Dynamic Integration of Foreign-Language Parsers into an NCL Player

Jorge P. Dodsworth
jorgepd@telemidia.puc-rio.br
PUC-Rio, Rio de Janeiro, Brazil

Lucas de Macêdo Terças
lucastercas@gmail.com
UFMA, São Luís, Brazil

Alan L. V. Guedes
alan@telemidia.puc-rio.br
PUC-Rio, Rio de Janeiro, Brazil

Guilherme F. Lima
glima@inf.puc-rio.br
PUC-Rio, Rio de Janeiro, Brazil

Carlos de Salles Soares Neto
csalles@deinf.ufma.br
UFMA, São Luís, Brazil

Sérgio Colcher
colcher@inf.puc-rio.br
PUC-Rio, Rio de Janeiro, Brazil

ABSTRACT

We describe how we modified an NCL player to accept as input, in addition to NCL documents, Lua scripts. These Lua scripts evaluate to a table in a particular format, called NCL-ltab, which is a Lua table encoding of the NCL player's internal model. One advantage of our modifications is that they allow the NCL parsing to occur in the Lua script, i.e., outside the formatter but integrated in its execution flow. The same applies for parsers of dialects of NCL or similar languages. Another advantage is that new parsers can be plugged dynamically into the formatter (if they are written in Lua or can be called from Lua). In this paper, we detail the internal model of the NCL player we are using and the NCL-ltab input format. To evaluate our proposal, we present two parser-integration use cases, one for NCL itself, using the DietNCL parsing toolkit, and another for sNCL, an alternative, user-friendlier syntax for NCL.

CCS CONCEPTS

• **Information systems → Multimedia content creation**; • **Applied computing → Hypertext / hypermedia creation**;

KEYWORDS

Document conversion, document parsing, NCL-ltab, DietNCL, sNCL

1 INTRODUCTION

NCL (Nested Context Language) [1, 7] is a XML-based, domain specific language for the description of interactive multimedia presentations. Recent work have pointed out the necessity of reducing the complexity of NCL players [10, 11], in particular, of its reference implementation, and of supporting alternative formats (syntaxes) for NCL [13, 16]. The arguments put forward in these works can be summarized as follows.

Structured decomposition of the NCL player. The process of playing an NCL document is complex. First, the NCL player (or formatter) must parse the XML document and convert it into a data structure suitable for presentation. Then the player must render this data structure while maintaining the QoE (Quality of Experience) defined in the original NCL document. The parsing step alone in PUC-Rio's player [9] takes ≈4000 lines of nontrivial C++ code. To keep complexity under control, NCL players should be structured as loosely coupled layers, each dealing with specific phases of the document presentation, and which communicate only through well-defined, simple interfaces.

Support for alternative NCL formats. NCL has usability issues caused by its XML syntax [16]. These issues motivated the development of alternative formats for NCL [13, 16] and of ways to generate parts of the document automatically via templates or scripts [2, 4, 15]. In each one of these proposals, however, a new parser must be written from scratch, with each specific input syntax converted back to the cumbersome XML format of NCL. To avoid this fallback to XML, NCL players should acknowledge the existence of alternative input formats for NCL and offer a better support for their integration into the system.

With these requirements in mind, in this paper we focus on decoupling the NCL parser from the NCL player presentation engine. That is, instead of making the NCL player perform both the parsing and the presentation tasks, we aim at: (1) defining a minimalist internal model for the NCL player which resembles NCM [14] (the conceptual model of NCL) but which is at the same time simpler and more expressive than NCM; and (2) exposing this internal model to the external world via a Lua-table format, called NCL-ltab, so that foreign-language parsers can use it as a target language. The proposed NCL player architecture is illustrated in Figure 1.

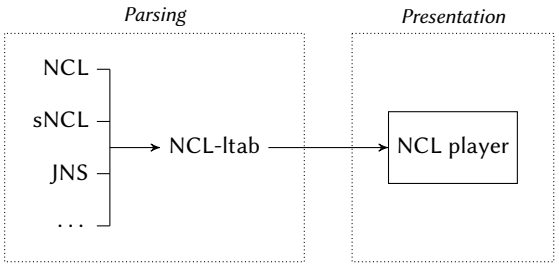

Figure 1: The proposed NCL player architecture.

WebMedia '18, October 16–19, 2018, Salvador-BA, Brazil

ACM ISBN 978-1-4503-5867-5/18/10…$15.00
https://doi.org/10.1145/3243082.3243095

In Figure 1, the NCL-ltab component stands not for a static Lua table but instead for an arbitrary Lua script, i.e., any Lua script [6] that evaluates to a table in the NCL-ltab format. One advantage of this scripting strategy is that it allows us to move the parsing code away from the formatter and embed it into the Lua script, which will ultimately produce the NCL-ltab to be played. This way, parsers for dialects of NCL, template languages, or any other format that can be represented by the internal model of the NCL player can run separately as part of the input Lua script.[1]

A further advantage of the scripting strategy is that, if the parser of the format to be integrated is written in Lua (or can be called from Lua), then the parser can be plugged dynamically into the formatter. That is, before running the input file, the formatter can load all parsers from a given parser directory and then use the appropriate parser to obtain an NCL-ltab from the input file. For instance, if the input file is a ".ncl" the formatter uses the NCL parser script, if it is a ".sncl" the formatter uses the sNCL parser script, and so on.

The rest of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we present the new internal model and architecture of PUC-Rio's NCL player (Ginga 1.0). In Section 4, we detail the NCL-ltab format, which is the syntactical representation of the new internal model. In Section 5, we present two parser-integration use cases. In the first use case, we describe how the NCL parser toolkit DietNCL [8] was integrated into Ginga 1.0 to be used in place of the default C++ NCL parser. In the second use case, we show how the sNCL parser was integrated into Ginga 1.0, allowing it to play sNCL files directly. Finally, in Section **??**, we draw our conclusions and discuss future work.

2 RELATED WORK

We organize our related works in two groups. The first group consists of [10, 11] which focus on reducing the complexity of the NCL player, and the second group consists of [2, 12, 13, 16] which focus on alternative formats for NCL.

In [11], the authors define a minimalist profile for NCL, called NCL Raw Profile. The Raw profile removes from NCL redundant elements, such as <descriptor> and <region>, but keeps the compatibility with the full language. For instance, it maintains the <causalConnector> and related elements, which means that link specification is still complex. Although the Raw profile reduces the number of elements in the language, the final NCL document is still a cumbersome XML file.

In [10], the authors define a NCL-like multimedia language, called Smix. Smix has a precise execution semantics and focuses on supporting the implementation of feature-rich languages, such as NCL and SMIL [3]. As Smix is only inspired by NCL, it does not support its structuring elements, such as <context> and <switch>, nor for its event-state machines—the central concept of the NCL model. This means that parsers for alternative formats of NCL, which tend to take these features for granted, need to perform a considerable amount of work to convert their NCL-like concepts to Smix. In contrast, the NCL-ltab proposal, although also minimalist, maintains the core abstractions of NCL intact with their usual meaning.

In the second group of related work, we have LuaTPL [4] and Luar [2], which evaluate Lua scripts inside NCL documents to generate more elaborated NCL constructs. Similarly, Lua2NCL [12] is a Lua library that generates NCL documents in the XML format. As alternative syntaxes for NCL, we can cite JNS [13] and sNCL [16]. JNS (JSON NCL Script) replaces the XML syntax of NCL by an equivalent JSON (JavaScript Object Notation) syntax. sNCL (Simpler NCL) is a declarative domain specific language with a syntax inspired by Lua language.

The works in the second group can benefit from the NCL-ltab proposal by avoiding to generate a final XML document, which can be cumbersome, especially for parsers of non-XML formats. A more significant benefit follows from the fact that most of these tools are written in Lua, which means that they can be called seamlessly by the NCL-ltab script. Some of these tools, such as the sNCL processor, even use a Lua table to represent the document internally. The conversion to NCL-ltab in this case consists simply by the manipulation of Lua tables. (We detail the sNCL to NCL-ltab conversion in Section 5.)

3 THE INTERNAL MODEL

In this section, we describe the internal model of the new version of PUC-Rio's NCL player, called Ginga 1.0. The internal model holds the media content, structure, and synchronization data of the input NCL document, and is used by the formatter to render and control the document presentation. The NCL-ltab format, which under our proposal is the input language of Ginga 1.0, is a Lua-table encoding of this internal model. Before detailing the internal model, we describe Ginga 1.0's architecture and highlight the central role played by the internal model in this architecture.

3.1 The Architecture of Ginga 1.0

The architecture of Ginga 1.0 (see Figure 2) consists of three main components: *Formatter*, *Parser*, and *Document*. The *Formatter* controls the life-cycle of the presentation. When the *Formatter* receives a *start*, it creates and uses a *Parser* to obtain a *Document* from the given NCL file. The *Document* is what we have been calling the "internal model"; it is an object that holds the state of the presentation defined by the input NCL file. After the *Document* is obtained, the *Parser* is no longer needed, so it is destroyed by the *Formatter*. (Because the *Parser* exists only during this brief moment—between the *start* call and immediately before the presentation is actually started—it is drawn with dashed lines in Figure 2.)

The *Document* contains an object tree consisting of container objects and media objects. This object tree holds the state of the whole presentation, and the *Document* abstraction functions as a top-level interface to query and modify this state.

After the *Document* is created and the *start* call returns to the caller, the *Formatter* becomes reactive and must be driven by the caller (external world). The following actions are performed by the *Formatter* in response to a given event or request from the external world:

[3]We have been using and will continue to use the term "parser" to mean parser plus converter, i.e., the component that not only parses the input character stream into a syntax tree but also converts this syntax tree into the target format (NCL-ltab).

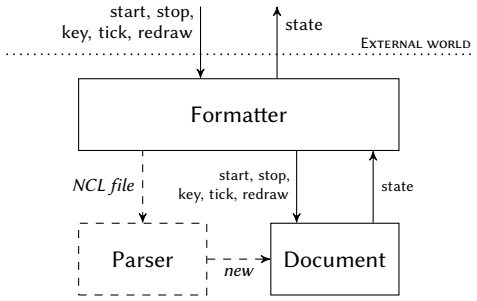

Figure 2: Ginga 1.0's architecture.

- If it receives a *key*, the *Formatter* delivers it to all objects in the *Document*.
- If it receives a *tick*, the *Formatter* advances the time of all objects in the *Document*. (This is the only place in which time advances, and it does so in lockstep: two objects that have been started in the same tick will always be synchronized.)
- If it receives a *redraw* request, the *Formatter* draws on the given screen the video frame corresponding to the current *Document* state.
- If it receives a *state* request, the *Formatter* checks if the *Document* is still running and sends this information back to the caller.
- Finally, if it receives a *stop*, the *Formatter*, stops the whole presentation and destroys the *Document*.

3.2 The *Document* Abstraction

The *Document* maintains the object tree of the NCL presentation. It has methods to query the tree contents and to evaluate actions over it, but not to change the tree structure. Figure 3 depicts the class hierarchy of the objects that can occur in a *Document*.

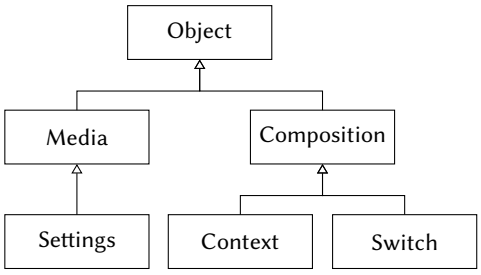

Figure 3: The class hierarchy of *Document* objects.

The classes *Object* and *Composition* are abstract. An *Object* has an id, a pointer to the parent object, a playback time, and a set of associated state-machines, called NCL events. A *Composition* is an object container; in addition to what it inherits from *Object*, the *Composition* maintains a list of child objects. The concrete classes *Media*, *Settings*, *Context*, and *Switch* stand for the corresponding elements of the NCL language.

Media is a presentation atom (the <media> element of NCL). Every *Media* has an underlying player, which renders the *Media* content during a *redraw*. The content of a *Media* is defined by its *uri* property. (A property is a type of NCL event that behave as a variable associated to the object. We defer the discussion of NCL events and related concepts to Section 3.3.)

Settings is a special *Media* object that maintains properties associated to the *Document* as a whole. There is exactly one settings object per *Document*.

Context stands for the <context> element of NCL. It is a container object that besides child objects maintains list of ports and a list of links. A port is simply an NCL event of some child object of the *Context* which is made visible to the *Context*'s parent. And a link is a synchrony relation between NCL events of the *Context*'s children.

Switch stands for the <switch> element of NCL. It is a container object that acts as a proxy to exactly one of its children. The *Switch* keeps a list of rules which it uses to determine which of its children it will impersonate.

Objects of the concrete types listed above make up the *Document* object tree. The root of this tree is always an instance of *Context* (the <body> element of NCL), and *Media* or *Settings* objects can only occur as leafs of the tree.

3.3 Events, Predicates, Rules, Actions, and Links

We now describe the concepts that determine the behavior of the objects in a *Document*. The first of these concepts is the *Event* (what we have been calling "NCL event").

3.3.1 Event. We begin by saying what an *Event* is not: an *Event* is *not* a short-lived, lightweight notification. In NCL terminology, an *Event* is a state machine associated with exactly one object (its container object) and which exists throughout the lifetime of this object. Figure 4 depicts the structure of an *Event*.

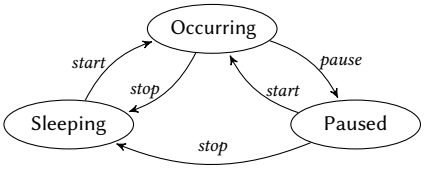

Figure 4: The structure of an *Event*.

At any time an *Event* is in one of three possible states: occurring (O), paused (P), or sleeping (S). Five transitions between these states are possible: $S \rightarrow O$ (*start*), $O \rightarrow S$ (*stop*), $O \rightarrow P$ (*pause*), $P \rightarrow O$ (*start*), and $P \rightarrow S$ (*stop*). The term *abort* is another name for transitions $O \rightarrow S$ and $P \rightarrow S$, and the term *resume* is another name for transition $P \rightarrow O$.

The exact meaning of *Event* states and transitions depends on the type of the *Event* and the type of its container object. There are three types of events:

Presentation event stands for the presentation of some part of the container object's content (or of its children if the object is a composition). The content associated with a presentation event is either being exhibited (the event is occurring), not being exhibited (the event is sleeping), or being exhibited

but frozen in time (the event is paused). Every *Object* has at least one presentation event, called lambda, which stands for the presentation of the object as a whole. *Media* objects can have other presentation events, which represent time segments of the *Media* content.

Attribution event represents a variable (or property) associated with the container object. The property is either being attributed (the event is occurring) or not being attributed (the event is sleeping). (In an attribution event, the state paused has no meaning.)

Selection event represents the selection of the container object via a given keyboard key or direct selection (mouse or focus navigation). The object is either being selected with the given key (the event is occurring) or not begin selected with the given key (the event is not occurring). (In a selection event, the paused state has no meaning.)

There are implicit dependencies between the various events of an object. These dependencies arise from the usual interpretation of the texts that define the semantics of NCL [1, 7]. For instance, a selection event of an object can only transition to state occurring if its lambda event is in state occurring or paused. In other words, the object can only be selected if its being presented or if its presentation is paused. We will not detail further dependencies, as that would be outside of the scope of this paper, but we remark that a correct understanding of event dependencies is essential for reasoning about the behavior of NCL documents.

Finally, as in NCL, every *Event* has an id which must be unique regarding the events of the same type in its container object. Ginga 1.0 uses the following naming convention for identifying events within a *Document* (this convention is also used in the NCL-ltab format):

- "$x@y$" stands for the presentation event y of object x.
- "$x.y$" stands for the attribution event (property) y of object x.
- "x<y>" stands for the selection event with key y of object x.

Now that we have defined what an *Event* is, albeit superficially, we proceed to describe the other concepts involved in determining the behavior of a *Document*.

3.3.2 Predicate. A *Predicate* is a boolean test involving properties and values. For instance, something like

$$(x.p \geq 5 \text{ and } x.q < y.p) \text{ or } \text{“red”} \neq y.q$$

which can be read as the statement: the value of property p of x is greater than or equal to 5 and the value of property q of x is less than the value of property p of y, or the value of property q of y is equal to the string "red". The actual syntax of predicates is described in detail in Section 4.

3.3.3 Rule. A *Rule* is a pair $\langle p, o \rangle$ where p is a *Predicate* and o is an *Object*. The *Switch* object keeps a list of such pairs to determine which of its children it will impersonate after the *Switch* is started. More precisely, when its lambda event transitions to state occurring, the *Switch* traverses its rule list, evaluating the predicate of each predicate-object pair, one at a time. If one of these predicates evaluates to true, then the traversal ends and the *Switch* starts the associated object which becomes the selected object. From this point on, the *Switch* acts as a proxy for the selected object—any actions performed over the *Switch* are transferred to this object.

3.3.4 Action. An *Action* is a triple $\langle e, t, p \rangle$, where e is an *Event* of some object, t is some of the five possible transitions of the event state machine (see Figure 4), and p is an arbitrary (possibly empty) *Predicate*. There are two uses for an action: either (i) it denotes a transition which we are waiting or (ii) it denotes a transition that we wish to perform. In both cases, predicate p acts as a condition that must be fulfilled for the action to take place.

3.3.5 Link. A *Link* is a pair $\ell = \langle A_1, A_2 \rangle$ where both A_1 and A_2 are nonempty lists of actions. We call A_1 the link head and we call A_2 the link tail. The head contains the actions whose execution the link is waiting for, and the tail contains the actions that the link will execute when triggered, i.e., when any of the actions in its head occurs.

Links exist only inside a *Context*—each *Context* maintains a (possibly empty) list of links which are reevaluated whenever an action is executed in the scope defined by the *Context*. More specifically, let $a = \langle e, t, p \rangle$ be an action such that e's container object is a child of context C. Then, whenever action a is executed (i.e., transition t of event e occurs and predicate p is true) each link of C is evaluated, one after another. By saying that a link is *evaluated* we mean that each action $a' = \langle e', t', p' \rangle$ in its head is compared with the action a which has just been executed and, if a' matches a (i.e., if $e' = e$ and $t = t'$ and p' is true), then each action in the link tail is executed, one after another, triggering nested link evaluations. A simplified version of Ginga 1.0's algorithm for action execution is depicted Figure 5. (This is essentially the stack-based algorithm for action execution in Smix [10].)

```
procedure exec(a)
    let a = ⟨e, t, p⟩
    let C be the parent context of e's container object
    if e can transition via t and eval(p) == true then
        transition(e, t)
        for each link ℓ = ⟨A1, A2⟩ in C do
            for each action a' = ⟨e', t', p'⟩ in A1 do
                if e' == e and t' == t and eval(p') == true then
                    for each action a'' in A2 do
                        exec(a'')
                    end
                    break
end
```

Figure 5: The action execution algorithm of Ginga 1.0.

3.4 Proposed Internal Model vs. NCL/NCM

We conclude the description of the internal model by remarking that many of the NCL elements (concepts of NCM) have no counterpart in the internal model. The *Parser* fills these gaps by converting the missing NCL elements into equivalent combinations of concepts of the internal model. For instance, The NCL elements <region>, <descriptor>, and <transition> are converted into *Media* properties, and the elements <rule> and <assessmentStatement> (and related elements) become *Predicate* objects. Similarly, the NCL elements <causalConnector> and <link> (and related elements) are converted into *Link* objects.

4 THE NCL-LTAB FORMAT

An NCL-ltab is a Lua script that when executed evaluates to a table in a specific format. In Lua, a table is an associative array containing key-value pairs of arbitrary values. Tables are created using table constructors which are expressions of the form "{...}". A valid NCL-ltab table is any Lua table which can be obtained by a constructor specified by the grammar below.

⟨NCL⟩ ::= ⟨CONTEXT⟩

⟨CONTEXT⟩ ::= **{ 'context',** ⟨ID⟩**,** ⟨PORTS⟩?**,** ⟨CHILDREN⟩?**,** ⟨LINKS⟩? **}**

⟨SWITCH⟩ ::= **{ 'switch',** ⟨ID⟩**,** ⟨CHILDREN⟩?**,** ⟨RULES⟩? **}**

⟨MEDIA⟩ ::= **{ 'media',** ⟨ID⟩**,** ⟨PROPS⟩?**,** ⟨AREAS⟩? **}**

⟨CHILDREN⟩ ::= **{** (⟨CONTEXT⟩**,** | ⟨SWITCH⟩**,** | ⟨MEDIA⟩**,**)∗ **}**

⟨PORTS⟩ ::= **{** (⟨EVT-ID⟩**,**)∗ **}**

⟨LINKS⟩ ::= **{** (⟨LINK⟩**,**)∗ **}**

⟨LINK⟩ ::= **{** ⟨ACTIONS⟩**,** ⟨ACTIONS⟩ **}**

⟨ACTIONS⟩ ::= **{** (⟨ACTION⟩**,**)+ **}**

⟨ACTION⟩ ::= **{** ⟨TRANS⟩**,** ⟨EVT-ID⟩**,** ⟨PRED⟩?**,** ⟨PARAMS⟩? **}**

⟨TRANS⟩ ::= **'start'** | **'stop'** | **'pause'** | **'resume'** | **'abort'**

⟨EVT-ID⟩ ::= **'**⟨ID⟩**@lambda'** | **'**⟨ID⟩**@**⟨AREA-ID⟩**'**
| **'**⟨ID⟩**.**⟨PROP-NAME⟩**'** | **'**⟨ID⟩**<**⟨KEY-NAME⟩**>'**

⟨PRED⟩ ::= **{ ('and'** | **'or'),** ⟨PRED⟩**,** ⟨PRED⟩ **}**
| **{ 'not',** ⟨PRED⟩ **}**
| **{** ⟨EXPR⟩**,** ⟨COMP⟩**,** ⟨EXPR⟩ **}** | **true** | **false**

⟨EXPR⟩ ::= **'$**⟨EVT-ID⟩**'** | ⟨*any Lua string or number*⟩

⟨COMP⟩ ::= **'=='** | **'~='** | **'<'** | **'<='** | **'>'** | **'>='**

⟨PARAMS⟩ ::= ⟨*any Lua table*⟩

⟨RULES⟩ ::= **{** (⟨RULE⟩**,**)∗ **}**

⟨RULE⟩ ::= **{** ⟨ID⟩**,** ⟨PRED⟩? **}**

⟨PROPS⟩ ::= ⟨*any Lua table*⟩

⟨AREAS⟩ ::= **{** (⟨AREA⟩**,**)∗ **}**

⟨AREA⟩ ::= **{** ⟨ID⟩**,** ⟨TIME-SPEC⟩?**,** ⟨TIME-SPEC⟩? **}**
| **{** ⟨ID⟩**,** ⟨NAME⟩ **}**

In the above grammar, nonterminals are shown between angle brackets and terminals are typeset in bold font. The symbol "::=" can be read as "is composed of", parentheses denote grouping, the vertical bar denotes choice, and the symbols "?", "∗", and "+" denote zero-or-one, zero-or-more, and one-or-more repetitions of the preceding nonterminal or group. In addition:

- The nonterminal ⟨ID⟩ is any Lua string whose contents is a valid XML id. The ⟨ID⟩ identifies or refers to single object (context, switch, or media) in the document.
- In the definition of ⟨EVT-ID⟩, the reserved suffix **@lambda** stands for the lambda event and ⟨AREA-ID⟩ is the id of some user-defined presentation event (which must not be called "lambda"). The nonterminals ⟨PROP-NAME⟩ and ⟨KEY-NAME⟩ are Lua strings whose contents are XML names. The nonterminal ⟨PROP-NAME⟩ stands for the name of some object property (e.g., "transparency", "volume", etc.) and ⟨KEY-NAME⟩ stands for the name of some input key ("ENTER", "CURSOR_UP", "RED", etc.).
- In the definition of ⟨AREA⟩, the first form denotes a presentation event with begin and end times and the second form denotes a presentation event with a label. The nonterminal ⟨TIME-SPEC⟩ is a Lua string representing a time value (e.g., "5s" or "02:20:15.258"). And ⟨NAME⟩ is any XML name.
- The nonterminal ⟨PARAMS⟩ is any Lua table containing parameters which affect the execution of the action (e.g., delay='5s', value=3.14, etc.) and ⟨PROPS⟩ is any Lua table containing the initial values for object properties (e.g., transparency='50%', width='128px', etc.).

When the formatter receives a Lua script, it doesn't call the NCL-ltab parser immediately; it first calls the Lua interpreter to execute the script and then calls the NCL-ltab parser to process the resulting table. There are some advantages to receiving and evaluating Lua script instead of receiving the target Lua table directly.

The first advantage is that the script can act as a template engine that creates the table dynamically. For instance, instead of writing a table constructor specifying n similar media objects, one can write a loop that inserts each of these objects in the table. This way the repetition is delegated to the script and what would be an extensive static document can now be written in a few lines of code. This technique is illustrated in Figure 6.

```
local children = {}
local links = {}
for i=1,1000 do
   children[i] = {
      'media', 'm'..i, {src='media/video'..i..'.ogv'},
   }
   links[i] = {
      {{'stop', 'm'..i..'@lambda', true}},
      {{'start', 'm'..((i%1000) + 1)..'@lambda'}},
   }
end
local ncl = {
   'context', 'ncl', {'m1@lambda'}, children, links,
}
return ncl
```

Figure 6: A Lua script that evaluates to an NCL-ltab table.

The Lua script of Figure 6 when evaluated generates a valid NCL-ltab table containing 1000 media objects and 1000 links. When this NCL-ltab is played, it first starts media "m1"; then, after "m1" finishes, it starts media "m2", and so on. Writing this in NCL would be cumbersome, but writing in NCL-ltab using the Lua interpreter as a template engine is trivial.

Another advantage of having a Lua script as input format is that the script may act as a converter for other input formats. That is, the script can read and process a document in a completely different format and generate a corresponding NCL-ltab table. This is precisely the approach we use to integrate foreign-language parsers into PUC-Rio's Ginga 1.0.

5 PARSER-INTEGRATION USE CASES

We now discuss the integration of two new parsers into Ginga 1.0. Both parsers use NCL-ltab as target format and are run on demand by the formatter depending on the type of input file. The first parser (discussed in Section 5.1) is a novel parser for NCL itself that uses the DietNCL toolkit to convert the NCL document into an equivalent NCL-ltab. The second parser (discussed in Section 5.2) is a parser for sNCL, an NCL-like language designed for greater usability.

5.1 DietNCL

The DietNCL toolkit [8] is a Lua library that transforms an input NCL document by pushing it through a sequence of filters (or *filter pipeline*). Each filter in the pipeline is self-contained and performs a specific transformation. To generate an NCL-ltab using DietNCL, we implemented a new filter that runs in the end of the pipeline and transforms the intermediate representation used by DietNCL into an NCL-ltab. The general structure of the filter pipeline we are using is depicted in Figure 7.

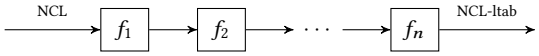

Figure 7: Filter pipeline for converting NCL to NCL-ltab.

The first filter in the pipeline parses the input NCL document (XML file or string) into an intermediate Lua table containing the document's element tree. The subsequent filters in the pipeline operate over this table, and perform the following transformations:

(1) Remove most of the elements in the <head> section of the document, such as <region>, <descriptor>, <transition>, etc., by transforming them into media object properties.
(2) Simplify the structure of connectors and links. That is, make sure that each <causalConnector> element in the document is referenced by exactly one <link>, and that <simpleCondition> and <simpleAction> elements are referenced by exactly one <bind> in the corresponding <link>.
(3) Put the resulting intermediate table into the NCL-ltab format. That is, process the document recursively, putting the subtables that represent elements in document's body (and head, in the case of link-connector pairs) into the format defined by NCL-ltab's grammar.

To illustrate the NCL to NCL-ltab conversion process, consider the NCL document depicted in Figure 8. This document contains two video objects that are started simultaneously when the document is started. When the first video reaches 1s, the second video is paused and its transparency is set to 50%, but only if the volume of the first video is greater than the volume of the second video; otherwise, nothing happens.

When the document of Figure 8 starts, media *m1* (ln. 23) is started; context *ctx1* (ln. 27) is also started and, consequently, media *m2* (ln. 32) starts playing as well. This happens due to ports *start-m1* (ln. 21), *start-ctx1* (ln. 22) and *m2-lambda* (ln. 28).

After one second, area *a1* (ln. 24) of media *m1* is started and the link (ln. 37–46) is triggered. However, before executing the link actions, the link assessment (ln. 6–9) is evaluated. If it evaluates to true, then the link actions are executed, i.e., property "transparency" of media *m2* is set to 50% (ln. 14, 41–44), and then media *m2* is paused (ln. 15, 45). Otherwise, nothing happens.

```
1  <ncl>
2  <head>
3   <connectorBase>
4    <causalConnector id="onBeginTestSet">
5     <compoundCondition operator="and">
6      <assessmentStatement comparator="gt">
7       <attributeAssessment role="left"/>
8       <attributeAssessment role="right"/>
9      </assessmentStatement>
10     <simpleCondition role="onBegin"/>
11    </compoundCondition>
12    <connectorParam name="var"/>
13    <compoundAction operator="par">
14     <simpleAction role="set" value="$var"/>
15     <simpleAction role="pause"/>
16    </compoundAction>
17   </causalConnector>
18  </connectorBase>
19 </head>
20 <body>
21  <port id="start-m1" component="m1"/>
22  <port id="start-ctx1" component="ctx1"/>
23  <media id="m1" src="media/video1.ogv">
24   <area id="a1" begin="1s"/>
25   <property name="volume" value="50%"/>
26  </media>
27  <context id="ctx1">
28   <port id="m2-lambda" component="m2"/>
29   <port id="m2-volume" component="m2" interface="volume"/>
30   <port id="m2-transparency" component="m2"
31         interface="transparency"/>
32   <media id="m2" src="media/video2.ogv">
33    <property name="volume" value="20%"/>
34    <property name="transparency" value="0%"/>
35   </media>
36  </context>
37  <link xconnector="onBeginTestSet">
38   <bind role="onBegin" component="m1" interface="a1"/>
39   <bind role="left" component="m1" interface="volume"/>
40   <bind role="right" component="ctx1" interface="m2-volume"/>
41   <bind role="set" component="ctx1"
42         interface="m2-transparency">
43    <bindParam name="var" value="50%"/>
44   </bind>
45   <bind role="pause" component="ctx1" interface="m2-lambda"/>
46  </link>
47 </body>
48 </ncl>
```

Figure 8: A simple NCL document.

To convert the document of Figure 8 into an NCL-ltab we proceed recursively. The transformations of <port>, <context> and <media> elements are direct. The tricky part is the transformation of NCL links into NCL-ltab links. This is the case because an NCL link is only meaningful when considered together with its corresponding connector, which is defined separately in the head part of the document.

The NCL-ltab script corresponding to the NCL document of Figure 8 is depicted in Figure 9. The link in lines 37–46 of Figure 8 checks if area *a1* of media *m1* was started and if the volume of *m1* is greater than the volume *m2*. In Figure 9, this link becomes the sub-table in lines 19–28, which specifies a link that waits for an "onBegin" transition with a predicate that compares the values of the volume property of *m1* and *m2*. When triggered, this link sets the transparency of *m2* to 50% and pauses *m2*, (ln. 25–26 of Figure 9).

We conclude by highlighting the syntactical economy of the NCL-ltab of Figure 9 when compared to the NCL document of Figure 8. For instance, in the body of the NCL document, to reference a property of media *m2*, which is a child of context *ctx1*, one has to insert a new <port> element and to come up with a new id, which must be unique in the document. In contrast, in NCL-ltab this is much simpler. All one has to do is to reference the id of the property directly, e.g., "m2.transparency", which is always unique within the document. The other feature that contributes to this syntactical economy is the absence of connectors in NCL-ltab. The link specification is compact and self-contained.

```
1  local ncl = {
2    'context', 'ncl',
3    {-- body ports
4      'm1@lambda',
5      'ctx1@lambda'
6    },
7    {-- body children
8      {'media', 'm1',
9       {src='media/video1.ogv',
10       volume='50%'}, {a1={'1s'}}
11     },
12     {'context', 'ctx1',
13       {'m2@lambda'},
14       {{'media', 'm2',
15          {src='media/video2.ogv',
16           volume='20%',
17           transparency='0%'}}
18       }
19       -- no links in this context
20     }
21   },
22   {-- body links
23     {
24       {-- conditions
25         {'start', 'm1@a1',
26           {'$m1.volume', '>', '$m2.volume'}}
27       },
28       {-- actions
29         {'set', 'm2.transparency', {value='50%'}},
30         {'pause', 'm2'}
31       }
32     }
33   }
34 }
35 return ncl
```

Figure 9: The NCL-ltab corresponding to Figure 8.

5.2 sNCL

NCL is a XML-based language. Although XML is supposed to be user friendly, when a XML document grows, the excessive, ornamented notation of XML becomes cumbersome and makes the document hard to follow [16]. sNCL is a proposal to reduce the problem of excessive notation in NCL. It improves NCL usability by replacing its XML syntax by a Lua-like syntax, which was developed based on an usability evaluation of the NCL language. A detailed description of this evaluation and of sNCL can be found in [16].

The sNCL compiler is written in Lua. When given a sNCL document, the compiler first parses it generating an intermediate Lua table which is then converted into a corresponding NCL document. The goal of sNCL is different from that of NCL-ltab; sNCL aims to provide a better authoring experience, and as a consequence it is not concerned with the complexity of the resulting NCL document. sNCL uses the LPeg library [5] to parse the input sNCL document and generate the intermediate table, whose format is similar to that of the one adopted by NCL-ltab.

Even though the Lua table maintained by sNCL parser and the table format used by NCL-ltab resemble each other, they are not the same. The fact that sNCL has to conform to the NCL EDTV standard [1, 7], i.e., must also be able to generate XML-based NCL documents, implies that it cannot generate a table that conforms to the NCL-ltab format straight away.

The sNCL compiler also has to generate the extra elements which are necessary to NCL, such as connectors and descriptors, but which have no counterpart in neither sNCL nor NCL-ltab. The sNCL compiler does this by inferring them from the other elements, for example, the connector elements can be derived from the links. This feature is not used when generating an NCL-ltab, since the NCL-ltab format has none of those elements.

One of the main differences between sNCL and NCL-ltab is illustrated by the sNCL document depicted in Figure 10. In this document, when media *startButton* (ln. 10) is selected, the link at the end of the listing (ln. 11–13) starts media *photo1*, which is a child of context *slideshow1* (ln. 4–8), which itself is inside context *album* (ln. 2–9). The link does this via multiple indirections using the port elements at lines 3 and 1. In sNCL and NCL, these indirections are needed because a link can only refer to elements declared in its scope. In NCL-ltab, however, such indirections are not needed as a link can refer to any event in the document.

```
1  port pBody startButton
2  context album
3    port pAlbumS1 slideshow1.portSlideshow1
4    context slideshow1
5      port portSlideshow1 m1
6      media photo1 end
7    end
8    context slideshow2 ... end
9  end
10 media startButton end
11 onBegin startButton do
12   start album.portAlbumS1 end
13 end
```

Figure 10: A simple sNCL document.

Figure 11 depicts the NCL-ltab corresponding to the sNCL document listed in Figure 10. As can be seen, the link in line 14–21 of Figure 11 references the lambda event of media *photo1* (ln. 9) directly by its id, no ports are needed. (In NCL-ltab ports cannot be referenced as they do not have ids; they exist only to start the mapped events in the parent context.).

In order to solve these references to port ids, in the translation of sNCL to NCL-ltab, the compiler resolves all elements that refer to ports recursively, until it finds an element that is not a port, and then substitute the first port by the id of the element found. This is the most complex part of the conversion process, since the Lua tables of NCL-ltab are similar to the ones that the sNCL compiler keeps internally. In overall, the sNCL to NCL-ltab conversion is pretty straightforward and takes only 167 lines of code.

```
1  return {
2     'context', 'ncl',
3     {-- body ports
4       'startButton@lambda'
5     },
6     {-- body children
7        {'context', 'album', {},
8           {'context', 'slideshow1', {},
9              {'media', 'photo1'}
10          }
11       }
12    },
13    {-- body links
14      {
15        {-- conditions
16          {'start', 'startButton@lambda'}
17        },
18        {-- actions
19          {'start', 'photo1@lambda'}
20        }
21      }
22    }
23 }
```

Figure 11: The NCL-ltab corresponding to Figure 10.

6 FINAL REMARKS

This paper presented a minimalist internal model for an NCL player and a corresponding Lua-table encoding for this model, called NCL-ltab. The NCL-ltab simplifies the implementation of the NCL player by allowing the migration of the parsing code from the formatter to the input document, which is essentially a Lua script that produces the NCL-ltab to be played.

Besides making the code of the formatter smaller and—by allowing it to concentrate on one thing (the presentation)—less error prone, our approach enables the dynamic integration of foreign-language parsers more easily and seamlessly into the formatter's execution flow, as we have shown in our use case examples for NCL itself and for sNCL.

As a future work, we aim at improving our proposal in two ways. First, we intend to extend the NCL-ltab format to also describe the state of the presentation. The NCL-ltab could store not only the representation of the source NCL structural components but also the state of this elements during presentation time. This capability could be used, for instance, to recover the presentation state of a document that was previously finalized.

Second, we intend to make the internal model even more customizable through the NCL-ltab format, i.e., by allowing arbitrary Lua code to be injected in to the model and evaluated at runtime by the formatter. Such injected code might be used, for example, to define new actions or types of events beyond those currently supported by NCL (e.g., "start", "onBegin", etc.). For instance, there are some efforts that propose a "recognition" event and the "onRecognize" condition to trigger links based on the recognition of user behavior or semantic evaluation of objects in a video or image content. Moreover, in scenarios outside the multimedia presentation context, such as IoT or CEP (Complex Event Processing), this extensibility feature might be useful to define new actions and events that are specific to these domains.

ACKNOWLEDGMENTS

This work was conducted during a scholarship supported by the Brazilian federal agency CNPq.

REFERENCES

[1] ABNT 15606-2. *Digital Terrestrial TV — Data Coding and Transmission Specification for Digital Broadcasting — Part 2: Ginga-NCL for Fixed and Mobile Receivers: XML Application Language for Application Coding.* ABNT, São Paulo, 2007.

[2] Bezerra, D. H. D., Sousa, D. M. T., Filho, G. L. d. S., Burlamaqui, A. M. F., and Silva, I. R. M. Luar: A language for agile development of NCL templates and documents. In *Proc. 18th Brazilian Symp. Multimedia and the Web* (2012), ACM.

[3] Bulterman, D., and Rutledge, L. W. *SMIL 3.0: Flexible Multimedia for Web, Mobile Devices and Daisy Talking Books*, 2nd ed. Springer, 2009.

[4] de Albuquerque Azevedo, R. G. LuaTPL: A simple lua-based template engine. https://github.com/robertogerson/luatpl, 2018. Accessed September 18, 2018.

[5] Ierusalimschy, R. A text pattern-matching tool based on parsing expression grammars. *Softw. Pract. Exper. 39*, 3 (mar 2009).

[6] Ierusalimschy, R. *Programming in Lua*, 4th ed. Lua.org, 2016.

[7] ITU-T Recommendation H.761. *Nested Context Language (NCL) and Ginga-NCL.* ITU-T, Geneva, November 2014.

[8] Lab. TeleMídia. DietNCL: A tool to remove the syntactic sugar from NCL documents. https://github.com/TeleMidia/DietNCL, 2018. Accessed September 18, 2018.

[9] Lab. TeleMídia. Ginga: The iTV middleware. https://github.com/TeleMidia/Ginga, 2018. Accessed September 18, 2018.

[10] Lima, G. F., de Albuquerque Azevedo, R. G., Colcher, S., and Haeusler, E. H. Converting NCL documents to Smix and fixing their semantics and interpretation in the process. In *Proc. 23rd ACM Brazilian Symp. Multimedia and the Web* (2017), ACM.

[11] Lima, G. F., Soares, L. F. G., Azevedo, R. G. d. A., and Moreno, M. F. Reducing the complexity of NCL player implementations. In *Proc. 19th Brazilian Symp. Multimedia and the Web* (2013), ACM.

[12] Moraes, D. d. S., Damasceno, A. L. d. B., Busson, A. J. G., and Soares Neto, C. d. S. Lua2NCL: Framework for Textual Authoring of NCL Applications using Lua, year=2016,. In *Proc. 22nd Brazilian Symp. Multimedia and the Web*, ACM.

[13] Silva, E. C. O., Santos, J. A. F. d., and Muchaluat-Saade, D. C. JNS: An alternative authoring language for specifying NCL multimedia documents. In *2013 IEEE Int. Conf. Multimedia and Expo Workshops* (2013), IEEE.

[14] Soares, L. F. G., and Rodrigues, R. F. Nested Context Model 3.0 part 1: NCM core. Monographs in computer science, Informatics Department, PUC-Rio, Rio de Janeiro, Brazil, 2005.

[15] Soares Neto, C. d. S., Soares, L. F. G., and de Souza, C. S. TAL—Template Authoring Language. *J. Brazilian Computer Society 18*, 3 (Sep 2012).

[16] Terças, L. d. M., Moraes, D. d. S., Ribeiro, D. d. S., Neto, M. C. M., and Neto, C. d. S. S. Usability-based language for authoring NCL documents. In *Proc. 23rd Brazillian Symp. Multimedia and the Web* (2017), ACM.