EBU
OPERATING EUROVISION AND EURORADIO

TECHNICAL REVIEW

Q2 2013

QEBU: AN ADVANCED GRAPHICAL EDITOR FOR THE EBUCORE METADATA SET

MAURO LEVRA, PAOLO PASINI, DENIS PATTI,
GIOVANNI PESSIVA, STEFANO RICOSSA
Polytechnic University of Turin

ABSTRACT

The creation and management of metadata documents can be quite a difficult task to accomplish manually. To address this issue in the context of the EBUCore v1.3 metadata set, we propose a GUI-based metadata editor, called QEbu, developed during the Multimedia Archival Techniques course, held at the Polytechnic University of Turin in collaboration with RAI.

QEbu was developed with the aim of providing a user-friendly graphical editor to create and manage XML documents, relieving the user from the burden of worrying about the structure of the data and letting him focus on the actual content. The editor is usable by both experienced users and novices in the field.

QEbu has been developed in C++ using the cross-platform and open source library Qt 4.8; this framework was chosen in order to exploit its natural features for developing interface-centred applications, running locally on the machine of the user under a desktop environment.

INTRODUCTION

In multimedia content analysis the word *metadata* refers to any piece of information used to describe a specific content entity. This kind of information requires the definition of a proper structure, in order to manage, exchange and exploit the available knowledge in a profitable way. A common choice is to use XML documents for this purpose, and this is exactly what is achieved with the *EBUCore Metadata Set* [1].

Besides the definition of the standard itself, it is equally important to provide convenient tools which are able to create and manage such data, since it is difficult to do this by hand, and likely to be error prone as well.

In order to address this last issue, QEbu has been developed and is presented in this article. QEbu is a GUI-based metadata editor built upon the EBUCore v1.3 specifications. The proposed tool was developed as an academic term project for the *Multimedia Archival Techniques* course held at Politecnico di Torino in collaboration with RAI, within the study plan of the Master of Science in Computer Engineering.

OBJECTIVES

QEbu has been developed with the aim of providing a user-friendly graphical editor to create and manage XML documents according to the EBUCore specifications.

The main idea is to relieve the end user from the burden of worrying about the structure of the data in order to let him focus on the content. The tool's function is to enforce all the required constraints and guide the user seamlessly toward the creation of a well formed, and both structurally and semantically coherent, document.

The tool is aimed at both experienced users and novices in this field. During the design phase, the fundamental objective has been to provide something that is easy to use without intense training, and which does not require any deep knowledge of the subject, thus gaining immediateness, without losing effectiveness. These aspects will be made clearer during the overview of the GUI in the following sections.

QEBU: OVERVIEW

In this section of the document, the main aspects of coding techniques and implementation details will be covered, analyzing both the conceptual model and the user interface.

In the following sections all elements worthy of attention are treated individually and, for some of them, a general code snippet is given, in the form of a template. The data model will be presented first, followed by parsing/serializing aspects and then finishing with details about the interface.

THE DATA MODEL

The data model finds its roots directly in the XML schema reference [2]; an almost one-to-one mapping for all data types defined in this document, with just few exceptions, can be found in the final outcome of the model itself.

Every named data type, either simple or complex, has its own class representing every single detail of the type itself; while for unnamed types the decision whether to define a specific class or not has been made considering the situation and context, as they occur in time sequence.

At the same time, by exploiting objects' composition and inheritance, we have tried to keep the number of defined classes to a reasonable minimum.

As a rule of thumb, the solution that has resulted in a less redundant code with improved re-usability has been chosen; but there is no absolutely right or wrong approach in this instance.

The programming style adopted, in terms of naming conventions, spacing and such, is the one assumed as the standard for Qt GUI framework development, which is already well documented by the official online community [3].

An explicative example of a general type class is the following:

Example 1: pattern followed in custom type class declaration

```
class ClassName
{
public:
    ClassName();  // Constructor
    ~ClassName(); // Destructor
    // Getter and Setter for a value member
    QString aString() const;
    void setAString(const QString &value);
    // Getter and Setter for a reference member
    CustomClass aClass();
    void setAClass(CustomClass *value);
    // Accessor/Mutator method to a live list
    QList<CustomClass*> &aList();
    // Override of the toString method for our custom class
    QString toString() const;
private:
    // Examples of private members
    QString m_aString; // a string variable
    CustomClass *m_aClass; // reference to a CustomClass object
    QList<CustomClass*> m_aList; // list of references to CustomClass objects
};
```

As seen in the code snippet above, basically all the classes defined to represent a type are composed of a public part, which contains the basic methods to create and destroy the class itself and to access its private members.

Depending on the type of inner members, an appropriate way of accessing them is defined, but in general objects use pointer logic: predefined types, such as strings, are handled by value; while lists and collections are treated like live elements, so a reference to the actual collection is returned by its accessors.

Given the wide range of formatting possibilities defined within the XML schema, a utility class called *TypeConverter* has been implemented, along with the model, to provide basic functionalities of type conversion between dates/times and strings, in support of parsing/serializing steps and for visualization purposes.

In this instance, the details of the actual implementation of the methods are omitted, since they are extremely trivial and would not add any relevant information to the present text.

In the case of highly correlated types or nested declarations, in order to mirror the relationships found in the reference schema, several classes may have been defined within the same file, but never with a nested approach: every single definition happens at the same level, without using inner declarations.

Before describing other aspects of the application, it is worth taking into account how internal references among objects are managed. There are some elements within the schema, such as *RightsType* and *PublicationType*, that use XML ID and ID-ref in order to enforce this kind of relationship; it has proven necessary to define a proper way to manage such a situation. Instead of a mere string-based reference, we decided to introduce an actual pointer among the objects, which provides a stricter constraint, but in turn requires more careful handling. Since the user is free to destroy referenced elements at any given time, an additional map of references has been

introduced in the model as well. This map contains entries indexed by reference ID, to which correspond the actual object pointer and a list of listener objects. Those objects are the ones that are required to be updated in the case of deletion of the referenced entity, and hence the name 'listener', in order to avoid breaking the implied relationships between them.

PARSER AND SERIALIZER

Once the model that provides a custom representation in memory of the schema was defined, two additional elements, a parser and a serializer, were developed in order to allow the transition between XML documents and custom in-memory representation.

These modules use Qt framework DOM functionalities to read and write XML files and expose simple interfaces to the GUI application code.

Although DOM APIs are notoriously slow and memory consuming, for the purposes of these modules they represented a valid choice, granting flexibility and easier code development.

Given the complexity of the underlying schema, and the need to have access for writing APIs, SAX was not chosen in this case in order to avoid having two different approaches for parsing and serializing.

It is also to be noted that the files with which this application works are not particularly big, so memory and performance related aspects should not really be an issue.

PARSER

Before being parsed into memory, every file is validated against the EBUCore schema by invoking `xmllint`, a command line tool provided by the free cross-platform C library libxml2.

The decision to use an external validator was made after an analysis of XML processing libraries provided by Qt, which assessed these as inadequate for the requirements of QEbu.

The `xmllint` executable should be reachable by the program at run-time. In a Windows environment this means it can be placed either in the QEbu directory or in a directory referenced by PATH.

The validator output is parsed, retrieving information about possible errors; such details are then forwarded to the user.

All validation functionalities are wrapped in the *Validator* helper class.

If the application cannot access the validator, it will show a warning message asking the user if he wants to continue with the parsing, even if the file could be invalid or not well-formed. If a non-validated file is loaded, the application outcome is not predictable; even though every item is checked for validity, which inherently enables crashes to be avoided, it is impossible to define the actual result of the parsing itself. In this case, it is up to the user to decide whether the process produced a sensible enough outcome to work upon.

This said, the basic approach to parsing a file into memory is exemplified by the following snippet of code, which illustrates the way it is performed given an arbitrary node element of the DOM tree representation.

Example 2: a basic snippet of code illustrating parser behaviour

```
ClassType *EbuParser::parseClassType(const QDomElement &element)
{
    // Sanity check for node element validity
    if (element.isNull()) {
        m_errorMsg = "ClassType element is null";
        return 0;
    }
```

```
    // Create custom memory representation for a given ClassType
    ClassType *obj = new ClassType();

    // Get attribute(s).
    T attributeName = element.attribute("attributeName");
    // Sanity check for attribute validity, according to the specific type
    if (!attributeName.isValid())
        obj->setAttributeName(attributeName);

    // ...more attributes.

    // Get element(s), a list for example...
    QDomNodeList nodeList = element.elementsByTagName(tagName);
    for (int i=0; i < nodeList.size(); ++i) {
        QDomElement el = nodeList.item(i).toElement();

        // In case of nested elements with a given name,
        // which are not direct  children of the current node
        if (el.parentNode() != element)
            continue;

        // Recursively parse the child element, like we just did with its parent
        T *child = parseChildType(el);
        // In case the returned child is not valid (i.e. null pointer)
        if (!child) {
            // Destroy the parent as well and return failure
            delete obj;
            return 0;
        }
        // In case of success append the child in the proper structure
        obj->tagName().append(child);
    }
        // ...more elements.
        return obj;
}
```

Parsing is nothing more than a full visit to the document tree, sequentially descending through every branch until a leaf is reached. At that point, the terminal element encountered is mapped into our custom model and appended to the underlying memory representation. This process iterates until nothing else is left unmanaged.

It may seem unnecessary to have a duplicate memory representation, given the fact that DOM already builds its own, but this makes for easier management in the following steps. When it is necessary to have the model interact with the user interface, it is much more convenient to have specific types and classes, for which it is possible to define customary constraints, sanity checks and provide proper accessor methods, rather than use generic nodes or elements in memory.

SERIALIZER

The serializer is the dual counterpart of the parser just introduced, and follows the very same principle. Once again it is just a full visit to our custom objects graph, to write on file what is held in memory.

In this case the word graph is used, instead of tree, because it is possible to find, in some situations, references among objects not linked with parent/child relations. In the schema it is possible to find elements of IDref type, which are handled like actual pointers in the proposed model. This results in slightly more complex management processes for the data, but simplifies the verifications of reference constraints and the general coherence of the structure, or at least makes

non well-defined references pretty evident, since non-handled exceptions would result in segmentation issues.

The snippet below provides a quick example of the pattern followed for serializing data.

Example 3: a basic snippet of code illustrating serializer behaviour

```
QDomElement EbuSerializer::serializeType(classType *obj)
{
    // Create an empty unnamed element
    QDomElement l = m_doc.createElement(" ");
    // Serialize attribute(s), performing sanity check prior to write data
    // The check is entirely dependant upon the kind of attributes
    if(obj->attribute1().isValid())
        l.setAttribute("attribute1", obj->attribute1());
    if(obj->attribute2().isValid())
        l.setAttribute("attribute2", obj->attribute2());

        // ...more attributes.
    if(obj->element1() != NULL) {
        // This is a child element node
        QDomElement e = serializeType1(obj->element1());
        e.setTagName("anotherInnerElementName");
        l.appendChild(e);
    }
        // ...more elements.
        return l;
}
```

Before concluding this section, it should be noted that parsing functionalities are employed in another aspect of the application as well, directly relating to auto-completion using dictionary files; but these will be illustrated specifically in their proper context, along with the description of the user interface in the following section.

GUI AND FUNCTIONALITIES

The EBUCore standard [1] comprises a great quantity of specific data types and so, for the sake of clarity and to avoid congesting the user interface, for each of these types a dedicated form has been designed and realized; similarly to the model a specific class has been implemented.

The form designed for the root element of the schema provides the only entry point to the underlying structure of custom views, and the user is able to navigate through all the items and screens in a consistent and, hopefully, sensible way.

As an example, a preview screenshot of one of the QEbu forms is presented in Figure 1. Further screenshots will be introduced in subsequent figures to illustrate additional details.

It has been chosen to design the interface so that only a single aspect, i.e. an element, of the XML document is to be considered at a time. The metadata standard is rich in content and covers a wide range of possible applications, so an improper management of these aspects could easily lead to a messy and overcrowded interface. To avoid this possibility, for every element of the schema a dedicated form has been developed, tailored in a way to maximize the coherence of its content, while still maintaining an attractive look and feel.

As a rule of thumb, the principal criteria followed while designing each view, can be summarized as follows:

- Whenever possible, attributes are managed via edit fields;
- For range-restricted values, pickers are employed, together with additional checkboxes to handle options;
- Child elements come with their own form, with a few exceptions (i.e. groups);
- If possible, attributes/elements are grouped together in a meaningful way;
- Form parts are reused where possible to give consistency of composition.

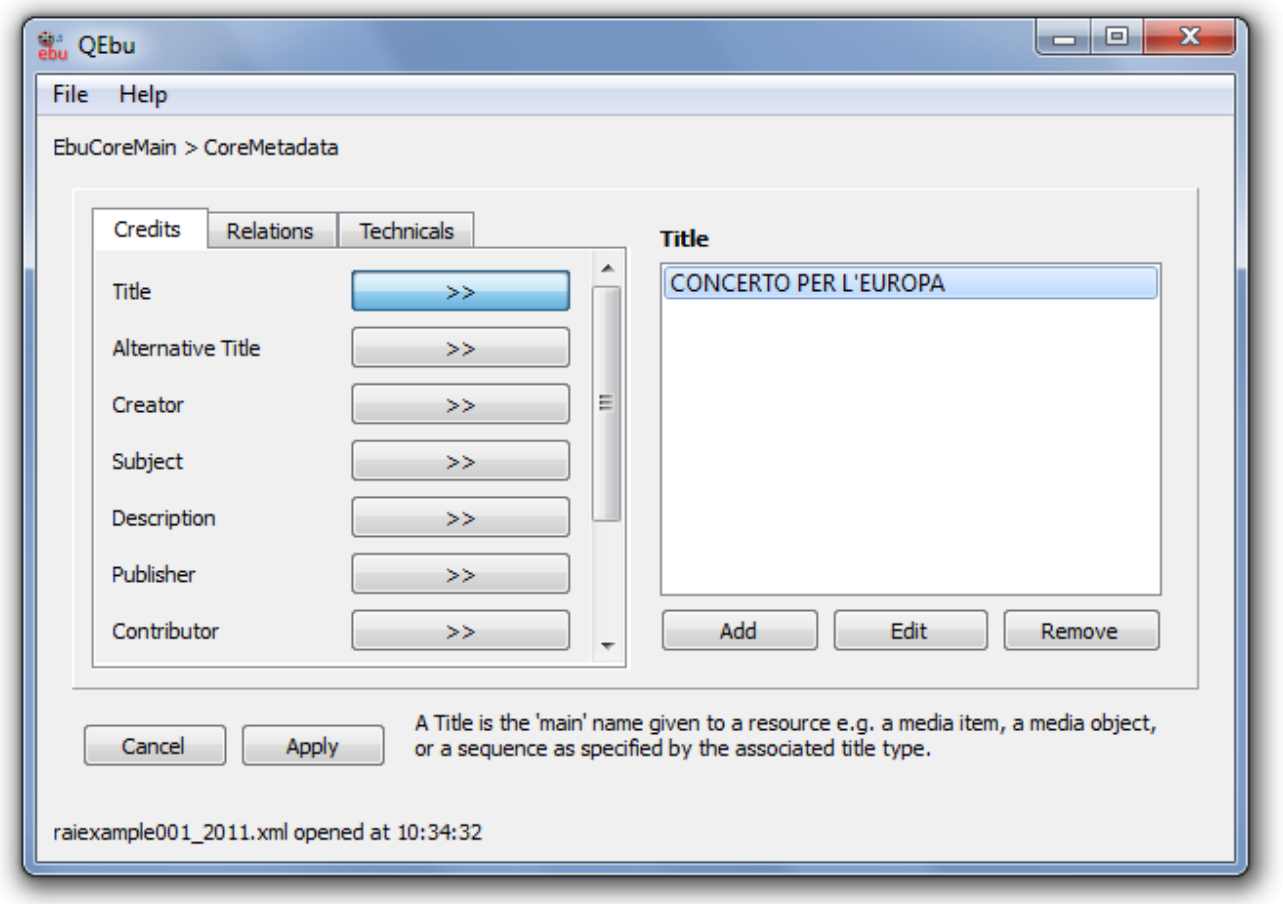

Figure 1: One of the forms of QEbu

Of course, given the variability of types and content, the above example might not be applicable to every item within the model, but it should be general enough to provide the generic look and feel that most forms share.

On the other hand, this approach was required also to manage effectively the transaction between the forms, with the aim to completely avoid popup dialogs, and so maintain a single evolving window without detached panels scattered all over the screen. Following the same idea that is used for mobile apps, we decided to implement a custom stack of forms, or widgets, to model the transactions between elements, in which only the topmost one is visible and interactive, while the others stay alive but frozen beneath the surface, waiting to reach the top again. An example of this approach is represented in Figure 2.

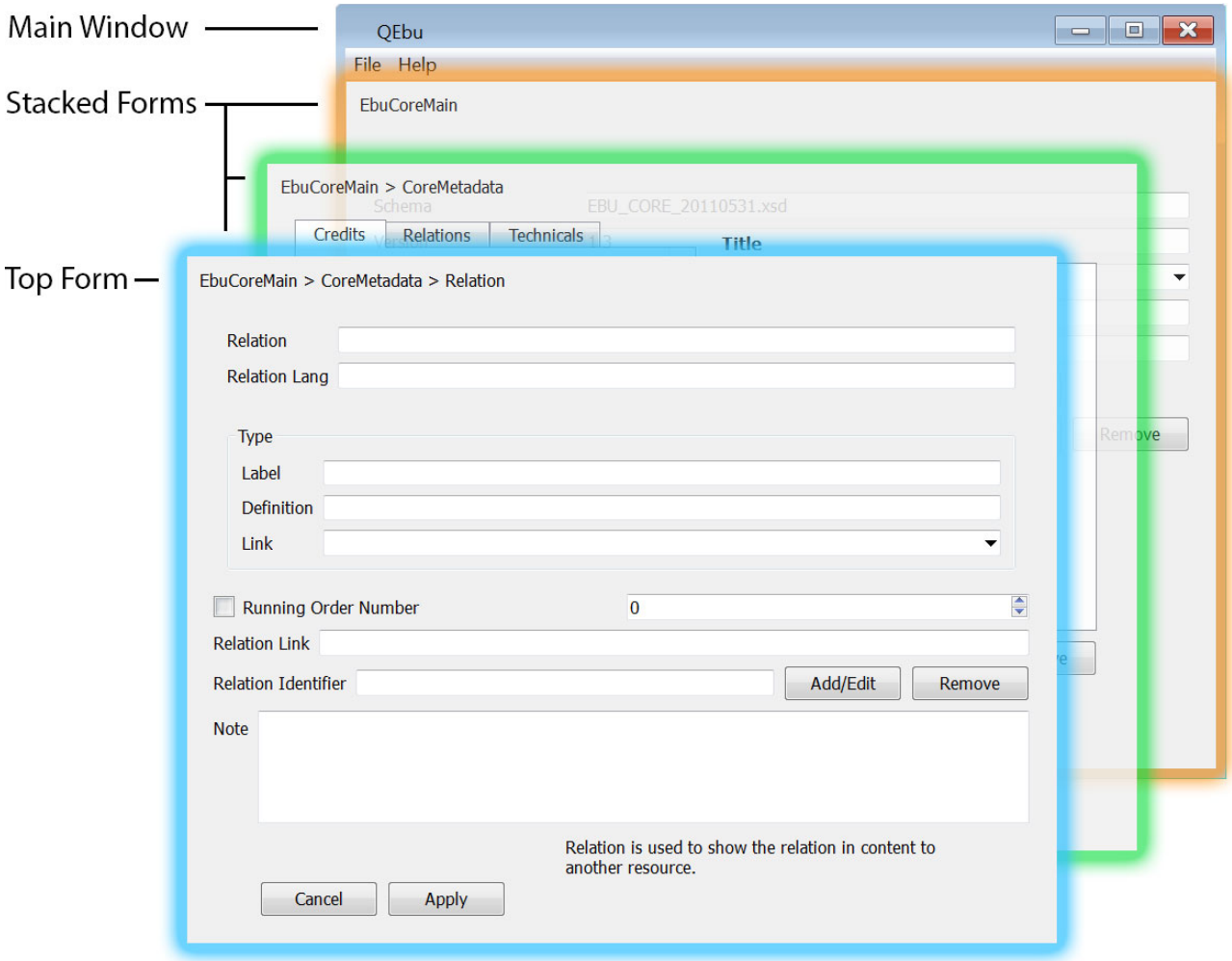

Figure 2: The structure of the stackable widget solution.

In order to keep navigation among forms consistent, we decided to keep the transaction model as simple as possible, as exemplified by the life cycle and flow of work representations in Figures 3 and 4.

The final user is allowed to navigate the data structure only in a strictly linear fashion, being forbidden to jump across the branches of the underlying tree representing the document being created of modified.

The logic behind this behaviour is defined within a custom widget, namely 'StackableWidget', from which all the forms of the application inherit, and that handles common events that may happen during the lifetime of a specific form, such as its creation, destruction or change of visibility.

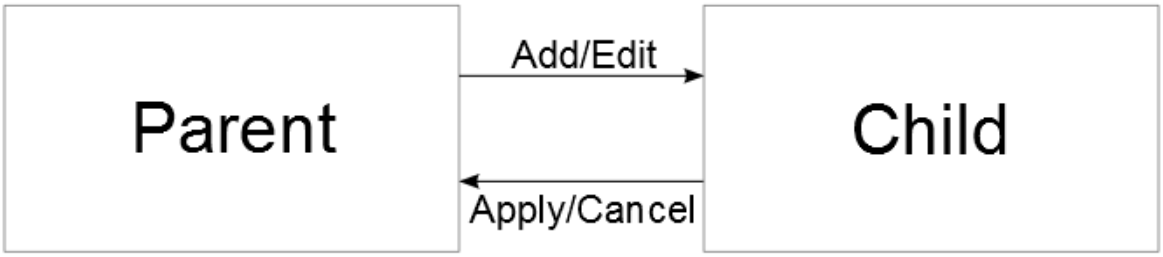

Figure 3: Simplified form life-cycle.

When the user chooses to Add/Edit a child element, a new form is created and becomes visible, while the parent is temporarily hidden. Then, using the Apply or Cancel button, it is possible to restore a previously frozen panel, causing the destruction of the current topmost form and the contextual release of its resources.

As stated previously, when possible, the single forms have been designed to employ the widgets made available by the Qt framework directly. However, in some specific cases, custom solutions needed to be designed, due to the limitations of the available widgets; for example, in the case of long integer values for which Qt's widgets offer no support, or the peculiar characteristics of some types or attributes. Specifically, some types with limited content, such as the ElementType, were embedded in fact directly in the parent object, in order to avoid the proliferation of low content forms.

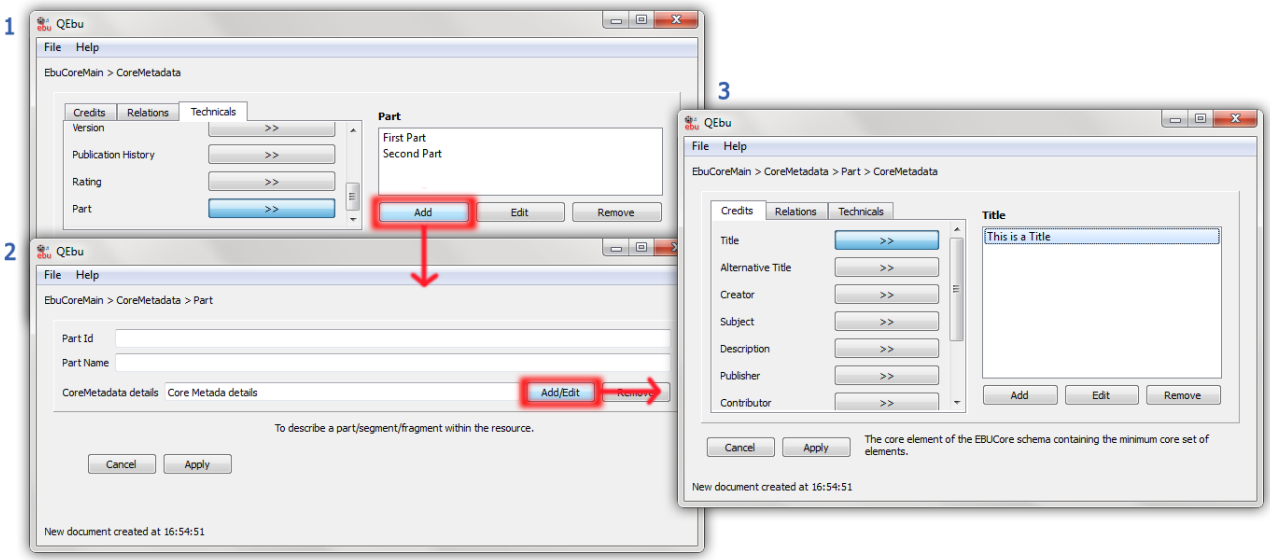

Figure 4: An example of the flow of work with the evolution of the user interface.

As shown in Figure 4, QEbu permits the handling of even complex situations, such as the management of parts, with ease. The specification allows the core element to include one, or more, instances of the same type as the core itself; but this is still straightforwardly manageable following the linear traversal mode implemented within the tool.

In addition to this, special attention has been paid to the management of the numerous sub-elements with unlimited cardinality [0..*] (i.e. lists of unrestricted variable length). The coexistence in a given element of many of these unbounded sub-elements, often along with other restricted ones, poses some problems in the design of a clear and user-friendly interface.

Looking at this in more detail, we may identify two critical aspects related to lists. Firstly, they can quickly lead to congestion in the interface, due to the amount of information that each list element can carry and that must therefore be represented. Secondly, without specific care, their coexistence with [0..1] elements could cause minor incoherencies in the interface, which in turn may prove confusing to the user.

A specific widget has been designed to deal with these problems. The surfeit of information is addressed by enabling only one list at the time for editing, avoiding the need to represent in the same view all the details carried by all the lists. The coherency between unlimited lists and single elements is then maintained by the use of the same widget in both cases, disabling the possibility of adding new elements after the first insertion, if necessary.

Given all of the aspects introduced here, and to provide further guidance to the final user while traversing the structure of a document, several devices have been embedded directly within the program interface. A sort of 'breadcrumb' bar is available at the top of each form, tracking the current position within the schema, to provide direct feedback about one's depth within the schema. Additionally, every form contains its own piece of documentation, to provide further indication on how to fill in each field properly. An example is provided in Figure 5.

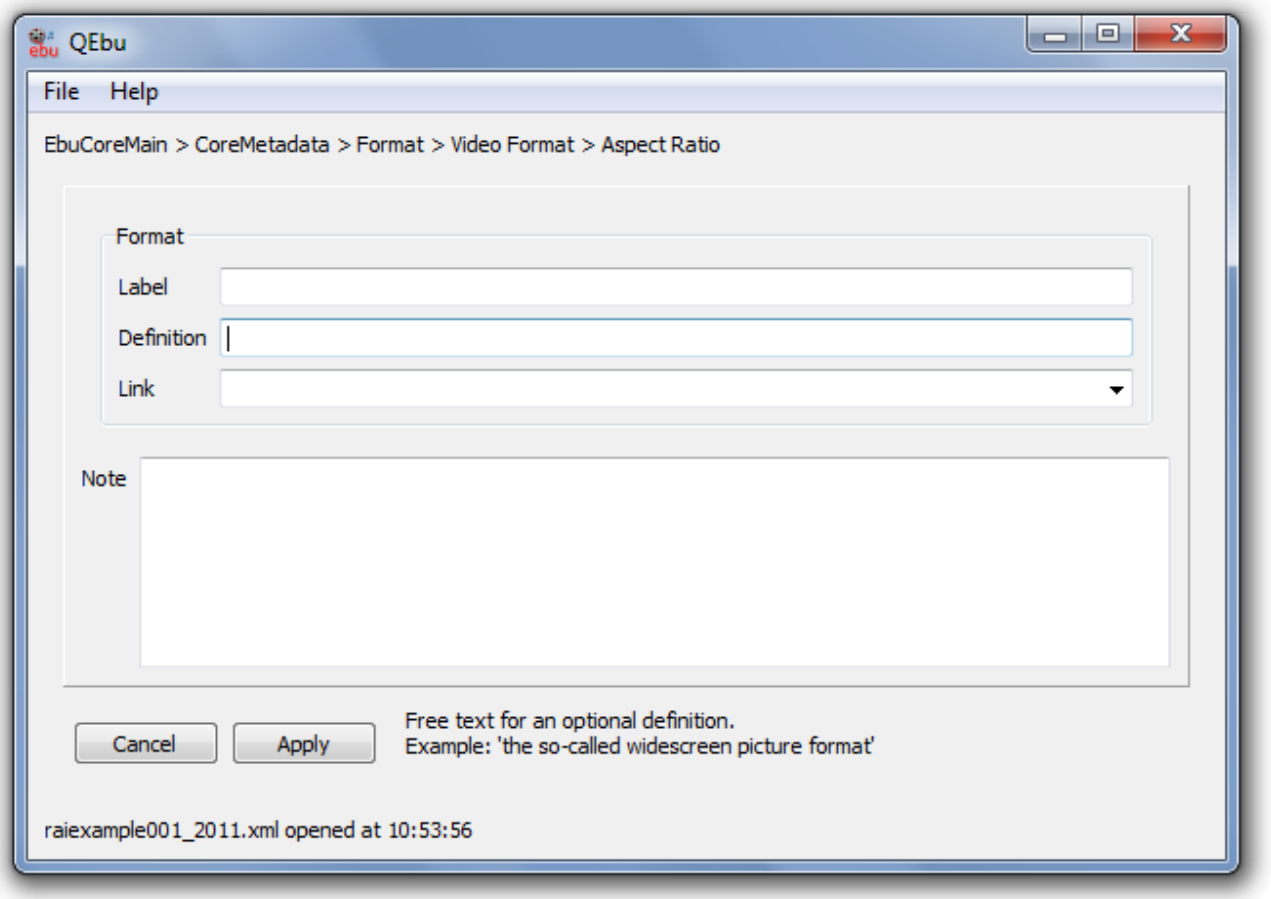

Figure 5: At the top of the screen a breadcrumb bar reminds the user of its current position, whilst at the bottom hints are provided about the currently active field.

When possible, instead of free text boxes, restricted value pickers or drop down selection menus have been included within the forms; however, even with the depth reached by the documentation of the standard, it has been impossible to provide full cover of all these aspects. Specifically, there are several fields that refer to standardized values, but only a few of them come with a corresponding dictionary to pick entries from, referenced in the EBUCore specification. In this case, QEbu fetches data from a set of XML files, which we retrieved from the EBU website.

In this specific case, given the need to load the data as fast as possible, an intermediate SAX parser is used to skim through these dictionary files, looking for just the minimum set of information required to fill in the combo boxes. In this way, it is possible to provide a response time from the application which goes unnoticed by the user, thus not detracting from the user experience. At the same time, an internal dictionary stores the fetched information for further reuse, avoiding multiple reading of the same external files while still providing a partial auto-completion feature.

FUTURE WORK

QEbu has been developed with regard to the EBUCore Metadata Set v1.3, so the limitations encountered refer to this specific version of the schema itself.

The main issue encountered is related to the management of time and duration values, which often appear in the same form in several different ways, i.e. both full date and year of publication, or hour of broadcast and full date/timezone reference, which could lead to potential inconsistencies. At the same time, the specifics do not restrict properly some of these fields, allowing for instance negative values of duration. In these cases, QEbu does not enforce further constraints, as we followed the rule that “the schema is always right”; but this is an area that requires more work.

Also, it would be really useful to determine a comprehensive set of values for those dictionary driven fields previously cited, in order to guide and restrict the user to just the specific set of valid values during the input phase.

As stated in the introduction, this tool has been developed as an academic term project, so it misses direct feedback from people with commercial experience in this field. We strongly believe that suggestions from experienced users could lead to still better tailored forms and a more functional application overall, given the review of QEbu from a wider perspective.

EBUCore is a still evolving standard, so it is natural that QEbu will evolve with it. We have done our best to make changes as simple as possible, so it should not be a difficult task to include new features, or to alter what is already available to embrace the changes introduced with newer versions of the standard itself.

CONCLUSIONS

This review describes a GUI-based metadata editor with respect to the EBUCore Metadata Set v1.3. Its purpose is to provide a way to easily manage and create XML instances, according to certain specifics, in a simple and coherent way, allowing the user to focus on the content and avoiding the need to worry about the structure of the documents themselves.

The aim is that fast and simple creation of this kind of documentation could lead to its usage in further applications related, for example, to the management of archived content, whether this is audio, video or any other kind of media. Providing tools to make these steps easier should have a very positive effect on the whole chain of content archiving and retrieval.

Because of the user-centred nature of this editor, it requires direct user feedback for its evolution and improvement. Further enhancement will thus be possible by collating responses from the content creation community relating to QEbu and this class of metadata editing tools in general.

QEbu is released as free software under terms of the GPL 3 license.

Source code is available, along with compiled Windows binaries, at the project repository [4].

REFERENCES

[1] EBU Core Metadata Set (EBUCore), ver 1.3, EBU Tech 3293, October 2011

[2] XML Schema reference documentation, http://www.w3.org/standards/techs/xmlschema

[3] Qt GUI framework reference documentation, http://qt-project.org/doc/qt-4.8/

[4] QEbu repository, *https://github.com/Nazardo/Qebu*

AUTHOR BIOGRAPHIES

MAURO LEVRA

Mauro Levra recently graduated cum laude with a M.Sc. degree in Computer Engineering from the Polytechnic University of Turin, Italy.

His thesis work concerned digital identity management in federated architectures, with particular focus on the European project STORK 2.0 which has aimed at developing an architecture that ensures interoperability between all European identification systems. Mauro's work addressed the issues that needed to be solved in order to allow and make use of cross-border user identities.

This research covered most existing federated models and architectures, and led to the proposal of a new model for STORK 2.0 infrastructure.

DENIS PATTI

Denis Patti is an assistant researcher at the Polytechnic University of Turin, Italy, where he obtained his M.Sc. degree in Computer Engineering in 2012.

He wrote his master thesis in cooperation with his colleague Paolo Pasini and the formal methods group of the Polytechnic of Turin. In his work he studied a multi-engine model checking suite, known as PdTRAV, under both algorithmic and architectural perspectives, in order to improve its overall performance in verifying digital designs.

Since March 2013, Denis has again joined forces with the formal methods group to continue his previous work in the model checking field.

PAOLO PASINI

Paolo Pasini graduated cum laude in 2012 with a M.Sc. degree in Computer Engineering from the Polytechnic University of Turin, Italy.

In his master's thesis, working with Denis Patti, he researched into aspects related to memory management and both algorithmic and architectural solutions in the context of model checking in multicore environments, under the supervision of the formal methods group of the Polytechnic of Turin.

As of March 2013, Paolo has undertaken a Ph.D. project in collaboration with the aforementioned research team, with the aim of further investigation into the refinement and transformation of digital circuits as a preliminary step to the verification phase itself.

GIOVANNI PESSIVA

Giovanni Pessiva gained his M.Sc. degree in Computer Engineering at the Polytechnic University of Turin, Italy, in 2012.

He wrote his master thesis in cooperation with his colleague Tao Su, on Android security, presenting an analysis of mobile security, and the tools available for malware detection. He developed a new framework capable of performing fast automated analyses of Android applications, in collaboration with Telecom Italia researchers. The analysis framework is currently used by Telecom Italia; Giovanni's static analysis module is released as open source.

He is currently working for the IT consulting group Reply S.p.A., specializing in Java EE technologies applied to the insurance industry.

STEFANO RICOSSA

Stefano Ricossa is currently a consultant with the IT consulting company Reply S.p.A., where he works in the information management area, specializing primarily in data warehousing and Big Data related topics.

He completed his course of studies at the Polytechnic University of Turin, Italy, receiving both the bachelor's and the master's degrees in Computer Engineering cum laude, respectively in 2010 and 2012.

Stefano's master thesis focused on formal verification algorithms and models, and led to the publication of a paper, co-authored with his colleagues Pasini and Patti, introducing an innovative technique for fast Cone-Of-Influence computation for multiple properties designs.

Published by the European Broadcasting Union, Geneva, Switzerland

ISSN: 1609-1469

Editor-in-Chief: Lieven Vermaele

Managing Editor: Eoghan O'Sullivan

Editor: David Crawford

E-mail: osullivan@ebu.ch