Considering Non-functional Aspects in the Design of Hypermedia Authoring Tools

Bruno Seabra Lima   Luiz Fernando G. Soares   Marcelo F. Moreno

Pontifical 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

{bslima, lfgs, moreno}@telemidia.puc-rio.br

ABSTRACT
Hypermedia content workflows involve many environments and actors, from the content producer to the service operator, from the editing studio to the service head-end. They all have different needs on hypermedia content creation and editing. Nowadays, even viewers are demanding tools to enrich content. But currently a single authoring tool cannot fulfill their different requirements. This paper deliberates on the importance of non-functional aspects in the development of new hypermedia authoring tools. It also proposes an architecture that enables tools to meet these different authors' requirements, relying on its extensibility, customizability, robustness and scalability.

Categories and Subject Descriptors
I.7.2 [**Document Preparation**]: Hypertext/hypermedia
D.2.6 [**Programming Environments**]: Integrated environments

General Terms
Design, Languages.

Keywords
Nested Context Language, NCL, hypermedia authoring tool, extensibility, scalability, non-functional requirements.

1. INTRODUCTION
In general, hypermedia applications can be partitioned into a set of declarative applications and a set of imperative applications, depending on the programming paradigm used to create their entry-point entities. However, hypermedia applications are usually hybrid, in the sense that they may have both declarative and imperative entities, which is a design decision taken by content creators based on their preferences, skills and, ideally, on how easy the required tasks can be specified and performed.

Authoring hypermedia applications using imperative languages like Java, C and C++ is more complex and more error-prone than using declarative domain specific languages, such as NCL [34], SMIL [8], SVG [40], and X3D [38]. Declarative descriptions are easier to be devised and understood than imperative ones, which usually require intended programming expertise.

No matter which programming paradigm is used, writing source code for complex programs without any help of development tools can be very difficult. Several works in the literature report the benefits of using authoring tools in the process of creating hypermedia applications. It is well known that web pages have got their popularity after sophisticated authoring tools have appeared.

Many reports also describe evaluations of hypermedia authoring tools based on author's needs, that is, on the functional aspects a tool should provide. Two kinds of tools are always on author's mind [36]: textual authoring tools that can help authors in coding (for example with autocomplete features) and debugging; and graphic authoring tools that abstract as much as possible the language complexity. Usually, expert programmers prefer the first tool type and report that they are easier than the second type, which is the preference of novice programmers that, in contrast, consider them much easier than the first type [35].

Graphic authoring tools usually offer multiple views, in order to represent different aspects of the language's conceptual model. A view can be considered an abstraction that allows for thinking about an application under a certain perspective, as for example, its flow in a timeline. Textual authoring tools usually also offer different views – indeed some graphic views besides the textual one – to easy the authoring process.

No matter its type, an authoring tool has a strict relationship with the semantic model it aims to support. In spite of having sophisticated graphic interfaces, with several resources for direct manipulation of application entities, an author usually cannot neglect the fundamental concepts of the underlying language of an authoring tool. The more complex the application, the greater is the need to master the language's conceptual model.

In an authoring tool, each view has its appropriate high-level data model. At the same time, the authoring tool has itself a central data model where all editing actions take place, usually a low-level data model suitable for machine manipulations. Different data models in a same tool might cause several non-functional problems.

Usually, the semantic gap between the central data model and a view's data model makes difficult the translation process between them. To bypass the problem, some tools are not extensible, working with a limited number of views; some others perform only one-way translation, without allowing synchronization between their views (changes on a view do not reflect automatically on the others); still others are resource consuming, requiring high-end platforms or working with excessive delay that compromises its use for live editing, and worse, presenting reliability problems.

Hypermedia authoring tools should be resilient and also adaptable to the storage or transport system. Once created, applications must be stored or transmitted. For example, a hypermedia authoring tool for Digital TV (DTV) must translate the syntax of its internal data model to the transfer syntax of the transport system in use, which varies from a DTV system to another one.

This paper discusses the importance of *non-functional* requirements in the design of hypermedia authoring tools. Non-functional requirements, hereafter NFRs, can be defined as system requirements that are related to constraints and qualities [27]. The literature lists a variety of NFRs [10], but this paper focuses on the following: performance (efficiency in memory and CPU use), customizability, extensibility, scalability, portability and reliability.

This work also proposes an architecture for hypermedia authoring tools that addresses these NFRs and presents its implementation in the second edition of *Composer*: an NCL authoring tool. NCL (Nested Context Language) is the declarative language of ISDB-T_B terrestrial DTV system [1, 24] and also an ITU-T Recommendation for IPTV services [25]. The remainder of the paper is organized as follows. Section 2 discusses some related work. Section 3 introduces the proposed architecture. Section 4 presents the Composer II as a proof of concept. Section 5 is reserved for conclusions and future work.

2. RELATED WORK

When using an imperative language, authors need to know standardized libraries and to be acquainted with the language programming paradigm (for example, object-oriented, in the case of Java and C++) to create applications. Authoring tools can be very helpful to easy this task, especially for non-programmer users.

JAME Author, *iTV Suite Author* and *AltiComposer* are authoring tools to create DTV applications based on Java language. Although these tools focus on imperative content production, they follow the declarative archetype, showing up the importance of this programming paradigm in developing hypermedia applications.

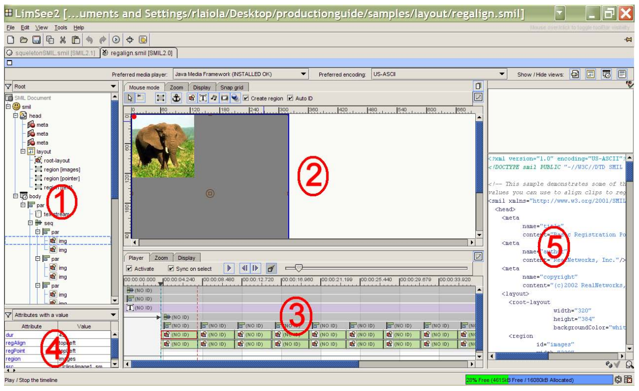
JAME Author [13] intends to easy the authoring task providing an abstraction (a view) composed of *pages*, similar to Web pages (*page-based services*). A page is composed of media objects and it is specified using an XML-based language called JAME PDL (*JAME Page Description Language*) [14]. A page view allows for adding pre-defined components to a page, for specifying the focus movement through the page's components via remote control key navigation, and for specifying links for navigation between pages. Another view allows for editing the properties of a selected component in a page. The tool also offers an emulator that allows authors to preview the results of their work. The authoring process is very similar to authoring HTML pages, which can ease its use by non-programmers. However, the set of functionalities provided is very limited. *JAME author* does not provide support for defining spatial and temporal relationship among media objects in a page, or for live editing an application, or any other Java computation support other than the pre-defined ones.

iTV Suite Author [9, 21] (former *Cardinal Studio*) is an intuitive authoring tool that has *acts* as its top-level abstraction. *Acts* are very similar to the *pages* of *JAME Author*. Indeed the tool paradigm and functionalities are very similar to the previous one. However, unlike *JAME Author*, *Cardinal Studio* provides support for defining spatial and temporal relationships among the media objects that compound an Act, although this task requires coding effort. Moreover, although there is no direct support for live editing, it is possible to specify stream events that can be used to trigger the execution of procedures to perform application changes in real time.

AltiComposer [3] uses a much more elaborated model consistent with film and TV industries, in which a *scene* has *planes*, and a *plane* contains *shots* and *actors*. The spatial and temporal synchronism of an object can be determined by specifying its *behavior* writing a script code in *AltiComposer Script Language*, a subset of ECMA-262 [4]. The support offered for live editing is like the one offered by *iTV Suite Author*.

Declarative domain-specific languages have been used to author hypermedia applications and among them are SVG [40], SMIL [8] and X3D [38], focusing on the Web domain, and MHEG and NCL, focusing on DTV domain in general. There are some authoring tools especially designed for these languages.

GRiNS [7] and *LimSee2* [29], authoring tools for SMIL, have integrated (synchronized) views (temporal, spatial, textual, etc.). Changes in a SMIL application made on a view are reflected on the others. In *GRiNS* the temporal view is the most important one and it is used to compose and manipulate the presentation of a SMIL document on a time axis. In *LimSee2*, the temporal and the spatial view together are powerful mechanisms to create applications. Figure 1 presents the Limsee2 graphic interface as an example.


LimSee2 [...uments and Settings/rlaiola/Desktop/productionguide/samples/layout/regalign.smil]
File Edit View Tools Help
regalign.smil [SMIL2.0]
Preferred media player: Java Media Framework (INSTALLED OK)
Preferred encoding: US-ASCII
Show / Hide views:
Mouse mode
Zoom
Display
Snap grid
Create region
Auto ID
Player
Activate
Sync on select
Attributes with a value
Attribute
Value
1
2
3
4
5
28% Free (4615kB Free / 16080kB Allocated)
Play / Stop the timeline

```
<?xml version="1.0" encoding="US-ASCII"?
<!DOCTYPE smil PUBLIC "-//W3C//DTD SMIL

<!-- This sample demonstrates some of th
values you can use to align clips to reg
<smil xmlns="http://www.w3.org/2001/SMIL
  <head>
    <meta
          name="title"
          content="Basic Registration Po
    <meta
          name="author"
          content="RealNetworks, Inc."/>
    <meta
          name="copyright"
          content="(c)2002 RealNetworks,
    <layout>
      <root-layout
                width="320"
                height="384"
                backgroundColor="whit
      <region
            id="images"
```

Figure 1. LimSee2 graphic interface.

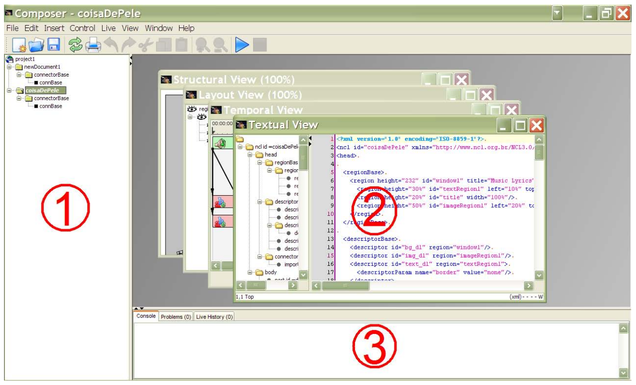
In Figure 1, region 1 presents the structural view, where a SMIL document is organized in a tree structure. Region 2 shows the spatial view, where media objects placement can be defined. The spatial view also allows for the pre-exhibition of an object (only text and images), in a specific moment in time selected in the temporal view. Region 3 is the temporal view, where media objects, represented by rectangles whose length is the media presentation duration, can be placed in a timeline. Region 4 is the property view, where attributes associated to a media object are displayed for modifications. Finally, region 5 represents the textual view. Unlike *GRiNS*, *Limsee2* does not have a SMIL player integrated to the tool. However, it recognizes native installed editors and allows users to select one of them for exhibition.

Unlike the previously mentioned tools for Java applications, both *GRiNS* and *LimSee2* favor the specification of temporal synchronization, without user interaction, among media objects. On the other hand, the specification of interactive actions is easier in those previous tools. In *GRiNS* and *LimSee2*, interactive relationships can be specified by using their textual views. In these views the SMIL code is shown and can be edited. However, their text editors are very simple, in comparison for example with *NCL Eclipse*, discussed afterward in this section. Finally, both *GRiNS* and *LimSee2* do not provide support for live editing. In *LimSee3* [12], authoring is based on archetypes, that is, on document models whose media content is the only thing an author should worry about.

Adobe Flash [2] is an authoring tool widely used for creating interactive content (Web pages, games and movies). By using facilities provided by the spatial view (called stage) and the temporal view (a timeline where media objects are placed), *Flash* provides a powerful spatiotemporal editor. However, synchronization relationships, including interactive ones, can only be defined through writing ActionScript [39] code.

NCL Eclipse [5] is a plug-in for the Eclipse IDE that provides a textual authoring tool for NCL programmers. Depending on the context where a code chunk is specified, code suggestions are presented, revealing the syntax and semantic aspects of NCL. The tool also validates the source code, providing error and warning indications. A hypertext facility allows for fast navigation between related NCL elements and attributes. Besides the textual view, NCL Eclipse has an outline graphic view that displays the NCL elements of an NCL document in a tree format to give authors an idea about the document structure and also allowing fast track on the document. In another graphic view, the initial displacement of an NCL media object can be displayed and graphically edited.

Composer I [17] offers high-level abstractions by means of graphic views, making easier the authoring process of NCL applications in conformance with the NCL Language Profile. *Composer I* allows many NCL documents to be edited simultaneously within a project. Figure 2 presents *Composer I*'s main window, as well as its textual view, temporal view, layout view, and structural view. In this figure, number 1 highlights the region where projects and their documents are managed. Number 2 shows the working area in which views are enabled. Number 3 presents an auxiliary area used for informing authors about the authoring process (e.g. error messages).



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="coisaDePele" xmlns="http://www.ncl.org.br/NCL3.0/
<head>

  <regionBase>
    <region height="232" id="window1" title="Music Lyrics"
      <region height="30%" id="textRegion1" left="10%" top
      <region height="20%" id="title" width="100%"/>
      <region height="50%" id="imageRegion1" left="20%" to
    </region>
  </regionBase>

  <descriptorBase>
    <descriptor id="bg_d1" region="window1"/>
    <descriptor id="img_d1" region="imageRegion1"/>
    <descriptor id="text_d1" region="textRegion1">
      <descriptorParam name="border" value="none"/>
```

Figure 2. Composer I graphic interface.

In *Composer I*, the structural view presents the NCL document structure. In NCL, a context element may contain media objects, switches and other contexts as its child elements, besides the spatiotemporal relationships among these internal elements. The structural view provides a fisheye-filtered view of nested contexts of an NCL application. The layout view provides the same functionalities already presented for *GRiNS*. The textual view provides a code editor better than those provided by *GRiNS* and *LimSee2*, but much more simple than that one provided by *NCL Eclipse*. The temporal view uses a model called Hypermedia Temporal Graphs (HTG) [11] to represent the temporal behavior of an application. HTG preserves relationships among events of an application presentation (including viewer interactions) and allows for event placements on a time axis, without losing the ability for expressing important structures present in DTV applications, like objects of indefinite duration, interactive events, and alternative contents (content adaptations). In the temporal view, relationships among events can be edited. *Composer I* allows authors to simulate the occurrence of unpredictable events in its temporal view. Also by using the temporal view, any time chain of events can be played from any starting point, through the NCL player integrated to *Composer I*. The resulting NCL document can be saved as an NCL file. In addition, any set of editing actions performed in a server side can be saved as an *NCLEditingCommand* that can be transmitted to change NCL documents in receiver (client) side. Thus, *Composer I* is the first mentioned tool that supports live editing.

It should be remarked that none of the previously mentioned work discusses non-functional requirements for the tools they propose, as discussed in the next section that supplements this related work discussion.

3. Hypermedia Authoring Tool Architecture

Non-functional requirements of a generic hypermedia authoring tool are discussed in Section 3.1, before presenting the proposed architecture in Section 3.2.

3.1 Non-functional Requirements

Several works discuss the desirable *functional* requirements of hypermedia authoring tools [25, 34, 18, 31, 33, 6]. In [35], a large group of users were evaluated regarding the NCL use. Several authoring tools were presented to them (not only those targeting NCL), among them a poorer version of *NCL Eclipse* and *Composer I*. It was not surprising that almost all expert programmers have reported that features regarding the textual authoring, which they consider much easier than using graphic tools, should be empowered. However, they also consider the use of some graphic views, in particular to define the positioning of media content. It was also not surprising that almost all naïve content producers pointed to the need of a tool that could abstract all the language syntax and semantics by means of graphic views. However, as they get more skill, they begin to use more and more the textual view.

As there are several different author profiles, from naïve to expert programmers and content producers, and since their needs change as they are more acquainted with the authoring language, an authoring tool should be adaptable to user needs. It should be **customizable** and **extensible**. However, almost all tools presented in Section 2 work with a fixed number of graphic views that together do not cover all the language expressiveness. The single exception is *Composer I* that allows third-party developers to create new views, which can be coupled to the tool through a predefined API. The API is offered as an abstract Java class that may be inherited by new views. However, there are many drawbacks in the *Composer I*'s approach.

Composer I encapsulates how its views interact among themselves, using a mediator module, as shown in Figure 3. When a specific view modifies an NCL document, its observer (obs1 in the figure) is notified. The observer then calls its associated compiler to reflect the changes in the internal data structure of *Composer I* (named JavaNCM) and also, as a consequence, in its NCL DOM tree. Each other mediator process is then notified (observer obs2 in the figure), which makes them to call its associated view compiler, updating all views. All this process is very heavy and does not scale. Even only with *Composer I*'s original views the delay caused by each mediator translation is unbearable when NCL documents are large. In low-end platform delays are unbearable even with medium size NCL documents.

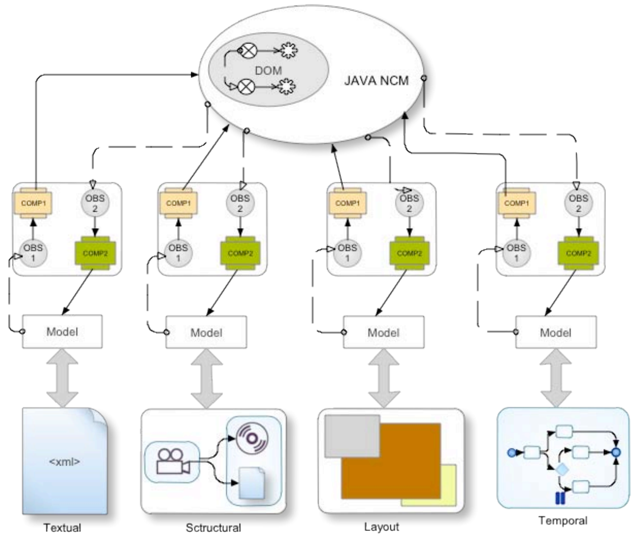


Figure 3. Synchronization of Composer I's views.

The mentioned problem comes from several reasons. First, Composer I was implemented in Java. Although this design decision brought the multiplatform advantage of Java, it also brought the inefficiency of an interpreted language. Second, Composer I uses two central models: the DOM tree and the JavaNCM. Both models have nearly the same information. The use of DOM is justified as a step to translate NCL specifications into the NCM model [33]; and to keep the NCL structure, since the translation from NCM back to NCL is not one-to-one. However, handling the DOM tree is inefficient with regards to both memory use and processing needed for all translations made by the mediators (see Figure 3). Still a third and main issue, authoring changes are not reflected on the several views incrementally. Each minor change made in a view causes the compilation of the entire document in all other views. In short, **efficiency** is a necessary requirement not only for minimizing delays presented to authors, but also for supporting the necessary **scalability** and **extensibility** that allow the use of several views for editing small to large size documents. Besides, efficiency is needed to allow for authoring in platform with scarce resources.

An authoring tool must also be adaptable regarding the production environment in which it is integrated. The tool should be customizable to work with different transmission systems. For example, in the Web, the resulting application is usually stored as a set of files; in a terrestrial DTV environment it is usually transmitted in a MPEG-2 TS [22]; IPTV services rely on RTP [32] and FLUTE streams [31]; etc. Ideally, the tool should also be customized with respect to its intended use and platform characteristics, as for example, to be used in a client side of a DTV system allowing for social TV applications. The use in different platforms also brings the **portability** requirement.

Summarizing, **performance** (efficiency in memory and CPU use), **customizability**, **extensibility**, **scalability**, and **portability** are the raised NFRs. Certainly, **reliability** and **maintainability** must be added to this list. The architecture should be **resilient** and **fault tolerant**, providing recovering mechanisms and preventing authors from losing their work.

3.2 Hypermedia Authoring Tool Architecture

Hypermedia application producers are usually inexperienced programmers. In addition, in some cases, as in Social TV applications [27, 15, 18], end-users can become producers or co-producers. Therefore, a comprehensive and lightweight graphic authoring tool is usually necessary. However, understanding the authoring language is essential for producing attractive applications. Declarative languages are valuable in this perspective. Although the proposed architecture can be used in authoring tools for imperative applications, its focus is on hypermedia applications defined using declarative languages.

The proposed architecture combines the smallest possible system functionalities in a microkernel, which serves as a host for future system extensions, coordinating the way these extensions collaborate. This architecture, indeed common in software systems that have to adapt themselves according to different requirements, meets the aspects and NFRs raised in the previous section.

In the architecture, microkernel **extensions** are made by means of plug-ins. As usual, plug-ins are software components that interact with host applications in order to extend them by adding features or specific resource types. By selecting plug-ins that fulfill their authoring needs, authors can build a **customized** instance of the tool that can also take into account the idiosyncrasies of the target environment.

The microkernel is responsible for controlling message exchanges between different plug-ins, maintaining the central data model that represents the hypermedia document under development, and notifying changes in this model to plug-ins interested in them.

It is worth to highlight the difference between plug-ins and views defined by usual authoring tools (see Section 2). Views allow for different approaches to manipulate a document, while plug-ins do not necessarily have a visible feedback. For instance, plug-ins related to transmission and validation tasks help the authoring process, but users do not modify documents by using them. However, a view certainly can be a plug-in. Moreover, a set of views can be grouped in a single plug-in.

Figure 4 illustrates the microkernel communication mechanism. The microkernel is the bridge between the central data model and the plug-ins. The moment an application author interacts with a particular plug-in (1), the plug-in notifies the microkernel about the changes occurred (2). The microkernel then validates the reported changes (3) and, if the validation does not return errors, it makes the necessary changes in the central data model (4). Afterward, the microkernel notifies the changes to the other plug-ins (5). Thus, plug-ins can incrementally make the necessary adjustments and report them to their users. If there is any validation error, the microkernel just notifies the source plug-in (4).

It is important to mention that the microkernel should use a single internal data model for the hypermedia document being authored. Handling changes using a single data model is very important both for **performance** and **scalability**, as discussed in Section 3.1.

There are two operation types targeting the central data model: query operations and modification operations. Query operations are performed by plug-ins accessing the central data model directly. On the other hand, modification operations are conducted through requests coming from plug-ins to the microkernel. This mechanism makes easier the access control to the central model. The microkernel is in charge of locking the document chunk under modification; performing the necessary changes and then unlocking it, ensuring the data model consistency.

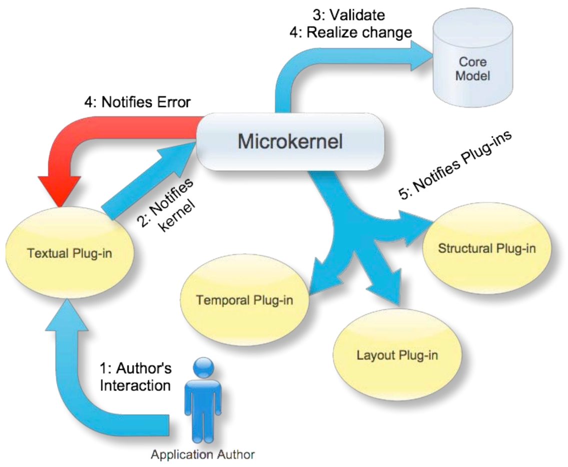


Figure 4. Communication between plug-ins and microkernel

The synchronism among views in the current tools (LimSee3 and Composer I, which has an architecture [12] similar to the proposed one) is usually performed on the whole data model, as discussed in Section 3.1. For each minimum change in a document, the entire views' internal models are synchronized with the central data model (a complete recompilation is necessary, since it is only notified that there is a new document representation and not what has been modified). This reduces the tool performance as the project (the document size) grows. As aforementioned, this approach is not scalable, besides being a critical point of failure.

One of the main advantages of the proposed architecture is the incremental synchronization of plug-ins. Changes in the central data model are notified to plug-ins as soon as they occur, through a well-defined API. In its turn, each plug-in can then incrementally update its graphic interface giving visual feedback to its user.

Another point deserves to be mentioned. As a plug-in may deal with only a subset of the target language's entities, it is notified of changes merely on these entities it is interested in. The microkernel provides this filtering mechanism. Additionally, the microkernel provides transaction control, with rollback support; plug-in control; project control; and model validation. The architecture modules related with these functions are illustrated in Figure 5.

Portability is a non-functional requirement fulfilled by the architecture that is built on top of the QT Framework [30]. QT supports several operating systems, including those for portable devices (Symbian, Maemo, Windows mobile). In addition, QT provides graphical user interface APIs, XML and multimedia (audio, video, images) handling APIs, and network access APIs.

Three main modules compound the architecture: *CoreModel*, *CoreControl* and *ComposerGUI*. The plug-ins are on top of them.

The *CoreModel* module contains the central data model, that is, the internal representation of application specifications (documents) under development. This data model is different from models used by application players (usually implemented as a plug-in), in which the application specification should remain consistent all the time. In contrast, the central data model is flexible and accepts temporary inconsistencies. Such inconsistencies occur when users are allowed to randomly edit parts of the document. Not allowing such inconsistencies brings premature commitment problems, as discussed in [16].

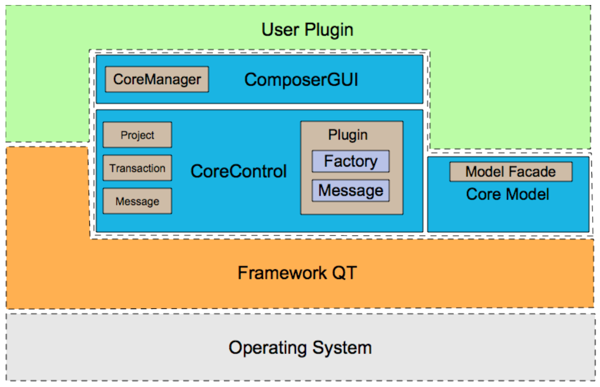


Figure 5. Modules of the architecture

The *CoreControl* is the microkernel itself. This module contains all essential microkernel's functions previously discussed. It also makes the bridge for message exchanges between plug-ins and the central data model. Four main modules make up the *CoreControl*: *Message*, *Transaction*, *Plug-in, and Project*.

The *Message* module is responsible for managing message exchanges between plug-ins and the microkernel. This module receives messages emitted by plug-ins, interprets them and delegates the required actions to the *ModelFacade*.

In order to enable the *CoreControl* module to act on the *CoreModel* module, a facade provides a simple API. *ModelFacade* is responsible for receiving requests to change the internal data model and for performing the necessary procedures to ensure that those changes are correctly carried out. When changes are successful, other plug-ins interested in those elements are notified.

Transaction module is responsible for managing transactions targeting the central data model. This module provides an API for a plug-in to open sessions and send block of messages to the microkernel. Such messages should be treated atomically. In addition, a plug-in may request the transaction rollback.

The *Plug-in* module is responsible for loading external plug-ins. This module also sets up the filtering function for each plug-in (in order to pass only relevant change notifications) and makes the connection between specific signals coming from the microkernel and the functions that handle these signals. The *Plug-in* module also defines interfaces that must be implemented when extending the microkernel. Section 3.2.1 details these interfaces.

The *Project* module handles operations targeting projects (open, close, delete and save), which contain files associated with a particular application, and manipulates documents that will be inserted or created within projects.

ComposerGui allows plug-ins to add graphic elements to their interfaces, which are incorporated into a QmdiArea - QT component (an area for displaying multiple graphical components). The graphic interface development for a plug-in is similar to developing a standalone application, when using the QT framework. In addition, *ComposerGUI* has a module, called

CoreManager, which is responsible for the communication between the plug-in's graphical interfaces and the microkernel internal modules.

The authoring tool reliability can be threatened since plug-ins are third part components. In order to improve the architecture resiliency, proactive **fault recovery** mechanisms, usually known as software rejuvenation [20], should be employed. Software rejuvenation consists on ending a component and starting it again from time to time, based on the assumption that its initial state is the most correct and consistent one in the component life cycle. In order to achieve a certain level of **reliability**, the plug-ins are treated as separated process, the microkernel is responsible to fork and spawn each plugin's process.

3.2.1 *API for microkernel extension*

Plug-ins incorporated to the architecture must interact with the microkernel through a single and orthogonal communication interface. This interface should be simple and straightforward so that a new feature implementer can keep its focus and abstract how plug-ins are integrated to the tool.

The QT framework provides means for extending its applications. Using QtPlugin, a new plug-in designer must develop a standalone QT application and use all QT development tools to assist its GUI implementation. In order to turn this standalone application into a plug-in, it suffices to specify the class that implements the interfaces defined by the microkernel: *IPluginFactory* and *IPluginMessage*.

The *IPluginFactory* interface must define how to create and destroy the plug-in's instances. Using this interface, the *CoreControl* allows multiple running instances of the same plug-in, each one tied to a document instance.

Listing 1 shows the *IPluginFactory* interface. The *createPluginInstance*() function must return an instance of the *IPluginMessage* object. The microkernel can then make the appropriate connections to exchange messages with the plug-in. The *releasePluginInstance*() function can be called to release the instance passed as a parameter, freeing memory when a document is no longer being edited by the plug-in.

```
class IPluginFactory {
  public:
   IPluginFactory();
   virtual ~IPluginFactory() = 0;
   virtual IPluginMessage*
          createPluginInstance()  = 0;
   virtual void
           releasePluginInstance
               (IPluginMessage *) = 0;
```

Listing 1 – Interface IPluginFactory

The *IPluginMessage* interface defines the messages and functions that support communication between the plug-in and the microkernel, as detailed in Listing 2. In addition, this interface defines a filtering function. The function, as previously mentioned, prevents plug-ins from receiving messages regarding elements that are out of the plug-in's scope of interest. The microkernel invokes the filtering (listenFilter) function passing an entity type as parameter. The plug-in must return true if the entity is in its scope of interest; otherwise, it must return false. The whole filtering mechanism is managed by the microkernel.

```
class IPluginMessage : public QObject{
 Q_OBJECT
 private:
  NclDocument *nclDoc;
 public:
  IPluginMessage();
  virtual bool listenFilter(EntityType ) = 0;
 public slots:
  virtual void onEntityAdded(Entity *) = 0;
  virtual void onEntityAddError(string) = 0;
  virtual void onEntityChanged(Entity *) = 0;
  virtual void onEntityChangeError(string) = 0;
  virtual void onEntityAboutToRemove(Entity *) = 0;
  virtual void onEntityRemoved(string) = 0;
  virtual void onEntityRemoveError(string) = 0;
 signals:
  void addEntity(EntityType entity, string
parentEntityId, map<string,string>& atts, bool
force);
  void editEntity(EntityType,Entity *,
map<string,string>& atts, bool force);
  void removeEntity( Entity *, bool force);
};
```

Listing 2 – Interface IPluginMessage

The microkernel sends messages to plug-ins when changes occur in language's elements (creation, modification or deletion). These messages must be processed by functions defined in the plug-ins. When an entity is created or modified in the central model, interested plug-ins receive a pointer referring to the entity instance.

Frequently, plug-ins require attribute values of specific elements. The central data model is designed so that query operations that do not modify elements can be performed directly over element instances received by the plug-ins. This mechanism avoids memory wasting, since it concentrates data in a single area, besides decreasing the number of message exchanges with the microkernel.

In contrast with query operations, when modifying an element, a plug-in must notify the microkernel. In messages sent by plug-ins notifying element modifications, the force variable informs the microkernel whether such changing operations should be performed or not, even if these operations bring the central data model to an inconsistent state. As previously discussed, keeping the central data model temporally inconsistent during authoring process is useful in order to avoid problems coming from premature commitment.

4. COMPOSER II

Hypermedia content production and distribution workflows have diverse environments and actors with different profiles and needs. From content producers, network and storage operators, to end-users, different requirements for content creation, edition and interaction arise. Each involved actor has a particular view of the whole process, which usually results in tools focused on particular actors aiming at meeting their specific needs.

Considering NCL use in the design of applications for interactive DTV, three main user groups are the targets: i) software programmers, who usually design complex applications (including those resident in DTV receivers), taking profit of all NCL features, including its Lua scripting language; ii) content producers, whose focus is on the development of visually rich applications, usually broadcasted to viewers; and iii) the viewers themselves, who usually build simple applications, based on graphical facilities and language simplifications offered by authoring tools embedded into their receivers. In order to support the needs of these different user groups the new version of Composer is based on the concept of multiple views, following the architecture presented in the previous section. Each view highlights a document particularity and may be more appropriate to a specific user profile, allowing for document manipulation without any NCL expertise.

Several plug-ins have been developed by different organizations for the new version of Composer. Among plug-ins for NCL application authoring there are: Textual View, Lua View, Structural View, Layout View, Temporal View, Spatial View, NCL Property View and Comment Editor. The textual plug-in integrates all features supported by NCL Eclipse [5], presented in Section 2. The Lua plug-in offers a textual editor for developing Lua scripts. The structural plug-in allows for structuring an NCL document. The layout plug-in allows authors to graphically specify the initial spatial positioning of NCL media objects (application's media contents). The temporal plug-in allows for working with the metaphor of program scenes. The spatial plug-in is similar to the layout plug-in but shows the spatial positioning of media objects in a given time chosen from the Temporal View. The property plug-in allows for changing media objects properties, like how they must be presented, etc.

There are also plug-ins for authoring NCL document families, which are specified using TAL [36] (Template Authoring Language), a domain specific language that provides a broad level of reuse. Based on families specified using TAL, a set of plug-ins named SAGGA (Support for Automatic Generation of Ginga Applications) is under development to allow for quick development of NCL applications.

There are still plug-ins under development for storage and transmission of NCL applications into specific transport systems. One of them allows for generating DSM-CC object carousels and their encapsulation into MPEG-2 transport streams. Another one allows for generating NCL stream events to transport NCL live editing commands [1].

In addition, Composer is being integrated to DTV broadcaster play-outs. This integration will make Composer a comprehensive authoring environment for DTV, encompassing from the design phase, to the encapsulation and transmission of applications.

5. CONCLUSIONS

The success of interactive DTV systems depends on if they have interesting and attractive applications. These applications appear when a massive number of producers are in scene. In order to allow the arising of such creative producers and not being limited to programming experts it urges to develop efficient authoring tools.

An efficient authoring tool must not only provide support to functional requirements but also non-functional requirements in order to help authors in their development tasks. Most authoring tools pay attention only to their functional aspects and fails to provide efficient, resilient and tailored support to authors.

This paper discussed several non-functional requirements coming from the analysis of several hypermedia authoring tools, and proposes an architecture to cover such raised aspects. The proposed architecture combines the smallest possible system functionalities in a microkernel, which serves as a host for future system extensions that are made by using plug-ins.

Besides all the architecture's advantages discussed throughout the paper, an additional one comes from the microkernel simplicity, robustness and easiness to be embedded into a device with scarce resources, such as a TV set-top box. The microkernel integrated into an STB opens a range of possibilities aiming at plug-ins for authoring on viewers' side, including those focusing on Social TV applications.

Composer II is used as a proof of concept of the proposed architecture. Composer II's microkernel and some of its main plug-ins are open source codes to be available under MIT license at www.softwarepublico.gov.br. A Web repository for developing and publishing plug-ins, commercial or not, is under development and will be integrated with Composer II, so that it can be possible to dynamically create distributions targeting specific users' needs and/or execution environments.

As a near future work we are planning to begin studies focusing on creating facilities for collaborative authoring of hypermedia applications.

6. ACKNOWLEDGMENTS

The authors would like to thank the TeleMídia Lab team who provided a thoughtful discussion of this work, in particular C. S. Soares Neto, R. G. Azevedo and E. Araújo. The authors also thank CNPq, CAPES, MCT and CETIC/RNP for their support.

7. REFERENCES

[1] Associação Brasileira de Normas Técnicas, ABNT NBR 15606-2. Data Codification and Transmission Specifications for Digital Broadcasting, Part 2 – GINGA-NCL: XML Application Language for Application Coding. Brazil. Nov., 2007. http://www.abnt.org.br/imagens/Normalizacao_TV_Digital/ABNT NBR15606-2_2007Ing_2008.pdf

[2] Adobe Systems. Adobe Flash CS3 Professional. In http://www.adobe.com/products/flash/.

[3] Alticast Inc. AltiComposer 2.0. USA. Available in http://www.alticast.com. Accessed in July 26, 2010.

[4] Alticast Inc. Script Guide - AltiComposer 2.0. USA. In http://www.alticast.com. Accessed in July 26, 2010.

[5] Azevedo, R. G. ; Soares Neto, C. S.; Teixeira, M. A. NCL Eclipse: Ambiente Integrado para o Desenvolvimento de Aplicações para TV Digital Interativa em NCL. *Simp. Bras. de Redes de Computadores*. 2009. Available in Portuguese.

[6] Azevedo, R.G.A.; Lima, B.S.; Soares Neto, C.S.; Teixeira, M.M. Uma abordagem para autoria textual de documentos hipermídia baseada no uso de visualização programática e navegação hipertextual. *XV WebMedia*. Fortaleza - Brazil. October, 2009. Available in Portuguese.

[7] Bulterman, D. C. A.; Hardman, L.; Jansen, J.; Mullender, K. S.; and Rutledge, L. GRiNS: A GRaphical INterface for creating and playing SMIL documents**.** *WWW7 Conference, Computer Networks and ISDN Systems*, v. 30, p. 519-529, Brisbane, Australia. April, 1998.

[8] Bulterman, Dick C.A., Rutledge, L. W. SMIL 3.0 - Flexible Multimedia for Web, Mobile Devices and Daisy. *Talking Books*. 2nd ed. Springer, 2009. ISBN: 978-3-540-78546-0.

[9] Cardinal Information Systems Ltd, Cardinal Studio 4.0. Finland. Available in http://www.cardinal.fi. Accessed in July 26, 2010.

[10] Chung, L., do Prado Leite, J. On Non-Functional Requirements in Software Engineering. 2009. Springer.

[11] Costa R.M.R, Moreno M.F., Soares L.F.G. Intermedia Synchronization Management in DTV Systems. *Proc. of ACM Symposium on Document Engineering* (Sao Paulo, Brazil, 2008), pp. 289-297. ISBN: 978-1-60558-081-4.

[12] Deltour, R. and Roisin, C. The limsee3 multimedia authoring model. *Proc. of the 2006 ACM Symposium on Document Engineering* (Amsterdam, The Netherlands, October, 2006).

[13] Fraunhofer Institute for Media Communication IMK. JAME Author 1.0. Schloss Birlinghoven – Germany. Available in http://www.iais.fraunhofer.de/jame.html. Accessed in July 26, 2010.

[14] Fraunhofer Institute for Media Communication IMK. User Manual - JAME Author 1.0. Schloss Birlinghoven – Germany. In http://www.iais.fraunhofer.de/jame.html. Accessed in July, 26, 2010.

[15] Geerts, D.; De Grooff, D. Supporting the Social Uses of Television: Sociability Heuristics for Social TV. *Proceedings of the 27th International Conference on Human Factors in Computing Systems* (CHI '09), pp. 595–604. ACM, Boston, Mass., USA. April, 2009.

[16] Green T. R. G., Petre M. Usability Analysis of Visual Programming Environments: A `Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing*. June, 1996.

[17] Guimarães, R.L.; Costa, R.R.; Soares, L.F.G. Composer: Authoring Tool for iTV Programs. *Proc. of 6th European Interactive TV Conference - EuroITV 2008.* Salzburg, Austria. July, 2008; pp.61-71. ISBN: 978-3-540-69477-9.

[18] Harboe, G.; Massey, N.; Metcalf, C.; Wheatley, D.; Romano, G. The Uses of Social Television. *Computers in Entertainment*, vol. 6, no. 1, pp. 1–15, 2008.

[19] Hardman, L., van Rossum, G., and Bulterman, D. C. Structured multimedia authoring. *Proc. of the First ACM international Conference on Multimedia*. August, 1993.

[20] Huang, Y. et al. Software rejuvenation: Analysis, module and applications. *International Symposium on Fault-Tolerant Computing*. Huang. June 1995; pages 381–390.

[21] Icareus Technology. Icareus iTV Suite Author. Avaiable in http://www.icareus.com/web/guest/technologies/itvsuite/author. Accessed in July 26, 2010.

[22] ISO/IEC 13818-1, "Information technology — Generic coding of moving pictures and associated audio information: Systems", 1996, ISO/IEC.

[23] ISO/IEC 13818-6, "Information technology — Generic coding of moving pictures and associated audio information: Digital Storage Media Command & Control", 1996.

[24] ITU-R Recommendation BT-1699. Harmonization of declarative content format for interactive TV applications. Geneva, 2009.

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

[26] Junehwa Song, Michelle Y. Kim, G. Ramalingam, Raymond Miller, Byoung-Kee Yi. Interactive Authoring of Multimedia Documents. *IEEE Symp. on Visual Languages*. 1996, p. 276.

[27] Malan, R., and Bredemeyer, D., "Defining Non-Functional Requirements", Available in http://www.bredemeyer.com/papers.htm

[28] Mantzari, E.; Lekakos, G.; Vrechopoulos, A. Social TV: Introducing Virtual Socialization in the TV Experience. *Proc. of the 1st International Conference on Designing Interactive User Experiences for TV and Video* (UXTV '08), vol. 291, pp. 81–84. Calif, USA. October 2008.

[29] LimSee2 – SMIL Editor. In http://limsee2.gforge.inria.fr. Also in LimSee2 - The cross-platform SMIL2.0 authoring tool, INRIA - Grenoble, France: http://limsee2.gforge.inria.fr/?goto=Download. Accessed in July 26, 2010.

[30] Nokia Trolltech. Qt - A cross-platform application and UI framework. http://qt.nokia.com/products/

[31] RFC 3926, FLUTE: File Delivery over Unidirectional Transport.

[32] RFC 3550, RTP: A Transport Protocol for Real-Time Applications.

[33] Soares L.F.G., Rodrigues R.F. Nested Context Model 3.0 Part 1 – NCM Core. *Technical Report. Informatics Department of PUC-Rio*, MCC 18/05. Rio de Janeiro. May, 2005. ISSN 0103-9741.

[34] Soares L.F.G., Rodrigues R.F. Nested Context Language 3.0 Part 8 – NCL Digital TV Profiles. *Technical Report. Informatics Department of PUC-Rio*, MCC 35/06. Rio de Janeiro. October, 2006. ISSN 0103-9741 http://www.ncl.org.br/documentos/NCL3.0-DTV.pdf.

[35] Soares Neto, C.S.; Soares, L.F.G.; de Souza, C.S. The Nested Context Language Reuse Features. *Journal of the Brazilian Computer Society.* DOI 10.1007/s13173-010-0017-z.

[36] Soares Neto, C.S. Autoria de Documentos Hipermídia Orientada a Templates. *Doctoral Thesis*, *Informatics Department of PUC-Rio*. September, 2010. Available in Portuguese.

[37] Vazirgiannis, M; Kostaas, I; Sellis, T. Specifying and Authoring Multimedia Scenarios. *IEEE Multimedia Magazine*. Vol6, no. 3. July, 1999

[38] WEB3D Consortium. Extensible 3D (X3D), ISO/IEC 19776-1.2:2009. In: http://www.web3d.org/x3d/specifications. Accessed in July 26, 2010.

[39] Williams, M., ActionScript Coding Standards, Macromedia White Paper. May, 2002. Available in http://www.adobe.com/devnet/flash/whitepapers/actionscript_standards.pdf. Accessed in July 26, 2010.

[40] W3C World-Wide Web Consortium. *Scalable Vector Graphics – SVG 1.1 Specification*, W3C Recommendation. 2003. http://www/w3/org/TR/SVG11.