Textual Authoring of Interactive Digital TV Applications

Roberto Gerson Albuquerque Azevedo¹,², Carlos de Salles Soares Neto¹, Mario Meireles Teixeira¹, Rodrigo Costa Mesquita Santos¹, Thiago Alencar Gomes¹

¹Laboratory of Advanced Web Systems
Departamento de Informática – UFMA
Av. dos Portugueses, Campus do Bacanga
São Luís/MA – 65080-040 – Brasil

²Pontifical Catholic University of Rio de Janeiro
Rua Marquês de São Vicente 225
22453-900 Rio de Janeiro, RJ, Brazil
+55 21 3527-1500 Ext: 3503

{roberto}@laws.deinf.ufma.br, {csalles, mario}@deinf.ufma.br,
{rodrim.c, thiago}@laws.deinf.ufma.br

© ACM, 2011. 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 EuroITV’11,
http://dx.doi.org/10.1145/2000119.2000169.
Textual Authoring of Interactive Digital TV Applications

Roberto Gerson Albuquerque Azevedo¹,², Carlos de Salles Soares Neto¹, Mario Meireles Teixeira¹, Rodrigo Costa Mesquita Santos¹, Thiago Alencar Gomes¹
¹Laboratory of Advanced Web Systems
Departamento de Informática – UFMA
Av. dos Portugueses, Campus do Bacanga
São Luís/MAC – 65080-040 – Brasil
{roberto}@laws.deinf.ufma.br, {csalles, mario}@deinf.ufma.br,
{rodrim.c, thiago}@laws.deinf.ufma.br

¹Pontifical Catholic University of Rio de Janeiro
Rua Marquês de São Vicente 225
22453-900 Rio de Janeiro, RJ, Brazil
+55 21 3527-1500 Ext: 3503

ABSTRACT
Authoring tools for hypermedia languages usually provide visual abstractions, which hide the source code from the author aiming to simplify and accelerate the development process. Among other drawbacks, these abstractions modify or even break the communication process between the author and the language designer, since these languages were designed to be readable and understandable by its target audience. This paper presents a textual approach to hypermedia authoring that does not have these inconveniences, but rather uses typographical accessories, such as program visualization, hypertextual navigation, and semi-automatic error correction. The proposed approach exploits concepts known to the author and does not imply in extra cognitive overload. A use case is presented, namely the NCL Eclipse authoring environment, for Nested Context Language, the Brazilian Digital TV and ITU-T standard.

Categories and Subject Descriptors
1.7.2 [Document Preparation]: Format and notation hypertext/hypermedia, Markup Language, Standards.

General Terms
Documentation, Design, Languages, Verification.

Keywords
Textual Authoring, Hypermedia Authoring, Nested Context Language, Ginga-NCL, NCL Eclipse.

1. INTRODUCTION
This paper proposes a set of techniques feasible for implementation by a textual authoring environment to declarative applications for Interactive Digital Television (iDTV). Commonly, the construction of iDTV applications employs some declarative languages, such as XHTML [1], SMIL [2], or NCL [3].

Declarative languages focused on creating non-linear presentations appear to be most suitable for iDTV applications development. Non-linear presentations are based on spatial and temporal synchronization specification of audio, video, image, text, and other media. Because these languages relate various media types, media content is often apart from the document itself, allowing the author to focus on what is most relevant in that authoring phase, namely the presentation behavior.

On the other hand, applying visual abstractions on the textual source code is frequently useful for guiding and accelerating the development process, since they aid and reduce inferential reasoning stages [4]. However, they also tend to be more restrictive since graphical notation cannot always support all types of abstractions. The visual abstraction is by its nature more restrictive than textual ones, since it implies in some concrete representation on the display device. This causes the problem of authors’ premature commitment, interfering with the abstraction process, thus diverting the author’s attention to details that are not important at that particular moment (such as object size, color and position on the screen) [5]. Furthermore, it is important to highlight two more potential drawbacks when using visual abstractions.

Firstly, hypermedia language design (as in the previously mentioned SMIL, NCL, and HTML) is already conceived in order to help the human conception of the document. Trying to use visual abstractions as a means to hide the source code is not always welcome, as it may hamper the communication process between the language designer and the programmer [6]. Visual abstractions may also imply in extra cognitive overload caused by the interpretation and association of a given symbol with concrete and specific aspects of the authoring language. Overall, it is essential to maintain the language designers’ view about the authoring process instead of creating a completely new language abstraction.

Secondly, the different visual abstractions are justified by the complexity of the hypermedia authoring process, together with the fact that there are also different user profiles. Graphic designers, for instance, expect to work at a higher abstraction level, closer to the hypermedia presentation. Conversely, programmers may be more pleased to work directly with the source code. Moreover, advanced users and beginners have different notation needs [7]. Thus, an authoring environment based on a sole visual abstraction may not satisfy all the specification requirements that a textual approach does.

Nevertheless, textual authoring also carries some pitfalls on its own. For instance, working directly with the source code is usually more error-prone than using visual abstractions. And
error-free applications are a wish that dates back to the beginning of computer science [8]. In practice, increasing source code size also increases the probability of finding programming errors. In order to help users detect and fix programming errors, two major approaches are found: formal software verification [9]; and the increasing usage of Integrated Development Environments (IDEs), which not only help users detect errors, but also help them fix those errors.

The usage of IDEs aims to minimize the effort during the code debugging phase, sometimes offering semi-automatic error correction. In other words, the IDE itself detects errors and points out to the author certain alternatives, one of which must be chosen to make the code correct. A lot of attention has been given to IDEs that offer this kind of feature in the domain of imperative languages [10] [11], but little has been done, though, in the declarative languages field.

This paper proposes to integrate some techniques into authoring environments in order to exploit the declarative paradigm itself. In the same way that the task of programming in declarative languages is alleviated by using domain metaphors, we believe that the techniques proposed in this paper also help facilitate the hypermedia application textual authoring process by using abstractions that are closer to the source code being written. Our contributions encompass hypermedia navigation within the document; media object preview; and program visualization techniques, among other aspects. Furthermore, we outline an error correction process and a contextual source code suggestion to be integrated into the authoring environment.

As a testbed for our proposals, we designed and implemented a plug-in for the Eclipse IDE, named NCL Eclipse, especially focused in NCL (Nested Context Language), the declarative Brazilian standard for Terrestrial Digital TV System and ITU-T standard for IPTV systems. This tool includes all the techniques to be described in this paper and has become a de facto standard for NCL development in Brazil and several other countries throughout South America.

The remainder of this paper is structured as follows. Section 2 discusses some related work. The proposed techniques to improve textual authoring are presented in Section 3. Section 4 discusses NCL and presents NCL Eclipse, illustrating its basic features and architecture. Finally, Section 5 concludes with some final remarks and comments on future works.

2. RELATED WORK

In recent years, concerns about visual programming languages have increased, though these systems have been discussed since a long time [12][13][14][15]. During the 50's and 60's several works showed the integration between documentation and source code [16][17]. Authoring based on visual languages is defined by Myers as visual programming [16]. Visual programming-based systems allow the specification of a program through visual elements, in two or more dimensions. According to Petre [8], many visual programming advocates say visual representations are better just because they are graphical. Moreover, the literature usually confuses the terminology used for visual authoring environments. In the context of this paper we will use that one defined by Myers. According to Myers [18] there is a crucial difference between visual programming (VP) and program visualization (PV). In VP, the graphical part of the system becomes the program itself, while in PV the program is specified textually and the graphical part is used to illustrate a few aspects of the program.

About iDTV authoring process, it is important to note that professionals involved in this process are not just programmers. There are also graphical designers, editors, and perhaps television reporters. In general, these other professionals are not particularly interested about programming code, but rather with the final presentation. Typically, a WYSIWYG approach sounds better to a non-programmer professional. The following subsections discuss the real necessity for, and some peculiarities, about visual and textual authoring for an iDTV development environment.

2.1 Visual Authoring

The main goal of visual authoring tools is the abstraction of programming related aspects. This goal, according to Price [19], is achieved through the usage of non-textual notations, establishing links between graphic symbols and individual syntactic parts of the language, besides representing implicit or explicit relationships through the usage of diagrams. After examining and reviewing the results from several studies [12][13][19][20][21][22], we can conclude that it is not always efficient the usage of visual programming. Navarro-Prieto [20] identifies some variables that should be taken into consideration when choosing a visual authoring environment, including: language goals, programming tasks (flow control, debugging, execution and maintenance), and the application of program visualization techniques. The authoring of hypermedia applications, for the Web or more recently for Digital TV seems to be a classic case where the use of visual tools, of WYSIWYG type, would be naturally applicable. Apologists of those tools argue there is nothing better than it to directly visualize the final product being developed, since that brings the author closer to the end user or TV viewer experiences. Moreover, such applications are not governed by a single timeline, but by numerous temporal chains, as many as there are opportunities for user interaction in the application. This makes it hard to navigate and edit the final product during authoring time, what may turn out a WYSIWYG approach impractical.

In fact, as Petre [7] mentions, text and graphics are not necessarily equivalent and should not be simply replaced for one another. In some cases, as Whitley [22] comments, a tabular or graphical representation has proved to be noticeably superior, as in the case of spreadsheets, which became a standard in place of using textual formulas. In other contexts, Whitley continues, as in the case of flowcharts, these ended up being gradually abandoned for pure and simpler textual representations. For instance, textual representation of algorithms, with appropriated indentation and code coloring, could be clear than visual ones.

2.2 Textual Authoring

Nowadays, one of the most widely used programming languages is Java [23]. There are several works proposing visual programming for Java [24][25], but Java authoring still occurs in a purely textual approach using Integrated Development Environments (IDEs). In the case of Java, there is Eclipse [10] platform and its countless contributions to textual authoring of imperative languages, such as the CDT [26] for C++, LuaEclipse [27] for Lua, and so on.

After a careful study of tools for textual authoring focused on imperative languages, we observed that a significant number of them implements mechanisms to identify errors during authoring.
phase [28][11]. Some of them go beyond, providing code correction features [28]. However, the same variety of tools is not observed in the declarative paradigm. One of the few studies in this area suggests an environment able to validate XML documents in general, based on an XML Schema [29] specification. Nevertheless, this approach is not applicable to all declarative languages, considering the existence of languages – like NCL – where an XML Schema would not be enough to describe all the language aspects.

There are a lot of researches indicating the reasons for the difficult in reading visual languages. According to the result of Green and Petre work [21], textual languages have an average of 50% faster reading response than visual languages. Another motivation for textual authoring is that reading and understanding programs is an important component of the professional programmer skills. Typical tasks when reading a program are its: navigation, execution, and debugging. According to Pennington [25], it is estimated that 50% of the programmers’ time is spent with maintenance and evolution of legacy code, whose main focus is reading and understanding the source code. Some authors [19][21] argue that the visual authoring problem is that programmers do not interpret the graphical notations used in visual languages the same way. Moreover, source code conveys information in an unambiguous and clear form. In the same way, Bouwhuis [30] adds that a professional programmer is a reader focused on the program’s primary goal, and guided by hypotheses. Programmers need authoring tools that, according to Price [31], should use typographic accessories, graphical notations, animations and even human-computer interaction techniques in order to quickly highlight and make available important information on the source code. For instance, in the website development, expert programmers tend to feel limited with the WYSIWYG visual mode and frequently switch to textual one, since they can express more precisely how they actually want the application to perform. Additionally, visual tools tend to generate poor code, full of redundancies, and often illegible. Conversely, designers or even beginner programmers, hardly venture into the field of programming commands, preferring to use visual tools.

There are several studies in this area focused on the benefits of textual authoring for imperative languages, however very few of them approach the same problem in declarative languages. Generally, the authoring of applications for interactive TV is carried out in a language that follows a declarative paradigm, such as XHTML [1], SMIL [2], or NCL [3]. This paradigm focuses on the ability of the author to solve the problem by describing its final intention rather than specifying a step by step sequence of commands. Other characteristic is its smooth learning curve and ease of use without loss of expressive power [32]. It is usually more straightforward and easier to visually abstract declarative languages because they have a higher abstraction level than their imperative counterparts. This can be confirmed by the great number of visual tools for the declarative paradigm, discouraging a purely textual authoring environment for hypermedia applications. One of the aspects discussed in this paper is the integration of typographic accessories and program visualization elements to support the textual authoring process of hypermedia documents – with no intention of being a substitute for that approach. When offering a textual and visual authoring environment, it is important to avoid hiding the textual source code while enriching it with auxiliary graphical representations.

### 3. Hypermedia-Based Textual Authoring

Hypermedia languages have, implicitly or explicitly, a lot of information that can be exploited by an authoring environment in order to improve the author’s interaction and reasoning on the source he/she is producing. This information may be used, for instance, to facilitate document navigation, preview information that is not in that same position in the source code or even to decide which information would be most relevant for someone editing a given portion of the code. Furthermore, hypermedia languages define certain structures that have a graphical representation more intuitive than its textual counterpart and thus may render the author’s work easier. The following subsections discuss some techniques that together aim at making available some graphical abstractions which are not very intrusive to the textual authoring task.

#### 3.1 Navigation in Hypermedia Documents

Hypermedia languages allow the creation of non-linear content using an intrinsically linear media – text – and thus need some mechanism to make reference to other parts of the document being edited in order to avoid textual redundancy. Firstly, it is necessary to divide the document into distinct parts called elements (the term element here has the same meaning used in markup languages). Those elements need unique identifiers so that they can be referenced in other parts of the document. This identification mechanism is common in languages such as XHTML and SVG [33] and sometimes is included in the language design to allow the reuse of some elements.

However, such a mechanism has some drawbacks as it may render the reading of the source code more complex. The author may need to move back and forth several times through different parts of the document so as to achieve a better understanding of that hypermedia document. It is neither an easy nor a quick task to find out in which part of the document are defined the elements which make reference to the piece of code currently being edited. This search also causes the additional inconvenience of diverting the author’s attention to a portion of the code distinct from the one being edited. The definition of the desired element may not ever be in the same file the author is currently working on. It is possible to mitigate this problem by allowing a non-linear navigation through the source code by using hyperlinks. The authoring environment could then infer the relationship information present within the source code based on information obtained from the source code itself. Figure 1 details this mechanism through an example where the reference to a given element YYY in document A can become a hyperlink to the definition of that element in document B. The choice of a hyperlink to perform this type of association is perfect as it does not distract the author and will only be used if the author needs to check the definition of the referenced element. In that case, he/she can easily navigate to the definition of the element and then come back to the previous edition spot as needed.
Figure 1. Hyperlink navigation applied to some hypermedia language.

When implementing the hyperlink mechanism and its corresponding navigation, the authoring environment is indeed realizing that the source is in fact a hypertext. This approach is common not only in the domain of hypermedia languages but also for general programming languages, such as the case of Java Development Tool [34] and the Visual Editor Project [35], where hyperlinks are used to access classes, functions and variable definitions. In this latter case, the use of hyperlinks is strictly unidirectional, from a source to a destination. However, in the context of hypermedia authoring another feature may be useful, namely the utilization of reverse hyperlinks. They allow the author of a document, when editing a given element, to receive feedback about what parts of the source code make reference to that element. Figure 2 details those reverse hyperlinks where the author, when editing element YYY, can trigger a hyperlink that will change the focus to parts of the document where that element is referenced, in this example by elements ZZZ and WWW. Aware of that information, the author can then follow those hyperlinks to verify how the change in element YYY could possibly have affected the ones that make reference to it.

This concept is not very much exploited in practice, probably because an element with many references to it can generate a great amount of reverse hyperlinks and they would be very difficult to be rendered on the display device. This problem can be solved with the use of filters but it may not be trivial to infer all the reverse references, mainly when working with more than a document file. One viable solution is to create project files to encapsulate all the relevant documents.

Figure 2. Reverse hyperlinks.

3.2 Media Preview

The utilization of hyperlinks allows a better navigation through the document when compared to a sequential reading. Nevertheless, navigating the document to check the referenced object can be quite uncomfortable and can potentially distract the author who is often interested only in understanding the semantics of that hypermedia document. He/she would perhaps rather have a quick preview of the referenced content without the need to move from his/her original position in the source code. The authoring environment can minimize this problem by providing a preview mechanism in order to be able to render the hyperlinked object in a small window, thus with very little intrusiveness in the authoring process.

A hypermedia language such as NCL usually has a clear separation between the program specification and the associated media objects. The preview mechanism could then be used to render the content of the media the program itself is using as shown in Figure 3. In such a case, the authoring environment is performing like a hypermedia system in its own. This is especially important since the author will come across familiar concepts when interacting with the authoring environment alleviating any extra cognitive overload.

Figure 3. Media preview.

Object preview has become a somewhat popular feature in modern user interfaces, such as those found in Web 2.0 applications, window preview in panels, URL preview in search engine results, operating system task bars, folder and document preview in file system explorers, and so on.

3.3 Program Visualization

As discussed in Section 2, the employment of program visualization (PV) can emphasize, through graphical abstractions, some peculiarities of the source code that can be more easily understood in visual form. In a textual authoring environment, PVs may be accessed through a shortcut key, by clicking an element or simply by passing the mouse over an element. In hypermedia languages, there are some elements that have direct or inferred mappings to PVs, such as region definitions, cascade relationships, pertinence relationships in elements defining contexts, media representation in a temporal line, and so on. The utilization of PV allows the author to view some information that graphically represents small excerpts of the source code. Those graphical representations cannot be edited by the author, serving rather as a support to enhance source code understanding by the author. However, PVs should be used in a way that does not hide the author’s text. Supplementary windows can again be used to the visualization of PVs. Another alternative would the employment of outline windows. Both alternatives have in common the characteristic of not overlapping text editing, which should always have the main focus of the author’s attention.

3.4 Semiautomatic Code Correction and Suggestion

The process of semiautomatic source code correction has been preferably used, instead of an automatic correction process,
because it considerably lowers the probability of inserting new logical errors (in other words, resulting in an application with a different purpose than the one the author initially intended). This is particularly true when there is more than one possibility of error correction, what may result in valid source codes, but with different meanings. For imperative languages, it is possible, although improbable, that even different corrective actions to a single semantic error may result in the same application logic. In the specific case of declarative languages, however, it does not apply, since the multiple possibilities of error corrections are intrinsically related to the resulting application logic.

The semiautomatic code correction process can be divided into six general steps (Figure 4): (i) identification of the errors present in the source code by a validation process (for declarative languages); (ii) computing and (iii) suggestion of the possible corrections (depending on the error, it is possible to fix it with more than one corrective action); (iv) choice of one the proposed corrections by the user; (v) execution of the chosen action; (vi) correction of any other part of the code affected by the correction application. Figure 4 presents this process in a schematic way, representing each one of these steps and the mutual dependence among them.

The user is generally the one responsible for triggering the correction, through the message (1) of Figure 4, either explicitly or through some change in the source code. In step (i), source code validation begins, resulting in error messages or alerts about that source code. The error messages, informed as a string or in a more elaborate data structure (that may include other information, such as the element related to that error, its position, etc.) are sent over message (2) in order to make it possible to determine the corrections. The unique identification of the error message becomes necessary in order to prevent ambiguities from appearing in the process of determining these corrections. Figure 5 presents a simple and explanatory example. In NCL, for instance, the attribute “region” of a <descriptor> element must have the same value of the attribute “id” of a <region> element. In this figure, the user typed “RGText”, but the possible values could be “rgScreen”, “rgVideo”, “rgImage” or “rgTexto1”. It is easy to notice that the user most likely intended to type “rgText1” and not the other values. Other possible corrections would be: add a new region with “RGTexto1” identifier; and even remove the attribute region of the <descriptor> element. Therefore it becomes clear that determining the possible values of an attribute and filtering it correctly (avoiding in the example the three improbable correction suggestions – “rgTela”, “rgVideo” and “rgImage”) is also part of step (ii).

Step (ii) then results in corrective actions that can be applied to the code, assembled in message (3). Step (iii) receives this list of actions and turns it into a list of messages that can be understood by the user, shows them on the computer screen, and leaves him/her to choose one of them. This group of user-friendly messages is then combined in message (4). By choosing one of these actions, step (iv) is performed by message (5) coming from the user. This action is then sent to message (6), resulting in its execution by step (v) and subsequent modification of the source code. However, it is possible that such a change is not atomic, requiring step (vi), which adapts the code according to the corrective action selected. A simple example, for instance, is the need to reformat the code after applying the correction.

Step (ii) is closely related to contextual content suggestion (autocomplete), namely to predict a phrase or text that the user would like to insert. It is very likely that the user has done something close to what he/she really intends and has made only some simple mistake, such as a typing error. In XML-based languages, which often use pointers for identifiers of elements, a common mistake is to erroneously type the name of a previously declared element. However, the mistyped name itself could be a clue for a solution to that mistake. To extend the process in Figure 5 to an automatic correction process is equivalent to removing message (5) and using some heuristics to infer the error correction to be used. The semiautomatic approach was chosen in our work to avoid that new errors be inserted by the correction process itself, what would further confound the author. This semiautomatic approach also allows the author to have full control and be informed about all changes that may occur in the text. To keep the author aware of any implications of his/her decisions on the code is an important premise. The semiautomatic code suggestion process, from step (i) to step (iii), may be performed dynamically at the time of authoring, that is, while the user interacts and modifies the source code, reflecting the latest changes. In an automatic process, once it is fired, it cannot be stopped at a specific time, resulting whenever necessary in modifications to the source code. Modifying the source code while the user is editing it tends to be an even greater disadvantage, which advises against the use of the automatic approach during authoring time, that is, at the same time the author is interacting with the code.
NCL AND NCL ECLIPSE

This Section discusses NCL and presents a proof of concept implementation of the above proposed techniques in an authoring environment for NCL, called NCL Eclipse.

4.1 NCL and NCM

NCM (Nested Context Model) is a conceptual model focused on hypermedia applications, providing support for defining temporal and spatial relationship among media objects, for content and presentation adaptation, and also for distributed exhibition of an application on multiple devices. As will be detailed bellow, a special kind of relationships among media objects is the user interaction. NCM is the conceptual model behind NCL (Nested Context Language), the language used as a use case by this paper.

As NCM, NCL does a strict separation between media content and their relationships. Media contents are individually codified using media specific codifications, for example, RTF for text, PNG or JPEG for images, MP4 or AVI for videos, and so on. These media contents are imported by the NCL document and represented by media objects. This way, NCL has its focus on specifying the temporal and spatial relationships among these objects, and it does not care about media content specification.

Because of that, NCL is commonly described as a 'glue language'.

An NCL document is divided in two main parts: <head> and <body>. These parts must be inside a root <ncl> tag. Elements inside <head> inform object position on the screen (<region>), how the objects are initially presented (<descriptor>), relations that can be used to relate objects (<causalConnector>), rules that can be followed to content adaptation (<rule>), transition effects (<transition>) and metadata (<meta> and <metadata>). In order to provide a better organization, these elements are grouped in bases (<regionBase>, <descriptorBase>, <connectorBase>, etc.).

Nodes and their relationships must be inside the <body> part. Media nodes (<media>) are the media objects discussed bellow. Inside these media nodes, it is possible to define content anchors (<area>) – representing a portion of the media object content – and property anchors (<property>) – representing some object property, like its positioning, etc.). Composite nodes group other nodes, and can be specialized in contexts group nodes that have semantic relation (defined by the user). Switch nodes allow for alternative presentations, depending on defined rules (e.g., based on user contextual information). Link elements specify the relationships among nodes, based on some previously defined relation (causalConnector).

NCL exhaustively uses references among elements. This is justified mainly because it improves the reuse of the language. For example, by defining a relation once (using the <causalConnector>), this relation can be reused by different relationships (<link>) among different nodes. Another example is the reuse of descriptors and regions. A media element can refer to a descriptor, which informs how this media will be initially presented. Also, descriptor elements can refer to regions, which inform in what position media objects that reuse this descriptor will be initially presented. All these references in the language bring hidden dependencies problems to programmers and can induce them to generate semantic mistakes (e.g. refer to elements that do not exist or to elements that are not of the required type). In this scenario, the semiautomatic correction and contextual content suggestion techniques discussed above could become truly important features capable of guiding programmers towards rapidly and correctly develop their applications.

4.2 NCL Eclipse

In order to experiment with the techniques proposed by this paper, we designed and implemented a textual editor focused on hypermedia languages named NCL Eclipse, made available as a free software at the NCL Eclipse website1. Its source code is available at Ginga Community2. This proof of concept implementation is specialized to NCL textual authoring and also implements some minor visual aids, like program visualization (PV) mechanisms. As proposed by this paper, all visual mechanisms attached to the editor do not interfere in the engineering-to-user communication through NCL language.

NCL Eclipse was developed as a plug-in to Eclipse IDE. Eclipse has become a popular IDE used by Java programmers. Moreover, because of its extensible architecture, it is not restricted to Java and we can find several other plug-ins to other languages, like PHP, HTML, and so on. It is also easy to find iDTV technology based plug-ins, e.g. Lua – Lua Eclipse [27]. In this way, the use of Eclipse as a starting point to implement NCL Eclipse creates also the possibility to integrate all those plug-ins into a single and robust IDE aimed at the development of iDTV applications.

The target audience of NCL Eclipse is mainly programmers. Advanced programmers may be benefited by NCL Eclipse in order to accelerate the development through the use of features such as content suggestion and syntactic and semantic validation. On the other hand, beginners may be benefited by improving their NCL skills. For example, content suggestion is able to show what a user will be able to do in his/her next step while developing an application. In the same way, all the validation features and content correction support teach beginners what is wrong with his/her code and how to correct it. Also, the integration with NCL Club [35] aims to make application examples available to beginners that can be used as a starting point to learn NCL.

In order to support marking error validation we have also developed in a previous work an independent library, called NCL Validator [36]. This library is independent from NCL Eclipse and also can be used by other tools. Nowadays, NCL Validator has been integrated with Composer [37] and it is also available as a standalone version [http://www.laws.deinf.ufma.br/nclvalidator/download.html]. It implements syntactic, structural and semantic validation to NCL language and has support to warning and error message.

1 http://www.laws.deinf.ufma.br/nclclipse
2 http://www.softwarepublico.gov.br
internationalization. Since its first version, NCL Eclipse is fully integrated with NCL Validator in order to support incremental validation while the user is editing NCL source code.

4.3 NCL Eclipse Evolution

The first version of NCL Eclipse (1.0) included support to: syntax highlighting, folding (which allows the author to hide parts of the source code), step-by-step tools to speed up creation of NCL document (commonly called Wizard tools), auto-formatting support, document marking error validation, contextual content suggestion and an outline view. Figure 6 shows the content suggestion support of NCL Eclipse running. As can be noted, no visual mechanism was supported by NCL Eclipse in this first version.

The second major change in NCL Eclipse came about in version 1.4. This version included support to: program visualization, media previews and hypertext navigation. Additionally, a new plug-in aimed to integrate NCL Eclipse with NCL Club was included in the same package. As pointed out before, the integration with NCL Club allows for beginners to start learning NCL based on real-world examples. The internationalization support for English, Spanish and Portuguese was also included in this version. Figure 7 shows a <region> program visualization example. As we can see, NCL Eclipse uses an auxiliary tooltip to present the program visualization, in order not to divert the programmer’s focus on the source code (which is the most important aspect about textual authoring).

The NCL Eclipse current version (1.5) has included semi-automatic code correction and contextual content suggestion improvements. Figure 8 shows the semi-automatic code correction running.

NCL Eclipse evolution has been based mainly on NCL programmer community feedback. Additionally, the ISDB-Tb standard growth, which has been being adopted by several countries, especially in South America, has contributed to increase the number of NCL programmers. Based on the number of NCL Eclipse download requests – more than 3,000 – and the number of Ginga Community users – about 10,000 users, including designers and programmers – we can argue that NCL Eclipse nowadays is perhaps the most widely used tool for NCL development.

5. CONCLUSION

The construction of interactive TV applications poses new challenges to designers and programmers alike. At a first glance, it could be considered as only a more specialized form of software development where we could use the same techniques, reasoning and tools that have been around for decades regarding computer application development.

As a matter of fact, iDTV applications not only demand a new approach for an efficient communication with the end user (now, the viewer) but it also introduces new languages, paradigms and tools. Generally, the authoring of iDTV applications uses some language that follows the declarative paradigm, such as XHTML, SMIL or NCL. However, although there are several tools (IDEs) to aid the development process in imperative languages such as Java and C++, the same is not true for the declarative language world. NCL, in particular, although based in XML, has some peculiarities that demand a more expressive formalism than XML Schema to fully describe a given NCL document.

Being the TV prominently a visual environment, many advocate that visual programming tools would be most adequate as they would bring the programmer closer to the final product. However, we argue otherwise because the use of visual
abstractions usually tends to interfere with the programmer’s understanding of the language and instead of accelerating the development process, it can actually render it slow and cumbersome.

Thus this paper proposed and discussed several techniques to enhance the programmer’s experience when performing the textual authoring of iDTV applications. These techniques have been discussed in the context of a declarative language, NCL and implemented as a plug-in for the Eclipse IDE. This tool, named NCL Eclipse has some very useful features, such as: tag and attribute coloring; element folding; automatic code formatting; document validation; code suggestion; and document playback. In more recent versions, due to the intense feedback from its user community, NCL Eclipse introduced new visual programming techniques; media objects preview; and hypertextual navigation within the NCL document itself. In this way, NCL Eclipse takes advantage of the intrinsic hypermedia nature of the NCL document and employs it to facilitate iDTV program development. Finally, in its latest version, 1.5, NCL Eclipse implemented the semi-automatic code correction and suggestion feature discussed in this paper which is innovative regarding the area of declarative hypermedia languages.

NCL Eclipse though is more than a mere proof of concept of the techniques presented in this paper. It has become the de facto standard tool for the development of NCL iDTV applications, in Brazil and many countries across South America, with a solid and enthusiastic user community.

Up to now, we have received strong positive feedback from NCL users regarding NCL Eclipse, which have somehow directed the tool’s evolution. However, we have been very careful to stick to the textual authoring approach which we believe is the backbone of our tool. As future work, we intend to perform usability tests with NCL Eclipse in order to fully assess the benefits (or even difficulties) it may bring to its users. We also have the intention of comparing both visual and textual approaches when applied to the authoring of iDTV applications.

6. REFERENCES


[29] Shivadas, A. Intelligent Correction and Validation Tool for XML. Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science. 2001.


