Pontifícia Universidade Católica
do Rio de Janeiro

Bruno Xavier Leitão

Web Templates Support in NCL Player

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Sérgio Colcher

Rio de Janeiro
September 2019

PUC-Rio - Certificação Digital Nº 1612822/CA

Pontifícia Universidade Católica
do Rio de Janeiro

Bruno Xavier Leitão

Web Templates Support in NCL Player

Dissertation presented to the Programa de Pós-graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee.

Prof. Sérgio Colcher
Advisor
Departamento de Informática – PUC-Rio

Prof. Carlos de Salles Soares Neto
Universidade do Maranhão – UFMA

Prof. Noemi de la Rocque Rodriquez
Departamento de Informática – PUC-Rio

Álan Livio Vasconcelos Guedes
Pesquisador Autônomo

Rio de Janeiro, September 20th, 2019

PUC-Rio - Certificação Digital Nº 1612822/CA

All rights reserved.

Bruno Xavier Leitão

The author is undergraduaded in Computer Engineering in Pontifícia Universidade Católica do Rio de Janeiro, PUC-Rio, in 2015.

PUC-Rio - Certificação Digital Nº 1612822/CA

Bibliographic data

Leitão, Bruno Xavier

Web Templates Support in NCL Player / Bruno Xavier Leitão; advisor: Sérgio Colcher. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2019.

v., 55 f: il. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

Ginga; NCL; Multimídia; Autoria baseada em Templates; Reuso; Autoria Hipermidia I. Colcher, Sérgio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

PUC-Rio - Certificação Digital Nº 1612822/CA

To my parents, for their unconditional support and encouragement that helped me achieve this goal.
To my boring brother, for always stays at my side.
To my relatives, for their kind lover and preoccupation.
And to my grandparents Lourdinha and Francisco (*in memoriam*) ————-

Acknowledgments

First of all, I would like to thank my advisor, Prof. Sérgio Colcher, for always believe and for never give up on me after the difficulties I faced.

To my friends and colleagues from TeleMidia for the excellent talks, cakes and coffee during the launch time, in special, to Álan for the discussions about this work and for helping me out reviewing the text.

To all my relatives for being always on my side and cheer me up when I needed. To my little brother, clonezinho André, for sharing all the moments, the goods and the bad ones.

To my long-date friends, for helping me chill and distract me when I requested. To my friends and professors from the gym to encourage me to keep my physical health.

The study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

PUC-Rio - Certificação Digital Nº 1612822/CA

Abstract

Leitão, Bruno Xavier; Colcher, Sérgio (Advisor). **Web Templates Support in NCL Player**. Rio de Janeiro, 2019. 55p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Ginga middleware and NCL language are standards for multimedia applications authoring for Digital TV. Some studies have indicated that NCL language is highly verbose. Such a factor increases the possibility of coding errors introduced by application authors. These mistakes can be reduced by reusing repeated elements on the code. In the multimedia field, is common to use templates to achieve such a goal. Templates describe a family of logically structured documents. Template language insertion ends up reducing the number of lines of codes written and thus make the final document less error-prone. On the web, the scenario is common the template usage in HTML development. In this scenario, developers commonly use specific templates engines that can even run on the client-side, such as Jinja2 and Mustache. This work aims at bringing web templates support for the NCL development. By running on the client i.e., Ginga, developers can provide adaptable template-based content to developed applications

PUC-Rio - Certificação Digital Nº 1612822/CA

Keywords

Ginga; NCL; Multimedia Template-oriented authoring; Reuse; Hypermedia Authoring.

Resumo

Leitão, Bruno Xavier; Colcher, Sérgio (orientador). **Suporte a Templates Web no Player NCL**. Rio de Janeiro, 2019. 55p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O middleware Ginga e a linguagem NCL são padrões para autoria de aplicações multimídia para TV Digital. Alguns estudos concluiram que a linguagem NCL apresenta um alto grau de verbosidade. Tal fator aumenta a possibilidade do autor da aplicação escrever um código errado. Uma maneira para reduzir tais chances de erro consiste em reusar elementos que se repetem na estrutura do código. Na literatura, dentro do campo de multimídia, é comum usar templates pra tal finalidade. Templates descrevem famílias de documentos com estrutura lógica em comum. No contexto da web é comum a utilização de templates na criação de páginas HTML. Nesse caso, os templates podem ser executados tanto no lado cliente como no lado servidor. A proposta desse trabalho é trazer linguagens de template web e suas ferramentas de execução (*engines*) para o universo da TV Digital. A execução rodando no cliente (Ginga) permite a seus desenvolvedores fornecerem applicação baseadas em templates com conteúdo adaptável.

PUC-Rio - Certificação Digital Nº 1612822/CA

Palavras-chave

Ginga; NCL; Multimídia; Autoria baseada em Templates; Reuso; Autoria Hipermidia

Table of Contents

PUC-Rio - Certificação Digital Nº 1612822/CA

List of Figures

PUC-Rio - Certificação Digital Nº 1612822/CA

List of Tables

PUC-Rio - Certificação Digital Nº 1612822/CA

List of Abreviations

ABNT – Associação Brasileira de Normas Tecnicas
API – Application Programming Interface
CDN – Cognitive Dimensions of Notation
DTV – Digital TV
DOM – Document Object Model
eRuby – embedded Ruby
HTML – HyperText Markup Language
IBB – Integrated Broadcast-Broadband
IPTV – Internet Protocol Television
ITU – International Telecommunication Union
JNS – JSON NCL Script
JSON – JavaScript Object Notation
JSX – JavaScript XML
NCL – Nested Context Language
OAR – Over-the-air
SBTVD – The Brazilian Digital TV System
sNCL – simple NCL
STB – set-top box
SRT – SubRip Subtitle file
PHP – Hypertext Preprocessor
TAL – Template Authoring Tool
XML – Extensible Markup Language
XSLT – eXtensible Stylesheet Language Transformations

PUC-Rio - Certificação Digital Nº 1612822/CA

PUC-Rio - Certificação Digital Nº 1612822/CA

Persistence is still the best way to achieve something in life.

Unknown Author.

1
Introduction

Digital TV (DTV) is an increment of traditional TV. In particular, it offers, besides higher quality image and sound, the ability to interact with the content using interactive applications running in the receiver device. This characteristic contributes to a much better watching television experience [1]. To enable such applications to execute, DTV systems use a standardized middleware layer. This layer focuses on software interoperability between different receivers to abstract different hardware and operating system configurations. In other words, the same application will work if it is running on conventional TV sets, high-definition devices, computers or cell phones from different vendors.

The Brazilian Digital TV System (SBTVD) Forum proposed in 2006 the Ginga middleware [2] for authoring its interactive applications. The Ginga main component is the Ginga-NCL subsystem. It supports the execution of applications developed in the Nested Context Language (NCL) [3]. Today, both Ginga and Ginga-NCL are ITU Recommendations for IPTV systems [4] and are used in Terrestrial Digital TV in many countries of South America and Africa.

NCL is based on XML[1] and comprises a domain-specific language for multimedia authoring. More precisely, it focuses on specifying multimedia applications with synchronized audiovisual media and key-based user interactions. According to Soares *et al.* [5], however, the XML syntax is verbose and error-prone. They highlighted, after conducting an usability analysis, that programming in NCL may be a hard job as applications' complexity increases. Therefore, some work aims at supporting alternative formats (i.e., syntaxes) for NCL, such as JNS [6] based on JSON (JavaScript Object Notation) [7] format. One may better support the development of NCL applications promoting code reuse.

The code reuse technique is based on taking advantage of a prior coding effort to reduce the amount of work to achieve a new accomplishment. It allows previously tested code usage, avoiding repetition and, therefore, reducing errors. In the multimedia field, it is common the use of *templates* to do so.

[1] `https://www.w3.org/XML/`

PUC-Rio - Certificação Digital Nº 1612822/CA

1.1 Templates

A *document template* describes a family of similar structured documents, i.e., applications. They are created by a *template author* and characterized for having "blanks" that must be filled in with some content according to rules and relationships. In general, these blanks are filled with information from another document called *padding* created by the *template user*. Templates isolate logical and structural information of a document from its presentation data, reducing the amount of work and error that might come out.

The system responsible for processing templates is called a *template engine*. It receives a Template document, described by some template language, together with a Padding document, and processes them to fill in the gaps, producing, as output, a new document in a final specific format. This output file may be, then, rendered by a Multimedia Player. The Figure 1.1 shows the common template engines' processing.

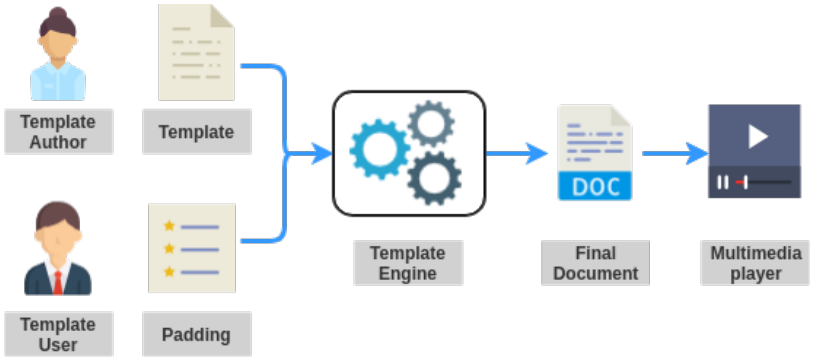

Figure 1.1: Template engines' processing

It is important to notice that this template engines' processing is not tied to happen in the same environment of the Multimedia Player. In this work, we consider two terms to denote where this processing takes place. We call *server-side* when the processing occurs before an application arrives at the Player environment. In this case, the client only receives the final document from a distribution channel. Differently, when processing runs at the *client-side*, the Multimedia Player receives both the Template and the Padding documents plus the Template Engine itself. To illustrate such difference, Figure 1.2 presents the template's execution on both server- and client-side.

To run on the client-side the Template Engine must assure compatibility with the multimedia player that will process the received information. In other words, it should be implemented in a language understandable by the multimedia player.

PUC-Rio - Certificação Digital Nº 1612822/CA

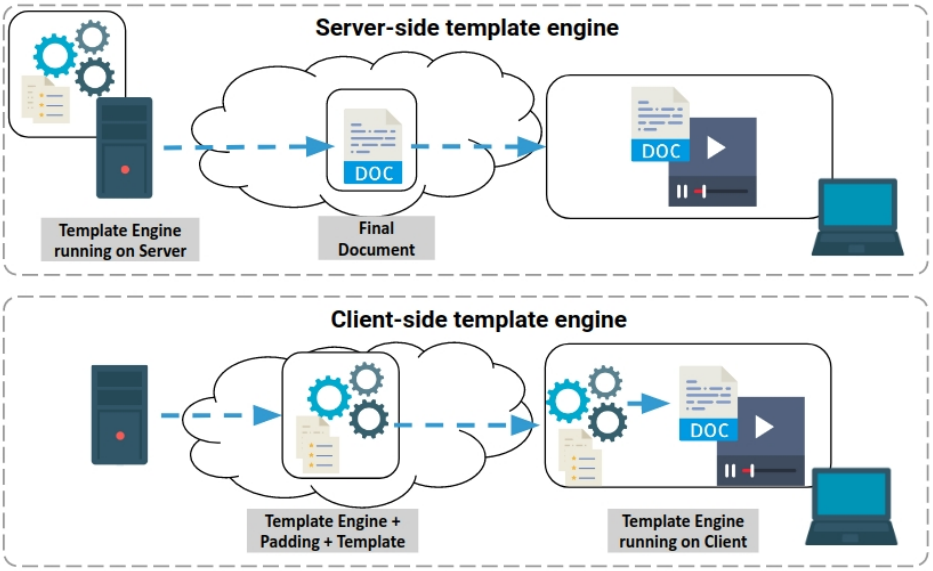

Figure 1.2: Template processing on client and server

1.2 Problem Definition

Several works have been studying templates for NCL. Among then, we cite LuaTPL [8] and Luar [9], which evaluates Lua scripts inside NCL documents to generate more elaborated NCL constructs. Also, Moraes *at. al* [10] implements a Lua library that generates NCL documents called Lua2NCL. Moreover, Terças *at. al.* [11] proposes a markup language with a Lua-like syntax called sNCL. Finally, [12] and [13] present XML-based template languages for hypermedia documents Xtemplate and TAL (Template Authoring Language), respectively.

Despite the aforementioned efforts, none of them obtained enough attention from NCL developers community. Given this scenario, we propose the following research question:

RQ1: How can we improve NCL development template usage?

To answer such a question, we take into account that, at the present moment, there is a shortage of NCL programmers. Nowadays, many existing NCL developers are actually experts that have a web background assigned to collaborate on DTV application development.

Template languages for NCL and HTML (HyperText Markup Language) emerged from the need of each respective community to focus on supporting themselves. However, differently from the NCL development, template-based applications on the web have gained more and more attention from its developers' community. We believe that bringing web template languages to

PUC-Rio - Certificação Digital Nº 1612822/CA

the development of applications in NCL strengthen this existing intersection by facilitating developers' work. So, we are looking at a field with broad template usage, the web. Figure 1.3 illustrates such an idea.

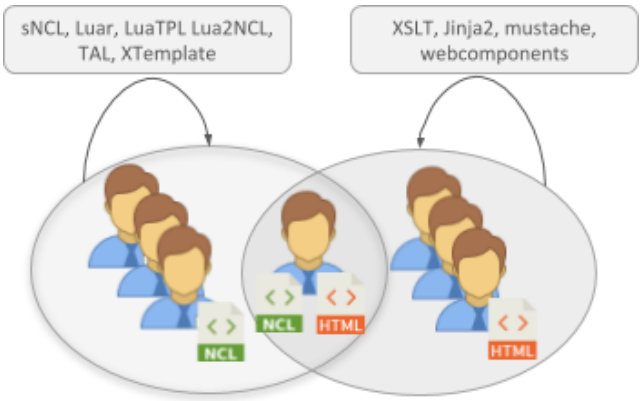

Figure 1.3: Intersection between NCL and HTML Template Developers

In the web scenario, we may cite template languages such as Jinja2 and Mustache. They have simplistic syntax and they are widely adopted by the web community and broadly used in frameworks. We also mention, XML-based templates for web such as XSLT and webcomponents[2]. These languages include relevant features such as inheritance and components (better discussed in Section 2.3).

We argue that an option to improve NCL template usage (*RQ1*) is to leverage the use of these web template-based languages in the NCL development environment, exploring the natural intersection between NCL and web developers.

Therefore, one of our objectives is *to evaluate web templates usage in NCL development.*

Motivated by this context, we also define the following more specific question:

> *RQ2: How can we support web templates processing in NCL development?*

Looking at the web field again, we can see several discussions regarding which is the best place for template processing: on the client-side or the server-side. However, in recent years, it has become increasingly common to run on the client-side. That is because, on the server-side, the server generates a new page for every interaction with a user. Each request has to travel all the way from the client to the server. Then, this page should be returned to the user. Such

[2] `http://www.webcomponents.org`

PUC-Rio - Certificação Digital Nº 1612822/CA

behavior can significantly increase the page loading time leading to latency issues.

On the other hand, the client-side processing avoids latency issues. More than that, it is not necessary to load all the page every time its content changes. Therefore, we also propose *the execution of template engines at player environment* i.e., client-side.

With web template language adoption, we aim at bringing its advantages to NCL development and, by enabling client-side execution, we may provide adaptable template-based NCL applications. This last case allows the execution of template-based applications over an IBB (Integrated Broadcast-Broadband) TV environment (better discussed in Chapter 4, Section 4.3).

1.3 Objectives

To summarize, this work addresses the following objectives:

1. **Allow web template engines execute at Ginga** (client-side in our context);

2. **To evaluate web templates usage in the development of applications in NCL**. To do this, we develop representative use cases, evaluate them and compare their susceptibility to errors against pure implemented NCL documents.

1.4 Organization

The remaining of this document is organized as follows. Section 2 discusses related work and compare them according to some features relevant to our work. Section 3 proposes an approach to allow web template engine execution on both server- and client-side. Then, section 4 presents an evaluation of web templates usage in NCL by addressing some use cases. Finally, Section 5 discusses final remarks and the next steps.

PUC-Rio - Certificação Digital Nº 1612822/CA

2
Related Work

The related work can be divided into two groups according to their propose. The first group comprises those works targeting the HTML; and the second, those developed for the NCL language.

2.1
HTML Templates

We organize related HTML templates into two groups. The first one focuses on processing tags in the HTML document to fill with information. They are XSLT and webcomponents.

XSLT (eXtensible Stylesheet Language Transformations) [14] is a language for document transformations. It is more applied in XML-based format of HTML called XHTML[1]. Although it is a powerful transformation language, creating transformations requires non-trivial programming skills and deep knowledge on the target language semantics and structure. The transformations defined with XSLT are done in style sheets. One style sheet is specific to a single transformation, not being possible to reuse it for the transformation of compositions containing elements different from the ones it was designed for. XSLT can operate over multiple input files in several distinct formats. The only requirement is the input file looks like XML.

Web components[2] allow developers to create new custom, reusable and encapsulated HTML tags. It consists of a set of JavaScript APIs (Application Programming Interface) that enables such tags creation. These news tags act as components and widgets that will work across modern browsers. They accept styling and are only rendered after being processed by the JavaScript code of the template engine. By themselves, they are not powerful due to logical structure limitations.

The second group aims at processing special syntax elements in the HTML document and fill it with information. They are: Jinja2, Mustache and React, and are described as follows.

[1] https://www.w3.org/TR/xhtml1/
[2] https://www.webcomponents.org/

PUC-Rio - Certificação Digital Nº 1612822/CA

Jinja2[3] template language is one of the web's most used template engines. Its language is inspired by Django framework[4] template system, however, Jinja2 extends it with a more expressive language giving to template authors a more powerful set of tools. It may be employed for both HTML and XML formats. Jinja2 offers a complete toolset for handle templates i.e., provides not only the semantic and syntax necessary to a language but an engine as well. Such an engine is implemented in python, nonetheless, third-party implementations supporting other programming languages might be found. Among them, Lupa[5] written in Lua.

Mustache[6] is a logic-less template syntax. Such term comes out from the fact that it has no "if" statements, "else" clauses, or "for" loops. Instead, there are only tags. It works by expanding these tags in a template using values provided in a hash or object. Such tags are replaced with a value, some nothing, and others with a series of values. The language can be employed in HTML, config files, source code—anything. And, since Mustache supports various languages, we don't need a separate template engine on the server- side. It is easier for non-programmers to manage once their logic is hidden behind tags. On the other hand, code became more difficult to read. Like Jinja2, it is implemented in many languages (e.g. Python, JavaScript, Lua). Listing 2.1 illustrates a "Hello World" template on Mustache. From the example, it is seen that Mustache template engine is embedded in the HTML document as a script (line 3).

PUC-Rio - Certificação Digital Nº 1612822/CA

```
1  <html>
2    <head>
3      <script src="mustache.min.js"></script>
4    </head>
5    <body>
6    <div id="target">Loading...</div>
7    <script>
8      var template = document.getElementId().innerHTML();
9      Mustache.parse(template);
10     var rendered = Mustache.render(template, {name: "Luke"});
11     template.html(rendered);
12   </script>
13   <script id="template" type="x-tmpl-mustache">
14     Hello {{ name }}!
15   </script>
16   </body>
17 </html>
```

Listing 2.1: Mustache "Hello World" example

[3] https://github.com/pallets/jinja
[4] https://www.djangoproject.com/
[5] https://github.com/zhsso/lupa
[6] https://mustache.github.io/

React[7] is a JavaScript library for building user interfaces on the web that runs on client- and server-side. It is based on components concept. A component is a self-contained element (a function or class in JavaScript) that produces an output when rendered. They might include other components to build more complex applications in React. Despite not being a template language, React allows expressions through JSX[8] (JavaScript XML), which embed XML code inside JavaScript. Listing 2.2 illustrates a "Hello Word" example. Similarly to Mustache, React is also embedded in the HTML code.

```
1  <html>
2    <head>
3      <meta charset="utf-8" />
4      <title>Hello React!</title>
5      <script src="https://unpkg.com/react@16/umd/react.development.
           js"></script>
6      <script src="https://unpkg.com/react-dom@16/umd/react-dom.
           development.js"></script>
7      <script src="https://unpkg.com/babel-standalone@6.26.0/babel.
           js"></script>
8    </head>
9    <body>
10     <div id="root"></div>
11     <script type="text/babel">
12       ReactDOM.render(
13       <h1>Hello, world!</h1>,
14       document.getElementById('root')
15     );
16     </script>
17   </body>
18 </html>
```

Listing 2.2: React "Hello World" example

When running on the web, Mustache and React works differently from Jinja2. Mustache and React load their engines as a script and use the DOM (Document Object Model) API to edit the HTML document. The DOM enables JavaScript code to access HTML elements as objects in a tree-based data structure. This process corresponds to line 8 in Listing 2.1 and line 14 in Listing 2.2. On the other hand, Jinja2 loads the engine through importing statements.

2.2 NCL Templates

This section details related work targeting templates in NCL. We grouped them into three categories. The first category focuses on handling NCL documents

[7] https://reactjs.org/
[8] https://reactjs.org/docs/introducing-jsx.html

PUC-Rio - Certificação Digital Nº 1612822/CA

with new markup elements that are processed and filled with information. They are TAL and XTemplate (3.0).

TAL [13, 15] supports template specifications called incomplete hypermedia compositions. TAL can define a set of documents that shares the same composition structure. However, it does not allow to assign its components information to the layout, nor does it provide any facility to create genre definitions of presentation characteristics. TAL owns template nesting, but its interfaces are not well-defined for this nesting, then properties can be violated.

XTemplate (3.0) [12] targets families of documents written in NCL 3.0. Unlike TAL, XTemplate 3.0 was developed to a specific target hypermedia language. On the other hand, TAL can be processed together with a padding document to generate applications in different target languages, depending only on the specific processor used. XTemplate targets on easing the authoring performed by experts. However, all XTemplate users need to have some technical pre-requisites such as XPath and XSLT [11] knowledge, even if they only need to instantiate composition templates. On the contrary, TAL has as one of its goals the reduction for the need of experts. TAL avoids the use of external notations different from those of the target-language conceptual model and notations that are beyond the abstraction level (like XSLT processing instructions of XTemplate 3.0 does).

The second group processes NCL documents with a special syntax to fill with information. They are LuaTPL [8] and Luar [9], which evaluate Lua scripts inside NCL documents to generate more elaborated NCL constructs. LuaTPL is very limited in its capability and therefore, the development of sophisticated templates becomes a very challenging task.

Luar implements embedded Lua snippets in the NCL document. In Luar, used templates are indicated inside the padding and treated as a media object, instead of a document. Luar came up with the template component concept, that allows distinct templates combination to form more elaborated applications. It defines two distinct processors: one for template documents and another for applications.

Finally, the third group aims at processing a different language than NCL to generate NCL documents. It comprises Lua2NCL only.

Lua2NCL [10] is a Lua framework. It builds NCL tags through Lua tables, instead of XML, and then uses information on these tables to produce NCL documents. In Lua2NCL, some original NCL tags are removed while others become invisible to programmers.

PUC-Rio - Certificação Digital Nº 1612822/CA

2.3 Discussion

To discuss the related work, we analyze them according to four characteristics, summarized in Table 2.1:

- **Control Structures Statements** indicates if the work supports control structures that change the template engine flow, such as loops and if-else statements.
- **Templates Inheritance Statements** means the capability of a template to inherit another. More precisely, one template acquires properties, states, and variables from its parents;
- **Components Statements** indicates the support to components. Components are independent elements with a self-contained structure that may be assembled to build more complex applications;
- **Template Engine Languages** indicates the languages used to implement the engine.

PUC-Rio - Certificação Digital Nº 1612822/CA

Work	*Control Structures Statements*	*Template Inheritance Statements*	*Components Statements*	*Template Engine Languages*
XSLT	for-each if-else	import[9]	×	Java, C, C++
webcomps.	JavaScript	×	API[10]	JavaScript
React	JavaScript	×	React.Component[11]	JavaScript
Jinja2	for-each if-else	extends[12]	×	Python, JavaScript, Lua...
Mustache	for-each	partials[13]	×	Ruby, JavaScript, Lua...
LuaTPL	Lua	×	×	Lua
Luar	Lua	×	includeComponent[14]	Lua
Lua2NCL	Lua	×	×	Lua
XTemplate	for-each	×	×	Java
TAL	for-each	extends[15]	×	Lua

Table 2.1: Related work features summary

[9] https://www.w3schools.com/xml/ref_xsl_el_import.asp
[10] https://developer.mozilla.org/en-US/docs/Web/API/CustomElementRegistry
[11] https://reactjs.org/docs/react-component.html
[12] https://jinja.palletsprojects.com/en/2.10.x/templates/#template-inheritance
[13] https://mustache.github.io/mustache.5.html#Partials
[14] http://ginga.lavid.ufpb.br/projects/wiki/wiki/_Cria%C3%A7%C3%A3o_de_aplica%C3%A7%C3%B5es_a_partir_de_templates_Luar
[15] [13]

Regarding Control Structures, some works support them by enabling developers to use its template engine language. It is the case of webcomponents and React based on JavaScript; and LuaTPL, Luar and Lua2NCL build upon Lua language.

Other works propose some syntax elements to iterate over collections and evaluate conditions. In particular, Mustache has its own "for-each" syntax (called "sections"), which index elements in the given padding document and Jinja2 uses a Python-like syntax. Moreover, Mustache, XTemplate, and TAL does not support "if-else" statements.

Usually, each work or supports inheritance or components statements. On one hand, XSLT, Jinja2, Mustache and TAL implement some inheritance statements. It happens mainly because they focus on creating a template for the entire document. On the other hand, webcomponents, React and Luar support components statements. Particularly, the first two uses JavaScript language to create them by inserting in the HTML DOM their processed data. The exceptions are LuaTPL, Lua2NCL and XTemplate which has no support for these two mechanisms.

Regarding the Template Engine Language, some HTML works have implementations in several languages. That happens because they can be processed on the server-side and hence should support languages that work on servers. XSLT, Jinja2 and Mustache are in this case. Jinja2 was originally coded in python and Mustache in Ruby. Nowadays, Jinja2 is assisted by languages like JavaScript and Lua, while Mustache is implemented in over 40 languages.

As one of our goals is to evaluate web-templates language to build NCL applications, it is important to highlight differences between Jinja2 and Mustache. Jinja2 is a template engine, so it offers in the same package a template language and a python-like API to process the padding data. In counterpart, Mustache depends on third-part engines as it is a template language specification only.

Concerning works targeting NCL, excluding Xtemplate, they are implemented in Lua. In them, templates are processed before their arrival at Ginga. Differently, from any other approach, we propose template processing at Ginga (client-side), without changing NCL standardized syntax neither creating any other language. More than that, our proposal consists of importing the appropriate engine and then process template documents using classes and methods imported.

PUC-Rio - Certificação Digital Nº 1612822/CA

3
NCL-formats tool

In order to improve the template usage in NCL, we define the objective of **Allow web template engines processing at Ginga**. By doing this, we take advantage of the intersection between NCL and web development. With this perspective, we intend to bring web programmers and their knowledge to NCL application's development. In that sense, NCL application templates may be built from web-template languages. The developer creates NCL applications by simply passing data (padding document) to a web-template engine.

To enable such execution, we propose the NCL-formats[1] tool. It aims at assisting the development of NCL applications based on template languages, especially those targeting the web. Nonetheless, it is important to highlight that NCL-formats can handle a wide range of template languages.

The remaining of this chapter is organized to present NCL-formats. In Section 3.1, we define which web-template language will be tested. Section 3.2 presents the most relevant execution configuration for NCL-formats while Section 3.3 addresses others possible scenario. At last, Section 3.4 comments about its implementation details.

PUC-Rio - Certificação Digital Nº 1612822/CA

3.1
Supported Web Template Languages

In the web context, it is recurrently necessary to take advantage of repetition elements. One example is page headers that might be shown on several pages across one website. So, a web programmer may opt to use a template to build such headers. From this need, many template languages have been developed and innumerable options are available.

To define our supported languages, we define the following requirements:

- license should *not be proprietary*;
- have a *significant users base*;
- *support control structures*;
- *support template inheritance*;
- be able to *run on TVD environment and at Ginga*.

[1] `https://github.com/TeleMidia/ncl-formats`

A web-template language with a large number of developers goes toward our proposal to *RQ-1 How can we improve NCL development based on templates?* The option for a widespread language encourages developers to implement applications for Ginga as well. Once this developer knows about developing with web-templates language, he/she can easier migrate to the development of NCL applications programming in the same language.

Control structures give more power to developers during coding by allowing them to handle their codes' behavior. In declarative languages, such as NCL, developers type what "the code should do" in their applications. Consequently, they cannot change their codes' flow of control.

Template inheritance is a desirable feature because it allows developers to spit their code in different parts. Their advantage is to provide templates reuse. In this way, developers may change one child template for another and still take advantage of the same base template.

Out of these five requirements, the most restrictive is the last one. In practical terms, it implies that the language should have an implementation in Lua. Such constraint eliminates a wide range of engines that targets languages directly related to the web development environment, such as JavaScript and PHP.

Based on these requirements, we chose initially to support Jinja2 and Mustache web-template languages.

PUC-Rio - Certificação Digital Nº 1612822/CA

3.2 Web-Template Processing Embedded in an NCL Document

Embedding templates processing in the NCL document is the main execution scenario for NCL-formats. It brings advantages compared to prepossessed templates. For instance, it allows configuration at run-time. This possibility gives more dynamism to templates. The use cases in Sections 4.2 and 4.3 from Chapter 4 explores more about this.

By *running template engine as a Lua Script*, Ginga receives one single NCL document responsible for setting up the environment. We call this NCL document as *configuration file*. Note that for this scenario there must be a mechanism to process the given data since the middleware can only handle NCL documents.

On the web, browsers can only render JavaScript. Therefore, to process templates documents in the client-side is it necessary an engine implemented in JavaScript as well. A similar approach occurs with Ginga that adopts Lua as its scripting language. For that reason, scripts in Lua has a straightforward integration and that is reason behind our choice.

Figure 3.1 below examples how the process works. Ginga middleware receives the aforementioned NCL document with three pieces of information: the template document; the padding; and the template engine, as properties of a media object. This media object is the NCL-formats tool NCLua script. The NCL-formats outputs an NCL document to be played as sooner as the script signalizes its ending.

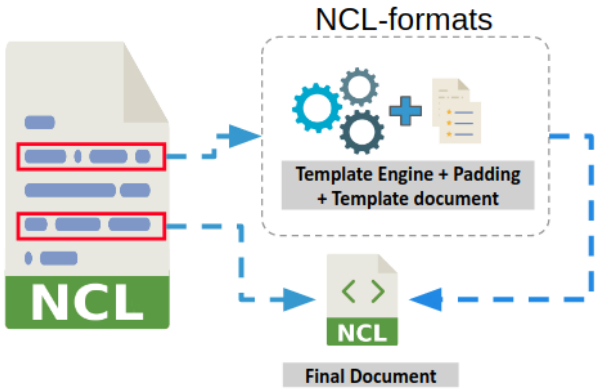

Figure 3.1: Template Processing through script languages

Listing 3.1 next details this NCL document. The NCL player starts the Lua script with relevant information as a *property* of *NCL-formats* script. The script processes the passed data to generate the `final_document.ncl` NCL document. In the end, the script signalizes its ending to the configuration file. Upon receiving this signal, the Ginga player starts to reproduce `final_documents.ncl`. “

NCL-formats is a Lua script responsible for handle all the template processing. In our approach, NCL-formats imports the engine and uses classes and methods from it to manipulate the given data and process the template document received.

PUC-Rio - Certificação Digital Nº 1612822/CA

```
<ncl>
  <head>
        ...
  </head>
  <body>
  <port id="template-handler" component="template-engine"/>
  <media id="template-engine" src="ncl-formats.lua">
    <property name="type" value="jinja2"/>
    <property name="template" value="slideShow_child.ncl.j2"/>
    <property name="padding" value="padding.json"/>
  <media>
  <media id="final-ncl" src="final_document.ncl"/>
  <link id="link" xconnector="conBase#onEndStart">
    <bind role="onEnd" component="template-engine"/>
    <bind role="start" component="final-ncl"/>
  </link>
  </body>
</ncl>
```

Listing 3.1: Configuration file used for template processing on the client-side with NCL-formats embedded as a media object

3.3 Web-template Processing outside NCL document

This section discusses two other possible scenarios for running NCL-formats: (1) as a standalone tool; (2) extending Ginga Player.

The standalone version enables developers to simulate their application on their workstation, for instance, making easier to create and test new template-based NCL applications. This scenario emphasizes that NCL-formats is self-contained and might be perfectly executed outside the Ginga environment. It can run on both: server- and client-side.

Listing 3.2 demonstrates NCL-formats standalone version execution through the command line. The Lua script receives three arguments: the padding data; the template engine to be executed; and the template file itself. NCL-formats uses the template file name to generate the outputted NCL document.

```
lua ncl-formats.lua padding.json template_engine=jinja2 template=
   slideShow_child.ncl.j2
lua ncl-formats.lua padding.json template_engine=mustache template
   =slideShow.ncl.mustache partial1.mustache partial2.mustache ...
```

Listing 3.2: Command line call to NCL-formats process Jinja2- and Mustache-based template for slideshow

By extending Ginga Player, it will handle, besides NCL, other formats. However, it will require modifications on Ginga specification that implies new in Forum and ABNT (*Associação Brasileira de Normas Tecnicas*) discussion

PUC-Rio - Certificação Digital Nº 1612822/CA

to release a new standard. Moreover, it takes time to current STB (set-top box) and TV to implement such features.

Ginga has been extended to support templates syntax. Figure 3.2, shows the proposed architecture. In this approach, the middleware receives any template-based document e.g., Jinja2, Mustache. The Parser takes care of checking document type and if it is a padding document it delivers both the padding and the template to NCL-formats.

NCL-formats Lua script determines the appropriate engine to execute and deliver the given data for template processing happens. As a result, NCL-formats produces the final NCL document.

This done, Ginga's Parser continues its natural workflow by passing the NCL document to NCL player.

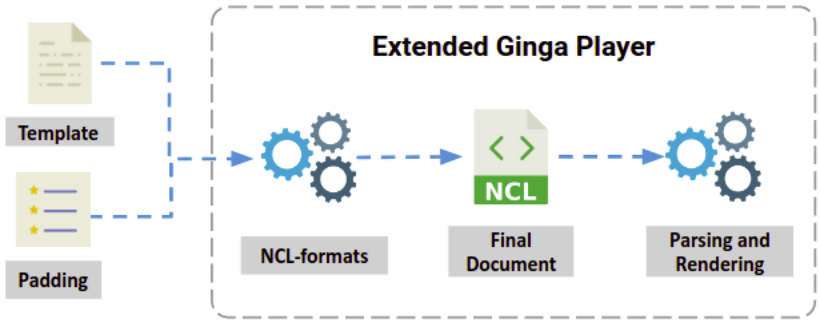

Figure 3.2: Extended Ginga Player architecture

To enable templates processing in the Parser, a new option was added to Ginga's command line entry list. Such an option expects a template document and only validates it if a padding document comes along. The Listing 3.3 illustrates its usage in the terminal.

```
1 ginga padding.json --template=slideShow_child.ncl.j2
```

Listing 3.3: PUC-Rio Ginga executing a Jinja2 template through command line

3.4 Implementation Details

To work properly NCL-formats need to load some dependencies: JSON library, Mustache and Jinja2 implementation.

The JSON library is necessary to deserialize JSON data from the padding document. In other words, this library is responsible for decoding JSON data and stores it in data type understandable by LUA language. It must be

PUC-Rio - Certificação Digital Nº 1612822/CA

implemented in pure Lua, otherwise, Ginga is not able to execute it. The developed examples in Chapter 4 use JSON Encode/Decode in Pure LUA[2].

Mustache implements template inheritance through the concept of partials. Partials came from embedded Ruby (eRuby) and are used in this sense to refer to templates that cannot be rendered by themselves. Each partial must correspond to a file.

Mustache Lua implementation is called Lustache[3]. It only handles templates as strings. So, in the scenario of a large Ginga application with n partials implemented, NCL-formats should read these partials (files) one by one saving them as strings. Only after this data type conversion, the engine becomes able to process the template.

Jinja2's implementation is Lupa. It allows templates to be loaded from the same folder besides strings. When loading from folders, the developer only passes one template. The engine deals with any other required template, as long as they are in the same directory from the given one.

Jinja2 implements variables that give more control to the programmers, such as `{{ loop.index }}` and `{{ loop.length }}` that counts the number of loop iterations starting from one and gets back the size of an iterable, respectively.

On Jinja2 it is also possible to set up variables. None of that Mustache can do. In the case of more complex applications, to generate the same NCL document, Mustache developers should:

- put extra information in the padding document;
- type it in NCL-formats.

The first option is not desirable since it would put not only semantic-related data but logical as well in the padding document. Such behavior breaks the concept of templates itself because of blends logical and semantics in one single file. The second option, despite not breaking the rules, would require developers to code extra data in NCL-formats to manage the necessary logic to build the same template as in Jinja2. In other words, NCL-formats would be responsible for handle more data that would change for each kind of template.

[2] `http://regex.info/blog/lua/json`
[3] `https://github.com/Olivine-Labs/lustache`

PUC-Rio - Certificação Digital Nº 1612822/CA

4
Web Templates Evaluation

This chapter presents an evaluation of web-templates languages behavior to built NCL applications. For that, it is necessary to discuss the evaluation procedure, first.

The **number of lines of code** is adopted as a metric for measuring the amount of work spent to produce an NCL application when using the chosen web-template languages. To determine the application's total gain, is used the equation below that gives the *total percentage score*:

$$Score = \left[1 - \frac{number\ of\ [TemplateLanguage]\ lines\ of\ code}{number\ of\ NCL\ lines\ of\ code}\right] \times 100\% \quad (4\text{-}1)$$

where:

$TemplateLanguage =$ indicates the proper web-template language.

Measuring the number of lines required for coding is not fair enough. Listing 4.1 illustrates that. It shows a snippet of code in Jinja2 responsible for setting a Lua table called `action_components`. The typed code is condensed to one single line.

```
1 {% set action_components = {set={media={'mNodeSettings' .. suffix,
      {bind_param={name='setValue', value='left'}, interface='
    direction'}}, {'mNodeSettings' .. suffix, {bind_param={name='
    setValue', value='3'}, interface='service.currentFocus'}}} ,
    stop={'ctx' .. suffix}} %}
```

Listing 4.1: Barely readable code

Certainly, this single-line instruction does not represent how most programmers would code it, as it is not very readable by humans. To turn the code more comprehensive, this line should be broken into many leading to a higher number of lines and so influencing the metric adopted. Nonetheless, logic does not changes in any aspect at all.

To demonstrate our point, Listing 4.2 shows the same code from Listing 4.1 in a way more likely to be typed by a developer (more human-readable).

PUC-Rio - Certificação Digital Nº 1612822/CA

```
{% set action_components =
  {set=
      {
         media= {
            'mNodeSettings' .. suffix, {
            bind_param= {
              name='setValue',
              value='left'
         },
         interface='direction'
         }
      },
      {
         'mNodeSettings' .. suffix, {
         bind_param={
            name='setValue',
            value='3'
         },
         interface='service.currentFocus'
         }
      }
        },
  stop=
      {
          'ctx' .. suffix
      }
  }
%}
```

Listing 4.2: More comprehensive code

So, taking that into account we also use the **number of instructions** as a metric. To this end, Table 4.1 summarizes a set of instructions in NCL, Jinja2 and Mustache languages and its use.

Language	*Delimiter*	*Usage Scenario*
NCL	<element attributes />	one-line instructions
	<element attributes > inner elements </elements>	multiple lines instructions
Jinja2	{% ... %}	statements
	{{ ... }}	expressions
Mustache	{ > name }	partials
	{{ ... }}	variables
	{{ #section name }} section content {{ /section name }}	sections

Table 4.1: Types of instructions on each language

Therefore, for this second metric, the percentage score becomes:

PUC-Rio - Certificação Digital Nº 1612822/CA

$$Score = \left[1 - \frac{number\ of\ [TemplateLanguage]\ instructions}{number\ of\ NCL\ instructions}\right] \times 100\% \quad (4\text{-}2)$$

To due NCL syntax, developers should type many instructions to specify their program and sometimes they do copying and pasting as a manner to reduce typing and gain time. The amount of code typed or the copy and paste process leads to errors in the final code that may pass unseen for developers until the application is tested.

Another issue appears in more sophisticated NCL codes that require more elaborated logic to develop. In such cases, due to their complexity, errors may come out during the application's developing stage. With templates, new developers do not need to reinvent the wheel. It is enough to use the templates.

The use cases were chosen to simulate very common application types and different transmission scenarios in DTV. In them, the NCL-formats was used to collect the padding; the template; and the engine and generate the NCL application.

Next sections discuss three use cases implemented: Section 4.1 delineates the slideshow example developed in Mustache and Jinja2; Section 4.2 details the additional content example that illustrates an application in a broadcast context; and Section 4.3 set forth a video recommendation application for the IBB (Integrated broadcast-broadband) scenario. Lastly, Section 4.4 discuss results.

PUC-Rio - Certificação Digital Nº 1612822/CA

4.1 Slideshow

The slideshow is a kind of presentation that changes the content displayed from time to time or after an action is triggered e.g., user presses a button to go back and forth.

The developed slideshow instance was implemented considering two cases, each one using a different web-template language. They consist of twenty-one images that change every 5 seconds. The final NCL document outputted i.e., the NCL application is the same for both.

Listing 4.3 shows a snippet of the padding document. It is a JSON document containing the media elements used in the application and is common for the two cases evaluated.

The slideshow example built with Jinja2 takes advantage of its template inheritance capacity. In this instance, there are two template documents: one called `slideShow_base.ncl.j2` and the other `slideShow_child.ncl.j2`.

```
[
  {
      "type":"directory",
      "name":"media",
      "contents":[
         {"type":"file","name":"image1.jpg"},
         {"type":"file","name":"image2.jpg"},
         {"type":"file","name":"image3.jpg"},
         ...
      ]
  }
]
```

Listing 4.3: padding.json

`SlideShow_base.ncl.j2` works as a base template in template hierarchy and is an NCL-based code with block tags in Jinja2 syntax. A `{% block %}` element indicates code replacement. So, in Listing 4.4 a `{% block medias %}` stipulates another template to handle the block named medias. The same happens for the block links (`{% block links %}`).

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- An NCL SlideShow example with embedded Jinja2 template -->
<ncl id="slideShow">
  <head>
      <connectorBase>
         <importBase documentURI="connectorBase.ncl" alias="
            conBase"/>
      </connectorBase>
      <regionBase>
         <region id="main" width="100%" height="100%" zIndex="1"/>
      </regionBase>
      <descriptorBase>
         <descriptor id="ImageDes" region="main" explicitDur="5s
            "/>
      </descriptorBase>
  </head>
  <body>
      <port id="startSlideShow" component="image1"/>
      {% block medias %}{% endblock %}
      {% block links %}{% endblock %}
  </body>
</ncl>
```

Listing 4.4: slideShow_base.ncl.j2

Logical structure is held in `slideShow_child.ncl.j2` (Listing 4.5) which is a child template. It establishes blocks behavior in its parent template. Inheritance is made through `{% extend %}` statement on `slideShow_base.ncl.j2` file. Medias "block" mounts the name of each media, gather its path and set them to `id` and `src` NCL attributes, respec-

PUC-Rio - Certificação Digital Nº 1612822/CA

tively. Links block, builds a link passing media objects `id` formed in media block.

```
{% extends "slideShow_base.ncl.j2" %}
{% block medias %}
  {% for i in files_list[1].contents %}
      <media id="{{'image' .. loop.index}}" src="{{'media/' .. i.
          name}}" descriptor="ImageDes"/>
  {% endfor %}
{% endblock %}
{% block links %}
  {% for i in range(#files_list[1].contents-1) %}
      <link id="{{'lMoveForward' .. loop.index}}" xconnector="
          conBase#onEndStart">
          <bind role="onEnd" component="{{'image' .. loop.index
              }}"/>
          <bind role="start" component="{{'image' .. (loop.index+1)
              }}"/>
      </link>
  {% endfor %}
{% endblock %}
```

Listing 4.5: slideShow_child.ncl.j2

PUC-Rio - Certificação Digital Nº 1612822/CA

Figure 4.1 shows a graphic comparing the number of lines needed in the slideshow instance implemented with Jinja2 to the NCL code. In overall 34 lines of code were written, 14 of them related to template syntax (`slideShow_child.ncl.j2`) while the others are in NCL (`slideShow_base.ncl.j2`). This code, when the final NCL document is generated, expands to a total of 119 lines. That represents a score of **71,43%** according to Equation 4-1.

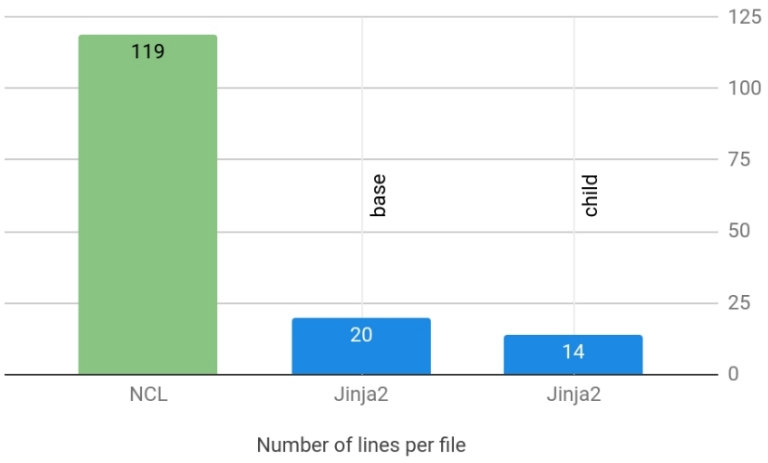

Figure 4.1: Number of lines coded per file employing Jinja2 as template language

From the overall number of lines, one is for inheritance and two are to delimiter the beginning and end of which block tag. Considering that

inheritance is not strictly necessary and that everything could have been done in only one file, we have 20 lines of code. In this case, Equation 4-2 enlarges the score to **77,31%**.

Measuring the number of instructions ends in similar results. However, the gains are a little higher. Figure 4.2 summarizes the amount of instructions required on each Jinja2's file compared to the generated NCL code. 12 lines were typed in `slideShow_base.ncl.j2` and nine in `slide_show.child.ncl` to generate 91 lines in NCL.

This case results in a higher gain. Applying Equation 4-1 ends in a *score* of **76,92%**. Without a hierarchy structure, the *score* goes to **82,42%** as pointed by Equation 4-2.

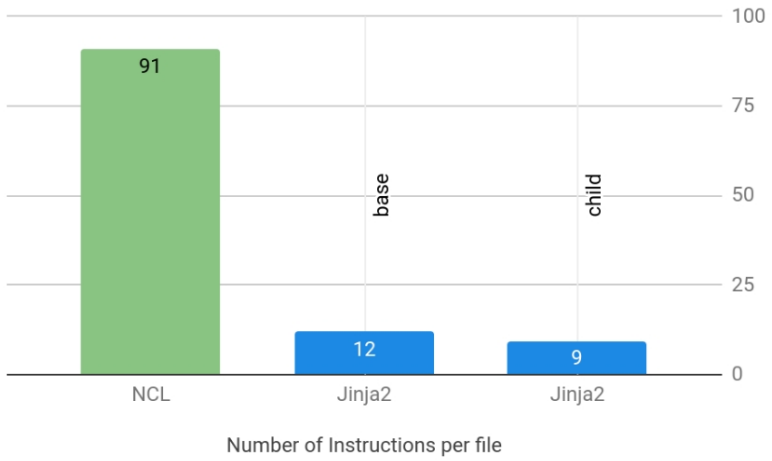

Figure 4.2: Number of instructions typed per file employing Jinja2 as template language

Table 4.2 summaries the *score* achieved on each case considering the two measured metric.

	Metric	
Case	*number of lines*	*number of instructions*
w/ hierarchy	69.75%	76.92%
w/o hierarchy	77.31%	82.42%

Table 4.2: Jinja2 slideshow *score* summary

As well as in the Jinja2 case, the slideshow implemented with Mustache also explores inheritance. Mustache code inheritance is implemented through partials, previously presented in Chapter 3, Section 3.4.

The slideshow proposed example was developed based on three files:

– `slideShow.ncl.mustache`: deals with NCL code that has not become a template. It is the main file;

PUC-Rio - Certificação Digital Nº 1612822/CA

– `medias.mustache`: builds the NCL media elements;

– `links.mustache`: creates each link.

Listing 4.6 presents the code in `slideShow.ncl.mustache`. It declares the NCL code and calls two partials: `{{<medias}}` and `{{<links}}`. Each partial corresponds to one inherited template file. The elements inside the NCL `head` tag were removed as they are the same as in Jinja2 instance.

```
<ncl id="slideShow">
  <head>
    ...
  </head>
  <body>
  <port id="startSlideShow" component="image1"/>
    {{>medias}}
    {{>links}}
  </body>
</ncl>
```

Listing 4.6: slideShow.ncl.mustache

PUC-Rio - Certificação Digital Nº 1612822/CA

Listing 4.7 and Listing 4.8 next exhibits, respectively, the `medias.mustache` and the `link.mustache`.

```
{{#contents}}
  {{#index}}
    <media id='image{{index}}' src='media/{{name}}' descriptor='
        ImageDes'/>
  {{/index}}
{{/contents}}
```

Listing 4.7: medias.mustache

```
{{#contents}}
  {{#next}}
    <link id='lMoveForward{{index}}' xconnector='conBase#
        onEndStart'>
      <bind role='onEnd' component='image{{index}}'/>
      <bind role='start' component='image{{next}}'/>
    </link>
  {{/next}}
{{/contents}}
```

Listing 4.8: links.mustache

Figure 4.3 summarizes the development with Mustache. In overall, 33 lines of code were typed: 20 in `slideShow.ncl.mustache` (main); five in `medias.mustache`; and 8 in `links.mustache`. That leads to a score of **72.27%**, according to Equation 4-1.

Figure 4.3: Number of lines coded per file employing Mustache as template language

This code implemented in one single file without hierarchy tags needs 31 lines. In this circumstance, the two removed lines are related to partials. Therefore, the score increases a little to **73.95%**, as Equation 4-1 indicates.

Taking into account the number of instructions the *score* with hierarchy is **72.53%**. In this case 12 instructions were typed in the main file; eight in the *links.mustache*; and five *medias.mustache*. Figure 4.4 illustrates the results for each file.

Figure 4.4: Number of instructions typed per file employing Mustache as template language

The removal of hierarchy elements from the code increases the *score* to **74.73%**, as stated by Equation 4-2. Table 4.3 summarizes the *score* obtained with Mustache as template language for the two cases proposed.

Creating templates with Jinja2 or Mustache results in a large reduction independently of the metric adopted. More than that, this reduction is pretty

PUC-Rio - Certificação Digital Nº 1612822/CA

	Metric	
Case	*number of lines*	*number of instructions*
w/ hierarchy	72.27%	72.53%
w/o hierarchy	73.95%	74.73%

Table 4.3: Mustache slideshow *score* summary

much the same.

A curious fact can be observed comparing the two cases. Using inheritance on both, the Jinja2 instance produced more lines in comparison to Mustache. That is because each `{% block %}` tag in Jinja2 generates 3 lines/instructions (one to denote the block on the parent template and two to mark its beginning and ending in the child template) plus one line for the extending tag. On the other hand, Mustache just requires the partial to be processed. Mustache separates one partial per file which eliminates the need for more tags.

Regarding what was mentioned in the previous paragraph, there is a drawback in the way Mustache works. For NCL applications that demand more elaborated templates, the number of files grows equally to the number of partials used in the template's logic. On Jinja2, a developer has free control of how many `{% blocks %}` statements he/she puts on each module.

We opt to develop the next use cases only in Jinja2. We do that because they are more elaborated applications and Mustache does not provide enough features to ease development in it, such as if-statement or declaration of variables. To code these on Mustache it would be necessary a different approach more sophisticated compared to Jinja2. Therefore, we consider it would not be worth to develop our templates in this language.

PUC-Rio - Certificação Digital Nº 1612822/CA

4.2 Broadcast Additional Content

Terrestrial Broadcast TV main characteristic is to deliver the same content to everyone tuned in the same TV station. It enables the transmission of other kinds of information in addition to the usual video and sound contents.

A very common application type in the Broadcast context is the additional content. In such applications, the user navigates among menus to gather more information about the audiovisual content displayed on the TV screen. For instance, an additional content for soup opera may show a character biography or a summary of the previous episodes; for a movie, an additional content may exhibit information about directors and actors.

Considering this scenario, we develop an additional content use case using a Jinja2 template. This use case simulates a TV station broadcasting a movie. To the viewer, it is possible to interact through a button. If the user chooses to interact, a four-button menu is displayed and he/she can navigate using the remote control.

Each button puts on view a different kind of data related to the movie in the TV programming: the movie plot; the characters; details (extra info such as budget, studio producer, release date, among others) and pictures. Figure 4.5 shows a screenshot from the application.

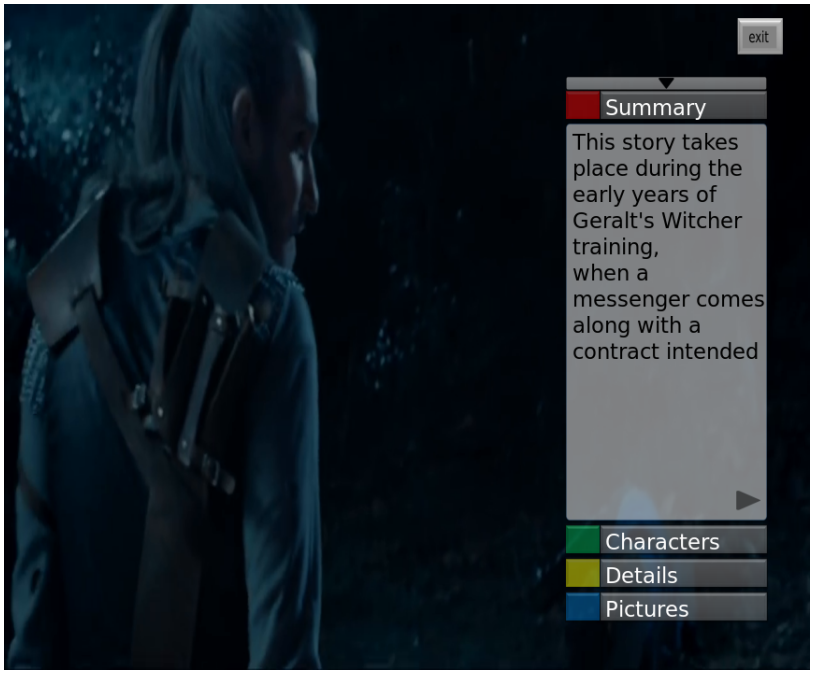

Figure 4.5: Additional content application screenshot

The padding document allows to each template author to specify some images used in the application and the content displayed on each button. Its data is as follow:

- video: the main video being played;
- begin: image to indicates that there is an application and the user can interact with it;
- close: finishes application;
- reg: minimizes each button context;
- button: information related to each button:
  - image: button image;
  - title: text title related to the button;

PUC-Rio - Certificação Digital Nº 1612822/CA

- info: first information window;
- infoPlus: second information window;
- background: image used as background;
- left: image indicating to return to the first window;
- right: image indicating that there is a second window.

Listing 4.9 shows the padding document. The *Green*, *Yellow* and *Blue* buttons information were omitted for being similar to the *Red* button.

```
{
  "video" : "video.mp4",
  "begin" : "info.png",
  "close" : "exit.png",
  "reg"   : "reg.png",
  "buttons" : [
      {
         "name"  : "Red",
         "image" : "red.png",
         "title" : "redTitle.txt",
         "info"  : "redInfo1.txt",
         "infoPlus" : "redInfo2.txt",
         "background" : "background.png",
         "left"  : "left.png",
         "right" : "right.png"
      },
      {
         ... // Green Button Data
      },
      {
         ... // Yellow Button Data
      },
      {
         ... // Blue Button Data
      }
  ]
}
```

Listing 4.9: Padding file used in the additional content use case

The template developed consists of three files. Starting from the same approach used in the slideshow use case (Section 4.1), there are three files:

- `additionalContent_base.ncl.j2`: contains NCL head elements and a set of block elements related to the application logic;
- `additionalContent_child.ncl.j2`: implements each block on its parent.

The third file contains all the macros used to support the development.

The blocks in `additionalContent_base.ncl.j2` are: one to begin application and another to end it; three to initialize each button context, finish

PUC-Rio - Certificação Digital Nº 1612822/CA

and minimize them; one to navigate to the second information window and another to navigate back to the first window; one to return to the menu; and four to handle each button content when its key is pressed.

The file `additionalContent_child.ncl.j2` implements the aforementioned blocks. Their code uses features presented in Jinja2 language syntax. Macros and variables creation are some of them.

Macros are comparable to functions in imperative programming languages and they put often typed code in reused functions avoiding repetition. A macro is defined as `{% macro name (param1, param2, ...) %}`. They must have a name and accept arguments as optional parameters. They might be called by simple typing `{{ name(param1, param2, ...) }}`.

Jinja2 also admits a macro to call another one. It is done through the `{% call %}` statement which calls the specified macro, executes its code and then returns to the same point it was invoked. After that, the code continues its natural workflow. In this procedure, a macro can return values to its caller.

The Listing 4.10 shows the *navigationRight* block which is used to control the behavior of the application when the right key is pressed and the first information window is active. In this block, a link is set up for each button through a call to a macro named *nutshell* at line 3.

The *nutshell* macro code is depicted in Listing 4.11. When reaching the command `{{ caller () }}` at line 7, it returns the control to its caller. *navigationRight* block then sets one table containing media elements that play *stop* role and another to those that play *start* role. These tables are used to another macro called *setAction*. At line 8 in Listing 4.10, the `{% endblock %}` tag, signalizes the ending of the "call" block. This way, the control is given back to the macro called that should also end its execution as well.

PUC-Rio - Certificação Digital Nº 1612822/CA

```
1  {% block navigationRight %}
2    {% for i in files_list.buttons %}
3    {%+ call nutshell("m".. key[loop.index] .. "Right" , "
        CURSOR_RIGHT", "onKeySelectionStartNStopN", "onSelection") %}
4        {%- set stopMedias = {'m' .. key[loop.index] .. 'Info', 'm'
            ..key[loop.index] .. 'Right'} %}
5        {{- setAction('stop', stopMedias) }}
6        {% set startMedias = {'m' .. key[loop.index] .. 'Info1', 'm'
             .. key[loop.index] .. 'Left'} %}
7        {{- setAction('start', startMedias) }}
8    {% endcall %}
9    {% endfor %}
10 {%- endblock %}
```

Listing 4.10: Example of macro invocation and call statement in additional content use case

```
1 {% macro nutshell(component, link_param, connector, role) %}
2   <link xconnector="{{'conBase#' .. connector}}">
3       <bind role="{{role}}" component="{{component}}"/>
4   {% if link_param ~= nil %}
5       <linkParam name="keyCode" value="{{link_param}}"/>
6   {% endif %}
7       {{ caller () }}
8   </link>
9 {% endmacro %}
```

Listing 4.11: Macro nutshell in additional content use case

Another feature from Jinja2 is variable creation. Variables are declared and initialized in a *set* statement. Line 4 and 6 from Listing 4.10 show two tables' initialization.

The template documents in addition to the padding when processed generate an NCL document that consists of *medias* elements and *links* that react to the user's interaction. Each button image and its title can occupy two different positions: one when the button context is opened (red button on Figure 4.5) and another when the button is closed or the application is in the main menu (the others buttons on Figure 4.5). These buttons' positions on the screen are set through *properties* of each corresponding media element.

In the NCL document, *Links* are compounded by media elements that assume *play* or *stop* NCL roles. They control events related to medias elements that change states in the NCL state machine and therefore, the application's behavior.

Figure 4.6 demonstrates the number of lines used per file. In `additionalContent_base.ncl.j2` template document 153 lines were coded, `additionalContent_child.ncl.j2` required 249 lines and `macro.jinja2` 15. These three documents sums 417 lines in total. After being processed, these templates expand to almost 700 lines of pure NCL code. That represents a *score* of **40.26%** mesured by Equation 4-1.

Figure 4.7 displays similar results measuring the number of instructions, instead. From it is seen that a total of 606 instructions were generated in the NCL document. The template documents required a total of 287 instructions: 111 in `additionalContent_base.ncl.j2`, 167 in `additionalContent_child.ncl.j2` and 9 for the macros. In this case, the score is **52,64%**.

Table 4.4 below summarizes results obtained with the developed templates. It considers the two metrics proposed and the code with and without hierarchy elements.

PUC-Rio - Certificação Digital Nº 1612822/CA

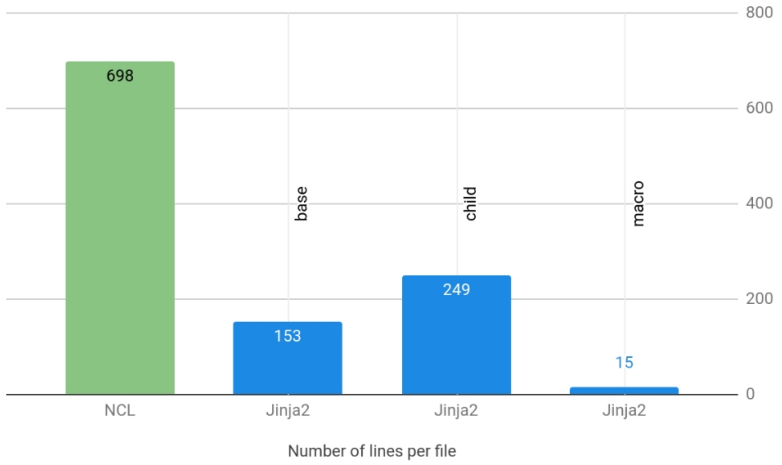

Figure 4.6: Number of lines coded per file employing Jinja2 as template language

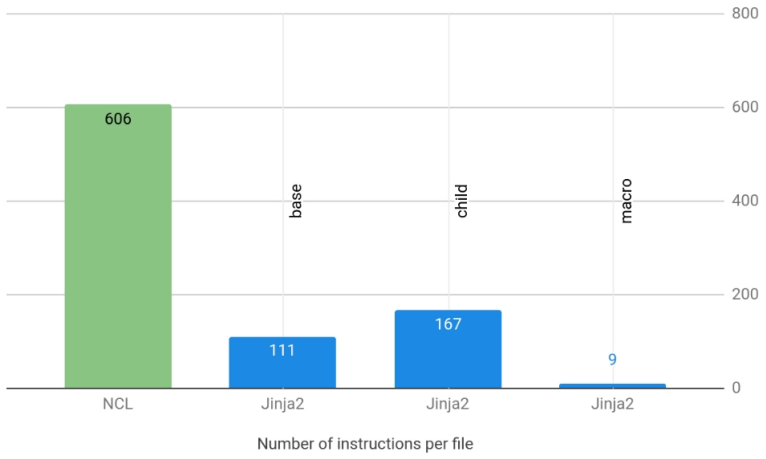

Figure 4.7: Number of instructions typed per file employing Jinja2 as template language

	Metric	
Case	*number of lines*	*number of instructions*
w/ hierarchy	40.26%	52.64%
w/o hierarchy	44.84%	63.70%

Table 4.4: Additional content *score* summary

4.3 IBB Video Recommendation

Differently from Broadcast, Broadband uses internet access to delivery audio-visual content. It became popular with the advent of smart TVs and video streaming services, such as Youtube and Netflix. Viewers are more active since it delivers a wide range of customized video streaming catalog in which they can navigate through.

PUC-Rio - Certificação Digital Nº 1612822/CA

The recently IBB (Integrated Broadcast-Broadband) scenario integrates seamlessly both Broadcast and Broadband contexts. In other words, it enables viewers to seamlessly navigate between contents from both scenarios. For instance, a viewer in the Broadband environment can receive a suggestion to watch a live Broadcast content related to what he/she is currently viewing. Alternatively, a user in the Broadcast environment can receive a suggestion to watch more content from Broadband sources related to what he/she is watching at that moment.

The integration brought in the IBB scenario leads to increase users' engagement and to maximize their satisfaction by offering a range of new services. Any IBB services should be able to extend the traditional Broadcasting using any Broadband mechanism available to bring new interactive and complementary content to the end-user [16].

In IBB scenarios, the biggest advantage is to enable applications to retrieve their content from servers. It occurs at application's running time, upon users' interaction. With client-side processing, NCL applications become able to respond to servers' answers and change the content displayed. In that way, it affords a wide range of information according to viewers' interaction.

The rest of this section presents an NCL video recommendation example for the IBB scenario developed using templates implemented in Jinja2.

The application shows a list of images related to the video on the screen. To the viewer is possible to go back and forth over this list. A focus indicates the currently selected image. If this user chooses one image from the list, the presentation of the current image ceases and the corresponding video starts to play. Figure 4.8 shows a screenshot taken from the application.

Figure 4.8: Video recommendation application screenshot

The developed template allows its author to specify the application's main video and recommended videos and images. Thus, the padding document

PUC-Rio - Certificação Digital Nº 1612822/CA

states as depicted in the Listing 4.12. In this case, the application advises five videos to its users.

```
{
  "primary" : "mainVideo.mp4",
  "secondary": [
      "video1.mp4", "video2.mp4", "video3.mp4", "video4.mp4", "
         video5.mp4"
  ],
  "images": [
      "image1.jpg", "image2.jpg", "image3.jpg", "image4.jpg", "
         image5.jpg"
  ]
}
```

Listing 4.12: Padding file used in the video recommendation use case

The template documents use the same Jinja2 features presented in Section 4.2 and are also split in the same structure: `videoRecommendation_base.ncl.j2` to load Jinja2 blocks and NCL code not required in the `videoRecommendation_child.ncl.j2` that holds the template logic. There is a macro file to help in the development.

In `videoRecommendation_base.ncl.j2` there are only two blocks: one to handle NCL contexts and the other to manager link elements that control these context presentation.

The NCL code resulted from the template processing, consists of NCL *contexts* and uses *links* to manager changes between such contexts. The intention behind it is to develop a carousel of videos. The figure 4.9 illustrates such idea.

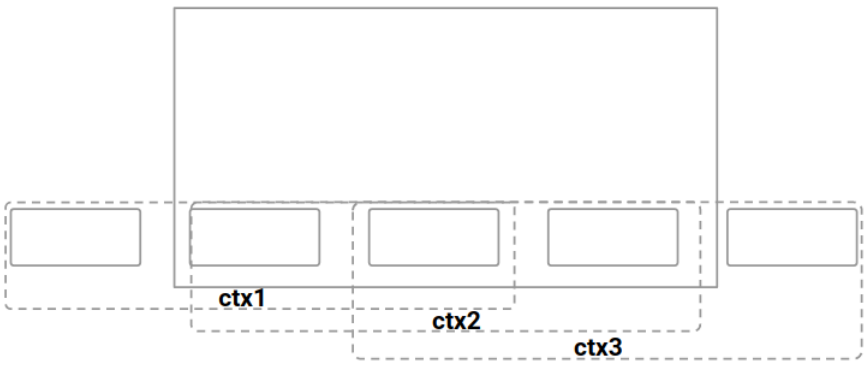

Figure 4.9: Video recommendation logic built through NCL contexts

Inside contexts, there are ambient variables used to control the application's logic. They are properties of Ginga's media object *application/x-ginga-settings*.

PUC-Rio - Certificação Digital Nº 1612822/CA

Below, we describe them:

- service.currentFocus: indicates the value of focusIndex element. It is an integer value;
- selected: indicates that a video from the recommendation list was selected. It is a binary value;
- direction: indicates users' navigation direction. Assumes the value of "left" or "right".

Each context has three ports. One for each video recommendation position: left, center and right. If any of these videos is chosen, the `selected` variable is set to true and a *switch* handles the chosen video according to the application's rule defined in the NCL *ruleBase* tag. These rules checks if the value of *service.currentFocus* is equal to 1, 2 or 3. In case of no video selection, `selected` variable remains false.

Links inside NCL contexts tests, for instance, if the user pressed the `RIGHT_CURSOR` key and if the `service.currentFocus` from the image is equal to 3, then, `direction` variable is set to right and the current context is ceased. Similarly, the same occurs when a user navigates to the left to go back to the previous videos list.

At the end of an NCL context execution, links outside them test the aforementioned ambient variables. If *selected* is true it means that a video from the context was chosen and any other context should not be initialized. Therefore, the application starts the chosen video and ends its execution. Otherwise, it is necessary to start the previous or the next context according to the user navigation direction. Variable *direction* value is tested to determine which context should start in this case.

The first and the last context are special cases because they require only one context initialization. Respectively, next and previous contexts.

Figure 4.10 and 4.11 shows the results for the developed Broadband instance considering, respectively, for the number of lines of code and the number of instructions.

When measuring the number of lines of code, 52 lines were written in `videoRecommendation_base.ncl.j2`, 121 in `videoRecommendation_child.ncl.j2` and 33 for the macros to generate 268 NCL lines. That lead to a *score* of **23.13%**. Employing the number of instructions the following results were achieved: 34 instructions in `videoRecommendation_base.ncl.j2`, 82 `videoRecommendation_child.ncl.j2` and 20 in the macros docu-

PUC-Rio - Certificação Digital Nº 1612822/CA

ment. In this case, 192 NCL instructions were generated and the *score* is **29.17%**.

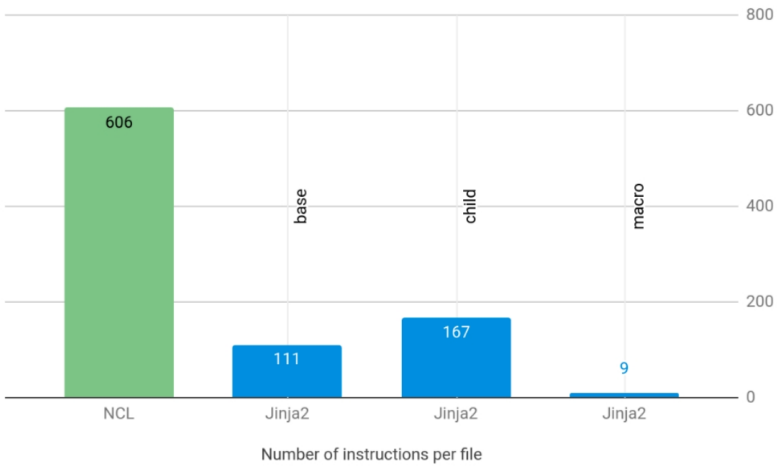

Figure 4.10: Number of lines coded per file employing Jinja2 as template language

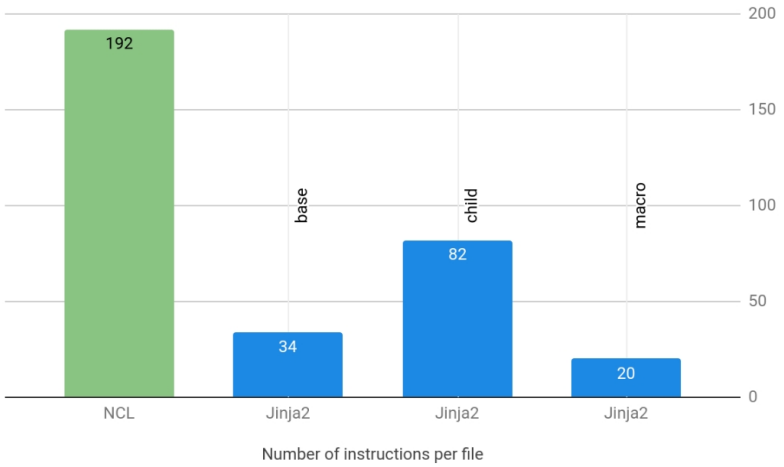

Figure 4.11: Number of instructions typed per file employing Jinja2 as template language

Table 4.5 presents the results for the video recommendation developed using templates in Jinja2. As in the previous scenarios, it considers the development with and without template hierarchy elements and for these two measures the score variable values according to the number of lines and the number of instructions.

PUC-Rio - Certificação Digital Nº 1612822/CA

	Metric	
Case	*number of lines*	*number of instructions*
w/ hierarchy	23.13%	29.17%
w/o hierarchy	26.12%	35.94%

Table 4.5: Video recommendation *score* summary

4.4 Discussion

This section debates about templates implementation using Jinja2 and Mustache. In this discussion, some features provided by these two languages, especially those concerning template inheritance and control structures, are examined. We also review the facility to program templates in them. At last, some remarks regarding the results are leverage.

Jinja2 and Mustache are very unlike in their syntax, especially for template inheritance and control structures, two of the specified requirements. Template inheritance affects directly the evaluation procedure and was discussed in Section 4.1. Regarding control structures, it is only possible to compare iterations, since Mustache only implements it.

The two template languages loop over collections in two different ways. Jinja2 uses the `{% for ... %}` statement to iterate, meanwhile Mustache renders special tags called "sections". Listing 4.13 and Listing 4.14 show snippets of code from Jinja2 and Mustache languages, respectively.

The syntax proposed in Jinja2 is very intuitive and similar to imperative programming languages such as Lua and Python and therefore becomes easier for developers to understand. The same does not occur in Mustache. It is necessary to know its syntax first and the context to understand code behavior.

PUC-Rio - Certificação Digital Nº 1612822/CA

```
1 {% for i in files_list[1].contents %}
2   <media id="{{'image' .. loop.index}}" src="{{'media/' .. i.name
      }}" descriptor="ImageDes"/>
3 {% endfor %}
```

Listing 4.13: Iteration in Jinja2

```
1 {{#index}}
2   <media id='image{{index}}' src='media/{{name}}' descriptor='
      ImageDes'/>
3 {{/index}}
```

Listing 4.14: Iteration in Mustache

Regarding template hierarchy, Mustache implementation only allows inheriting from the main file. However, in Jinja2 any template can inherit from others. Another difference between them is related to their engine implementation. Mustache's engine requires all templates to be explicitly given to it, otherwise, it cannot process data. Jinja2's engine can find and load templates as long as they are in a specified folder. That way, only the main template can be passed and the engine takes care of loading the others.

As stated in the slideshow example in Section 4.1, developing using Mustache language becomes more difficult in more elaborated applications due to its logic-less syntax. For instance, in the video recommendation, it is necessary to make a difference for the first and last NCL contexts, as well as set variables to build the links that control the application's logic. Mustache does not implement "if" statements or any similar feature to declare and handle variables.

Developing these applications using Mustache, requires developers to implement a completely distinct logic compared to the one employed in Jinja2-based templates. Particularly, the NCL document outputted after template processing is different. Comparing these two distinct logic would not be fair.

Therefore, we had two alternatives:

- develop with a similar approach on both languages and compare one to another, but without exploring all the power that Jinja2 offer;
- use more features from Jinja2 and not compare it against Mustache.

We opt for the second and did not implement applications built with Mustache for the last two use cases.

The developed applications use the following Jinja2 features: template inheritance; blocks; macros; calls to macros; iterators and its control variables; conditionals; variable creation and manipulation; arithmetic and Lua code, such as functions to manipulate tables (range, pairs, ipairs, table.insert), Lua length (`#`) and concatenation (`..`) operators.

Based on the three use cases developed it is possible to assure that web-template processing reduces the amount of typed code, as it was expected.

From Equation 4-1 and 4-2 it is seen that the *score* increases when one of these two happens:

- the number of instructions/lines goes down;
- the number of lines/instructions in the NCL code goes up.

Once this last is fixed for an application, so the fraction numerator must change. And this is what happens when hierarchy elements are removed from the code.

PUC-Rio - Certificação Digital Nº 1612822/CA

Also, the two equations demonstrate that the higher the number of instances the higher the *score* is. That explains the high gain acquired in the slideshow in which we use 21 images against the 5 videos from the video recommendation. The additional content has its instances quantity fixed (4) which implies in a fixed *score* as well.

Jinja2 needs exactly 3 instructions for each inherit information plus one to indicate the inheritance itself. Also, each of these instructions takes the same amount of lines. Therefore, the removal of hierarchy elements results in a larger *gain* independently from the metric adopted. A high *gain* in the *number of instructions* in comparison to the *number of lines* can be explained by instructions that take more than one line of code.

PUC-Rio - Certificação Digital Nº 1612822/CA

5
Final Remarks

NCL is a declarative language based on XML. As a consequence, it is verbose which may lead to errors, especially in large documents. With that in mind, many studies have been conducted. Once NCL has many code repetition, one approach is to reuse duplicated elements. One common reuse concept in multimedia documents are templates. They generate a family of documents based on the same common structure. More than easing the work of NCL applications' authors, templates reduce errors and make easy to identify them if any happens.

Given this context, we proposed to answer two research questions introduced in Chapter 1.

First, we asked *RQ-1 How can we improve NCL development based on templates?* We try to answer it by arguing that the usage of web templates in the development of NCL applications is an option to reduce typing errors. To address this question, we define the objective of **Evaluate web templates usage in NCL development**.

In Chapter 4, we developed three use cases: (a) slideshow; (b) broadcast additional content and (c) IBB video recommendation. More precisely, we discussed how the use of Jinja2 and Mustache template languages can support the development of NCL applications. These examples confirmed that web templates can reduce code-writing, independently if the metric used is the number of lines or the number of instructions in the document.

Jinja2 template-based applications avoid code repetition and are less error-prone. Nonetheless, differently from some related works, it did not make it easier. To its authors, it still requires a certain expertise in programming. It is a tough task to generate the final NCL document properly indented with Jinja2. The outputted NCL document is still correct, nonetheless, it becomes less readable and not ease developers task during coding.

In Chapter 4, Section 4.4 we mentioned all Jinja2 features used in our examples. Except for arithmetic, all of them were widely used. Some of them like iterators, conditions, variables manipulation, and the Lua embedded code are indispensable, while others such as template inheritance and macros creation and invocation are used to support the development making it more

PUC-Rio - Certificação Digital Nº 1612822/CA

organized, cleaner and less repetitive.

Despite targeting web development, we were not sure if web engines may be easily applied to develop DTV applications. For instance, on the web, some of these engines run on the server-side, while others run on the client-side only and some can execute on both.

Therefore, we also asked *RQ-2 How can we support web templates processing in the NCL development?* To address this question, we define the objective of **Allow web template engines processing at Ginga**.

To fulfill such an objective, we developed the NCL-formats tool. It aims at assisting the development of NCL applications based on web-template languages. As discussed in Chapter 3, it can be executed as a standalone tool. We also suppose two possibilities for executing it on the client-side.

The first possibility consists of extending Ginga's parser to support templates syntax. It was extended to also handle template documents in Jinja2 or Mustache languages. In this context, the Parser receives the template documents and the padding document. With that, the NCL-formats tool is called to process the given data and generate an NCL document ready to be played by the NCL player. As pointed out in Section 3.3, this way implies modifying Ginga's standard which takes time to be approved and adopted.

Having in mind the drawbacks of the aforementioned proposal, another approach was envisaged. This solution consists of embedding NCL-format as an NCLua script. This way, an NCL Player receives a simple NCL document. The NCL-formats becomes a media element of this NCL and the required information (padding document, template document, and the engine) is passed as a property of this media element. The NCLua script gathers the passed data and calls the required web template engine. The engine fills out gaps on template document with information from padding to generate the final NCL document. After processing, the NCLua script signalizes its ending and the NCL final document is played.

A version of NCL-formats was packed in a rock[1] using LuaRocks package manager.

Reviewing Table 2.1 from Section 2.3, this work uses control structures and template inheritance, however, does not support template components as it was intended. Particularly, it focuses on web developers but does not exclude NCL developers, once its aim at generating applications in NCL. The biggest advantage brought in our proposal is: template-based applications in NCL are not tied to be prepossessed. They can be dynamic and adjustable at exhibition time.

[1]https://luarocks.org/modules/bxl/ncl-formats

PUC-Rio - Certificação Digital Nº 1612822/CA

5.1 Limitations and Future Work

This work has limitations in both the web template evaluation and the NCL-formats tool.

Our evaluation focused only on web-template languages' impact on the code itself. We did not concern the programmers. So as a suggestion for future works, we consider measuring the impact of our work targeting its developers. In particular, we may measure the coding time.

Regarding the NCL-formats tool, it was tested against Jinja2 and Mustache template engines. However, it may be applied to process template documents in any language. Therefore, future works may enlarge it by supporting other template languages. For instance, NCL templates engines such as XTemplate[2], sNCL[3], TAL[4], LuaTPL[5], Lua2NCL and Luar[6]. Moreover, we can also extend it to include as well other NCL formats, such as jNCL[7], NCL-ltab[8] and RAW NCL[9].

In this work we do not explore scenarios in witch the template engine acts as a media. It can be used to create new types of media players. For instance, to create templates that receive a SRC file as a pading document to render subtitles.

Templates usage brings the possibility of processing data to generate NCL applications. With that, the template's engine accepts padding documents with rules in them. Therefore, the engine is responsible for recognizing different kind of input formats, understand its rules and process them.

5.2 Publications

This work was approved to the "VIII Iberoamerican Conference on Applications and Usability of Interactive TV (jAUTI 2019)"[10].

PUC-Rio - Certificação Digital Nº 1612822/CA

[2] https://github.com/joeldossantos/aXT
[3] https://github.com/lucastercas/sncl
[4] https://github.com/TeleMidia/tal-processor
[5] https://github.com/robertogerson/luatpl
[6] https://code.google.com/archive/p/luar-template-engine
[7] http://www.midiacom.uff.br/~caleb/jns/
[8] http://www.telemidia.puc-rio.br/files/biblio/2018_09_dodsworth.pdf
[9] http://github.com/TeleMidia/dietncl
[10] https://webmedia.org.br/2019/en/viii-iberoamerican-conference-on-applications-and-usability-of-interactive-tv-jauti2019/

Bibliography

[1] C. Montez and V. Becker, *TV digital interativa: conceitos, desafios e perspectivas para o Brasil*, 2nd ed. Florianópolis: Ed. da UFSC, 2005.

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

[3] L. F. G. Soares and S. D. J. Barbosa, *Programando em NCL 3.0 2a. Edição Versão 2.1*, 1st ed. Elsevier Campos, May 2011.

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

[5] C. de Salles Soares Neto, C. S. de Souza, and L. F. G. Soares, "Linguagens Computacionais Como Interfaces: Um Estudo Com Nested Context Language," in *Proceedings of the VIII Brazilian Symposium on Human Factors in Computing Systems*, ser. IHC '08. Porto Alegre, Brazil, Brazil: Sociedade Brasileira de Computação, 2008, pp. 166–175. [Online]. Available: http://dl.acm.org/citation.cfm?id=1497470.1497489

[6] E. C. O. Silva, J. A. F. d. Santos, and D. C. Muchaluat-Saade, "JNS: An alternative authoring language for specifying NCL multimedia documents," in *2013 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*, July 2013, pp. 1–6.

[7] E. International, "The json data interchange syntax," December 2017. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[8] R. G. de Albuquerque Azevedo, "LuaTPL: A simple lua-based template engine," 2018. [Online]. Available: https://github.com/robertogerson/luatpl

[9] D. H. D. Bezerra, D. M. T. Sousa, G. L. d. S. Filho, A. M. F. Burlamaqui, and I. R. M. Silva, "Luar: A language for agile development of ncl templates and documents," in *Proceedings of the 18th Brazilian Symposium on Multimedia*

PUC-Rio - Certificação Digital Nº 1612822/CA

and the Web, ser. WebMedia '12. New York, NY, USA: ACM, 2012, pp. 395–402. [Online]. Available: http://doi.acm.org/10.1145/2382636.2382718

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

[11] L. de Macedo Terças, D. de Sousa Moraes, T. de Sousa Lima, M. C. M. Neto, and C. de Salles Soares Neto, "Introducing different levels of reuse to a hypermedia authoring language with macros and templates," in *Proceedings of the 24th Brazilian Symposium on Multimedia and the Web*, ser. WebMedia '18. New York, NY, USA: ACM, 2018, pp. 117–124. [Online]. Available: http://doi.acm.org/10.1145/3243082.3243117

[12] J. A. F. dos Santos and D. C. M. Saade, "Xtemplate 3.0: Adding semantics to hypermedia compositions and providing document structure reuse," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 1892–1897. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774490

[13] C. d. S. Soares Neto, L. F. G. Soares, and C. S. de Souza, "Tal—template authoring language," *Journal of the Brazilian Computer Society*, vol. 18, no. 3, pp. 185–199, Sep 2012. [Online]. Available: https://doi.org/10.1007/s13173-012-0073-7

[14] W3C, "XSL Transformations (XSLT) Version 1.0," 1999, 00020. [Online]. Available: http://www.w3.org/TR/xslt

[15] C. S. Soares Neto, H. F. Pinto, and L. F. G. Soares, "Tal processor for hypermedia applications," in *Proceedings of the 2012 ACM Symposium on Document Engineering*, ser. DocEng '12. New York, NY, USA: ACM, 2012, pp. 69–78. [Online]. Available: http://doi.acm.org/10.1145/2361354.2361369

[16] ITU, "Integrated broadcast-broadband systems," ITU, Geneva, Tech. Rep. ITU-R BT.2267-6, October 2016. [Online]. Available: https://www.itu.int/dms_pub/itu-r/opb/rep/R-REP-BT.2267-6-2016-PDF-E.pdf

PUC-Rio - Certificação Digital Nº 1612822/CA