Reducing the Complexity of NCL Player Implementations

Guilherme A. F. Lima Luiz Fernando G. Soares Roberto G. A. Azevedo Marcio F. Moreno
Department of Informatics
PUC-Rio, Rio de Janeiro, Brazil
{glima,lfgs,razevedo,mfmoreno}@inf.puc-rio.br

ABSTRACT

In this paper, we present an approach for reducing the complexity of NCL player implementations. This approach consists, basically, in introducing in the player's architecture an initial conversion step that removes all syntactic sugar and reuse features from the source language. The output of this step, a redundancy-free version of the original input, is then fed to the player that interprets it and creates a corresponding multimedia presentation. In particular, we propose the use of the NCL Raw profile as this intermediate language. The Raw profile is an (almost) redundancy-free profile that is compatible with the NCL 3.0 EDTV (Enhanced Digital TV) profile, a property that guarantees a seamless integration with current EDTV profile implementations. The main targets of the proposed approach are NCL players running on HTML browsers. We discuss how the solutions presented by NCL4Web, WebNCL, and Ginga Plug-in can be tuned to overcome some problems pointed their authors. The same problems arise in similar contexts for other declarative languages, e.g., SMIL, and the solutions presented here can also be extended to those systems.

Categories and Subject Descriptors: I.7.2 [Document Preparation]: Hypertext/hypermedia, Language and systems, Standards

General Terms: Algorithms, Languages, Standardization

Keywords: NCL Raw profile; NCL players; web-based players

1. INTRODUCTION

Authoring hypermedia applications using imperative languages like Java, C, and C++, or scripting languages like JavaScript is usually more complex and error-prone than using declarative, domain-specific languages such as NCL [2, 8] or SMIL [4]. Declarative descriptions are generally easier to be devised and understood than imperative ones, which usually require programming expertise. The declarative approach also has advantages from an engineering point of view: It makes easier to maintain and reuse content, as opposed to the purely imperative approach. So is the case of NCL (Nested Context Language).

NCL is a declarative language for the specification of interactive multimedia presentations. NCL has a strict separation between application content and application structure. The language does not define any media itself; instead, it defines the glue that relates media objects in time and space, during multimedia presentations. The language flexibility, its reuse facility, multi-device support, presentation adaptability, API for building and modifying applications on-the-fly, and mainly, its ability for easily defining the spatial and temporal synchronization of media objects (including viewer interactions) make it an adequate solution for different multimedia application domains. For particular procedural needs, e.g., when complex, dynamic content generation is needed, NCL provides support for the use of imperative scripts written in Lua [7], a fast and lightweight scripting language.

NCL has been primarily used for developing interactive multimedia applications in digital TV (DTV) domain. Indeed, NCL is the ITU-T declarative language recommendation for IPTV services [8] and the declarative language of the Brazilian ISDB-T terrestrial DTV standard [2]. Since other DTV standards use HTML-based languages as their declarative support, in 2011, the ITU-T started a liaison process to harmonize the NCL and HTML-based approaches. On the one hand, NCL is a glue language that does not restrict nor prescribe media-object type, so it can embed HTML applications naturally as one of its media objects. (This feature is natively supported in both the ISDB-T standard and the ITU-T H.761 recommendation.) On the other hand, embedding NCL into an HTML-based DTV middleware is also possible and has been done, e.g., in the HbbTV [12] proprietary platforms. However, a platform independent solution is still missing.

NCL has also been available for publishing multimedia applications on the Web, but the lack of widely deployed tools has limited its impact. Trying to fill this gap, our group developed a browser plug-in, the Ginga Plug-in [13], for the Firefox and Chrome Web browsers, which contains the player of the reference implementation of Ginga-NCL. More recently, with the advance of HTML5 [3] and, in particular, of its graphic, audio, and video support, it became possible to develop standard-based applications that run natively in Web browsers. NCL4Web [14] and WebNCL [11] follow this approach.

However, the aforementioned solutions for embedding NCL into HTML pages (Ginga Plug-in, NCL4Web, and WebNCL) have persisted in the same shortcoming: They try to embed a player for the EDTV (Enhanced DTV) profile of NCL. The NCL EDTV profile has redundant constructs, which can be removed, giving rise to a simplified profile, called the NCL Raw profile. A player for the NCL Raw profile is easier to be designed and implemented than a player for the NCL EDTV profile. Moreover, such a player might implement a well-defined API for media objects, including media

objects with HTML code and media objects with embedded NCL code (i.e., nested NCL applications), which allows for embedding more than one NCL into an HTML page, as NCL4Web and WebNCL do, and also makes possible to relate both the HTML host application and the embedded NCL application—similar to the solution adopted by Ginga Plug-in.

In this paper, we discuss how the solutions presented by NCL4Web, WebNCL, and Ginga Plug-in can be tuned to overcome some problems pointed their authors. The same problems arise in similar contexts for other declarative languages, e.g., SMIL, and the solutions presented here can also be extended to those systems.

The rest of the paper is organized as follows. Section 2 discusses how a foreign language player can be embedded into an HTML browser, since this is the target engine we discuss in this paper. Section 3 presents some related work. Section 4 introduces the NCL Raw profile. Section 5 discusses the conversion of EDTV profile constructs into equivalent Raw profile constructs, and presents the approach we propose for embedding NCL players into HTML browsers. Finally, Section 6 concludes the paper.

2. RUNNING FOREIGN LANGUAGE APPLICATIONS IN HTML BROWSERS

There are at least three usual approaches to run a foreign language application in an HTML browser.

1. Use a browser plug-in containing the foreign language player.
2. Implement the foreign language player in JavaScript.
3. Use XSLT[1] to translate the foreign language application into an equivalent application in HTML+CSS+JavaScript.

The main disadvantage of approach (1) is that it is platform dependent. Nevertheless, plug-ins usually have a better performance and this can make a difference, especially when the presentation of several media objects must be synchronized in time and space. Ginga Plug-in [13] and SMIL State [9] together with the Ambulant SMIL Player[2] are examples of this approach, as discussed in Section 3.

In approach (2), there are, basically, two possibilities: (i) use the HTML browser media players to exhibit the media content of the foreign language application; or (ii) implement custom media players in JavaScript using its Canvas API. At least with the current state of HTML5 canvas API, the first variant seems to be more feasible. In this variant, the foreign language application entities are converted into HTML entities and a JavaScript library controls the multimedia presentation. The conversion to HTML can be done in its entirety before the starting of the application, or it can be done incrementally, during the presentation of the application. This is the approach followed by SmilingWeb [6] and WebNCL [11].

In approach (3), the foreign language application is converted into an equivalent HTML+CSS+JavaScript application. Since most HTML browsers support XSLT, this solution may be considered platform independent, like approach (2). Furthermore, in this variant we can explore the possibility of having the conversion done in server side, and not only in the client side. Indeed, if the conversion is done in the server side there is no need to be attained to an XSLT converter to be platform independent; one could use any language, which then is converted to HTML+JavaScript+CSS. NCL4Web [14] follows this approach, realizing the conversion at the client side, i.e., by the HTML browser via XSLT.

3. RELATED WORK

SMIL State [9] combines the SMIL [4] language with an external data model, specified in the XForms[3] declarative language, allowing for this data model to be shared with other components and, effectively, enabling its use as an API between components of an application. In particular, the data model is proposed as the way SMIL can communicate with its plug-ins and how SMIL can be embedded in another host player as a plug-in. The use of the shared data model as the communication paradigm between components decouples dependencies between these components: they only depend on a common understanding of the data model. The external data model allows for communication through setting values to variables by one side that can cause some action on the other communication side. This language bridge works fine from the SMIL side. The problem is on the other side of the bridge: As only SMIL State and XForms currently share the proposed data model, the integration into other languages requires some glue code. Following this direction, SMIL State [9] was implemented in the open-source Ambulant SMIL player plug-in for WebKit browsers. In the prototype, the glue is implemented with JavaScript code, which is triggered by DOM events when the data model changes.

A slightly different approach is used by Ginga Plug-in [13]. It defines an API that enables the host system (the HTML browser) to execute a series of actions, including those for controlling the plug-in life-cycle, e.g., `pause`, `resume`, or `abort` actions. The same API is used to answer to commands coming from the host system and to internal events, that occur during the plug-in execution. Unlike SMIL State, the reported events are not limited to those resulting from the attribution of variables. The NCL player API, part of the general API defined by the Ginga Plug-in, is inherited by the approach proposed in Section 5.

Timesheet.js [5] is an open-source library that supports the common subset of the SMIL Timing[4] and SMIL Timesheets [15] styling specifications. The approach uses HTML5+CSS3 for structuring and styling the multimedia content. Inline SMIL Timing elements can be inserted in the HTML document to handle timing, media synchronization, and user interaction. SMIL time constructs can also be inserted via an external timesheet. Timesheet.js, however, does not translate the entire SMIL language; it translates only its timing functionality, so it can be used on the Web. Moreover, in Timesheet.js, the SMIL time containers expose a significant part of the *HTMLMediaElement* API, which enables JavaScript code to control the SMIL time containers via the usual `play` and `pause` methods, check the current time via `currentTime` property, and receive `timeupdate` DOM events. As a consequence, a SMIL time container can be related to any other HTML5 element or other embedded SMIL time containers. This feature is also supported by Ginga Plug-in.

SimilingWeb [6] is a SMIL player that runs on any Web browser that supports HTML5. It uses a JavaScript library to control the presentation of the application. The current version of the player does not support all SMIL tags.

WebNCL [11] takes the same direction of SmilingWeb, in this case, however, to embed an NCL player into HTML5 browsers. In WebNCL, the NCL parser component translates the NCL document into the *NCL Representation Model*, an internal data structure that represents the NCL elements, e.g., link, media, connector, etc. The

[1] `http://www.w3.org/TR/xslt`
[2] `http://www.ambulantplayer.org/`
[3] `http://www.w3.org/MarkUp/Forms/`
[4] `http://www.w3.org/TR/SMIL3/smil-timing.html`

Table 1. Characteristics of the embedment approaches.

	SMIL State	Ginga Plug-in	Timesheet.js	SmilingWeb	WebNCL	NCL4Web
Source language	SMIL	NCL	SMIL (Timing)	SMIL	NCL	NCL
Embedment approach	plug-in	plug-in	inline	JavaScript compilation	JavaScript incremental compilation	XSLT compilation
Supports all features of the language	Yes	Yes	No	No	No	No
Can embed more than one player	Yes	Yes	Yes	?	Yes	No
Defines a control/notification API	Yes	Yes	Yes	No	Partially	Partially
Uses the media players of the browser	No	No	Yes	Yes	Yes	Yes

NCL Player Manager component controls the creation and destruction of HTML elements corresponding to the NCL specification and keeps track of the active media players, i.e., the media players controlled by the browser. The HTML5 elements corresponding to the NCL elements in the representation model are created incrementally (on demand) by the WebNCL player.

Melo et al. [11] argue that, in an NCL presentation, computer power is required more to media presentation than to media orchestration. Since the HTML browser is responsible for media presentation, and since browsers are usually implemented in native language, the performance difference between WebNCL and a native-code NCL player plugin would be irrelevant. This argument must be pondered considering synchronization issues in low-end platforms, like those found in DTV receivers. Nevertheless, WebNCL is certainly an adequate solution for high-end receivers.

NCL4Web [14], unlike WebNCL, transforms all NCL code into HTML+CSS+JavaScript before the whole presentation starts. A JavaScript file, called "ncl-complements.js," is inserted in the translated document; this file contains common function used by every translated NCL document, and manages user interactions. Instead of using JavaScript, the translation is done using an XSLT stylesheet. NCL4Web supports more NCL tags than WebNCL, e.g., switches and rules. Although NCL4Web proponents only explore client side conversion, the translation could also be realized in server side. This would allow for using the approach in Web browsers that do not support XSLT, e.g., some Android browsers.

More than one WebNCL presentation machine can be embedded in the same HTML page. NCL4Web was designed for presenting a standalone NCL application, but it can embedded in a Web page through using the HTML `<iframe>` element. However, in both solutions, an API for relating the embedded NCL applications was only partially defined. Both solutions rely on DOM events to report events coming from NCL presentations, although WebNCL also provides an API to post events to NCL entities.

The related works presented in this section rely on a language player engine embedded in HTML browsers that can be either implemented in some platform dependent language, which is the case of the plug-in approaches, or implemented in JavaScript. With the exception of Timesheet.js, which is intended for incorporating into HTML5 only the timing functionality of SMIL, all works aim at playing its source hypermedia language embedded in HTML. Moreover, these works try to convert a language profile that was conceived to help application authors, and thus is full of syntactic sugar and reuse features. This is one of the reasons why, except in the case of plug-ins, none of the players was able to contemplate the full expressiveness of its source language. Table 1 goes over the main points of these approaches.

In this paper, we argue that even in the case of plug-ins, the syntactic sugar and reuse features of the target language should be removed in an initial conversion step, before any of the solutions take place. Moreover, we also advocate the use of the Ginga Plug-in API by presentation engines targeting NCL. The next sections discuss the advantages of the proposed approach.

4. NCL RAW PROFILE

The authoring and presentation of hypermedia documents can be considered as the answer to four general questions: what to present, where, how, and when. To help authors in answering these four questions, several hypermedia specification languages have been proposed, like HTML, NCL, SMIL, etc. All these languages define much more than the needed entities to answer the questions, trying to help authors to logically structure the document specification, to choose between content alternatives, etc. All these languages also have reuse features and syntactic sugar add-ons to easy even more the authoring process.

However, language players should use data models as close as possible to the presentation engine's execution platform, to make a better use of the platform resources and to achieve an efficient and reliable implementation.

The different goals of the specification language data model and the player data model make them semantically distant. As a consequence, the process of converting one into another, during document presentation, can be complex and error-prone.

The solution usually adopted by language players, including all related work discussed in Section 3, is to sacrifice the presentation engine conception, obliging it to perform using a high-level data model. This alternative, although possible, especially in high performance platforms can lead to implementations that are more prone to efficiency and reliability problems due to code with redundant logic in the interpreter program.

An alternative for this approach come from identifying and defining procedures for redundancy removal in hypermedia specification languages. In doing this, we are, at the same time, giving a precise semantics for redundant elements and identifying which parts of our interpreter's code are likely to contain duplicated logic.

On the semantics side, we gain by trimming our specification. Each redundancy removal procedure functions as a precise definition for the removed element. Thus, given these procedures, we get a complete specification by just defining the behavior of the primitive elements. This approach also helps in formalizing, debugging, and maintaining the specification, because it keeps primitive behavior clearly separated from derived behavior. Here we consider primitive those concepts that are kept (they should be as minimum as possible) in the original conceptual model; and we consider derived the redundant concepts, identified by the removal procedures.

On the presentation engine side, the separation between what is primitive form what is derived induces, in terms of program logic, a similar separation between what should be compiled, or converted, from what should be interpreted.

In other words, the solution can come from introducing an intermediate conceptual data model. The problem is then moved to correctly choose this new data model to have both a translation process from the authoring data model to this intermediate data model, and another translation process, as a second step, towards a presentation data model simple enough that makes the implementation of converters and presentation engines simpler.

This intermediate conceptual data model can define an abstract syntax notation to which an authoring language can be compiled and from which the engine presentation model can be extracted. The intermediate syntax notation must have at least the same expressiveness of the authoring language. Ideally, it could have the expressiveness of different target authoring languages, allowing for application written in these different languages to share the common intermediate abstract syntax notation and thus share the same application player.

The first step in this direction was the definition of the NCL Raw profile [10]. The NCL Raw profile has been designed to allow simple converters for the NCL 3.0 EDTV profile and to allow a simpler Ginga-NCL implementation. It should be stressed that the profile goal is not the authoring process but to act as an intermediate language withing the conversion process.

The definition of the NCL Raw profile allows for its use not only in the client side, but also as the basis of a new transfer syntax. In this case, the conversion process could be done in the server side. This approach has some advantages. First, it allows for a simpler interpretation procedure at the client side. Since the client (receiver device) is usually a platform with limited resources, this can be an advantage. Second, as the Raw profile is not tailored for authoring, since it is less structured and has few reuse features, applications written in this profile are usually more difficult to be understood and thus, to be reverse engineered. Third, and the main one, the Raw profile can act as an intermediate notation for converters of other declarative languages. Thus the authoring phase can use languages other than NCL, without imposing any additional load on the receiver. Therefore, the NCL Raw profile can act as a liaison transfer syntax among several declarative hypermedia languages. This is an interesting point to be worked in the future.

The NCL Raw profile is backwards compatible; thus a Raw profile application should be able to run in any EDTV compatible player. In fact, this compatibility principle guided the profile design. Therefore, Ginga Plug-in, WebNCL, and NCL4Web are already able to run applications developed in the NCL Raw profile. In doing this, some of the limitations of the current implementations would vanish. For example, WebNCL would support, indirectly, the <switch> element. However, in Section 5 we go further: we propose the incorporation of the NCL Raw profile converter in the architecture of any NCL EDTV player, including those mentioned in this paper.

4.1 NCL Raw Profile Schema

Most redundant elements and attributes of NCL 3.0 EDTV profile were removed in defining the NCL Raw profile. Table 2 presents the elements of the NCL 3.0 Raw profile. In the table, parentheses are used for grouping, the symbol (|) read as "or," (?) read as "zero or one," (*) read as "zero or more," and (+) read as "one or more." Child element order is not specified. Moreover, for simplicity, only the <link> and <causalConnector> elements of the *Linking* and *Extended CausalConnectorFunctionality* modules of NCL 3.0 are presented.

The schemas of the NCL Raw profile can be downloaded from

`http://www.ncl.org.br/NCL3.0/RawProfile`

In short, the 45 elements of NCL 3.0 EDTV profile were reduced to 22 in the Raw profile. Section 5.1 details the conversion process of NCL 3.0 EDTV profile documents into equivalent NCL 3.0 Raw profile documents.

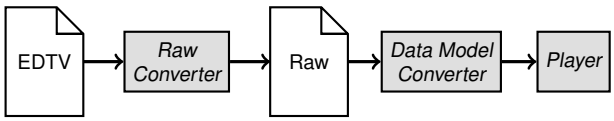
Table 2. The NCL 3.0 Raw Profile.

Element	Content
<ncl>	(<head>?, <body>?)
<head>	(<causalConnector>)*
<body>	(<port>|<property>|<media> |<context>|<link>)*
<media>	(<area>|<property>)*
<context>	(<port>|<property>|<media> |<context>|<link>)*
<area>	-
<port>	-
<property>	-
<link>	(<bind>)+
<causal-Connector>	((<simpleCondition>|<compoundCondition>), (<simpleAction>|<compoundAction>))

5. EMBEDDING NCL APPLICATIONS

We propose a new approach for the architecture of NCL players with, at least, two translation phases: from NCL EDTV profile to NCL Raw profile, and then to the data model of the NCL Raw player. This process is illustrated in Figure 1, in which the *Raw Converter* module is in charge of the first step, and the *Data Model Converter* is in charge of the second step.

Figure 1. The conversion flow.



Note that if the NCL Raw player is embedded into another language player, like an HTML browser, the *Data Model Converter*, in Figure 1, should be divided into two further steps: (i) translating (usually, compiling) from NCL Raw profile to the language of the host environment, and then (ii) translating (interpreting) the result of the previous step into the execution data model of the host environment. Note also that the *Raw Converter* module can be implemented in the server side, as discussed in Section 4.

5.1 NCL Raw Profile Converters

In the NCL Raw profile, <media> and <context> properties must be defined using only <property> elements. Moreover, the *Layout*, *Descriptor*, and *DescriptorControl* modules of NCL 3.0 were removed. Since all properties must be defined in <property> elements, the *externable* attribute is used to determine if the given property may be referenced by links and ports.

As an example conversion, consider the conversion of <region> elements. The <region> element defines the position and size parameters of <descriptor> elements, which can be associated to <media> elements. Regions are declared within <regionBase> elements and may be nested to any level, so that a child region may

define its attributes in relation to it's parent's attributes. We eliminate a region by converting its attributes (except *id*) into parameters of the associated descriptors, which later are converted into media properties. (Note that descriptors may also be referenced by <bind> elements; thus the need for the two conversion steps.) The transformation of <region> attributes into <descriptorParam> elements is straightforward if (i) the attribute belongs to a root region, i.e., a region that is an immediate child of a <regionBase> element; (ii) the attribute is *zIndex*; or (iii) the attribute is *width* or *height* and its value is given in pixels. In these cases, all we have to do is to insert a corresponding <descriptorParam> element into the associated descriptors.[5] Thus, e.g., the excerpt

```
<region id='r' width='10px' zIndex='1'/>
...
<descriptor id='d0' region='r'/>
<descriptor id='d1' region='r'/>
```

reduces to

```
<descriptor id='d0'>
  <descriptorParam name='width' value='10px'/>
  <descriptorParam name='zIndex' value='1'/>
</descriptor>
<descriptor id='d1'>
  <descriptorParam name='width' value='10px'/>
  <descriptorParam name='zIndex' value='1'/>
</descriptor>
```

If, however, conditions (i)–(iii) are false, we must calculate the value of the region attribute in relation to its parent's attribute, which is either obtained by rules (i)–(iii) or must be calculated from its parent's parent's attribute, and so on. Let $P(r)$ denote the parent region of a child region r, and let $r[a]$ denote the value of attribute a of r. Then the value of $r[a]$ in relation to $P(r)$ is given by the "unnest" function U such that

$$U(r,a) = \begin{cases} r[a] & \text{if (i), (ii), or (iii) hold for } r \text{ or } a \\ r[a] \cdot U(P(r),a) & \text{if } a = A_{wh} \\ r[a] + U(P(r),a) & \text{if } a = A_{tblr} \text{ and } r[a],\ U(P(r),a) \text{ are in pixels} \\ r[a] \cdot U(P(r),A_h) + U(P(r),a) & \text{if } a = A_{tb} \text{ and } r[a] \text{ is in \% and both } U(P(r),A_h),\ U(P(r),a) \text{ are in pixels or \%} \\ r[a] \cdot U(P(r),A_w) + U(P(r),a) & \text{if } a = A_{lr} \text{ and } r[a] \text{ is in \% and both } U(P(r),A_w),\ U(P(r),a) \text{ are in pixels or \%} \\ \uparrow \textit{(undefined)} & \text{otherwise,} \end{cases}$$

where $A_{x_1 x_2 \ldots x_n}$ stands for *width*, *height*, *top*, *bottom*, *left*, or *right* whenever $x_i = w, h, t, b, l, r$, for $1 \le i \le n$. The cases where $U(r,a)$ is undefined are those where we end up adding a pixel value to a percentage of screen's width or height, which cannot be done in conversion time because screen dimension is unknown.

The *TransitionBase* and *BasicTransition* modules were also removed from the EDTV profile in defining the Raw profile. Transitions are assumed to be media object's exhibition properties to be defined in <property> elements.

The Raw profile does not support any importing facility. Thus elements can only refer to elements defined in the same NCL document. This means that the *refer* attribute can only have an *id* value defined in the same document. Moreover, syntactic reuse is not allowed. Reuse is only allowed for presentation objects. Thus the *instance* attribute of <media> elements can only contain the values "instSame" or "gradSame."

We could have removed the <context> element, since it does not have a relevant role in presentation scheduling. But we have decided to keep it because it is required for document structuring, which can be helpful in the client side if live-editing is allowed and if link actions may be applied to a set of objects. The <body> element was kept in the Raw profile only for compatibility with the EDTV profile.

In the NCL Raw profile, content alternatives are not chosen from a <switch> element, instead, they are selected by links that test the global *settings* node of NCL. Thus the *TestRule*, *TestRuleUse*, *ContentControl*, and *SwitchInterface* were removed.

As another example conversion, we present the conversion of <switch> elements. The <switch> element defines a mutually exclusive set of <media>, <context>, or other <switch> components whose presentation depends on the evaluation of associated rules. A rule is a Boolean expression defined in the header section of the document, by <rule> or <compositeRule> elements; these are associated with the switch components via <bindRule> elements.

When we start a switch, its rules are evaluated in order of declaration. If a valid rule is found, the component associated with this rule is presented and no other rule is evaluated. If, however, no valid rule is found, either the default component, defined by the <defaultComponent> element, is presented, or no component is presented, in case there is no default.

We eliminate a switch by converting it into a specially crafted context that use links to implement the switch's logic. Before presenting the conversion algorithm, we define the semantics of the switch element. We deal first with the simplified case of switches that do not contain <switchPort> elements. The semantics and conversion algorithm for switches containing <switchPort> elements builds on the simplified case and will be given later.

Let S be a switch containing a list $x_1, x_2, \ldots, x_m$ of <media>, <context>, or other <switch> elements, and a list $b_1, b_2, \ldots, b_n$ of <bindRule> elements. Without loss of generality, assume that $m = n$ and that, for all $1 \le i \le n$, component x_i is associated with bind-rule b_i, i.e., $b_i[\textit{constituent}] = x_i[\textit{id}]$. (We may safely assume this because the switch semantics ignores dangling <bindRule> elements or components that are not referenced by any <bindRule> or <defaultComponent> elements.) Assume further that b_i is declared immediately before b_{i+1} in S. Then the algorithm for selecting which component of S is started when S is started is given by function G such that

$$G(S) = \begin{cases} x_1 & \text{if } V(R(b_1)) \\ x_2 & \text{if } \neg V(R(b_1)) \wedge V(R(b_2)) \\ \vdots & \vdots \\ x_n & \text{if } \neg V(R(b_1)) \wedge \cdots \wedge \neg V(R(b_{n-1})) \wedge V(R(b_n)) \\ x_d & \text{if } \neg V(R(b_1)) \wedge \cdots \wedge \neg V(R(b_{n-1})) \wedge \neg V(R(b_n)) \text{ and } x_d \text{ is the default component of } S \\ \varepsilon\ (\textit{none}) & \text{otherwise,} \end{cases}$$

where R is a function that returns the <rule> or <compositeRule> element referenced by a given <bindRule> element, and V is a unary predicate on the set of rules such that $V(r)$ holds iff (if, and only if) at the moment S started, rule r evaluates to true.

[5] Note that a region may be referred by multiple descriptors, but each descriptor may refer to at most one region.

Let S be a switch containing no <switchPort> element. Then the algorithm for removing S from an arbitrary NCL document consists of the following six steps.

Step 1. Create an empty context C such that $C[id] = S[id]$.

Step 2. Insert into C a <media> element, with unique identifier w, of the form

```
<media id=w refer=w' instance='instSame'/>
```

where w' is the identifier of the document settings node, i.e., the media object that contains the global properties to be tested.

Step 3. Insert into C an empty <media> element, with unique identifier t of the form

```
<media id=t/>
```

and a <port> element, with unique identifier t', pointing to t, of the form

```
<port id=t' component=t/>
```

Media t is used to trigger the links inserted in the next step.

Step 4. For each <bindRule> element b_i, such that b_i is the i-th element in the ordered list of <bindRule> elements of S, insert into context C a new link of the form

```
<link xconnector=K>
  <bind role='onBegin' component=t/>
  <bind role=b_i[rule]1 component=w
     interface=R_1(R(b_i))[var]/>
  <bind role=b_i[rule]2 component=w
     interface=R_2(R(b_i))[var]/>
  ...
  <bind role=b_i[rule]k component=w
     interface=R_k(R(b_i))[var]/>
  <bind role='start' component=b_i[constituent]/>
  <bind role='stop' component=t[id]/>
</link>
```

where R_j is a function that returns the j-th <rule> element in the tree defined by a given <rule> or <compositeRule> element. In addition, insert into the document's connector base a new causal connector, with unique identifier K, of the form

```
<causalConnector id=K>
  <compoundCondition operator='and'>
    <simpleCondition role='onBegin'/>
    <compoundStatement operator='and'>
      Ā(R(b_1)) Ā(R(b_2)) ··· Ā(R(b_{i-1})) A(R(b_i))
    </compoundStatement>
  </compoundCondition>
  <compoundAction operator='seq'>
    <simpleAction role='start'/>
    <simpleAction role='stop'/>
  </compoundAction>
</causalConnector>
```

where A is a function that converts a given rule into an equivalent <assessmentStatement> block and $\bar{A}$ denotes the negation of A; these functions are precisely defined below.

Step 5. If S contains a <defaultComponent> element x_d, then insert into C a new link-connector pair, similar to those inserted in the previous step, which starts x_d and whose main assessment statement is a conjunction of

$$\bar{A}(R(b_1))\ \bar{A}(R(b_2))\ \cdots\ \bar{A}(R(b_{i-1}))\ \bar{A}(R(b_i)).$$

Step 6. Finally, replace the switch S by context C.

The aforementioned function A, which converts a rule r into an equivalent <assessmentStatement> block, is defined as follows. If r is the j-th <rule> element encountered so far (starting at 1), then $A(r)$ is equal to

```
<assessmentStatement comparator=r[comparator]>
  <attributeAssessment role=r[id]j
    eventType='attribution'/>
  <valueAssessment value=r[value]/>
</assessmentStatement>
```

Otherwise, if r is a <compositeRule> element containing a list $r_1, r_2, \ldots, r_n$ of member rules, then $A(r)$ is equal to

```
<compoundStatement operator=r[comparator]>
  A(r_1) A(r_2) ··· A(r_n)
</compoundStatement>
```

The function $\bar{A}$, which converts a rule r into an <assessmentStatement> element that is equivalent to the negation of r, is defined by adding the attribute-value pair *isNegated*="true" to the root element of the tree returned by $A(r)$.

We proceed to prove (informally) the correctness of the previous switch removal algorithm. Let S be a switch containing no <switchPort> element, and let C be the context generated by the preceding algorithm. We want to show that

1. $G(S) = x$ iff x is the only component of C that gets started after C is started; and
2. $G(S) = \varepsilon$ iff no component gets started after C is started.

We shall prove the if-part of statement (1); the proof of the only-if-part of (1) and of both parts of (2) proceed in a similar fashion. Suppose $G(S) = x$, for some x in S. There are two possibilities: Either (i) there is a <bindRule> b_i such that b_i refers to x and

$$(\dagger) \qquad \neg V(R(b_1)) \wedge \neg V(R(b_2)) \wedge \cdots \wedge \neg V(R(b_{i-1})) \wedge V(R(b_i)),$$

or (ii) x is the <defaultComponent> of S and, for all i, $\neg V(R(b_i))$. We proceed to prove case (i); again, the proof of case (ii) is similar. Suppose that case (i) holds. Then, by the fourth step of the preceding algorithm, there is a link ℓ in context C that starts x immediately after C only if the conjunction of the series of statements

$$(\ddagger) \qquad \bar{A}(R(b_1))\ \bar{A}(R(b_2))\ \cdots\ \bar{A}(R(b_{i-1}))\ A(R(b_i))$$

evaluates to true. By construction, for all r, $A(r)$ evaluates to true iff $V(r)$; and $\bar{A}(r)$ evaluates to true iff $\neg V(r)$. Thus $(\dagger)$ implies $(\ddagger)$. Therefore, if we start C the condition of link ℓ is satisfied and x is started; moreover, no other link of C is satisfied, since they contain either an assessment statement of the form $A(R(b_j))$, for some $1 \leq j < i$, which evaluates to false since $\neg V(R(b_j))$, or an assessment statement of the form $\bar{A}(R(b_i))$, which also evaluates to false since $V(R(b_i))$. Q. E. D.

We can easily extend the previous algorithm to deal with switches containing <switchPort> elements. This general version consists of the following five steps. Step 1. Create a context C to represent S. Step 2. For each <switchPort> element p in S, apply steps 1–5 of the previous algorithm to obtain a context C_p that represents the sub-switch defined by the switch-port p, i.e., that containing only the <media> elements referenced by the <mapping> elements of p and the corresponding <bindRule> elements, and insert C_p into C. Furthermore, insert into C a <port> element p' such that $p'[id] = p[id]$. Step 3. Use steps 1–5 of the previous algorithm to obtain a context C_S, which will represent S, but this time considering all its components and <bindRule> elements, and insert C_S into C. Moreover, insert into C a <port> element, with

unique identifier q, pointing to C_S. STEP 4. Update the links that reference S directly, i.e., without specifying a switch-port, to point to port q of C. STEP 5. Finally, replace S by C.

5.2 NCL Player API

In implementing a player for the Raw profile of NCL it is important to use the *Player API* defined by Ginga Plug-in, which is recommended for embedded NCL applications in the ITU-T reference implementation of Ginga-NCL. In obeying this API, it will be possible to relate more than one NCL embedded application to each other, and to relate an NCL application to host language elements, e.g., HTML elements. It is also important to implement the *inputControlNotification API* to allow the NCL player to pass an gain control of the input devices. Reference [13] presents these APIs and discusses their use by the Ginga Plug-in implementation, together with some application use cases.

6. CONCLUSIONS

This paper recognizes the potential of HTML5 browsers in turning JavaScript into a target language for compilers (or converters) of other high-level languages. The wide distribution of HTML browsers can make JavaScript an important tool for bursting other language applications in the Web. Taking this into account, this paper also recognizes the importance of works like Ginga Plug-in, WebNCL, and NCL4Web in helping disseminate NCL applications. Contributing to those works, this paper encourages the use of NCL Raw profile as the target profile for NCL players, to allow for a simplified but complete implementation of the presentation engine.

Even if the developers of Ginga Plug-in, WebNCL, and NCL4Web do not follow our suggestions, the conversion of NCL EDTV applications to NCL Raw profile application in the server side could use those players with advantages. For example, some features not supported today would become available.

However, the re-factoring of the current NCL players to support the "conversion flow" depicted in Figure 1 would allow for a simpler interpretation procedure in the client side, and therefore, probably smaller and less prone to bugs. Moreover, the NCL Raw profile can act as an intermediate syntax notation for converters of other declarative languages. Hence, in the authoring phase, languages other than NCL, more tailored to user's flavor, could be used without imposing any additional load on receivers. We are currently working on this issue in the definition of CASyNo, a common abstract syntax notation for hypermedia languages. For now, NCL Raw profile does not introduce features particular to other languages neither features to support the new NCL 4.0, e.g., those features coming from 3D object support.

We have a partial implementation of an EDTV to Raw profile converter tool, called DietNCL.[6] We are also working on an new NCL player for the Raw profile. These implementations are helping us to fine tune the semantics of NCL EDTV profile, since each redundancy removal procedure functions as a precise definition of the removed element. This approach also helps in formalizing, debugging, and maintaining the specification, because it keeps the primitive behavior clearly separated from the derived behavior.

In another direction, we are also planning to use a non-XML syntax for Raw profile by translating it, e.g., into MPEG-4 BIFS [1].

[6]Available at `http://www.telemidia.puc-rio.br/~gflima`

ACKNOWLEDGMENTS

This work was partially supported by the Brazilian CNPq, CAPES, and FAPERJ funding agencies, and by the Brazilian Ministry of Science, Research, and Innovation.

REFERENCES

[1] 14496-11:2005, I. Information technology – Coding of audio-visual objects – Part 11: Scene description and application engine. 2005.

[2] ABNT NBR 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, SP, Brazil, November 2007.

[3] Berjon, R., Faulkner, S., Leithead, T., Navara, E. D., O'Connor, E., Pfeiffer, S., and Hickson, I. HTML5: A vocabulary and associated APIs, for HTML and XHTML. Candidate recommendation, W3C, August 2012.

[4] Bulterman, D. C., and Rutledge, L. W. *SMIL 3.0: Flexible Multimedia for Web, Mobile Devices and Daisy Talking Books*, 2nd ed. Springer, 2008.

[5] Cazenave, F., Quint, V., and Roisin, C. Timesheets.js: when SMIL meets HTML5 and CSS3. In *Proceedings of the 11th ACM Symposium on Document Engineering - DocEng'11* (2011), ACM, New York, NY, USA, pp. 43–52.

[6] Gaggi, O., and Danese, L. A SMIL player for any web browser. In *Proceedings of the 17th International Conference on Distributed Multimedia Systems - DMS 2011* (Forence, Italy, August 2011), ACM, New York, NY, USA, pp. 114–119.

[7] Ierusalimschy, R. *Programming in Lua*, 2nd ed. Lua.Org, 2006.

[8] ITU-T Recommendation H.761. Nested Context Language (NCL) and Ginga-NCL for IPTV Services. ITU-T, Geneva, Switzerland, April 2009.

[9] Jansen, J., and Bulterman, D. C. A. SMIL State: An architecture and implementation for adaptive time-based web applications. *Multimedia Tools and Applications 43*, 3 (2009), 203–224.

[10] Lima, G. A. F., Soares, L. F. G., Neto, C. S. S., Moreno, M. F., Costa, R. R., and Moreno, M. F. Towards the NCL Raw Profile. In *II Workshop de TV Digital Interativa (WTVDI) - Colocated with ACM WebMedia'10* (Belo Horizonte, MG, Brazil, October 2010).

[11] Melo, E. L., Viel, C. C., Teixeira, C. A. C., Rondon, A. C., Silva, D. d. P., Rodrigues, D. G., and Silva, E. C. WebNCL: A Web-based presentation machine for multimedia documents. In *Proceedings of the 18th Brazilian Symposium on Multimedia and the Web - WebMedia'12* (São Paulo, SP, Brazil, October 2012), ACM, New York, NY, USA, pp. 403–410.

[12] Merkel, K. HbbTV: A hybrid broadcast-broadband system for the living room. In *Proceedings of the European Broadcasting Union* (Geneva, Switzerland, 2010).

[13] Moreno, M. F., Marinho, R. S., and Soares, L. F. G. Ginga-NCL architecture for plug-ins. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins - TOPI'11* (Waikiki, Honolulu, HI, USA, 2011), ACM, New York, NY, USA, pp. 12–15.

[14] Silva, E., Saade, D. C. M., and Santos, J. D. NCL4WEB - Translating NCL applications to HTML5 Web pages. In *Proceedings of the 13th ACM Symposium on Document Engineering - DocEng 2013* (Florence, Italy, September 2013), ACM, New York, NY, USA.

[15] Vuorimaa, P., Bulterman, D., and Cesar, P. SMIL Timesheets 1.0. Working draft, W3C, January 2008.