EBU
OPERATING EUROVISION AND EURORADIO

TECHNICAL REVIEW

Q2 2013

# QEBU: AN ADVANCED GRAPHICAL EDITOR FOR THE EBUCORE METADATA SET

**Внимание!**
Данный перевод **НЕ** претендует на аутентичность
и может содержать отдельные неточности.
Оригинал документа на сайте https://tech.ebu.ch

# QEBU: ПРОДВИНУТЫЙ ГРАФИЧЕСКИЙ РЕДАКТОР ДЛЯ НАБОРА МЕТАДАННЫХ EBUCORE

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

РЕЗЮМЕ

Создание и управление документами метаданных может быть очень трудно выполнять вручную. Для решения этой проблемы в контексте набора метаданных EBUCore v1.3 мы предлагаем редактор метаданных на базе GUI под названием QEbu, разработанный на курсе Multimedia Archival Techniques, проведенном в Политехническом университете Турина в сотрудничестве с RAI.

QEbu был разработан с целью обеспечения удобного графического редактора для создания и управления документами XML, освобождая пользователя от забот о структуре данных и позволяя ему сосредоточиться на самом контенте. Редактор пригоден как для опытных пользователей, так и для новичков в этой области.

QEbu разработан на C++ с использованием межплатформной библиотеки с открытым источником Qt 4.8; эта структура была выбрана с целью использования ее натуральных свойств для разработки интерфейсных приложений, работающих локально в компьютере пользователя в настольной среде.

tech.ebu.ch/techreview

## ВВЕДЕНИЕ

В анализе мультимедийного контента слово *метаданные* относится к любому фрагменту информации, используемому для описания определенного объекта контента. Этот вид информации требует определения должной структуры для эффективного управления, обмена и использования имеющихся знаний. Общий выбор – использовать для этой цели документы XML, и именно это достигается с *EBUCore Metadata Set* [1].

Помимо определения самого стандарта, так же важно обеспечить удобные инструменты, способные создавать и управлять этими данными, т.к. вручную это делать трудно и, скорее всего, ненадежно.

В целях решения последней проблемы был разработан QEbu, представленный в данной статье. QEbu – это редактор метаданных на базе GUI, построенный на спецификациях EBUCore v1.3. Предлагаемый инструмент разработан как семестровый проект для курса *Multimedia Archival Techniques* в Politecnico di Torino в сотрудничестве с RAI в рамках учебного плана магистра наук вычислительной техники.

## ЗАДАЧИ

QEbu разработан с целью обеспечения удобного графического редактора для создания и управления документами XML согласно спецификациям EBUCore.

Главная идея – освободить конечного пользователя от забот о структуре данных, позволив ему сосредоточиться на контенте. Функция инструмента – применять все необходимые ограничения и незаметно направлять пользователя к созданию правильно сформированного и структурно и семантически согласованного документа.

Инструмент предназначен как для опытных пользователей, так и для новичков в этой области. На этапе проектирования фундаментальной целью было обеспечить то, чем легко пользоваться без интенсивного обучения и что не требует глубоких знаний о предмете, давая таким образом незамедлительность поз потери эффективности. Эти аспекты будут уточняться в ходе обзора GUI в следующих разделах.

## QEBU: ОБЗОР

В этом разделе документа будут рассмотрены основные аспекты методов кодирования и детали реализации с анализом концептуальной модели и пользовательского интерфейса.

В следующих разделах все элементы, достойные внимания, рассматриваются по отдельности, и для некоторых дан общий фрагмент кода в форме шаблона. Сначала будет представлена модель данных, затем аспекты синтаксического анализа / сериализации и затем детали интерфейса.

## МОДЕЛЬ ДАННЫХ

Модель данных уходит корнями прямо в эталон схемы XML [2]; преобразование всех типов данных, определенных в этом документе, почти один к одному, лишь с несколькими исключениями, можно найти в окончательных результатах самой модели.

Каждый именованный тип данных, простой или сложный, имеет собственный класс, представляющий каждую деталь самого типа; а для неименованных типов решение о том, определять класс или нет, принимается с учетом ситуации и контекста, т.к. они происходят во временной последовательности.

В то же время, используя состав и наследование объектов, мы постарались свести число определенных классов к разумному минимуму.

Как правило, выбирается решение, дающее менее избыточный код с большей возможностью повторного применения; но в этом примере нет абсолютно верного или неверного подхода.

Принятый стиль программирования, в смысле соглашений об именах, расположения и т.п., предполагается как стандарт для развития структуры Qt GUI, который уже задокументирован официальным онлайновым сообществом [3].

Вот пояснительный пример класса общего типа:

*Пример 1: шаблон, соблюдаемый в декларации класса пользовательского типа*

```
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
};
```

Как видно в этом фрагменте кода, по существу, все классы, определенные для представления типа, состоят из публичной части, которая содержит базовые методы для создания и разрушения самого класса и для доступа к его частным членам.

В зависимости от типа внутренних членов, определяется соответствующий способ доступа к ним, но вообще объекты используют логику указателей: предопределенные типы, например, строки, управляются по значению; в списки и коллекции – как «живые» элементы, поэтому ссылка на саму коллекцию возвращается средствами доступа к ней.

Учитывая широкий диапазон возможностей форматирования, определенных в схеме XML, был реализован утилитный класс *TypeConverter,* вместе с моделью, для обеспечения базовых функций преобразования типов между датой / временем и строками, в поддержку синтаксического анализа / сериализации и в целях визуализации.

В данном примере детали фактической реализации методов опущены, т.к. они чрезвычайно тривиальны и не добавят никакой уместной информации в данный текст.

В случае высоко коррелированных типов или составных деклараций, для отражения отношений, найденных в опорной схеме, можно определить в том же файле несколько классов, но с составным подходом каждое определение никогда не бывает на том же уровне без использования внутренних деклараций.

Прежде чем описывать другие аспекты приложения, стоит рассмотреть, как организованы внутренние ссылки между объектами. В схеме есть некоторые элементы, например, *RightsType* и *PublicationType*, использующие XML ID и ID-ref для принуждения такого рода отношений; Оказалось необходимым определить правильный способ решения такой ситуации. Вместо простой ссылки на основе строки мы решили ввести среди объектов фактический указатель, который дает более строгое ограничение, но в свою очередь требует более тщательного управления. Поскольку пользователь может в любое время разрушать элементы, указанные по ссылке, в модель также была введена дополнительная карта ссылок. Эта карта содержит объекты, индексированные по ID ссылки, которому соответствует фактический указатель объекта и список объектов-приемников. Эти объекты требуется обновлять в случае удаления объекта по ссылке, отсюда название «приемник» (listener), во избежание нарушения предполагаемых отношений между ними.

## СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР И СЕРИАЛИЗАТОР

После определения модели, обеспечивающей персонализированное представление в памяти схемы, были разработаны два дополнительных элемента, синтаксический анализатор и сериализатор, для возможности перевода между документами XML и персонализированным представлением в памяти.

Эти модули используют функции DOM структуры Qt для чтения и записи файлов XML и подвергания простых интерфейсов прикладному коду GUI.

Хотя API DOM медленные и потребляют много памяти, для этих модулей они представляли эффективный вариант, давая гибкость и облегчая разработку кодов.

Учитывая сложность лежащей в основе схемы и необходимость доступа к записи API, SAX не был выбран в данном случае во избежание двух разных подходов к синтаксическому анализу и сериализации.

Следует также заметить, что файлы, с которыми работает это приложение, не особенно большие, поэтому аспекты, связанные с памятью и производительностью, не должны создавать проблем.

## СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР

Перед синтаксическим разбором в память каждый файл проверяется по схеме EBUCore, активизируя xmllint, инструмент, запускаемый из командной строки и обеспечиваемый бесплатной межплатформной C-библиотекой libxml2.

Решение об использовании внешнего агента проверки было принято после анализа библиотек вычислений XML, обеспечиваемых Qt, которое оценило их как неадекватные для требований QEbu.

Исполняемая программа xmllint должна быть доступна программе во время работы. В среде Windows это значит, что ее можно поместить либо в папку QEbu, либо в папку, на которую ссылается PATH.

Выход агента проверки синтаксически анализируется, выбирая информацию о возможных ошибках; затем эти детали передаются пользователю.

Все функции проверки упакованы во вспомогательный класс *Validator*.

Если приложение не имеет доступа к агенту проверки, оно покажет сообщение с вопросом, хочет ли пользователь продолжать синтаксический анализ, даже если файл может быть поврежден или неправильно сформирован. Если загружен непроверенный файл, то результат приложения непредсказуем; даже если проверен каждый элемент, что позволяет избежать отказов, фактический результат самого синтаксического анализа определить невозможно. В этом случае пользователь решает, работать ли с процессом, производящим такой чувствительный результат.

Базовый подход к синтаксическому анализу файла в память иллюстрируется следующим фрагментом кода, который показывает способ его выполнения с произвольным сетевым элементом древовидного представления DOM.

*Пример 2: базовый фрагмент кода, иллюстрирующий поведение синтаксического анализатора*

```
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;
}
```

Синтаксический анализ – это не более чем полный обзор дерева документа с последовательным спуском по каждой ветви до листьев. В этой точке оконечный элемент преобразуется в нашу персонализированную модель и прилагается к представлению в оперативной памяти. Этот процесс повторяется до тех пор, пока ничто не останется неуправляемым.

Дубликат представления в оперативной памяти может показаться необязательным, т.к. DOM уже строит собственное, но это облегчает управление на следующих этапах. Когда нужно взаимодействие модели с пользовательским интерфейсом, гораздо удобнее иметь определенные типы и классы, для которых можно определить общепринятые ограничения, проверку работоспособности и обеспечить должные методы доступа, а не использовать общие узлы или элементы в памяти.

## СЕРИАЛИЗАТОР

Сериализатор – двойник синтаксического анализатора, который следует тому же принципу. Это тоже полный обзор нашего графика объектов для записи в файле, что содержится в памяти.

В этом случае вместо дерева используется слово график, т.к. в некоторых ситуациях можно найти ссылки между объектами, не связанными родительскими / дочерними отношениями. В схеме можно найти элементы типа IDref, которые управляются как фактические указатели в предлагаемой модели. Это ведет к чуть более сложным процессам управления данными, но упрощает верификацию опорных ограничений и общей согласованности структуры или, по крайней мере, делает не четко определенные ссылки METADATA явно выраженными, т.к. необработанные исключения ведут к проблемам сегментации.

Следующий фрагмент дает быстрый пример шаблона для сериализации данных.

*Пример 3: базовый фрагмент кода, иллюстрирующий поведение сериализатора*

```
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;
}
```

Прежде чем закончить этот раздел, следует заметить, что функции синтаксического анализа используются и в другом аспекте приложения, непосредственно связанном с автозавершением с помощью файлов словаря; но они будут проиллюстрированы в своем должном контексте вместе с описанием пользовательского интерфейса в следующем разделе.

## GUI И ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ

Стандарт EBUCore [1] содержит большое количество специфических типов данных, поэтому для ясности и во избежание перегрузки пользовательского интерфейса для каждого из этих типов разработана и реализована специальная форма; аналогично, для модели реализован определенный класс.

Форма для корневого элемента схемы дает только точку входа в лежащую в основе структуру персонализированных представлений, и пользователь может согласованно и, надеемся, разумно переходить по всем элементам и экранам.

Для примера на Рис. 1 представлен скриншот предварительного просмотра одной из форм QEbu. На следующих рисунках будут показаны дальнейшие скриншоты для иллюстрации дополнительных деталей.

Было решено проектировать интерфейс так, чтобы в каждый момент времени рассматривался только один аспект, т.е. элемент, документа XML. Стандарт метаданных имеет богатое содержание и охватывает широкий спектр возможных приложений, поэтому неправильное управление этими аспектами легко может привести к беспорядочному и перегруженному интерфейсу. Во избежание этого для каждого элемента схемы была разработана специальная форма, приспособленная для максимизации согласованности содержимого, сохраняя привлекательный вид.

В качестве общего правила, принципиальные критерии, соблюдаемые в разработке каждого представления, можно сформулировать следующим образом:

- По возможности атрибуты управляются через поля редактирования;
- Для значений с ограниченным диапазоном используются указатели вместе с дополнительными флажками для управления опциями;
- Дочерние элементы имеют собственную форму, за некоторыми исключениями (т.е. группы);
- При возможности атрибуты / элементы группируются значащим способом;
- Части форм по возможности используются повторно, давая согласованность композиции.

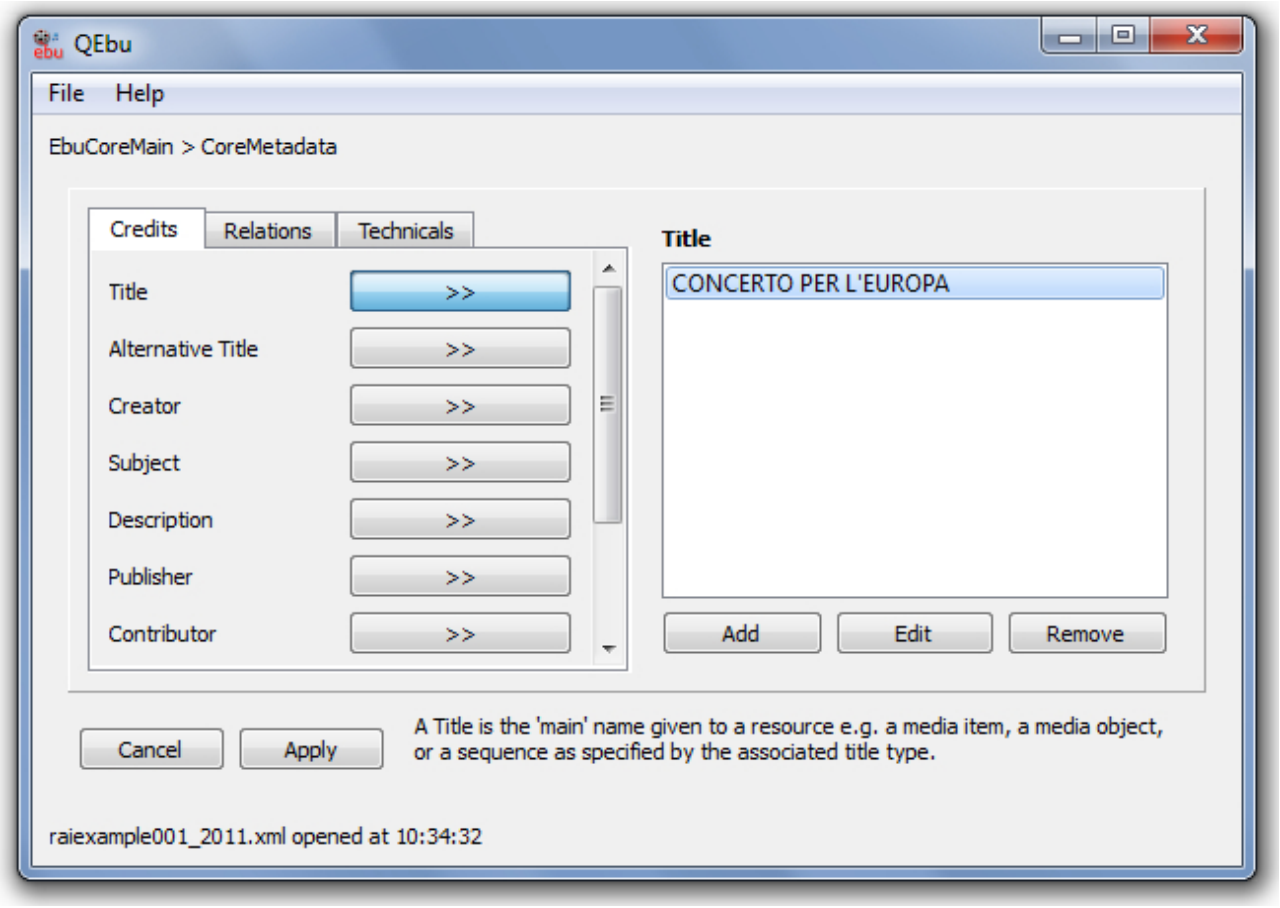


*Рис. 1: Одна из форм QEbu*

Конечно, при разнообразии типов и содержания вышеуказанный пример может не быть применим к любому элементу в модели, но он достаточно общий, чтобы показать обычный вид большинства форм.

С другой стороны, этот подход также требовался для эффективного управления транзакцией между формами с целью полного устранения всплывающих диалогов и сохранения одного разворачивающегося окна без отдельных панелей, разбросанных по всему экрану. Следуя тому же принципу, что и для мобильных приложений, мы решили реализовать персонализированную стопку форм, или виджетов, для моделирования транзакций между элементами, в которых виден и активен только самый верхний, а остальные живы, но заморожены под поверхностью, в ожидании снова попасть наверх. Пример такого подхода представлен на Рис. 2.

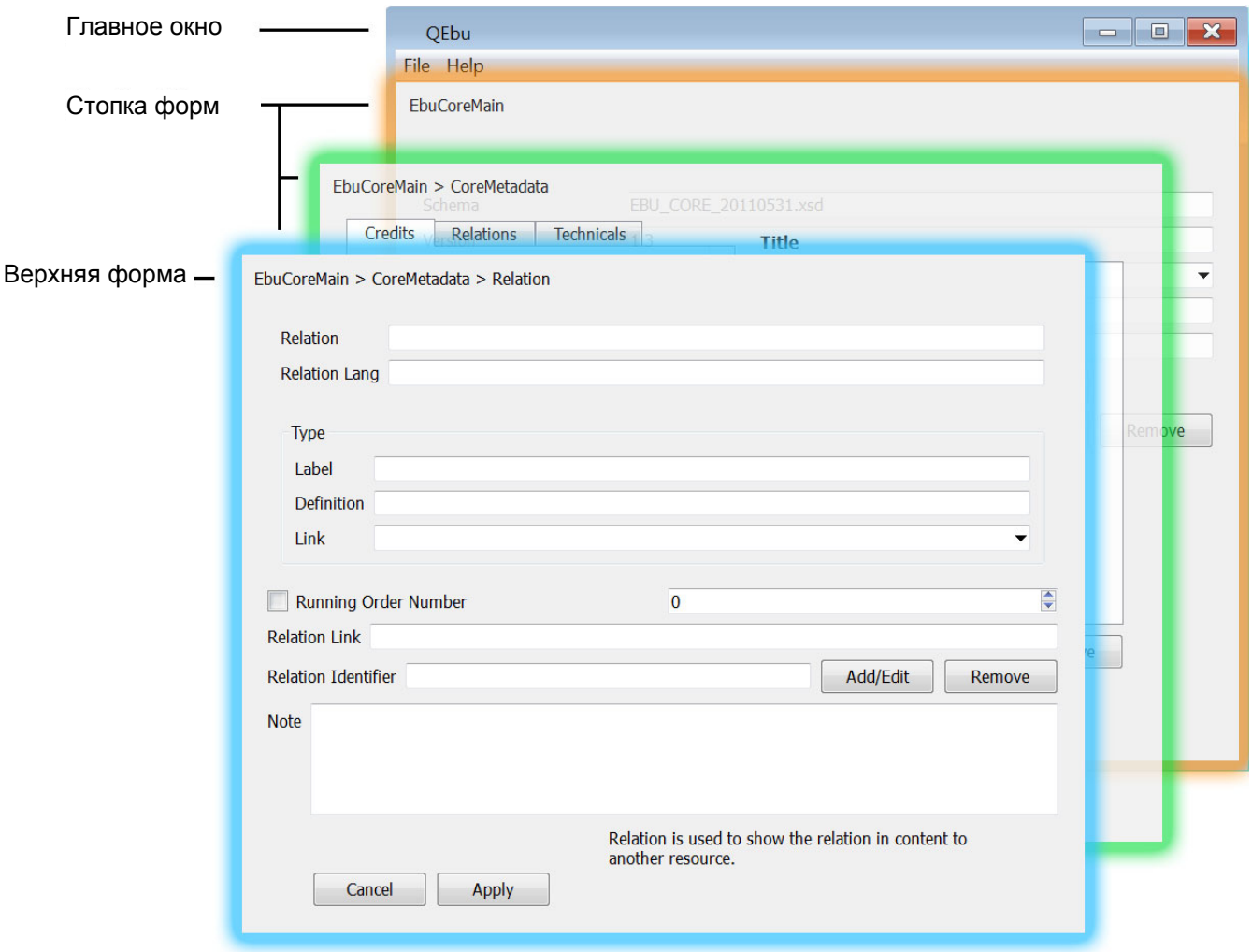


*Рис. 2: Структура штабельного решения виджетов.*

В целях сохранения согласованной навигации между формами мы решили сделать модель транзакций как можно проще, как показано представлениями жизненного цикла и схемы работ на Рис. 3 и 4.

Конечному пользователю можно двигаться по структуре данных только строго линейно и нельзя перепрыгивать через ветви дерева, представляющего создаваемый или модифицируемый документ.

Логика такого поведения определяется в персонализированном виджете, а именно ‘StackableWidget’, из которого наследуются все формы приложения и который управляет общими событиями, который могут произойти в течение жизни определенной формы, например, ее создание, разрушение или изменение видимости.

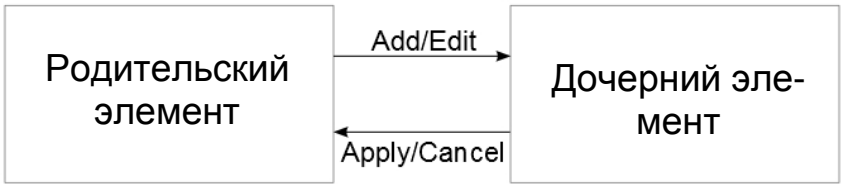


*Рис. 3: Упрощенный жизненный цикл формы.*

Когда пользователь выбирает Add/Edit дочерний элемент, новая форма создается и становится видимой, в то время как родитель временно скрыт. Затем с помощью кнопки Apply или Cancel можно восстановить ранее замороженную панель, вызвав разрушение самой верхней формы и контекстный роспуск ее ресурсов.

Как уже говорилось, по возможности отдельные формы сконструированы для использования виджетов, обеспечиваемых напрямую структурой Qt. Однако в особых случаях нужно разрабатывать персонализированные решения из-за ограничений имеющихся виджетов; например, в случае длинных целых значений, которые виджеты Qt не поддерживают, или необычных характеристик некоторых типов атрибутов. Так, некоторые типы с ограниченным содержанием, например, ElementType, были встроены прямо в родительский объект во избежание распространения форм с низким содержанием.

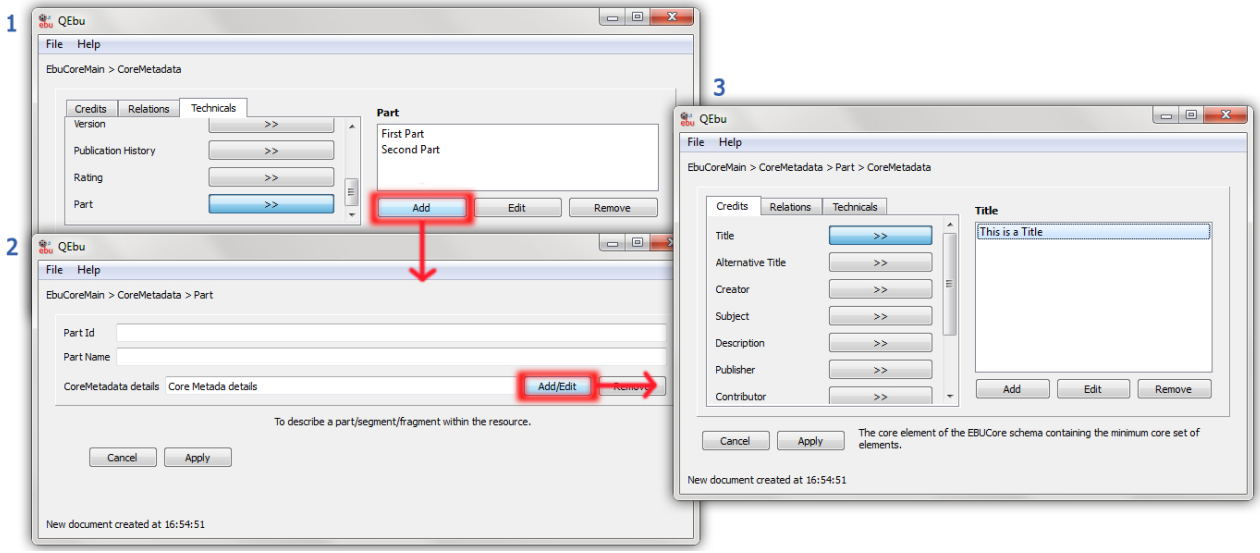


*Рис. 4: Пример схемы работ с эволюцией пользовательского интерфейса.*

Как показано на Рис. 4, QEbu позволяет легкое решение даже сложных ситуаций, например, управление частями. Спецификация позволяет базовому элементу включать один или более экземпляров того же типа, что и само ядро; но этим по-прежнему легко управлять, следуя линейному поперечному режиму, реализованному в инструменте.

Кроме того, особое внимание было уделено управлению многочисленными субэлементами с бесконечной кардинальностью [0..*] (т.е. списками неограниченной переменной длины). Сосуществование в данном элементе множества таких неограниченных субэлементов, часто с другими неограниченными элементами, создает некоторые проблемы в дизайне четкого и удобного интерфейса. Если взглянуть повнимательнее, можно идентифицировать два критических аспекта, связанных со списками. Во-первых, они могут быстро привести к перегрузке интерфейса из-за объема информации, который может передавать каждый элемент списка и который поэтому должен быть представлен. Во-вторых, без специальных мер  их сосуществование с элементами [0..1] может вызвать небольшие несоответствия в интерфейсе, которые, в свою очередь, могут запутать пользователя.

Для решения этих проблем был разработан специальный виджет. Избыток информации устраняется всего одним списком в каждый момент времени для редактирования, без необходимости представлять там же все детали, передаваемые всеми списками. Согласованность между неограниченными списками и отдельными элементами поддерживается использованием в обоих случаях одного и того виджета, при необходимости не давая возможности добавлять новые элементы после первой вставки.

С учетом всех перечисленных аспектов и для обеспечения дальнейшего руководства конечному пользователю во время навигации по структуре документа, несколько устройств были встроены прямо в программный интерфейс. Наверху каждой формы есть что-то вроде строки навигации, отслеживающей текущее положение в схеме для прямого сообщения о глубине позиции внутри схемы. Кроме того, каждая форма содержит собственный фрагмент документации для дальнейшего указания, как правильно заполнять каждое поле. Пример дан на Рис. 5.

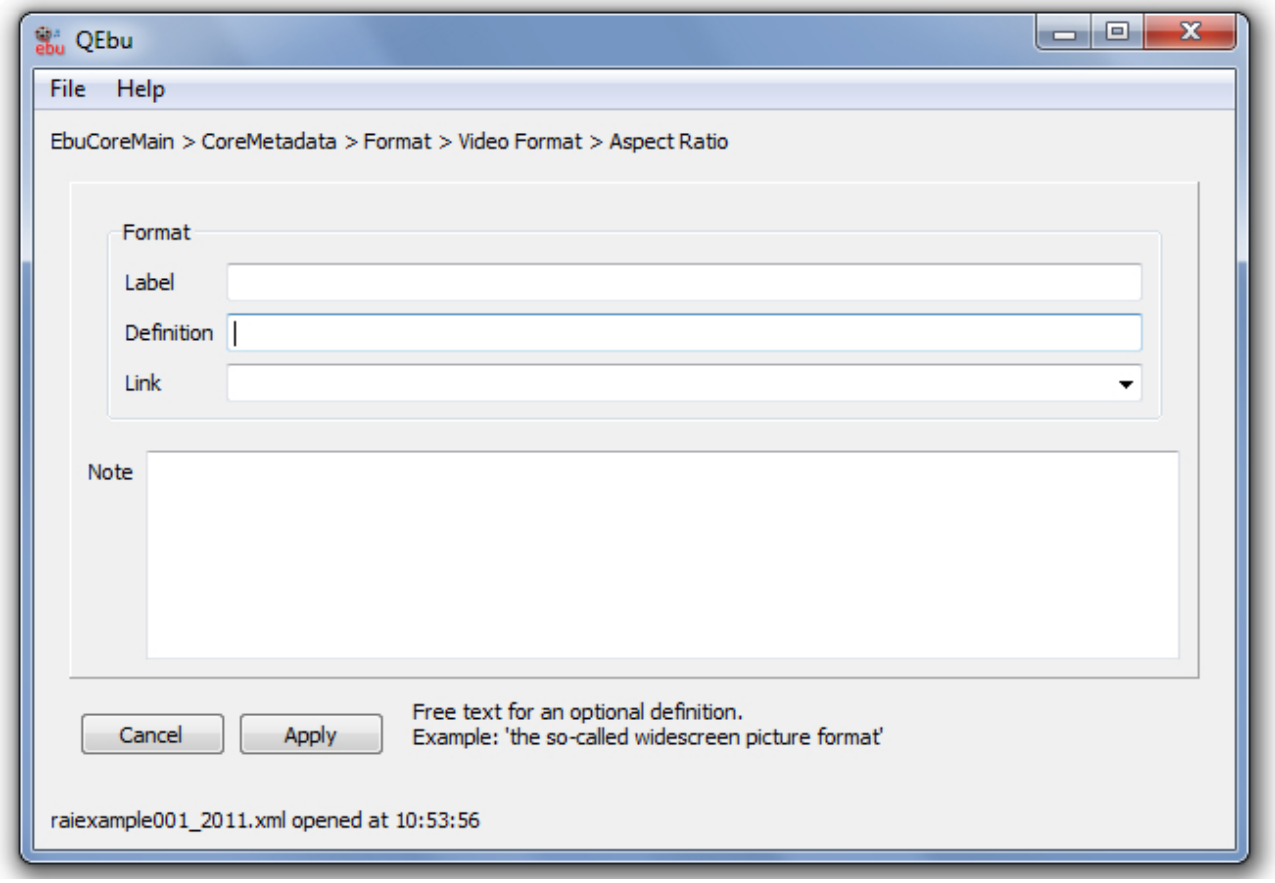


*Рис. 5: Наверху экрана строка навигации напоминает пользователю о его текущей позиции, а внизу даны подсказки о текущем активном поле.*

По возможности, вместо окна со свободным текстом, в формы включены указатели с ограниченным значением или выпадающие меню выбора; однако даже при всей глубине документации стандарта было невозможно полностью охватить все эти аспекты. Например, есть несколько полей, относящихся к стандартизированным значениям, но лишь несколько из них имеют соответствующий словарь для выбора входных данных, к которым обращается спецификация EBUCore. В этом случае QEbu выбирал данные из набора файлов XML, которые мы взяли с сайта EBU. В данном случае, при необходимости загружать данные как можно быстрее, для беглого чтения этих файлов словаря используется промежуточный синтаксический анализатор SAX, который ищет лишь минимальный набор информации, необходимой для заполнения комбинированных полей. Таким образом, можно обеспечить время реакции от приложения, работающего незаметно для пользователя, не отвлекая его. В то же время внутренний словарь хранит выбранную информацию для повторного использования, избегая повторного чтения тех же внешних файлов и обеспечивая при этом функцию частичного автозавершения.

## БУДУЩАЯ РАБОТА

QEbu разработан в соответствии с EBUCore Metadata Set v1.3, поэтому его ограничения связаны с данной версией самой схемы.

Главная проблема связана с управлением значениями времени и хронометража, которые часто бывают в одной и той же форме по-разному, т.е. полная дата и год публикации или час трансляции и полная дата с часовым поясом, что может вести к потенциальным несоответствиям. В то же время специфика не ограничивает некоторые поля должным образом, позволяя, например, отрицательные значения хронометража. В таких случаях QEbu не налагает дальнейших ограничений, т.к. мы следуем правилу, что «схема всегда правильна»; но эта область требует больше работы.

Также было бы полезно определить всеобъемлющий набор значений для вышеупомянутых полей, управляемых словарем, для руководства и ограничения пользователя определенным набором действительных значений на этапе ввода.

Как было сказано  во введении, этот инструмент был разработан как проект семестра, поэтому не имеет прямых отзывов от людей с коммерческим опытом в данной области. Мы верим, что предложе-

ния опытных пользователей могут дать более специализированные формы и более функциональное применение, если смотреть на QEbu с более широкой перспективы.

EBUCore еще развивается, и естественно, что с ним будет развиваться и QEbu. Мы приложили все усилия, чтобы сделать изменения как можно проще, поэтому должно быть нетрудно включать новые характеристики или менять то, что уже есть, для включения изменений, введенных с новых версиях самого стандарта.

## В ЗАКЛЮЧЕНИЕ

В этом обзоре описан редактор метаданных на базе GUI, связанный с EBUCore Metadata Set v1.3. Его цель – обеспечить легкий способ управления и создания экземпляров XML в соответствии с определенной спецификой, просто и согласованно, позволяя пользователю сосредоточиться на контенте и не заботиться о структуре самих документов.

Задача состоит в том, что быстрое и простое создание такого рода документации может привести к ее использованию в дальнейших приложениях, связанных, например, с управлением архивным контентом, будь то аудио, видео или другой вид медиа. Обеспечение инструментов, облегчающих эти этапы, должно очень положительно повлиять на всю цепочку архивирования и поиска контента.

Вследствие характера этого редактора, сконцентрированного на пользователе, он требует прямых отзывов пользователей для эволюции и улучшения. Таким образом, дальнейшая доработка будет возможна при сопоставлении отзывов сообщества создателей контента относительно QEbu и данного класса инструментов редактирования метаданных в целом.

QEbu выпущен как бесплатное программное обеспечение по условиям лицензии GPL 3.

Имеется код источника вместе со скомпилированными двоичными файлами в хранилище проекта [4].

## ССЫЛКИ

[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*

## БИОГРАФИИ АВТОРОВ

**MAURO LEVRA**

Mauro Levra недавно с отличием получил диплом магистра наук по вычислительной технике в Политехническом университете Турина, Италия.

Его дипломная работа касалась цифрового управления идентичностью в объединенных инфраструктурах, с особым фокусом на европейском проекте STORK 2.0, направленном на развитие архитектуры, гарантирующей взаимодействие между всеми европейскими системами идентификации. Работа Mauro направлена на проблемы, которые необходимо решить для возможности использования международной идентичности пользователей.

Это исследование включало большинство существующих объединенных моделей и архитектур и привело к предложению новой модели инфраструктуры STORK 2.0.

**DENIS PATTI**

Denis Patti – научный ассистент в Политехническом университете Турина, Италия, где ы 2012 г. получил диплом магистра наук по вычислительной технике.

Свой диплом он писал в сотрудничестве с коллегой Paolo Pasini и университетской группой формальных методов. В своей работе он изучал комплект для проверки многопроцессорной модели, известный как PdTRAV, в алгоритмической и архитектурной перспективе, в целях улучшения его общей производительности в проверке цифровых дизайноа.

С марта 2013 г. Denis снова объединил усилия с группой формальных методов для продолжения предыдущей работы в области проверки моделей.

**PAOLO PASINI**

Paolo Pasini с отличием получил диплом магистра наук по вычислительной технике в Политехническом университете Турина, Италия в 2012 г.

В своем дипломе, работая с Denis Patti, он изучал аспекты, связанные с управлением памятью и алгоритмическими и архитектурными решениями в контексте проверки моделей в многопроцессорной среде, под руководством группы формальных методов университета.

В марте 2013 г. Paolo занимается кандидатским проектом в сотрудничестве с вышеупомянутой группой с целью дельнейшего исследования детализации и трансформации цифровых схем как предварительного этапа самого процесса верификации.

**GIOVANNI PESSIVA**

Giovanni Pessiva получил диплом магистра наук по вычислительной технике. в Политехническом университете Турина, Италия, в 2012 г.

Он писал свой диплом в сотрудничестве с коллегой Tao Su по безопасности Android, представляющий анализ мобильной безопасности и инструменты для обнаружения вредоносных программ. Он разработал новую структуру, способную выполнять автоматический анализ приложений Android, в сотрудничестве с исследователями Telecom Italia. Структура анализа чейчас используется Telecom Italia; Модуль статического анализа Giovanni выпущен как открытый источник.

Сейчас он работает в консультационной группе IT Reply S.p.A., специализирующейся на технологиях Java EE для страховой индустрии.

**STEFANO RICOSSA**

Stefano Ricossa работает сейчас консультантом IT компании Reply S.p.A. в области управления информацией, специализируясь на информационных хранилищах и темах, связанных с Big Data.

Он прошел курс обучения в Политехническом университете Турина, Италия, с отличием получив дипломы бакалавра и магистра по вычислительной технике в 2010 и 2012 г. соответственно.

Диплом магистра Stefano был сфокусирован на алгоритмах и моделях формальной верификации и привел к публикации документа в соавторстве с коллегами Pasini и Patti, где был введен новаторский метод быстрого вычисления конуса влияния для дизайнов со множеством свойств.

Опубликовано European Broadcasting Union, Женева, Швейцария

ISSN: 1609-1469

Ответственный редактор: Lieven Vermaele
Главный редактор: Eoghan O'Sullivan
Редактор: David Crawford
E-mail: osullivan@ebu.ch

*Ответственность за мнения, выраженные в данной статье, лежит исключительно на авторе.*