Architecture for Hypermedia Dynamic Applications with Content and Behavior Constraints

Luiz Fernando G. Soares¹ Carlos de Salles Soares Neto² José Geraldo de Sousa¹
lfgs@inf.puc-rio.br csalles@deinf.ufma.br jgeraldo@laws.deinf.ufma.br
¹ Depto. de Informática – PUC-Rio
Rua Marquês de São Vicente, 225
Rio de Janeiro – 22453-900 – Brazil
0055-21-3527-1500 Ext: 4330
² Depto. de Informática – UFMA
Av. dos Portugueses, Bacanga
São Luís/MA – 65080-040 – Brasil
0055-98-3301-8224

© ACM, 2012. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in DocEng’12,
http://dx.doi.org/10.1145/2361354.2361403.
Architecture for Hypermedia Dynamic Applications with Content and Behavior Constraints

Luiz Fernando G. Soares\textsuperscript{1} Carlos de Salles Soares Neto\textsuperscript{2} José Geraldo de Sousa\textsuperscript{1}
lfgs@inf.puc-rio.br csalles@deinf.ufma.br jgeraldo@laws.deinf.ufma.br
\textsuperscript{1} Depto. de Informática – PUC-Rio
Rua Marquês de São Vicente, 225
Rio de Janeiro– 22453-900 – Brasil
0055-21-3527-1500 Ext:4330
\textsuperscript{2} Depto. de Informática – UFMA
Av. dos Portugueses, Bacanga
São Luís/MA – 65080-040 – Brasil
0055-98-3301-8224

ABSTRACT
This paper deals with the generation of dynamic hypermedia applications whose content and behavior their authors may not be able to predict a priori, but which must conform to a strict set of explicitly defined constraints. In the paper, we show that it is possible to establish an architecture configuration to be followed by this special kind of dynamic applications. In the proposed architecture, templates are responsible for specifying the design patterns and the constraints to be followed. Some alternatives for distributing (from the client side to the server side) the components that comprise the architecture are discussed, and one of them is used to exemplify an instantiation of the architecture. In the instantiation, TAL (Template Authoring Language) is used to define templates. In TAL, templates are open-compositions, that is, special set of patterns for compositions, whose content must obey some explicitly defined constraints. The paper also shows how the architecture instantiation could be used to build dynamic digital TV applications.

Categories and Subject Descriptors
D.2.11 [Software Architectures]: Domain-specific architectures, languages, patterns.

General Terms
Design, Languages.

Keywords
Dynamic DTV applications, Templates, TAL, NCL.

1. INTRODUCTION
Hypermedia applications comprise a set of media data to be presented and a specification document (their source code) relating these data in time and space. Dynamic hypermedia applications are those that can be modified (or created) on-the-fly, that is, while their data content is being presented on the client side, as a consequence of some internal or external event occurrences.

We call “modifier agent” the software module in charge of receiving the aforementioned events and accomplishing the necessary tasks to make the desired changes (in the document specification or in data content).

The triggering events can be generated by human beings or automatically by the application itself. They can be generated at the client side (for example, a digital TV – DTV – receiver, in the case of DTV applications), or at the server side (the broadcaster, for example, also in the case of DTV applications). Events are sent to modifier agents that can be located in the server side, in the client side or in a proxy environment.

Dynamic applications are very important in digital TV domain. They allow for the personalization of application content, or the way application content is presented, depending on the viewer profile, the receiver profile, or even the viewer localization. Moreover, it is common in DTV applications having some media content that is only known at exhibition time. For example, during a soccer game, statistics about a goal can be shown in the moment the goal occurs (and if it occurs), and it depends on who makes the goal. It is also very common in DTV applications to have spatial and temporal relationships among media content that are not known a priori and that may depend on some live content. For example, in a talk show, depending on the matter under discussion, a link with other related matters can be established commanding their presentation when a viewer interacts. In addition, it is not unusual to have new content to be presented depending on viewers answers.

Modifier agents can be part of the language entities used to specify applications. For example, NCL (Nested Context Language) [1] and SMIL (Synchronized Multimedia Integration Language) [2] languages define the <switch> element to specify set of alternative media objects, and rules for their selection. In this paper we do not focus on this kind of dynamic application however. We are much more interested in dynamic applications whose alternatives cannot be predicted a priori.

Moreover, we are interested in dynamic hypermedia applications whose structure and content must observe some constraints that can be explicitly defined. There are many good reasons for establishing explicit constraints. First, they promote coherent application branding, enabling content producers to define and
In client-side Web browsers, plug-ins, scripts (in particular JavaScript/ECMAScript [3]) and applets are the usual technologies applied for modifier agent implementations. In HTML-based DTV middlewares, we usually find the ECMAScript approach [4] [5] [6] [7].

Scripts can have a series of constraints imposed by a system, for example, in DTV systems, for security reasons they do not have right to read or write files in the client receiver. Beyond these implicit system constraints, additional constraints cannot be imposed to application authors, for example, forcing them to follow some design patterns.

1 We are using the term design pattern in this paper in its broad sense: a general reusable solution to a commonly occurring problem within a given context in software design. It is a description or template for how to solve a problem that can be used in many different situations.

2 The current version of ISDB-T standard implements the XML markups, the stylesheets properties, the ECMAScript engine and the DOM API for the BML for basic services ("fixed terminal profile").
from external actions, like viewer interactions, etc. Like in HTML-based solutions, Lua scripts can have a series of implicit constraints imposed by the system.

NCL provides an API that allows for building and modifying applications on-the-fly through live editing commands. NCL editing commands are much more powerful than the events coming from the server-side in HTML-based solutions. NCL editing commands allow for dynamically changed applications by adding or removing any NCL element or attribute, without needing to trigger imperative scripts. Moreover, NCL editing commands can add and trigger Lua objects on-the-fly. That is, Lua scripts cannot only act as static ECMAScripts does in HTML-based solutions, but can also be dynamically loaded. Moreover, NCL editing commands can come from the server side and can also be generated by embedded Lua scripts.

Despite having a richer model to generate dynamic content for hypermedia applications, NCL, by itself, does not allow incomplete application specifications that follow explicitly defined constraints, from which applications can be derived following some implicitly defined design pattern.

### 2.2 Applications with Explicit Constraints

Several hypermedia applications embed common design patterns. This is very usual in DTV domain. Design patterns have been intensively studied and proposed in the literature [18], including those targeting hypermedia applications [19] [20]. However, to the best of our knowledge, all structure-based hypermedia languages, including those used in DTV domain, fail to let authors create applications in which unspecified internal content or unspecified relationships have to follow some design patterns.

Some authoring environments for hypermedia applications use common patterns to guide and easy application authoring [21] [22]. However, they do not focus on dynamic application generation beyond those already mentioned in this Section 2.

In line with design patterns principles, SMIL Timesheets [23] [24] is an example of technology that allows for adding temporal behavior to hypermedia applications independently of the specification language used.

Dynamic DTV applications can be defined using design patterns, both in client and server sides. However, no DTV middleware offer this kind of support, although they can be roughly implemented using their scripting languages.

Unlike usual design pattern definitions, TAL (Template Authoring Language) [25] allows for defining not only common patterns but also a series of constraints on their uses. TAL can be considered as a specification language to a set of high level hypermedia design patterns expressed as a template. So, one of the main basis of TAL are templates for composition, proposed in previous versions of XTemplate language [26]. A template is formally described by means of a vocabulary of allowed child-object types, a set of relations allowed between those types, rules that constraint the instantiation of these child-object types and relations, and a set of fixed components of the composition (media objects, other composite objects and relationships). In this sense, a template is an incomplete hypermedia composition (an open-composition) that has certain blanks that must be filled out in accordance with rules that constrain the content and relationships that authors can insert.

TAL specifications can be used to generate hypermedia applications in declarative languages such as SMIL, HTML/ECMAScript, NCL/Lua, etc. Using TAL, dynamic hypermedia applications can be developed constrained to explicitly specified semantics.

Although any other language for designing pattern specifications could have been used in the architecture we propose in Section 3, TAL has been our starting point, as presented in Section 4.

### 3. TEMPLATE-BASED ARCHITECTURE

In order to have dynamic hypermedia applications following explicitly defined design patterns and whose modifier agents must follow explicitly defined constraints, the architecture depicted in Figure 1 is proposed. In the architecture, a template is responsible for specifying the design patterns and the constraints to be followed, as defined in the previous section.

![Figure 1. Template-based generation of dynamic applications.](image)

It should be noted first that the template itself can be changed by a modifier agent (Ag2 in Figure 1). Therefore, even the design patterns that an application must follow and the explicitly defined constraints that other modifier agents must obey can be dynamically changed by the Ag2 component.

Second, the architecture also defines a processing flow, but it is agnostic to where (client side, server side, or both) any modifier agent runs, as discussed further on.

Third, as previously mentioned in Section 1, modifier agents are requested for changes by events that can be produced by human beings, that can be automatically generated by the applications themselves, or that can be caused by the modifiers agents themselves.

The Ag4 component is the usual modifier agent embedded in existing applications, which is responsible for creating dynamic content that has no explicitly defined constraint. They are the modifier agents discussed and exemplified in related work presented in Section 2.1: scripting objects, switch elements, etc.

The other components compound the core of our architecture. Some event coming from the hypermedia application starts the Ag1 processing. This event can be the result of user interactions; can come from the presentation of some application content (like its start, end, pause, etc.); can result from setting values to properties of some content (for example, establishing a new position for its presentation); can come from the processing of...
some object that contains imperative (probably scripting) code; etc.

The event triggering Ag1 can come with some input parameter. For example, the event can report a viewer interaction and which object was selected by the interaction. Based on the parameter received, Ag1 can start to reconstruct the application, which must follow a given Template, also present in Figure 1. The result of the Ag1 processing is a padding document (padDock in Figure 1).

The padding document is written in any language understood by the template processor (TP in Figure 1). This document must fill the blanks of the open-composition template (Template in the figure) with dynamic content that Ag1 generates based on the input parameter it has received.

After finishing its task, Ag1 calls the service of the template processor, passing the template to be used and the padding document as parameters. TP can then generate the completely new application.

A template must be processed together with a padding document giving rise to a new document in some specific language, called target language. Usually, specific processors are required for each target language and for each language used to specify padding documents. Figure 2 illustrates the process.

Authors of dynamic applications are usually only concerned with authoring applications with their Ag4 modifier agents, and authoring the Ag1 modifier agents. Ag2, Ag3, TP, and Filter components, besides the template document, are usually under the responsibility of another player: the one that requires that applications follow the template. For example, in DTV domain, broadcasters can require that applications follow a given template. Therefore, they are responsible to implement the TP and Filter components to check if after each application change the Core Application remains in agreement with the template.

We must emphasize that the granularity of the components of the architecture were defined only to make clear the processing flow of the Core Application generation. An implementation of the architecture does not need to follow strictly the components defined, but only their functionalities. For example, it can be more efficient to have the TP template processor incorporating the functions of the Filter, creating editing commands directly.

Focusing on Figure 1, we can devise some alternative distributions for their components. Figure 3 shows some possibilities, no matter where the template document is stored.

The first alternative (Figure 3.a) builds all dynamic content at the client side. Indeed, in this alternative, the client-side can consider the Filter, TP, Ag1 and Ag4 as components (usually scripting objects) of the whole application. The dynamic content to be presented is the Core Application, also part of the whole application. The example given in Section 3.1 follows this alternative of distribution.

The second alternative (Figure 3.b) builds the dynamic content ruled by the template at the server side (usually a broadcaster station in DTV systems). In this case the event to trigger the Ag1 modifier agent must be sent from the client side to the server side. In terrestrial DTV systems, for example, this can be usually done using the return channel. In this case, the necessary editing commands to change the old application to the new one are usually sent in data carousels [27].

In the third alternative (Figure 3.c) the template processor and the Filter run at the server side. Ag1 and Ag4 run at the client side, which has to communicate with the template processor. Again, in terrestrial DTV systems the return channel can be used with this purpose. Note that in this alternative, as well as in the previous one, if the editing commands are sent by broadcasting, a client user can command an application change that will affect all other client users.

Of course other alternatives are possible including ones using a proxy provider to perform the dynamic content generation. This can be very interesting in social TV applications. For example, assume that in Figure 3.c, the server processing is now performed by a proxy server of some social network. Changes made by a member of this network, for example, the addition of a comment to a movie, will also reflect on the movie presentation in all other members of the same network. Note that all changes would still be controlled by who have implemented the template, the template processor, and the Filter.
3.1 Architecture Instantiation Example

This section presents an implementation example of the proposed architecture. The architecture presented in Figure 1, with the component distribution presented in Figure 3.a, has been implemented targeting dynamic NCL applications running on Ginga middleware [14] [15] [16].

In the implementation, TAL [24] is used to define templates. TAL templates are generated and stored in some place accessible from client-side receivers. External modifier agents (Ag2) can change these templates, but are completely independent from the client-side implementation, as shown in Figure 3.a.

As aforementioned and illustrated in Figure 2, TAL templates are processed together with padding documents to generate applications in different target languages, depending only on the specific processor used. In this example case, the padding document is an NCL document and the target document is also an NCL application, named “CoreApplication” in Figure 3.a.

Any other pattern language and any other language used to define constraints could be used to define templates. We have chosen TAL because it has a template processor targeting NCL applications. In addition, this processor is implemented in Lua language. As presented in Section 2, Lua is the scripting language of NCL. This fact has an impact on the implementation, as discussed in what follows.

TAL Processor [28] for generic NCL applications is used in the implementation. The Ag1 modifier agent is also an NCL media object with Lua code, as well as the Filter component. Note thus that the Ag1, Filter, TP, and the “CoreApplication” are objects of an NCL encompassing application (as a glue language, NCL applications can embed objects with Lua code, and other NCL applications). Thus we have here a case of reflective application: an application that adapts part of itself (the embedded “CoreApplication” NCL application), as presented in Listing 1.

As usual in declarative specification, we can understand how the encompassing NCL application works in performing its tasks, even without knowing the language syntax. So, let us browse the Listing 1 document, without worrying about the language details.

This media object contains two interfaces: an entry property (“Agentry”) and an output property (“output”).

Similarly, lines 15 to 18 define the TP (template processor) media object (<media id="TP" src="../tp.lua" type="application/x-ncl-NCLua" .../>), and lines 19 to 21 define the Filter media object (<media id="Filter" src="../filter.lua" type="application/x-ncl-NCLua" .../>). Note that the “Filter” does not have the “output” property since its output are NCL editing commands [16].

Lines 22 to 35 define the “CoreApplication”. The initial “CoreApplication” is generic and has no content to be exhibited. It just starts the application building. The “CoreApplication” starts through its entry port (“appEntry”) defined in line 23. The application has an output port, defined in line 24, externalizing the “shared.startChanging” property of its “globalVariables” object (defined in lines 25 to 27) to be used in the relationship with the “Ag1” media object.

Lines 7 to 9 define that “Ag1”, “TP”, and “Filter” must be started when the application begins. The starting of these event-oriented Lua objects call their initializer procedures that creates all code spans and data that may be used during the Lua-object execution and, in particular, registers one (or more) event handler.

Line 10 defines the “CoreApplication” as another opening component: the one that starts the whole dynamic process. The “CoreApplication” begins (line 23) with the exhibition of a fake component (represented by the “initializer” <media> element in line 28 without source content specified). The beginning of this fake content produces an event triggering the relationship specified in the NCL <link> element (lines 29 to 34), which sets the “initData” value to the “shared.startChanging” property (defined in line 26), starting the dynamic-changing process.

The end of the “initData” value attribution to the “shared.startChanging” property produces an event triggering the relationship specified in the NCL <link> element (lines 36 to 42), which sets the “Ag1entry” variable of Lua and calls the corresponding event handler.

Upon receiving the setting event, the “Ag1” lua object generates the NCL padding document, based on the “initData” information received. This padding document in then set by the Lua code to the “output” property (line 13) of “Ag1” together with the template location to be used: the “(padDoc, template)” pair value.

The end of this attribution produces an event triggering the relationship specified in the NCL <link> element defined in lines 43 to 49, which sets the “TPentry” property (line 13) of the Lua...
template processor (TP) component, calling its corresponding event handler.

Using NCL editing commands [16] an external Ag2 modifier agent can change (remove and add) “TP”, as is usual with any other NCL application entity [1].

After creating the target document, the “TP” Lua code sets this document to its “output” property (line 17). The end of this attribution produces an event triggering the relationship specified in the NCL <link> element defined in lines 50 to 56, which sets the “Filterentry” property of the “Filter” element to “targetDoc”, calling its corresponding event handler. The Lua code of “Filter” compares the new NCL Core Application (“targetDoc”) with the old one. Based on this comparison, NCL editing commands are issued changing the previous “CoreApplication” to the new one.

The whole process restarts when a new Core Application calls Ag1 to create a new dynamic application, by setting a “data” value to its “shared.startChanging” property. If the setting value is equal to “end”, the Lua code of “Ag1” must generate the NCL editing command that finishes the whole reflexive application.

### Listing 1. Reflexive NCL application.

```xml
1. <xml version="1.0" encoding="ISO-8859-1"?>
2. <ncl id="nclReflexiveApplication" xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
3. <head>
4. ...
5. </head>
6. <body>
7. <port id="startAg1" component="Ag1"/>
8. <port id="startTP" component="TP"/>
9. <port id="startFilter" component="Filter"/>
10. <port id="startApp1" component="CoreApplication" interface="appEntry"/>
11. <media id="Ag1" src="../ag1.lua" type="application/x-ncl-NCLua">
12. <property name="Ag1entry"/>
13. <property name="output"/>
14. ...
15. <media id="TP" src="../tp.lua" type="application/x-ncl-NCLua">
16. <property name="TPentry"/>
17. ...
18. <media>
19. <media id="Filter" src="../filter.lua" type="application/x-ncl-NCLua">
20. <property name="FilterEntry"/>
21. </media>
22. <context id="CoreApplication">
23. <port id="appEntry" component="initializer"/>
24. <port id="startChanging" component="globalVariables" interface="shared.startChanging"/>
25. <media id="globalVariables" type="application/x-ncl-settings">
26. <property name="shared.startChanging"/>
27. </media>
28. <media id="initializer" type="video">
29. <link xconnector="onBeginSet">
30. <bind role="onBeginSet" component="initializer"/>
31. <bind role="set" component="globalVariables" interface="shared.startChanging"/>
32. <bindParam name="var" value="initData"/>
33. </bind>
34. </link>
35. </context>
36. <link xconnector="onEndAttSet">
37. <bind role="onEndAttribution" component="CoreApplication" interface="startChanging"/>
38. <bind role="getValue" component="CoreApplication" interface="startChanging"/>
39. <bind role="set" component="Ag1" interface="startChanging"/>
40. <bindParam name="var" value="getValue"/>
41. </bind>
42. </link>
43. <link xconnector="onEndAttSet">
44. <bind role="onEndAttribution" component="Ag1" interface="output"/>
45. <bind role="getValue" component="Ag1" interface="output"/>
46. <bind role="set" component="TP" interface="TPentry"/>
47. <bindParam name="var" value="getValue"/>
48. </bind>
49. </link>
50. <link xconnector="onEndAttSet">
51. <bind role="onEndAttribution" component="TP" interface="output"/>
52. <bind role="getValue" component="TP" interface="output"/>
53. <bind role="set" component="Filter" interface="Filterentry"/>
54. <bindParam name="var" value="getValue"/>
55. </bind>
56. </link>
57. </body>
58. </ncl>
```

### 4. EXAMPLE OF DYNAMIC APPLICATION

Let us now use the architecture instantiation of Section 3.1, to create a dynamic application example.

The application is deliberately very simple, just to illustrate the concepts. Assume that an IPTV channel requires that every third-party movie it transmits must have its logo (TeleMídia) and also an image button (Zappiens) representing the channel’s video repository, as shown in Figure 4.
Of course the IPTV channel, when tuned, also requires that only its logo can be exhibited and only its video repository is available to download movies.

The application also has a set (composition) of image buttons, each one associated with a video of the channel’s repository that has some semantic relationship (same genre, same actors, etc.) with the movie being presented. This set of buttons is presented when the “Zappiens” button is selected, as shown in Figure 5.

![Screenshot of the dynamic application example after selecting the repository.](image)

If one of the semantically related video is chosen, the process of dynamic changes is started. The movie is substituted by the chosen one, and a new list of correlated videos is created. The new CoreApplication is then restarted: causing the presentation of the new chosen movie, the “TeleMídia” logo, and the Zappiens” button, similar to Figure 4. The application must finish if the IPTV channel logo is selected.

Going on this section, we discuss in Section 4.1 the template used to create the application. Section 4.2 gives the padding document created by the modifier agent $Ag_1$. Finally, Section 4.3 presents the application reconstruction.

4.1 The Template Example

Let us now move our attention to the TAL template, in which all the application complexity is defined.

In TAL, template is an open-composition (an incomplete composition). More precisely, in TAL template is an especially pattern for composition, whose content is given by:

- Vocabulary: defining the allowed types of child-objects (the components) of the template, the allowed types of interfaces for these child-objects and for the template itself, and the allowed relations to be used in relationships among child-objects;
- Constraints: defining rules on the types defined in the vocabulary;
- Resources: defining common instantiated child object types that shall be inherited by all compositions that use (follow) the template;
- Relationships: defining common instantiated relation types, relating child-object types and resources that shall also be inherited by all compositions that follow the template.

Our open-composition is the “CoreApplication”, and its template is defined in lines 2 to 59 of Listing 2. As usual in declarative specifications, we can understand how TAL templates work, even without knowing the language syntax. So, let us superficially browse the Listing 2 document, without worrying about the language details. The structural view of the template, shown in Figure 6, also helps in its understanding.

![Figure 6. Structural view of the template.](image)

The template has four <port> elements (lines 3 to 6), mapped to the video type component, to the two button resources (“Zappiens” and “TeleMídia”), and to the “shared.startChanging” interface of the “globalVariables” resource.

The video type component is defined in line 7. Its selects attribute establishes to which target padding language element the <component> type must be applied.

The two button resources (“Zappiens” and “TeleMídia”) are specified in lines 9 and 10, respectively. They must follow the button type specified in line 8, as defined by their class attribute.

Another resource (“globalVariables”) is specified in lines 11 to 13; its interface <property name=”shared.startChanging”> is the one mapped to the “startChanging” port, defined in line 6.

Two other component types are defined. The first one is the composition (“relatedContext”) defined in lines 14 to 28, which contains the button images representing the several videos semantically related with the video being presented. These button images follow the other component type (“relatedContent”), specified in lines to 25 to 27. The “relatedContext” component type has interfaces (“contentInterface”) to each instantiated “relatedContent” type element, as defined in lines 15 to 19. The “relatedContent” component type has also interfaces (“dataInterface”) to each “data” interface of instantiated “relatedContent” type elements, as defined in lines 20 to 24.

The CoreApplication behavior defined by the template is dictated by its links. The first one (lines 29 to 32) establishes that if the button corresponding to the Zappiens repository is selected, all buttons corresponding to the videos of the Zappiens site that are semantically related with the video in current exhibition have their exhibition started.

The second relationship (“videoSelection” in lines 33 to 37) is a TAL link relating child component types of the template. It will result in as many links as “relatedContent” type instances (children of the “relatedContext” instance) exists in the target
document. For each one of these instances, it establishes that when the instance is selected, the “shared.startChanging” property of the “globalVariables” component is set to the corresponding “video-locator” value of the “data” property of the selected instance.

Finally, the third relationship (lines 38 to 43) determines that when the “TeleMidia” button is selected, the “shared.startChanging” property of the “globalVariables” component is set to the “end” value. As we have discussed in Section 3.1, this will end the application.

Constraints are defined on the components and relationships of the template by using TAL <assert> elements. The <assert> element requires that the test evaluation returns “true”, otherwise an error message should be presented. All the constraints in Listing 3 can be understood by reading their corresponding error messages.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>&lt;tal:tal id=&quot;template_repository&quot;&gt;</code></td>
</tr>
<tr>
<td>2.</td>
<td><code>&lt;tal:template id=&quot;coreApplication&quot;&gt;</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>&lt;por id=&quot;pVideo&quot; component=&quot;video[1]&quot;/&gt;</code></td>
</tr>
<tr>
<td>4.</td>
<td><code>&lt;por id=&quot;pZappiens&quot; component=&quot;Zappiensi&quot;/&gt;</code></td>
</tr>
<tr>
<td>5.</td>
<td><code>&lt;por id=&quot;pTeleMidia&quot; component=&quot;TeleMidia&quot;/&gt;</code></td>
</tr>
<tr>
<td>6.</td>
<td><code>&lt;por id=&quot;startChanging&quot; component=&quot;globalVariables&quot; interface=&quot;shared.startChanging&quot;/&gt;</code></td>
</tr>
<tr>
<td>7.</td>
<td><code>&lt;tal:component id=&quot;video&quot; selects=&quot;media[class=video]&quot;/&gt;</code></td>
</tr>
<tr>
<td>8.</td>
<td><code>&lt;tal:component id=&quot;button&quot; selects=&quot;media[class=button]&quot;/&gt;</code></td>
</tr>
<tr>
<td>9.</td>
<td><code>&lt;media id=&quot;Zappiens&quot; src=&quot;ZappiensButton.png&quot; class=&quot;button&quot;/&gt;</code></td>
</tr>
<tr>
<td>10.</td>
<td><code>&lt;media id=&quot;TeleMidia&quot; src=&quot;TeleMidiaButton.png&quot; class=&quot;button&quot;/&gt;</code></td>
</tr>
<tr>
<td>11.</td>
<td><code>&lt;media id=&quot;globalVariables&quot; type=&quot;application/x-ncl-settings&quot;/&gt;</code></td>
</tr>
<tr>
<td>12.</td>
<td><code>&lt;prop name=&quot;shared.startChanging&quot;/&gt;</code></td>
</tr>
<tr>
<td>13.</td>
<td><code>&lt;/media&gt;</code></td>
</tr>
<tr>
<td>14.</td>
<td><code>&lt;tal:component id=&quot;relatedContext&quot; selects=&quot;context[class=relatedContext]&quot;/&gt;</code></td>
</tr>
<tr>
<td>15.</td>
<td><code>&lt;tal:interface id=&quot;contentInterface&quot; selects=&quot;port[class=pContent]&quot;/&gt;</code></td>
</tr>
<tr>
<td>16.</td>
<td><code>&lt;tal:forEach instanceOf=&quot;contentInterface&quot; iterator=&quot;it&quot;&gt;</code></td>
</tr>
<tr>
<td>17.</td>
<td><code>component=relatedContext[it]</code></td>
</tr>
<tr>
<td>18.</td>
<td><code>&lt;/tal:forEach&gt;</code></td>
</tr>
<tr>
<td>19.</td>
<td><code>&lt;/tal:interface&gt;</code></td>
</tr>
<tr>
<td>20.</td>
<td><code>&lt;tal:interface id=&quot;dataInterface&quot; selects=&quot;port[class=pContentData]&quot;/&gt;</code></td>
</tr>
<tr>
<td>21.</td>
<td><code>&lt;tal:forEach instanceOf=&quot;dataInterface&quot; iterator=&quot;it&quot;&gt;</code></td>
</tr>
<tr>
<td>22.</td>
<td><code>component=relatedContext[it], interface=data[1]</code></td>
</tr>
<tr>
<td>23.</td>
<td><code>&lt;/tal:forEach&gt;</code></td>
</tr>
<tr>
<td>24.</td>
<td><code>&lt;/tal:interface&gt;</code></td>
</tr>
<tr>
<td>25.</td>
<td><code>&lt;tal:component id=&quot;relatedContext&quot; selects=&quot;media[class=relatedContext]&quot;/&gt;</code></td>
</tr>
<tr>
<td>26.</td>
<td><code>&lt;tal:interface id=&quot;data&quot; selects=&quot;property[name=data]&quot;/&gt;</code></td>
</tr>
<tr>
<td>27.</td>
<td><code>&lt;/tal:component&gt;</code></td>
</tr>
<tr>
<td>28.</td>
<td><code>&lt;/tal:component&gt;</code></td>
</tr>
<tr>
<td>29.</td>
<td><code>&lt;link xconnector=&quot;onSelectionStart&quot;&gt;</code></td>
</tr>
<tr>
<td>30.</td>
<td><code>&lt;bind role=&quot;onSelection&quot; component=&quot;Zappiens&quot;/&gt;</code></td>
</tr>
<tr>
<td>31.</td>
<td><code>&lt;bind role=&quot;start&quot; component=&quot;relatedContext&quot;/&gt;</code></td>
</tr>
<tr>
<td>32.</td>
<td><code>&lt;/link&gt;</code></td>
</tr>
<tr>
<td>33.</td>
<td><code>&lt;tal:link id=&quot;videoSelection&quot;&gt;</code></td>
</tr>
<tr>
<td>34.</td>
<td><code>&lt;tal:forEach instance=&quot;relatedContent&quot; iterator=&quot;it&quot;&gt;</code></td>
</tr>
<tr>
<td>35.</td>
<td><code>onSelection relatedContext.contentInterface[it] then</code></td>
</tr>
<tr>
<td>36.</td>
<td><code>set globalVariables = dataInterface[it] end</code></td>
</tr>
<tr>
<td>37.</td>
<td><code>&lt;/tal:forEach&gt;</code></td>
</tr>
<tr>
<td>38.</td>
<td><code>&lt;/tal:link&gt;</code></td>
</tr>
<tr>
<td>39.</td>
<td><code>&lt;link xconnector=&quot;onSelectionSet&quot;&gt;</code></td>
</tr>
<tr>
<td>40.</td>
<td><code>&lt;bind role=&quot;onSelection&quot; component=&quot;TeleMidia&quot;/&gt;</code></td>
</tr>
<tr>
<td>41.</td>
<td><code>&lt;bindParam name=&quot;var&quot; value=&quot;end&quot;/&gt;</code></td>
</tr>
<tr>
<td>42.</td>
<td><code>&lt;/link&gt;</code></td>
</tr>
<tr>
<td>43.</td>
<td><code>&lt;/tal:template&gt;</code></td>
</tr>
<tr>
<td>44.</td>
<td><code>&lt;tal:assert test=&quot;#video==1&quot;&gt;</code></td>
</tr>
<tr>
<td>45.</td>
<td><code>CoreApplication must have just one video.</code></td>
</tr>
<tr>
<td>46.</td>
<td><code>&lt;/tal:assert&gt;</code></td>
</tr>
<tr>
<td>47.</td>
<td><code>&lt;tal:assert test=&quot;#button==2&quot;&gt;</code></td>
</tr>
<tr>
<td>48.</td>
<td><code>It must have only two button images.</code></td>
</tr>
<tr>
<td>49.</td>
<td><code>&lt;/tal:assert&gt;</code></td>
</tr>
<tr>
<td>50.</td>
<td><code>&lt;tal:assert test=&quot;#relatedContext==1&quot;&gt;</code></td>
</tr>
<tr>
<td>51.</td>
<td><code>Coreapplication must have just one repository context.</code></td>
</tr>
<tr>
<td>52.</td>
<td><code>&lt;/tal:assert&gt;</code></td>
</tr>
<tr>
<td>53.</td>
<td><code>&lt;tal:assert test=&quot;#relatedContext &gt; 0&quot;&gt;</code></td>
</tr>
<tr>
<td>54.</td>
<td><code>It must have at least one related content in each related context.</code></td>
</tr>
<tr>
<td>55.</td>
<td><code>&lt;/tal:assert&gt;</code></td>
</tr>
<tr>
<td>56.</td>
<td><code>&lt;tal:assert test=&quot;#relatedContent == #data&quot;&gt;</code></td>
</tr>
<tr>
<td>57.</td>
<td><code>Each related content must have an associated data.</code></td>
</tr>
<tr>
<td>58.</td>
<td><code>&lt;/tal:assert&gt;</code></td>
</tr>
<tr>
<td>59.</td>
<td><code>&lt;/tal:template&gt;</code></td>
</tr>
<tr>
<td>60.</td>
<td><code>&lt;/tal:tal&gt;</code></td>
</tr>
</tbody>
</table>

Listing 2. TAL template example.

### 4.2 Padding Document Example

For generating new “CoreApplications”, Ag1 is a Lua component that searches in the repositories the correlated videos (based on their metadata), creating the NCL padding document of Listing 3.

In Listing 3, the “padding.generatedByAg1” composition indicates to the TP template processor the template to be followed. Note in the listing how the media and composition objects are related with corresponding template types through their *class* attributes.
4.3 Application Reconstruction

As aforementioned, TF is a template processor targeting NCL documents. Its open-source Lua code can be downloaded from [28].

Based on the new application generated by TF, the Filter Lua component just send several NCL editing commands [16] stopping the “CoreApplication” presentation, removing all child elements of <context id=“CoreApplication”> element, except the two image buttons and the context collecting the correlated videos (but without child elements). Then several NCL editing commands add new child elements to <context id=“CoreApplication”> element. Finally, this context is restarted using another NCL editing command.

5. CONCLUSIONS

The architecture instantiation presented in Section 3 has been intensively and successfully used for developing dynamic DTV application in the Brazilian SAGGA (Support to Automatic Generation of Ginga Applications) project [29]. Indeed, the architecture presented in Section 4 is the first application developed in SAGGA, with minor simplifications. In this application, the design of the Ag1 modifier agent encompassed the developing of a search machine which was not trivial, since there were very few metadata to be consulted in files of the Zappiens mentioned site.

All SAGGA dynamic applications have structure and content that must be in line with some explicitly defined constraints, usually established by TV broadcasters, to promote coherent application branding. The proposed architecture has supported quite well the design of these applications, enabling content producers to define and follow the same design pattern that, in addition, are more usable for those who view and interact with different applications using the same TAL template. Moreover, constraint-based authoring reinforced by the architecture promotes reuse, making the modifier agents design much easier and structured then when they are built without any structured design, since the converter complexity is in the TAL processor, which is the same for all kind of target language application.

It should also be stressed that the architecture allows for event-based behavior change in applications, which has been very suited to live events and live content generation.

In SAGGA, the opening CoreApplication is the one that Ag1 modifier agent must build when it receives the first “initData” value in its “Ag1entry” property, as discussed in Section 3.1. The “Ag1” Lua element of the reflexive NCL application of Listing1 is generic and specific wizards have been designed to help authors in defining the opening parameters, and to choose the template to be used by the TAL processor. For example, in the application of Section 4, the wizard only asks authors to enter the first video URI and the template URI to be used.

Given the similarities of these specific wizards, we have just started a work of developing a graphical authoring view (a plug-in) for Composer (an extendable NCL authoring environment) [30] to easy the design of general dynamic DTV applications that follows the architecture and distribution of Figure 3.a.

The next step in SAGGA is to use the architecture for creating health, educational and government applications. Templates will play a very important role in these application domains. The Brazilian public TV broadcaster has started its first steps in this direction.

Finally, until now we have concentrated our efforts in client-side dynamic content generation, following the distribution of modifier agents depicted in Figure 3.a. In a near future work we are planning to explore the server-side solution presented in Figure 3.b in developing social inclusive applications in cooperation with Brazilian public TV broadcasters. Of course the architecture presented in Figure 3.c is the one that motivate us most, especially when the server processing is performed by a proxy server of some social network. We have already started a work in this direction using Ginga multiple device support [16].

6. ACKNOWLEDGMENTS

Our thanks to CNPq and MCT, Brazilian research grant organizations, for the financial support to this work. We would like also to thank the TeleMídia Lab team who provided a thoughtful discussion of this work. We would like also to thank the reviewers that help to clarify several issues.

7. REFERENCES


[28] Available at: http://www.telemidia.puc-rio.br/?q=en/projetoSAGGA

[29] Available at: http://www.telemidia.puc-rio.br/?q=en/projetoSAGGA