Generating an Ontology Specific Editor

Hannes Niederhausen, Sven Windisch, Lutz Maicher
Topic Maps Lab
University of Leipzig
Leipzig, Germany
{*niederhausen, windisch, maicher*}*@informatik.uni-leipzig.de*

***Abstract*—Semantic technologies like Topic Maps provide a generic way of structured data representation. These technologies can be used to create data stores of any kind. To use them, it is necessary to fill them with data and therefore an editor is needed which provides input masks for the data. In this paper we present an Editor Generator Toolkit that enables developers to easily create small and fast editor applications for multiple platforms that allow easy collation of data.**

***Keywords*-Topic Maps, ontology, editor, TMCL, Eclipse RCP**

I. Introduction

Semantic technologies provide a generic way of structured data representation and thus are highly applicable as data stores for applications of all kinds. Beyond that, data stores must be filled with data and as most data is not available in an easily transformable format, an editor application is needed for the manual handling of the data.

In this paper, we present an *Editor Generator Toolkit* that allows the generation of ontology specific editor applications for topic map ontologies in a flexible multi-step process. The whole wizard-driven process is based on an ontology that is specified by the user as a Topic Maps schema and leads the user to the finished editor desktop application.

The Editor Generator Toolkit is an extension for the well-known *Eclipse IDE*, which not only provides the environment to develop applications in many programming languages, but also provides a basic architecture for standalone desktop applications. This architecture, the so-called *Eclipse RCP*, is used by the Editor Generator Toolkit as the basic application framework and is extended by an ontology specific domain model.

Section III contains an overview of the Editor Generator Toolkit architecture and its individual parts. Section IV describes the workflow that leads from the initial Topic Maps schema to the finished editor application. In Section V we introduce the Yacca editor as an example application. Section VI summarizes our results and provides an outlook on future work.

II. State of the Art

The tool we present here is no ontology editor in terms of products like Protégé or OntoStudio. Whereas these products offer features for creation and editing of knowledge bases with one of the popular ontology languages as OWL or RDFS, we present a toolchain to create an instance editor as a desktop application for a specific Topic Maps schema ontology [1]. However, an ontology editor is required to create the basic Topic Maps schema from which the editor application is built. This can be done with another Eclipse extension, called *Onotoa* [2], [3].

All currently existing Topic Maps domain editors like Ontopoly or Topincs are web-based applications and thus require both a central application server and a reliable network connection. As these requirements are not always available, we focus on desktop applications to avoid any obstacles for the users of the editor application.

Furthermore, we separate the process of ontology creation from the process of editing the actual data, so that users without knowlegde of the Topic Maps schema language are able to use the editor application that was previously created by a topic maps expert.

III. Architecture

The editor applications that are provided through the Editor Generator Toolkit consist of several components and are based on the Eclipse RCP. See Figure 1 for an overview of these components. The individual components, which were developed at the Topic Maps Lab, will be explained in this section.

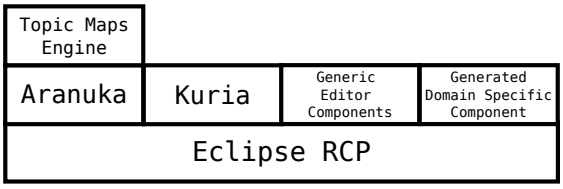

Figure 1. The components of a generated editor application. Based on the Eclipse RCP, Aranuka handles the mapping of the domain model to the Topic Map engine while Kuria, the generic editor components, and the generated domain specific component provide the visible parts of the editor application.

A. Eclipse RCP

Eclipse is a plug-in based software development system which initially was created as integrated development environment (IDE) for Java. The environment is enhanced

regularly by the work of the large community that grew around Eclipse. With version 3.0, the Eclipse Foundation released the Eclipse RCP (Rich Client Platform), which consists of a stripped version of the IDE and can be used to create new applications that make use of the architecture of Eclipse. The Eclipse RCP components form the foundation of the editor applications that are generated with the Editor Generator Toolkit.

Using the Eclipse RCP also results in platform independent applications. The *Standard Widget Toolkit* (SWT) is a Eclipse plugin and responsible for rendering the user interface. It is directly connected to the platform specific user interface libraries. The Eclipse Foundation provides a set of bindings for a lot of platforms, which can be used to create Eclipse RCP applications for different operating systems and hardware architectures. A complete description how to develop Eclipse RCP applications can be found in [4].

B. Aranuka

Semantic technologies like Topic Maps provide a generic way to represent data. On top of that, programming interfaces of Topic Maps engines like the *Topic Maps Application Programming Interface* (TMAPI) [5] need to be generic too and therefore developers must have a good understanding of Topic Maps even if they are supposed to write a domain specific application.

Aranuka, an open source project from the Topic Maps Lab, provides a way to map a domain model that was developed in Java to an underlying Topic Maps engine [6]. The developer then uses his model classes and a helper class from Aranuka called *Session*. This Session is able to retrieve topics from the topic map and persist topics back into the topic map. Aranuka uses connectors to associate Aranuka to a specific Topic Maps engine. These connectors bind the Aranuka core to any TMAPI supporting Topic Maps engine. Right now, Aranuka was tested with the Topic Maps engines *tinyTiM* and *Ontopia* and works flawlessly with both of them.

The configuration of the domain model is done via Java annotations. For every construct of the Topic Maps Data Model (TMDM) an specific annotation can be used in classes or attributes. Every annotation contains several properties which can be used to configure the mapping. The annotation **@Occurrence** for instance, has a type property that specifies the subject identifier of the occurrence type of the mapped attribute. More information about Aranuka annotations and its use can be found in [7].

After annotating the model classes a Configurator instance is used to tell Aranuka which classes should be mapped to topics and which connector should be used. It is also possible to add names to the types specified in the annotations. This is done via an internal mapping between the subject identifier set of the annotation and the value of the name that should be added to the used topic type. The Configurator provides additional methods for adding and removing prefixes, too. These prefixes can be used in any URI which is used as identifier in annotations and instances. Every instance which is mapped to a topic must have at least one identifier which should be a URI.

C. Kuria

The W3C created a group to analyze the need of model-based user interfaces (see [8]). In its report the groud states that the development of web applications should use utilities to build the final user interface based on the model of the application and the target platform. Instead of developing for different platforms, a description language based on XML should be used to map the domain model to specific user interface elements. With this description different layouts and designs can be generated, based on the target platform, which could be a mobile device or a standard browser.

A similar approach is implemented with *Kuria*, an open source project from the Topic Maps Lab [9]. Instead of web applications, Kuria generates input masks for desktop applications. It is modularized to support different approaches of declaration and generation. The Editor Generator Toolkit uses three Kuria modules, the *Kuria Runtime* module, the *Kuria Annotation* module, and the *Kuria SWTGenerator* module.

Kuria Runtime: The Kuria Runtime module is the core of Kuria. User interfaces are composed of elements like buttons, windows, labels, dialogs, which are called widgets. For every widget exists a descriptor, which is called binding. Bindings contain the model specific information of the widget, for instance which text is valid for a text field and an accessor and mutator method. With the binding it is possible to set the value of an attribute of an object instance. In addition, bindings for tree nodes and table columns exists.

Kuria Annotation: The Kuria Annotation module is used to create widget bindings based on annotations on the model classes. If no annotation exists, a binding based on the datatype and the name of the field is created. It is also possible to hide an attribute, which can be done with the annotation **@Hidden**. For a complete list of annotations and their attributes refere to [10].

Kuria SWTGenerator: The *Standard Widget Toolkit* (SWT) was developed by IBM to create an efficient Java widget toolkit which uses the libraries of the underlying operating system. One other well known user interface (UI) library for Java is Swing, which is part of the Java SDK. Swing renders UI elements by itself, which results in a consistent look and feel, because applications using Swing look very much the same on every system. However, these applications look kind of alien in most operating systems.

In contrast, the SWT wraps UI elements of the operating system, and thus all applications that rely on SWT need platform dependent libraries for every system. Though this has the advantage of providing the look and feel of the

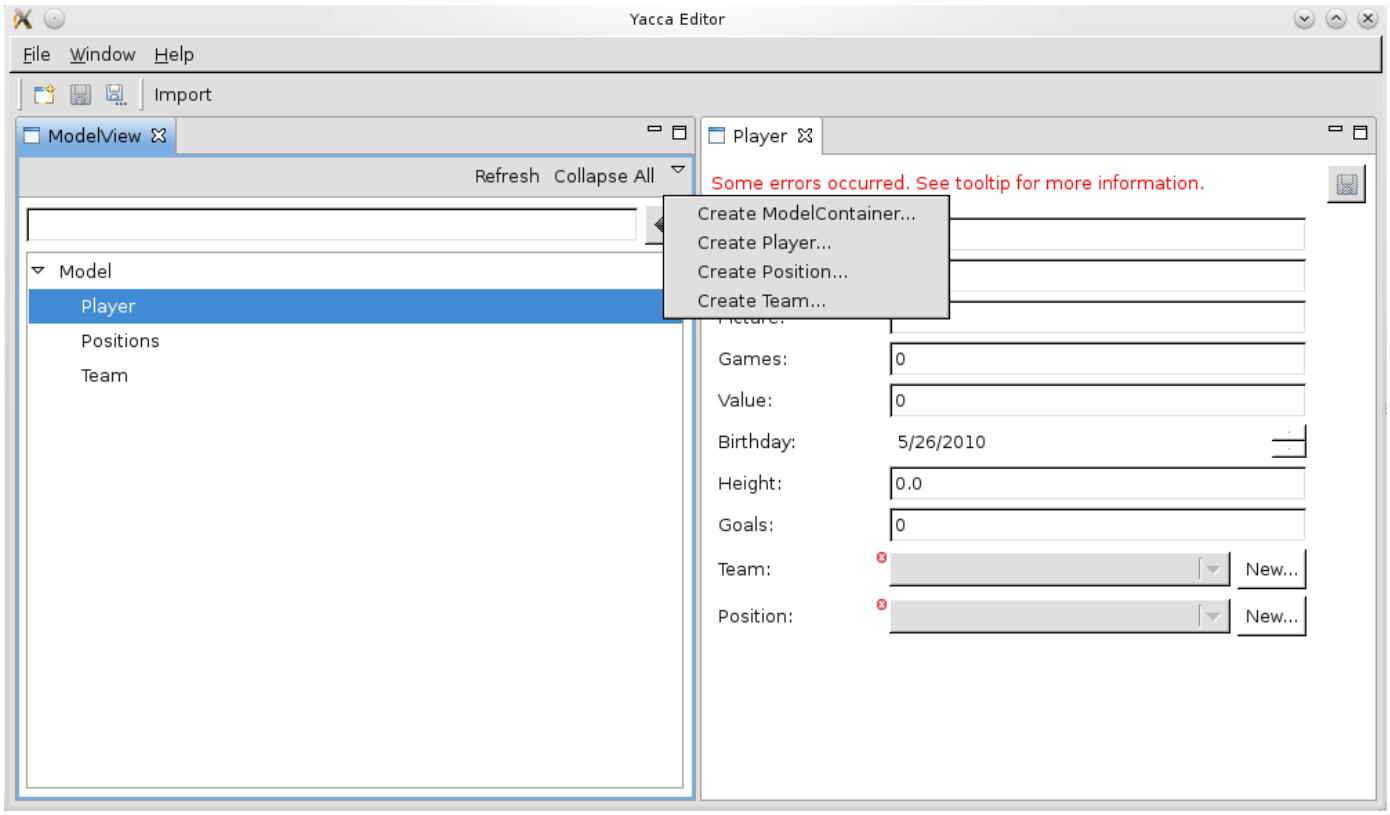

Figure 2. Empty editor window. On the *left side* is the modelview. On the *right side*, for every opened model instance an editor tab is opened.

underlying operating system. Another advantage is the increased speed that comes while using the operating system's native libraries, which is much faster than using the emulated ones, like Swing does.

The SWTGenerator module is used to generate a user interface based on the widget bindings using SWT. In addition, it provides methods to easyly generate tree and table widgets. To create an input mask for an instance, the SWTGenerator uses the parent widget and the class of the instance. The SWTGenerator then checks whether a binding for the class exists and generates the input mask according to the bindings.

The Editor Generator Toolkit produces Kuria annotations for every generated domain model. In the resulting editor application, the overview tree and all input masks are generated by Kuria.

D. Generic Editor Components

The *Generic Editor Components* provide the containers for the user interfaces. These components are fixed and configured by the generated annotated domain model.

In Figure 2, an empty application window is shown. On the left side is the *ModelView*. This window contains a tree representing the model. For every type in the ontology a node exists and its children are the instances of the type. New instance of a model type can be created with the provided context menu. Alternatively, the view provides a menu on the left side of the title bar with options for every topic type. Already existing instances can be edited by simply clicking on them.

The individual editors for the instances are placed on the right side. It is possible to open multiple editors. If an editor is activated the *Save*-button in the toolbar persists the edited instance into the topic map. Every editor provides a *Save and Close* button inside the editor window which persists the instance in the topic map and closes the editor.

Generated Domain Specific Component: The last component of the Editor Generator Toolkit (cf. Figure 1) is the *Generated Domain Specific Component*. This plug-in contains the generated domain model classes and additional product configurations. The latter are necessary for the configuration of the *Eclipse IDE* and contain information about the plug-ins that are part of the editor and thus must be exported together with the editor application. The configuration of Aranuka is also part of this plug-in and can be found in the **ModelHandler** class. After generating the code, the names for types can be added there. The selection of the Aranuka connector is also possible in this

class and can be changed any time after the generation process.

IV. Generate an Editor

This chapter explains how to generate an editor application, what prerequisites are needed and what steps are necessary to create the application. An overview of the steps are shown in Figure 3.

Prerequisites

The Editor Generator Toolkit consists of a set of Eclipse plug-ins that add the generation facility to the Eclipse IDE. In order to work with these plug-ins, a working Eclipse with the *Java Development Toolkit* (JDT) is needed. To create the base ontology, it is strongly recommended to use the Topic Maps schema editor Onotoa.

Create the Ontology

The first step is to create the ontology. This can be done via text editors writing plain TMCL in CTM-notation or any visual editor. We advise to use Onotoa, a visual editor which can be installed directly into the Eclipse development environment.

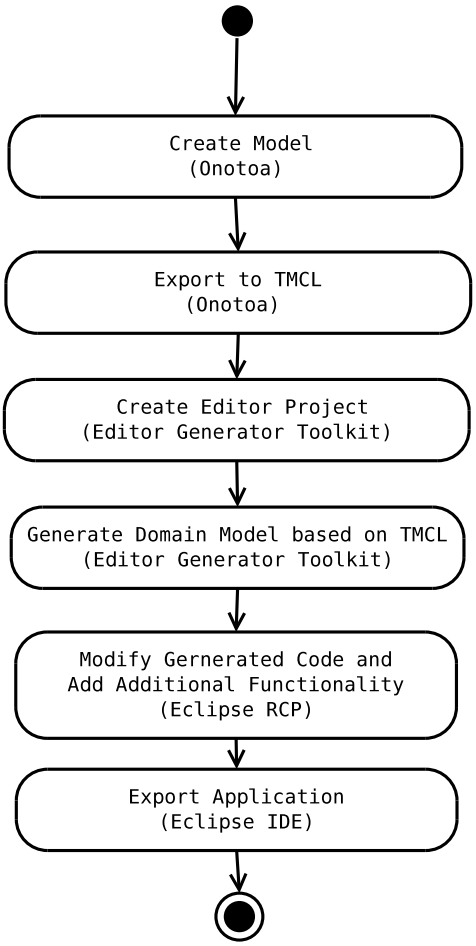

Figure 3. Chain of activities to develop generate an ontology specific editor.

After creating the model in Onotoa it should be exported to TMCL which is indicated by step 2 in the activity diagram in Figure 3.

Create an Editor Project

The Editor Generator Toolkit adds a new wizard to the *New Project Wizard* list of Eclipse. Create a new project and fill in all required data into the first page of the wizard. This page asks for a project name, which should have a form like a Java package name. The name of the application will be used in the title bar of the application. It will be used as name of the executable binary of the application, too. The third entry is a drop box which allows selection of the used Topic Maps engine.

In the second editor page the name of the schema file is required. If this field is empty no model will be created and the developer has to create it on his own.

Modify Generated Code and Add Additional Functionality

The generated domain model is a set of Java classes which are generated on the basis of the topic types in the given TMCL schema. In addition, these classes have attributes annotated for Kuria and Aranuka. The generated classes can be revised and modified to tailor the editor and receive the expected input masks for the models. Examples for modifications are:

- A topic has an occurrence of type string. In the class an attribute is generated with a **@Textfield** annotation of Kuria. This annotation indicates the use of a text field with one line. To have a multiline text area an additional attribute must be added to the annotation.
- For topics with associations in every generated class an attribute for the counter player exists. This is because of the bidirectional nature of associations in topic maps. In most cases, only one direction is needed in the editor interface. It is recommended to remove one of the counter player attributes.

The generated editor is an eclipse application and therefore provides some possibilities to extend the application. These can be done inside the generated domain specific plug-in or in additional plug-ins. All new dependencies should be added to the product configuration, which is responsible for the correct export of every required plug-in. Developing additional functionality is optional and not needed to export a working editor application.

Export the Application

The editor for the product configuration provides an export function, which is a link in the first page of the editor. By activating the link, a wizard opens and asks for the target directory and the desired target platforms.

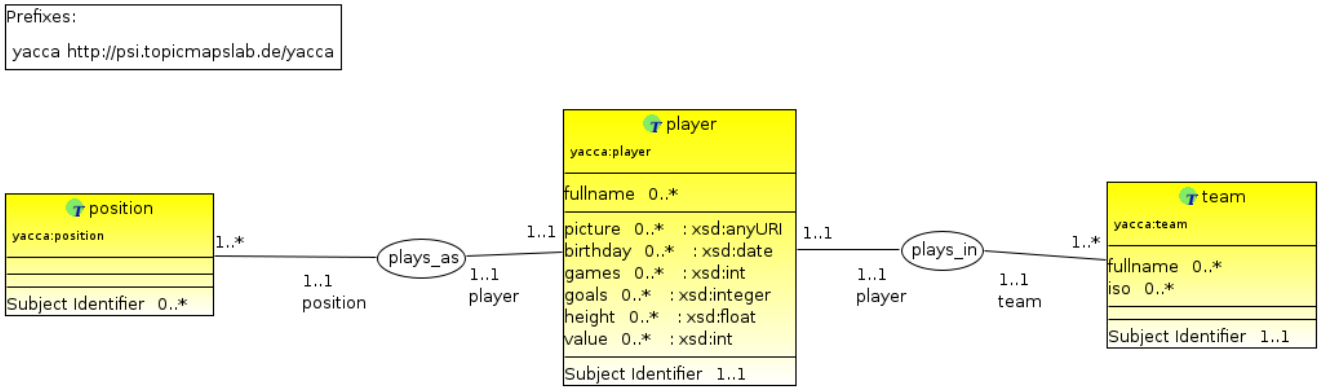

Figure 4. The Yacca ontology as it was created with Onotoa.

After the export is finished, the target directory contains the created applications, one for each platform, which can then be deployed to the target machines.

V. Example Editor: Yacca

In this section, we present an example on how a generated editor application can be used in the developmental process of Topic Maps driven web applications. The example we provide is called *Yacca* and was voted into top ten at the 2010 ESWC AI Mashup challenge.

In Yacca, the power of structured data – that comes with the use of Topic Maps – as data store is combined with the flexibility of TMQL as a query language that allows to extract just the desired amount of data from the Topic Map. The so called "Yacca cards" are HTML snippets, produced from the structured data and provided by *Maiana*, the social Topic Maps explorer from the Topic Maps Lab [11]. The topic map for the data store was created with an editor that was generated with the Editor Generator Toolkit.

The Yacca topic map contains three topic types and two associations (cf. Figure 4). Based on this Topic Map schema, the editor in Figure 5 was created. The ModelView shows the three topic types that are now classes in the Java application. On the right side, an editor for a player is visible. The dialog on the left contains another input mask for the position topic. This dialog opens when pushing the *New* button next to the position field in the editor. A similar dialog opens for the team of the player.

The Yacca editor has an additional feature: An import of players from a comma separated file. This function was used after creating the team and position topics.

After creating the topic map with the editor, it was uploaded to Maiana [12]. The editor is still used for updating the topic map, for instance in case a player gets injured and can not play.

VI. Conclusion and Future Work

Summary

The Editor Generator Toolkit provides an simple and fast way to generate small and fast Topic Maps editor applications that are feasible for easy collation of data. Based on the popular Eclipse Platform it integrates into most established development processes. With the used base technologies – Eclipse RCP and the Standard Widget Toolkit – the generated editor can be used on almost every platform. With our approach, Topic Maps experts with little to no experience in Java programming are able to build editor applications based on their Topic Maps schema.

In this paper, we have explained the architecture of the Editor Generator Toolkit and its base modules. Furthermore, we showed the process of creating the editor application and have given an example for successful use of a generated editor.

Future Work

The generated editor is an application with a simple user interface. Especially with a lot of topics the tree in the ModelView gets to large. Search facilities can be implemented to provide ways to find topics with a specific property. This could be done by specifying the property inside a dialog or entering a TMQL query [13].

In the current state, the generator produces default Kuria annotations and attributes for every site of an association. In future release some additional schema elements in form of reification or occurrences of specific types should be introduces and supported by the schema editor. With these additional elements the manual modification of the generated code would be unnecessary.

References

[1] M. Weiten, *Semantic Knowledge Management*. Springer, 2009, ch. OntoSTUDIO as a Ontology Engineering Environment, pp. 51–60.

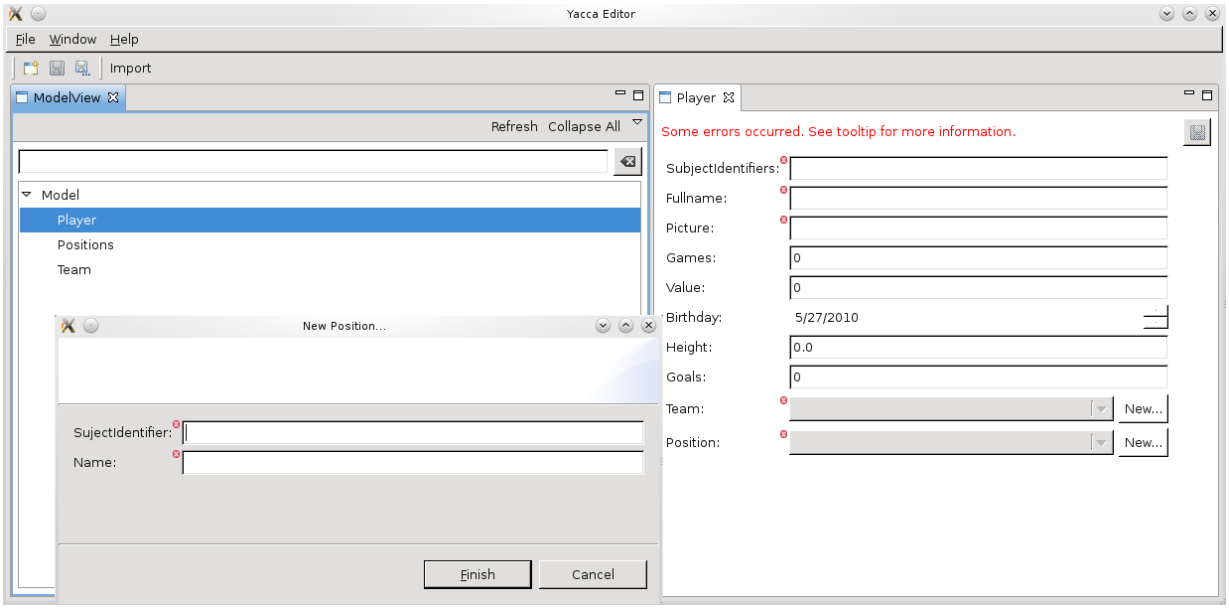

Figure 5. The Yacca editor application with ModelView, Editor and Dialog to create a new position.

[2] H. Niederhausen, "Onotoa – TMCL-basierter grafischer Schema-Editor für Topic Maps," Master's thesis, University of Leipzig, 2009.

[3] H. Niederhausen, *Onotoa Handbook*, last checked: July 19 2010. [Online]. Available: http://docs.topicmapslab.de/onotoa

[4] J. McAffer and J.-M. Lemieux, *Eclipse Rich Client Platform*. Addison-Wesley Professional, 2005.

[5] L. Heuer and J. Schmidt, "Tmapi 2.0," in *TMRA – Subject Centric Computing*, 2008, p. 129.

[6] H. Niederhausen, *Aranuka Project Site*, last checked: July 19 2010. [Online]. Available: http://code.google.com/p/aranuka

[7] H. Niederhausen, "Aranuka documentation," last checked: July 19 2010. [Online]. Available: http://docs.topicmapslab.de/aranuka

[8] J. M. C. Fonseca, "W3C Incubator Group Report 04 May 2010," 2010, last checked: July 19 2010. [Online]. Available: http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui-20100504/

[9] H. Niederhausen, *Kuria Project Site*, last checked: July 19 2010. [Online]. Available: http://code.google.com/p/kuria

[10] H. Niederhausen, "Kuria Documentation," last checked: July 19 2010. [Online]. Available: http://docs.topicmapslab.de/kuria

[11] Topic Maps Lab, *Maiana*, last checked: July 19 2010. [Online]. Available: http://maiana.topicmapslab.de

[12] Sven Windisch, *Yacca Topic Map*, last checked: July 19 2010. [Online]. Available: http://maiana.topicmapslab.de/u/yacca/tm/yacca

[13] L. M. Garshol and R. Barta, "Topic maps query language," last checked: July 19 2010. [Online]. Available: http://www.isotopicmaps.org/tmql/tmql.html