (through `<area>` elements) and properties (through `<property>` elements), as all NCL media objects can do. Imperative code chunks may be associated with `<area>` elements using the `label` attribute. In this case, the `label` value must identify a chunk of code (a function, a method, etc.). A `<property>` element can be mapped to a code function or to a code variable. The `name` attribute of the `<property>` element must be used to identify the function or the variable in the imperative code. As usual, `<area>` and `<property>` elements can be used as interface points of `<link>` elements, establishing a two-way bridge between the declarative and imperative environment.

To declare an NCLua media object that defines two properties (`inc` and `counter`) associated with homonym entities in the Lua code (in this case, a function and a variable, respectively), we could specify the following:

```
<media id="clicks" src="clicks.lua"
      type="application/x-ginga-NCLua">
  <property name="inc"/>     <!-- inc function -->
  <property name="counter"/> <!-- counter variable -->
</media>
```

6.2 Lua Code Interaction with other NCL Objects

The NCL user agent controls the lifecycle of an imperative NCLua object. The user agent is responsible for triggering the execution of an imperative object and for mediating the communication between this object and other NCL objects of an application.

As with all media object players (e.g., video player, audio player, image player, etc.), once instantiated, the NCLua player (the language engine) shall execute an initialization procedure. However, different from other media players, this initialization code is specified by the author of the Lua code. This initialization procedure is executed only once, for each instance, and creates all Lua code chunks and data that may be used during the imperative-object execution and, in particular, registers at least one event handler for communication with the NCL user agent.

After the initialization, the execution of the NCLua object becomes event oriented in both directions. That is, any action commanded by the NCL user agent reaches the registered event handlers, and any NCL event state change notification is sent as an event to the NCL user agent (as for example, the natural end of a Lua code chunk execution).

Application authors may define NCL links to start, stop, pause, resume or abort the execution of a Lua code. A Lua player shall interface the imperative execution environment with the NCL user agent. Analogous to conventional media content players, Lua players shall control event state machines associated with the NCLua object. As an example, if the code finishes its execution, the player shall put the event presentation state machine corresponding to the code execution in the sleeping state. However, different from media content players, a Lua player does not have sufficient information to control by itself all event state machines, and shall rely on the Lua application to command these transitions.

NCL links can be bound to NCLua object interfaces (content anchors and properties). If an NCL link starts, stops, pauses, resumes, or aborts the presentation of an anchor, callbacks in the Lua code shall be triggered.

On the other hand, a Lua code can also command the start, stop, pause or resume of its associated NCL content anchors through an event driven API offered by the language. The corresponding changes in the event state machine associated with the anchor may be used as conditions of NCL links to trigger actions on other NCL objects of the same application. Thus, a two-way synchronization can be established between the imperative code and the remainder of the NCL document.

A Lua code may also be synchronized with other NCL objects through `<property>` elements (NCL variables). When the `<property>` element is mapped to a Lua *code chunk* through its `name` attribute, a link action "`set`" applied to the property shall cause the code execution, with the set values interpreted as parameters passed to the code chunk. When the `<property>` element is mapped to a Lua *variable*, the action "`set`" shall assign the value to that variable. As usual, the event state machine associated with the property shall be controlled by the Lua player but, in some situations, commanded by the Lua application.

In addition to interactions through `<area>` and `<property>` elements, the imperative language used with NCL shall offer an API that allows an imperative code to query any pre-defined or dynamic properties of the NCL *settings* node (see Section 3). 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 through using NCL links, in order to allow an easier consistency control of the application flow done by the NCL user agent.

6.3 NCL Editing Commands

An imperative-code player should also offer an API that provides a set of methods to manipulate the NCL document structure (*editing commands*) and the NCL's private base (see Section 5). In Lua, this API is provided by the *event* Lua module [1] in its *edit* class. The editing commands allow any NCL element to be added to or removed from an NCL document presentation. The commands to manipulate the private base are equivalent to those presented in Section 5. In addition to those commands, the *ncledit* Lua module also provides a command (*ncledit.setPropertyValue*) to set values of NCL variables.

It is important to note that the facility provided by editing commands is not equivalent to the direct manipulation of the DOM representation of an NCL document. These commands represent more abstract operations, which are defined based on the abstractions provided by the NCL document model. The NCL user agent is responsible for performing all these commands and for guaranteeing the overall consistency of the document presentation after each command execution.

7. FINAL REMARKS

The goal of NCL profiles has been to define a set of functionalities rich enough to allow the development of a wealth of time-dependent applications without resorting to imperative procedures. However, NCL recognizes that an imperative programming support may be required, especially for those applications that demand intensive generation of dynamic content, as result of a complex computation. Thus NCL also provides support for objects containing imperative code, but under control, in order to avoid inconsistencies in the application temporal flow.

With the exception of the presentation state control, all ideas presented in the paper are employed in the reference implementation of the Ginga middleware of the Brazilian digital TV system: an open source implementation that can be obtained from http://www.softwarepublico.gov.br. They are also implemented in all commercial implementations of Ginga. A prototype of an NCL user agent including the presentation state control is now in testing, and will be also available as an open source implementation. However, further work has yet to be done involving multiple devices in resuming a stored application.

Several applications have been developed in order to test the usefulness and the expressiveness of the proposal. The result has been quite satisfactory. Indeed, most of DTV applications have been making use only of NCL's declarative primitives and, for the more complex applications, the support offered by NCL imperative objects has been more than sufficient.

Finally, authoring support needs to be defined for hiding the syntax of variable definitions and manipulations for non-expert developers. A graphical support is in its initial stage of development aiming at being incorporated to the NCL Composer [5] tool, also offered as open source software.

8. ACKNOWLEDGMENTS

The authors would like to thank Carlos Salles, Romualdo Costa, Marcio Moreno, Marcelo Moreno and Francisco Sant'anna who provided a thoughtful discussion of this work, tracked down and fixed problems in the initial reference implementation of Ginga. Luiz Fernando Soares and Simone Barbosa thank CNPq for its support. Simone Barbosa also thanks FAPERJ, project #E-26/100.685/2007.

9. REFERENCES

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

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

[3] ECMA International - European Association for Standardizing Information and Communication Systems. 1999. ECMA – 262 – ECMAScript Language Specification. 3rd Edition. DOI= http://www.ecma-international.org/publications/standards/Ecma-262.htm.

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

[5] Guimarães, R.L; Costa, R.R.; Soares, L.F.G. 2008. Composer: Authoring Tool for iTV Programs. In Proceedings of European Interactive TV Conference (Salzburg, Austria, July 2008). EuroiTV 2008. DOI= http://dx.doi.org/10.1007/978-3-540-69478-6_7

[6] Ierusalimschy R., Figueiredo L.H., Celes W. 2006. Lua 5.1 Reference Manual, Lua.org, ISBN 85-903798-3-3.

[7] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. 2007. The evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California, June 09 - 10, 2007). HOPL III. ACM, New York, NY, 2-1-2-26. DOI= http://doi.acm.org/10.1145/1238844.1238846.

[8] ISO/IEC International Organization for Standardization 14496-1:2001. Coding of Audio-Visual Objects – Part 1: Systems. 2nd Edition.

[9] King P., Schmitz P., Thompson S. 2004. Behavioral reactivity and real time programming in XML: functional programming meets SMIL animation. In Proceedings of ACM Document Engineering (Milwaukee, Wisconsin, USA, October, 2004). DOI=http://doi.acm.org/10.1145/1030397.1030411

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

[11] W3C World-Wide Web Consortium. 1998. Cascading Style Sheet, level 2 – CSS. W3C Recommendation.

[12] W3C World-Wide Web Consortium. 2004. Document Object Model – DOM Level 3 Specification. W3C Recommendation.

[13] W3C World-Wide Web Consortium. 1998. Timed Interactive Multimedia Extensions for HTML. W3C Submission. DOI= http://www.w3.org/TR/NOTE-HTMLplusTIME.

[14] W3C World-Wide Web Consortium. 2005. Synchronized Multimedia Integration Language – SMIL 2.1 Specification, W3C Recommendation. DOI= http://www.w3.org/TR/2005/RECSMIL2-20051213/.

[15] W3C World-Wide Web Consortium. 2008. Synchronized Multimedia Integration Language – SMIL 3.0 Specification, W3C Candidate Recommendation. DOI= http://www.w3.org/TR/SMIL3/.

[16] W3C World-Wide Web Consortium. 2003. Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation. DOI=http://www.w3.org/TR/SVG11/.

[17] W3C World-Wide Web Consortium. 1999. XML Path Language (XPath). W3C Recommendation. DOI= http://www.w3.org/TR/xpath/.