TAL Processor for Hypermedia Applications

Carlos de Salles Soares Neto[1, 2] csalles@deinf.ufma.br
Hedvan Fernandes Pinto[1] hedvan@laws.deinf.ufma.br
Luiz Fernando G. Soares lfgs@inf.puc-rio.br

[1] Departamento de Informática – UFMA
Av. dos Portugueses, Campus do Bacanga
São Luís/MA – 65080-040 – Brasil
0055-98-3301-8224

[2] Departamento de Informática – PUC-Rio
Rua Marquês de São Vicente, 225
Rio de Janeiro/RJ - 22453-900 – Brazil
0055-21-3527-1500 Ext: 4330

© 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/10.1145/2361354.2361369.

TAL Processor for Hypermedia Applications

Carlos de Salles Soares Neto[1] Hedvan Fernandes Pinto[1] Luiz Fernando G. Soares[2]
csalles@deinf.ufma.br hedvan@laws.deinf.ufma.br lfgs@inf.puc-rio.br

[1] Departamento de Informática – UFMA
Av. dos Portugueses, Campus do Bacanga
São Luís/MA – 65080-040 – Brasil
0055-98-3301-8224

[2] Departamento de Informática – PUC-Rio
Rua Marquês de São Vicente, 225
Rio de Janeiro/RJ – 22453-900 – Brasil
0055-21-3527-1500 Ext:4330

ABSTRACT

TAL (Template Authoring Language) is a specification language for hypermedia document templates. Templates describe application families with structural and semantic similarities. In TAL, templates not only define design patterns that applications must follow, but also constraints on the use of these patterns. A template must be processed together with a padding document giving rise to a new document in some specification language, called target language. TAL supports the description of templates independently of the languages used to specify target and padding documents. Usually a specific processor is required for each target language and for each padding document used. This paper concerns TAL processors. However, we should note that the proposal can be easily extended to any other solution used to define templates. Any pattern language and any language used to define constraints could be used instead of TAL. The TAL processor architecture is general and it is discussed when presenting the processor framework. As an instantiation example, an implementation of a TAL Processor targeting NCL (the declarative language of Ginga DTV middleware) is examined, and also another one targeting HTML-based middleware. The use of wizards for defining padding documents is also discussed in the examples of the proposed architecture instantiation.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Processors - *Code generation, Interpreters, Parsing*.

General Terms

Documentation, Standardization, Languages, Verification.

Keywords

Digital TV Applications, iDTV, Nested Context Language, NCL, TAL, Ginga.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DocEng'12, September 4–7, 2012, Paris, France.
Copyright 2012 ACM 978-1-4503-1116-8/12/09...$15.00.

1. INTRODUCTION

Hypermedia documents usually share common design patterns[1]. To take profit of this characteristic, TAL (Template Authoring Language) [1] has been conceived to allow for developing common templates to be followed by applications. We call family of applications to the set of applications that follow the same specific set of templates.

However, it is not sufficient to define common design patterns that applications must follow. Sometimes we also need to set a series of constraints on the design pattern uses. In several situations, template authors are different from application authors, and the first want not only to be assured that their design patterns will be followed but also that some add-ons will not be allowed. For example, a particular application provider can require that every application it transmits must have its logo in the right upper corner of the screen. However, in addition, it can require that no other logo may be present in the application. Taking these scenarios into account, TAL extends the usual template concept to define not only common design patterns but also constraints on their uses.

Nevertheless, to guarantee that the template will be strictly followed, the final desired application must be checked against the template specification. This is one of the main roles of a *template processor*.

A *padding document* must fill at least the blanks (hot spots) of a template. Then the template must be processed together with the padding document giving rise to a new document in some specification language, called *target language*. Ideally, the padding document is written in any language understood by the template processor. Usually, a specific processor is required for each target language and for each padding document used.

We should stress that the final document is generated by the template processor. However, the process is completed only if the padding document does not deviate from the template specification. Figure 1 illustrates the process using TAL templates.

[1] We employ 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.

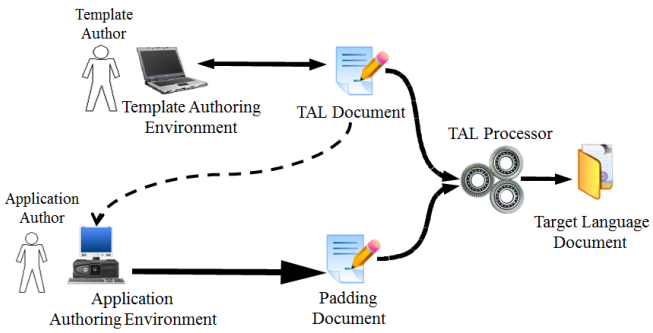

Figure 1. TAL Processing Flow.

In TAL, templates group a set of design patterns for hypermedia compositions. Hypermedia compositions include media objects and other hypermedia compositions, recursively, in addition to relationships (usually temporal relationships) among these elements. Media objects contain data to be processed and presented when their parent hypermedia compositions run. They also contain set of properties to control their presentations, as for example, the positioning of the exhibited content. Hypermedia compositions may implicitly define relationships among their child elements, as is the case of "par" and "seq" SMIL [2] containers with their embedded temporal semantics. But they can also have relationships explicitly defined, as is the case with NCL [3] links. In either case, hypermedia compositions encapsulate semantic relationships among objects. Note that even with languages that do not support composition abstraction, the whole body of the document denotes a composition. In other words, the concept of composition still prevails, although not allowing composition nesting.

Therefore TAL template is an open-composition that defines a family (a set) of compositions. TAL is independent of any authoring language used to specify hypermedia applications that can benefit from its templates to define a unique member of the composition family. TAL processors are in charge of generating this unique member, assuring that it is in agreement with the template.

This paper concerns TAL processors. However, we should note that the proposal can be easily extended to any other solution used to define templates. Any pattern language and any language used to define constraints could be used instead of TAL. The processor architecture proposed is general. It is also import to reiterate that the proposal is also independent of the target language used for the application specification.

As an instantiation case, a TAL processor implementation targeting NCL (the declarative language of Ginga DTV middleware [4]) is discussed, and also another one targeting HTML-based DTV middleware. The continuation of this paper briefly presents, in Section 2, some related work. In Section 3, TAL is overviewed. The TAL Processor architecture is discussed in Section 4. Section 5 presents two processor implementations having NCL as target language: one in which the padding document is obtained through a graphical wizard, and another one also using NCL as the padding document language. In addition, Section 5 discusses an implementation for HTML-based target languages. Finally, Section 6 presents some conclusions.

2. RELATED WORK

There are many good reasons for template-based development. First, templates promote coherent application *branding*, enabling content producers to define and follow the same hypermedia-application pattern. Second, as a consequence of having hypermedia presentations following the same interface patterns, thanks to a common source template, hypermedia applications can be more usable for those who view and interact with different documents of the same family. Third, template-based authoring promotes reuse, allowing authors to concentrate on filling out only the blanks that make a particular document unique within the family to which it belongs. Finally, templates can also encode domain concepts across related applications, creating a specific vocabulary and defining a set of constraints on this vocabulary, to be followed by all documents of a given family.

Some authoring tools are based on template approach. PageJokey [25] describes templates as classes of components to be embedded on target documents. It focuses mainly on application design and layout characteristics. Templates apply variables to denote what differ from one instance to another. GRiNS [26] is another tool that can use templates. It allows for using sample code as basis for a new document and provides graphical abstractions to help users to customize new document instances.

Several hypermedia applications embed common design patterns. However, to the best of our knowledge, all structure-based hypermedia languages (NCL, SMIL, SVG, etc.) fail to let authors create compositions with unspecified internal content or unspecified relationships. However, extension languages have been defined to increase the facilities of those languages in line with design patterns principles, like SMIL Timesheets [5] aiming at allowing any language to incorporate the XML elements and attributes of the SMIL temporal control modules. In this section we focus only on the process (and processor) used to define the resulting application when these extension languages are used.

Timesheet.js [6] is a Javascript library that incorporates SMIL Timesheets in HTML5 [7] or SVG [8] documents. This library processes the embedded SMIL Timesheets using the JavaScript engine present in browsers. After interpreting the temporal relationships, the library estimates each media duration and creates time containers to control each media execution time. The library architecture can be extended with new functions or Javascript libraries, if greater presentation control or if the inclusion of new temporal behavior is needed. Different of TAL, it is not possible to specify constraints on the application structure and behavior but only adding temporal relationships.

LimSee 3 [9] is a template-oriented authoring tool for multimedia documents. New markups are added to the document language, indicating where changes should be made when instantiating the final document. The instantiation may be incremental, and each time a document area is filled, the template markups are removed or altered to represent the new document configuration. In this approach there is not a clear separation between the template and the multimedia document, which reduces the possibility of reuse.

A model-driven approach for DTV application, called StoryToCode is presented in [10]. The tool makes use of template-based authoring concepts to standardize the structure of

application requirements, to simplify the development process and to reduce the rework when coding interactive applications. The StoryToCode development process starts from an abstract template to which successive transformations are applied, in order to obtain the final hypermedia application. These successive transformations require processors at various abstraction levels, which can lead to a high cost development environment. However, the approach allows for having target document in different languages, simply by adapting the step-by-step processors.

XTemplate [11] is the predecessor language of TAL. However, unlike TAL, the XTemplate 3.0 version was developed to a specific target hypermedia language, the NCL. XTemplate requires specific knowledge about XPath [12] and XSLT [13]. This demands that the padding document author, who usually is a non-expert user, understands XSLT transformation, which is not easy and desirable. On the other hand, at the expense of greater complexity for its use, the development of XTemplate processors is simple, since a generic XSLT processor can be used to help generating NCL code.

It should be emphasized the correlation between TAL processors and the generic XSLT processors, since both have similar purposes. XSLT processors focus on generic XML transformations applied to a XML-based application in order to obtain new representations in different XML-based languages. On the other hand, TAL processors combine an incomplete document specification with a template to generate a complete target application. Usually, XSLT converters focus much more on performance issues [14] than in provided facilities to adapt these processors for creating new tools. In other words, the scope of XSLT transformations is generic and does not care about its processor specialization to specific uses.

3. TAL SPECIFICATION: AN OVERVIEW

In TAL, template content is given by:

- Vocabulary: defining the allowed classes of child-objects (the components) of the template; the allowed classes 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 classes defined in the vocabulary;
- Resources: defining common instantiated child object classes that shall be inherited by all compositions that use (follow) the template;
- Relationships: defining common instantiated relation classes, relating child-object classes and resources that shall also be inherited by all compositions that follow the template.

In defining the vocabulary we are also defining the basic hierarchy imputed to the child-objects, given by the composition nesting. Child-objects of templates can be media objects or other nested compositions, as anticipated. An interface can define part of the content of a media object, or can define a child-object property, like its positioning on the screen, etc. Child composite-objects and the template itself may also have interfaces that externalize the interfaces of their internal child-objects.

Figure 2 shows two applications of the same family that we propose as examples. Both applications start presenting an invitation icon. If the icon is selected, a quiz starts presenting a series of questions. Each question is related with up to three incorrect answers and a right one, which can be selected by the color buttons of the remote control (red, green, yellow and blue). For each answer selected, a possible different message including the right solution is presented by a short period of time, followed by the next question of the quiz. Figures 2a. and 2b. show the first application (a talk show about health), in which each questions have three possible answers. Figures 2c. and 2d. show a documentary about a touristic state in Brazil; in it each question has four possible answers.

Although very simple, coding these applications is tedious, since there are many repetitive structures of relationships among their components: the selection of an option must be followed by the respective right answer message, and soon after by the new question with its possible answers. This repetitive code increases as the number of questions and options increase.

In TAL, applications are modeled based on component classes, besides component instances. Relationships applied on these classes reflect in every type instance, decreasing the authoring work load. Figure 3 shows the structural view of the template defined by the two applications of Figure 2, as modeled in TAL.

Indeed, we can have several models for the structure of this template of questions and answers. We have chosen the one in Figure 3 because it is simple and easy to be reused. The quiz family is represented by an open composition (the most external circle), containing another open composition, corresponding to each question of the quiz. So, we have here a case of a template that includes another template in its definition, as state in Listing 1. The root element is <tal> (line 1), which defines the template library. Note in line 4 how TAL specifies that a component (in the case id="subject") must follow a template (specified in the *template* attribute).

```
1.  <tal:tal id="set_of_templates">
2.   <tal:template id="quizTemplate">
    ...
4.     <tal:component id="subject"
                selects="context[class=subject]"
                template="questionTemplate"/>

19. </tal:template>
20. <tal:template id="questionTemplate">
    ...
81. </tal:template>
82. </tal:tal>
```

Listing 1. TAL template including other TAL template.

(a) (b)

(c) (d)

Figure 2. Application Examples.

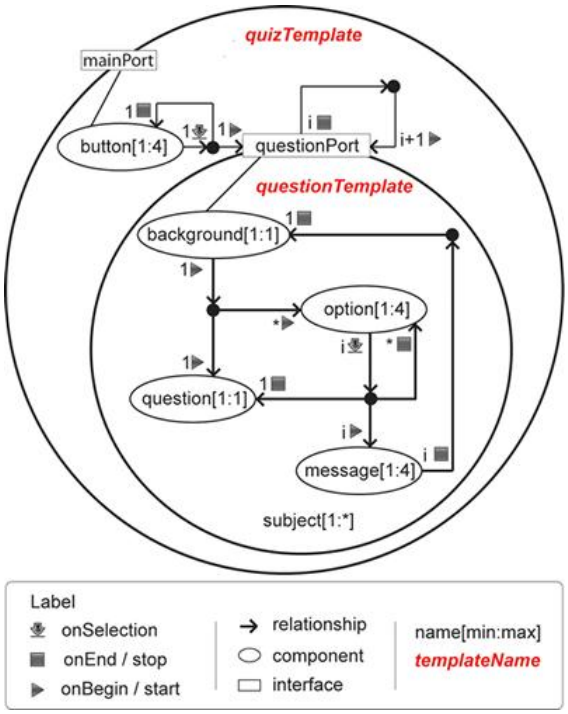

Figure 3. Template for the Applications in Figure 2.

In Figure 3, ellipses correspond to components of the templates; indeed, component classes. Every component class has a name followed by its cardinality: the minimum and maximum number of instances allowed for the class. For example, the quizTemplate must have one and only one "button" component (the invitation icon of the application quiz), and must have at least one "subject" component (the quiz's question), which follows the "questionTemplate". In Figure 3, the "background" component is associated to the background image over which the quiz question and its color button options (instances of the "option" component class) are placed. Depending on the selected option, a message, instance of the "message" component class, is presented. In TAL, <component> elements define component classes. Components can be media objects or nested compositions.

Constraints on the classes defined in the template vocabulary are specified using <assert>, <report> and <warning> elements. These elements establish constraint rules similarly to Schematron [15]. In all three elements the test attribute specifies the logical test to be evaluated. The error or warning message is defined in the content of these elements. The <assert> element requires the test evaluation returns "true", otherwise its error message should be presented. The <report> element is similar but requires that the test be evaluated as "false" to not exhibit its error message. The <warning> element requires that the test be evaluated as "false" to show its warning message. When an error

message occurs, the template evaluation is aborted and no final document is generated by the template processor. On the other hand, a warning message does not stop the template processing.

Listing 2 adds components and constraints to the templates of Listing 1. Templates and classes defined in TAL may use selectors similar to CSS selectors [16]. The selector role is to identify which elements of the padding document must be processed in agreement with the class or template they are associated. Line 3 defines the component class (the media object class) that will be associated to the invitation icon of the application. The selector indicates that elements of the padding document that have their *class* attributes with value equal to "button" are instances of this class. Similarly, line 22, 23, 24 and 25 define the component classes for the template "questionTemplate": the set of questions, the set options, the background and the set of messages, respectively.

```
1.  <tal:tal id="template">
2.   <tal:template id="quizTemplate">
3.     <tal:component id="button"
         selects="media[class=button]"/>
4.     <tal:component id="subject"
                 selects="context[class=subject]"
                template="questionTemplate"/>
    ...
13.    <tal:assert test="#button==1">
14.      It must have one BUTTON element.
15.    </tal:assert>
16.    <tal:assert test="#questionTemplate>=1">
17.      It must have at least one
         QUESTIONTEMPLATE element.
18.    </tal:assert>
19. </tal:template>
20. <tal:template id="questionTemplate">
     ...
22.    <tal:component                id="question"
                 selects="media[class=question]"/>
23.    <tal:component                  id="option"
                 selects="media[class=option]"/>
24.    <tal:component              id="background"
               selects="media[class=background]"/>
25.    <tal:component id="message"
                 selects="media[class=message]"/>
    ...
66.    <tal:assert test="#background==1">
67.      It must have only one BACKGROUND element.
68.    </tal:assert>
69.    <tal:assert test="#option==#solution">
70.      The number of OPTION elements must be the
         same of SOLUTION elements.
71.    </tal:assert>
72.    <tal:assert test="#option<=4">
73.      The number of OPTION elements must be at
         most four.
74.    </tal:assert>
75.    <tal:assert test="#option>=1">
76.      It must have at least one option element.
77.    </tal:assert>
    ...
81. </tal:template>
82. </tal:tal>
```

Listing 2. Components and constraints in TAL.

In lines 13 to 18 of Listing 2 we can see two <assert> elements, establishing constraints on the cardinality of the "button" and "questionTemplate" component classes, in this order. In lines 66 to 77 we can see the constraints on the cardinality of the components of the "questionTemplate".

Figure 3 shows several causal relationships among components. For example, the one linking the "button" establishes that when this component is selected, its presentation must be stopped and the first question must be presented. In TAL, relationships are defined by using <link> elements. As relationships can be established among classes, <forEach> child elements can be used to iterate on these class instances. In causal relationships, conditions defined on events must be satisfied in order to trigger action on events.

The current version of TAL has the following event types: presentation event, which is defined by the presentation of a subset of the information units (an interface) of a media object, or, in case of compositions, the presentation of the information units of any object inside it; selection event, which is defined by the selection of a subset of the information units of a media object being presented; attribution event, which is defined by the attribution of a value to a property (an interface) of an object.

As described in the previous paragraph, events are defined on interfaces of child components of a composition. Interface classes, both for the template itself and for its child components, are defined by <interface> elements. Every child component of a template and the template itself has an interface defined by default representing the whole content of the object. Except for the "questionTemplate", this is the case all other components in Listing2, which have only this implicitly defined interface class.

An <interface> element for a composition, in particular the template itself, can also define mappings to interfaces of child-objects of the composition. As mappings can be established among classes, <forEach> child elements can be used to iterate on these class instances. Listing 3 specifies the interface of "questionTemplate" template. Note that as the mapping is to the unique instance of the "background" component class, the mapping is established by using the *component* attribute of the <interface> element, without needing to define the <forEach> child element.

```
21.    <tal:interface id="questionPort"
         selects="port[class=questionPort]"
         component="background[1]"/>
```

Listing 3. Interface for the open-composition.

There are two relationships defined by the "quizTemplate", as shown in Listing 4. Lines 5 to 7 define the first relationship. It establishes that the selection of the invitation icon stops its presentation, and starts presenting the first "subject" instance, that is, the first question. The second relationship, in lines 8 to 12, establishes that when each "subject" instance finishes, the next instance of this component class must be started. Note in Listing 4 that the <forEach> in line 9 is responsible for the iteration on the set of "subject" instances.

```
5.     <tal:link id="beginQuiz">
6.       onSelection button[1] then start subject
    [1]; stop button[1] end
7.     </tal:link>
8.     <tal:link id="nextQuestion">
9.         <tal:foreach
              instance="subject" iterator="i">
10.           onEnd subject[i] then start
                                 subject[i+1] end
11.        </tal:forEach>
12.    </tal:link>
```

Listing 4. Relationships defined in the "quizTemplate".

Finally, there are six relationships defined by the "questionTemplate", as shown in Listing 5. The first, in lines 27 to 33 establishes that when the "background" component (the one that starts when the "questionTemplate" starts) begins, then the first question ("question[1]") must start, together with all its answering options. The <forEach> in line 29 is responsible for the iteration on the set of "option" instances. The second relationship defines that when the "option[1]" is selected with the RED remote control key, then the question presentation must be stopped, the "message[1]" must be presented (as specified in line 35), and that every "options" must be stopped (as specified in lines 36 and 37). The third, fourth and fifth relationships are similar to the second one, when "option[2]", "option[3]", and "option[4]" are selected, respectively with GREEN, YELLOW and BLUE remote control keys. The sixth relationship, in lines 62 to 65, establishes that when any "message" instance ends, the background" must be stopped, and, in consequence, the parent "subject" instance. In agreement with the link defined in line 8 to 12 of Listing 4, the next question must starts, until the last one is presented.

```
26.    <tal:relation id="link" selects="link"/>

27.    <tal:link id="beginQuestion">
28.        onBegin background[1] then start
    question[1];
29.        <tal:forEach instance="option"
                              iterator="i">
30.            start option[i];
31.        <tal:foreach>
32.        end
33.    </tal:link>
34.    <tal:link id="option1Selection">
35.        onSelection option[1] with key = "RED"
    then start message[1]; stop question[1];
36.        <tal:forEach instance="option"
                              iterator="i">
37.            stop option[i];
38.        <tal:foreach>
39.        end
40.    </tal:link>
41.    <tal:link id="option2Selection">
42.        onSelection option[2] with key =
    "GREEN" then start message[2]; stop
    question[1];
43.        <tal:forEach instance="option"
                              iterator="i">
44.            stop option[i];
45.        <tal:foreach>
46.        end
47.    </tal:link>
48.    <tal:link id="option3Selection">
49.        onSelection option[3] with key =
    "YELLOW" then start message[3]; stop
    question[1];
50.        <tal:forEach instance="option"
                              iterator="i">
51.            stop option[i];
52.        <tal:foreach>
53.        end
54.    </tal:link>
55.    <tal:link id="option4Selection">
56.        onSelection option[4] with key =
    "GREEN" then start message[4]; stop
    question[1];
57.        <tal:forEach instance="option"
                              iterator="i">
58.            stop option[i];
59.        <tal:foreach>
60.        end
61.    </tal:link>
62.    <tal:link id="endQuestion">
63.      <tal:forEach instance="message"
                              iterator="i">
64.        onEnd message[i] then stop
                        background[1] end
65.      </tal:forEach>
66.    </tal:link>
```

Listing 5. Relationships defined in the "questionTemplate".

It is important to stress that TAL also allows for defining constraints on relations. Note that in line 26 of Listing 5 a relation class is defined for relationships defined by <link> elements. This allows for defining the constraint of Listing 6.

```
78.    <tal:assert test="#link==0">
79.      There cannot be any <link> element on the
         application apart from those defined by
         the template.
80.    </tal:assert>
```

Listing 6. Constraints on relations.

Listing 1 to 6 define our template example. Finishing this section, Listing 7 gives the padding document body for the application of Figure 2.a. and 2.b (just one question is shown). Note that the padding document is all that an application author must fulfill.

```
    …
30.  <body template="template.tal#quizTemplate">
31.    <port id="pinit" component="initButton"
                          class="mainPort"/>
32.    <media id="initButton" … class="button"/>
33.    <context id="quizSubject1" class="subject">
34.      <!-- port to background at template -->
35.        <media id="backgroundS1"…
               class="background"/>
36.        <media id="questionS1" …
               class="option"/>
37.        <media id="option1S1" …
               class="option"/>
38.        <media id="option2S1" …
               class="option"/>
39.        <media id="option3S1" …
               class="option"/>
40.        <media id="message1S1" …
               class="message"/>
41.        <media id="message2S1" …
               class="message"/>
42.        <media id="message3S1" …
               class="message"/>
43.    </context>
         …
85.  </body>
86. </ncl>
```

Listing 7. Padding document example.

4. TAL PROCESSOR ARCHITECTURE

As aforementioned, TAL specifies its templates independent of the target document language and independent of the language used in defining the padding document. This requires a template processor framework with hot spots to be specialized to each input and output language. In agreement with this principle, TAL Processor architecture is divided in three modules, as presented in Figure 4. These modules and their components follow the analysis-synthesis paradigm usually adopted in modern compilers [17]. In Figure 4, components in dark grey are the hot spots.

The first module is the Interpreter. The Padding Document Parser component initiates the whole process converting the

padding document to an internal data structure, called Padding Document Model in Figure 4. This intermediate data structure is easier to be handled when generating the final coding, and it is completely independent of the padding document language.

The Padding Document Model is based on the NCL Raw profile [18]. This conceptual model has been chosen because it can represent the majority of hypermedia declarative language models. During the parser process, XML elements of the padding document that have *template* attributes are used to identify which templates TAL Parser component must use.

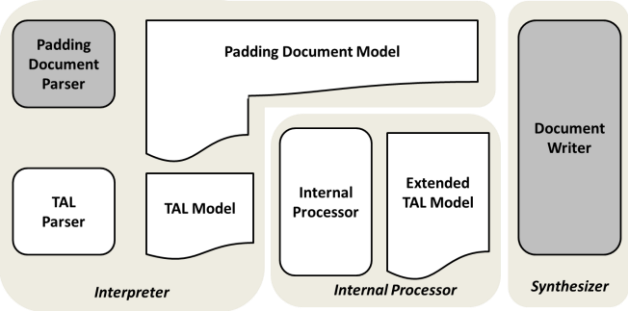

Figure 4. TAL Processor Architecture.

The TAL parser component interprets the templates referred by the padding document. The component is responsible for validating template syntaxes. Validation cannot be performed only using XML Schema [19, 20], since TAL presents the particularity of combining three additional distinct syntaxes within the same TAL document: CSS-like syntax to bind components to padding document entities; Schematron-like for constraint definitions; and TAL own syntax for relationship and interface mapping definitions. After checking if a document complies with the TAL Type Definition, TAL Parser generates an internal data structure, called TAL Model in Figure 4.

TAL Model is an intermediate data structure that reorganizes the different parts of TAL documents to make them easier to be handled in generating final coding. TAL Parser gets TAL *selects* attributes, TAL *Constraints* and TAL *Relationships*, processes them, and generates the corresponding internal structures for each one.

Selectors are converted into functions, responsible for searching the corresponding elements in the Padding Document Model that refers to the selectors. Constraints are converted to functions to evaluate the cardinality of components defined in the template vocabulary. Relationships are transformed into other data structures ready to be easily combined with information defined in the Padding Document Model.

Based on the Padding Document Model and the TAL Model, the Internal Processor component validates the template constraints, process all relationships and all interfaces defined in the template, and uses the functions created when processing the selectors to retrieve the corresponding structures from the Padding Document Model.

In validating the constraints, if any of them is not satisfied, the whole process is stopped. The message defined in the invalid constraint is then presented to the TAL Processor user. In case of "warnings", as defined in Section 3, the message is also presented to the user, but without stopping the processing.

As discussed in Section 3, <relation>, <link> and <forEach> TAL elements are used to define causal relationships. Some of these relationships are transformed in new ones by the Internal Processor component. New relationships are generated when <forEach> elements are found in relationships of the TAL Model that refer to elements in the TAL vocabulary. The example presented in Figure 5 can help in understanding the process.

The top of Figure 5, denoted by (1), represents the relationship as originally defined in TAL. Let us now assume that the padding document has three elements referring (using TAL selectors) to the "subject" component class. Based on the Padding Document Model and the description of the TAL relationship in the TAL Model, the Internal Processor component resolves the iteration creating three new relationships as depicted in part (2) of Figure 5.

Similar converting procedure is employed by the Internal Processor component in transforming TAL <interface> elements having <forEach> elements as children.

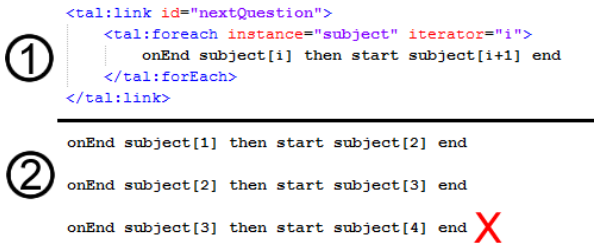

Figure 5. Converting TAL relationships.

If the Internal Processor component comes across an invalid index value when solving <forEach> expressions, it stops the generation of the relationship (or interface) and ignores the invalid relationship (or interface). Figure 5 shows a relationship generated by referencing an inexistent element (exemplified by subject [4]). In this case, the relationship is discarded.

In the Extended TAL Model generated by the Internal Processor, constraints are removed and selector functions are replaced by the set of elements of the Padding Document Model that they refer. New generated relationships and interfaces are added, replacing the old ones, and formatted according to the structure of the NCL Raw Profile.

The Synthesizer Module has just one component: the Document Writer. This component creates the target document in a given specification language based on the Padding Document Model and the Extended TAL Model.

Note in Figure 4 that components filled in dark grey are those that depend on the padding and target document languages. Note also that the architecture is loosely coupled, in the sense that it is possible to change the Padding Document Parser and the Document Writer components and reuse all other elements. Therefore, in order to incorporate new target languages to a TAL Processor, only hot spots of the Document Writer component must be filled. Likewise, in order to incorporate other input padding languages, or include communication with other tools for data entry (as for example wizards, IDEs, etc., as discussed in

the next section), only hot spots of the Padding Document Parser component must be filled. Section 5 explores this feature instantiating the architecture for two target languages and using two ways of defining the padding document.

5. TAL PROCESSOR INSTANCES

Tal Processor has been designed to be smoothly integrated to other tools. This integration can be done either at the front-end (Padding Document Parser component in Figure 4) and the back-end (Document Writer component in Figure 4) of TAL Processor.

Regarding the front-end integration, TAL Processor can be added to a tool that assists the construction of padding documents, possible making use of graphical abstractions to simplify the data entry process.

As for the back-end integration, the target document generated can be directly sent to a second authoring tool as its input, for example, or to a playout station.

No matter if in the front-end or in the back-end, the integration can take place either directly or indirectly. Integration is said to be direct when the integrated tool acts as a component of the TAL Processor, replacing part of its tasks. Indirect integration is through using TAL processor without interfering with the integrated tool operation, thus using it as a black box. In this second case, the tool input or output is used as a means to establish the integration.

To clarify the integration with other tools, some examples are presented in the following subsections. In 3.1, the integration of TAL Processor with a wizard tool is presented. In 3.2 the generation of NCL and HTML documents is discussed. This paper does not present examples of indirect integration since this type of integration is straightforward and does not change any TAL Processor modules.

5.1 Front-End Integration: NCLWizard

Usually, the input information of a TAL processor is defined in a document specified using the same language used to specify the target document. Listing 7 in Section 3 illustrates a Padding Document written in NCL 3.0 [3] for the template presented in the same section, resulting in the NCL application shown in Figure 2.a and 2.b.

The specification of padding documents directly in NCL or other languages requires certain degree of experience and knowledge from the application author. As a consequence, the time needed for application development can increase. Generally, content producers and writers of TV programs do not have any experience with programming languages, even with declarative ones. To enable their participation in all stages of application developments, it is necessary to use tools possibly with graphical abstractions close to their daily live perceptions.

A simple and powerful abstraction is the interface pattern known as wizard. As an example, we have done the integration of TAL Processor with the NCLWizard [21] framework, combining the ease of using templates with the clarity of wizard interfaces. In complex and repetitive tasks it is recommended to use wizards to assist users [22]. They provide a step by step, clear and direct interface, highlighting and directing their users to points that deserve their attention.

In a direct integration with TAL Processor, wizards can take over the functions of the Padding Document Parser component of Figure 4. The result is graphical interfaces used to communicate with application authors rather than the direct textual authoring of documents. Figure 6 shows two screenshots coming from the integration of TAL Processor with the NCLWizard framework. They are gotten during the Padding Document Model generation for the TAL template defined in the Section 3.

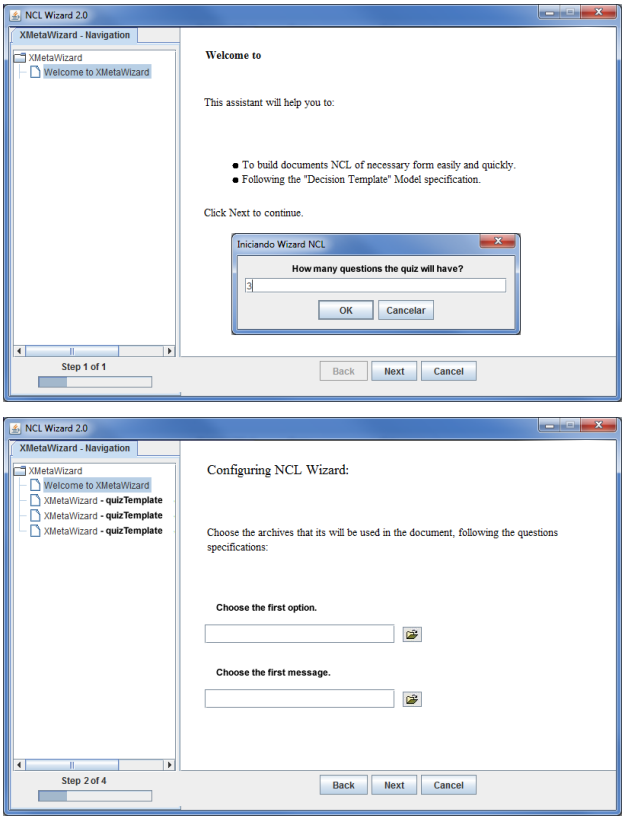

Figure 6. Wizard for the Template of Listing 1.

5.2 Creating NCL and HTML Applications

The key task performed by the Document Writer component of Figure 4 is translating the TAL Model relationships to the temporal presentation semantics of the target language conceptual model. NCL [3], for example, defines the temporal semantic by using NCL <link> elements. On the other hand, SMIL [2] use <seq> and <par> temporal containers, besides temporal attributes defined in SMIL media objects. In HTML, scripts will be necessary, and so on.

Relationships defined using TAL has a one-to-one translation to <link> entities used in NCL version 3.1. This contrast with the NCL language profile version 3.0 used in the standardized Ginga middleware [4], which requires the translation of TAL <links> to two NCL 3.0 entities: <connector> to define relations, and <link> to define relationships referring to <connector> pre-defined elements. Figure 7 shows an example: in 7 (1) we have the TAL relationship and in 7 (2) we have the generated NCL 3.0 elements.

```
(1)
onSelection button[1] then
stop video[1]; stop video[2]; stop video[3];
start video[1] end
```

```
<causalConnector id="onSelectionStartStop">
    <simpleCondition role="onSelection"/>
    <compoundAction operator="seq">
        <simpleAction role="stop" max="unbounded"/>
        <simpleAction role="start" max="unbounded"/>
    </compoundAction>
</causalConnector>
(2)
<link xconnector="conn#onSelectionStartStop">
    <bind role="onSelection" component="mthumb1"/>
    <bind role="stop" component="video1"/>
    <bind role="stop" component="video2"/>
    <bind role="stop" component="video3"/>
    <bind role="start" component="video1"/>
</link>
```

Figure 7. TAL Relationship Translated to NCL elements.

The structured organization of TAL, in which a component can contain other components, has a one-to-one correspondence to NCL <context> elements. Interfaces with mapping specifications in the Extended TAL Model have a one to one correspondence to <port> elements of NCL. The other interface types are translated to NCL <property> and <area> elements.

Indeed, TAL has been designed aiming at having NCL as its client target language. That is why it is so easy to translate the Extended TAL Model (referring to and using the Padding Document Model) into an NCL application. This makes the Document Writer component implementation very simple.

The translation from the Extended TAL Model to HTML documents is not straightforward as it is to NCL, but it is still easy to implement. The structured organization of TAL can be used by and HTML elements. However, some TAL relationships cannot be described by HTML declarative tags and may require the use of scripts. Figure 8 shows an example: in 8 (1) we have the same TAL relationship of Figure 7, and in 8 (3) we have the script generated.

```
(1)
onSelection button[1] then
stop video[1]; stop video[2]; stop video[3];
start video[1] end
```

```
(3)
document.getElementById("thumb1Link").click(
   function(e){
       e.preventDafault();
       stop("video1, video2, video3");
       start("video1");
   });
```

Figure 8. TAL Relationship Translated to ECMAScript.

Using ECMAScript library, the Document Writer component implementation becomes workable. Building this complete library is left to future work. Similar work has been done for SMIL Timesheet [5].

Table 1 shows how TAL conditions and actions are related to those of NCL and HTML. Note that there is a one-to-one correspondence between TAL and NCL, as previously mentioned. HTML relationships are based on events captured by its scripting language, especially when they handle attribution events.

Table 1. Mapping TAL conditions and actions.

	TAL	NCL	HTML
Actions	Start	start	play/show
	Stop	stop	stop/hide
	Pause	pause	pause
	Resume	resume	play/show
	Abort	abort	abort
	Set	set	Script hadling
Conditions	onEnd	onEnd	Ended
	onBegin	onBegin	Playing
	onAbort	onAbort	Abort
	onPause	onPause	Pause
	onResume	onResume	Playing
	onEndAttribution	onEndAttribution	Script hadling
	onBeginAttribution	onBeginAttribution	Script hadling

6. CONCLUSIONS

In this paper we have presented the TAL Processor architecture and how it can be easily specialized to be integrated with other application developing tools.

Currently we are working on the integration of TAL Processor with other tools that ease the specification of padding documents. In particular, we are integrating TAL as a plug-in of Composer [23]. Composer provides high-level abstractions by means of graphical views, making easier the authoring process of NCL applications in conformance with the language DTV Profile. Integration with Composer will allow a complete development cycle, in which Composer can be used to create NCL padding documents. In this case, TAL Processor will be used for the generation of NCL applications based on predefined templates. These NCL applications could then be enhanced using other Composer plug-ins. Moreover, since TAL allows template nesting, the final document generated by TAL Processor can feedback the whole process as a new padding document, and so on, until the DTV application is finally ready. In addition, as there are playout-station plug-ins for Composer, TAL processor could also take profit of this facility to transmit applications besides the automatic publication of produced applications in NCL specialized sites, like the one in www.club.ncl.org.br.

Several experts in NCL language have attested the usefulness of TAL in speeding the development of DTV applications, in special applications with dynamically generated content [24]. A usability study is planned as future work to validate the use of TAL by non-expert programmers and obtain further evidence on the perception of users about the ease and applicability of TAL.

TAL Processor has been implemented in Lua, the scripting language of NCL. Therefore, TAL Processor is able to be executed by Ginga middleware in the client side (viewer side). Another future work targets the creation of applications (indeed padding documents) by viewers at the client side, based on pre-

defined templates. In this case, it will be possible for viewers to play the role of application authors even using the limited resources of set-top boxes. Today, some applications dynamically generated at the client side by TAL Processors have already been developed [24].

Finally, a template repository is being created for Ginga-NCL, with extensive semantic documentation (metadata) about families of applications addressed by each template. This can speed up the use of TAL and get faster the development of NCL DTV applications.

7. ACKNOWLEDGMENTS

This research has been partially supported by CNPq and MCT.

8. REFERENCES

[1] Soares Neto, C. S.; Soares, L. F. G.; and Souza, C. S.. TAL - Template Authoring Language. To be published in *Journal of Brazilian Computer Science*. 2012.

[2] W3C. *Synchronized Multimedia Integration Language (SMIL 3.0) W3C Recommendation*, 2007. Available at: http://www.w3.org/TR/SMIL

[3] NBR 15606-2. ABNT NBR 15606-2. *Digital terrestrial television – Data coding and transmission specification for digital broadcasting – Part 2: Ginga-NCL for fixed and mobile receivers – XML application language for application coding*. September, 2007.

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

[5] W3C. *SMIL Timesheets 1.0. W3C Working Draft*. Available at http://www.w3.org/TR/timesheets/. 2011.

[6] 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, Mountain View: United States. 2011.

[7] W3C. *HTML5 A vocabulary and associated APIs for HTML and XHTML* W3C Working Draft. Available at: http://www.w3.org/TR/html5/. 2011.

[8] W3C. *Scalabe Vector Graphics*. W3C Recommendation. 2009. Available at: http://www.w3.org/TR/SVG11/.

[9] R. Deltour and C. Roisin. The limsee3 Multimedia Authoring Model. In: *Proceedings of the ACM Symposium on Document Engineering*, DocEng, p. 173-175, New York: United States. 2006.

[10] Kulesza, R., Meira, S. R. L., Ferreira, T. P., Lívio, Á., Filho, G. L. S., Marques Neto, M. C., Santos, C. A. S. Uma Abordagem Dirigida por Modelos para Integração de Aplicações Interativas e Serviços Web: Estudo de Caso na Plataforma de TV Digital (in Portuguese). In: *18th Simpósio Brasileiro de Sistemas Multimídia e Web*, Florianópolis, Brasil. 2011.

[11] Santos, J. A. F., Muchaluat-Saade, D. C. XTemplate 3.0: spatio-temporal semantics and structure. In: *Multimedia Tools and Applications*. 2011. DOI 10.1007/s11042-011-0732-2.

[12] W3C. *XML Path Language (XPath) 2.0 (Second Edition)*. W3C Recommendation. 2011. Avaliable at http://www.w3.org/TR/xpath20/

[13] W3C. *XSL Transformations (XSLT) Version 1.0* W3C Recommendation. 1999. Avaliable at http://www.w3.org/TR/xslt

[14] Bittner, T. *Performance Evaluation for XSLT Processing*. Tech. Report, University of Rostock. 2004.

[15] ISO/IEC 19757-3. *Information technology -- Document Schema Definition Language (DSDL) -- Part 3: Rule-based validation – Schematron*. 2006

[16] W3C. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. 2009. Available at: http://www.w3.org/TR/CSS2/.

[17] Grune, D., Bal, H., Jacobs, C., Langendoen, K. *Modern Compiler Design*. Wiley. 2000.

[18] Lima, G.; Soares, L.F.G.; Soares Neto, C.S.; Moreno, M. F.; Costa, R. R.; Moreno, M. F. Towards the NCL Raw Profile. In: *Simpósio Barsileiro de Sistemas Multimídia e Web*, WebMedia 2010, Belo Horizonte. 2010.

[19] W3C. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C Recommendation. Available at: http://www.w3.org/TR/xmlschema11-2/. 2012.

[20] W3C. *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. W3C Recommendation. Available at: http://www.w3.org/TR/xmlschema11-2/. 2012.

[21] Soares Neto, C. S.; Soares, L. F. G. Autoria orientada a arquétipos para TV digital: uma abordagem restritiva e direcionada (in Portuguese). In *Proceedings of the Conferencia Latino Americana de Informatica*, CLEI 2008, Santa Fe. 2008.

[22] Parvan, M., Maurer, M., Lindermann, U. Software Wizard Design for Complexity Management Application. In: *International Design Conference*, Design 2010, Dubrovnik, Croatia, May 2010.

[23] Lima, B. S., Soares L. F. G., Moreno, M. F. Considering Non-functional Aspects in the Design of Hypermedia Authoring Tools. In: *ACM Symposium on Applied Computing*, SAC 2011, Taiwan. 2011.

[24] Soares, L.F.G.; Soares Neto, C.S.; Sousa, J.G. *Architecture for DTV Dynamic Applications with Content and Behavior Constraints*. Technical Report. Informatics Department of PUC-Rio. Rio de Janeiro. January, 2012. ISSN 0103-974. Submitted to DocEng 2012.

[25] S. Fraïssé, J. Nanard, and M. Nanard. Generating hypermedia from speci cations by sketching multimedia templates. In *ACM Multimedia 96*, pages 353-364, Boston, MA, EUA, 1996.

[26] Bulterman, D. C. A., Rutledge, L., Hardman, L., Jansen, J. and Mullender, K. S., GRiNS: an authoring environment for web multimedia, *World Conference on Educational Multimedia, Hypermedia and Educational Telecommunications*, ED-MEDIA 99, Seattle, WA, USA, 1999.