ActiveTM: A Topic Maps – Object Mapper

Benjamin Bock

University of Leipzig, Johannisgasse 26, 04103 Leipzig, Germany

bb--tmra2008-activetm@bock.be

Abstract. Currently, the most common way to programmatically access Topic Maps data is the use of a Topic Maps API, like TMAPI. Another approach, besides the use of a query language like TMQL, is the encapsulation of the Topic Maps related code in domain-specific model classes. This concept is similar to object-relational mapping (ORM) which encapsulates access to a relational database inside the model classes. These techniques decouple the data store specific code from the business logic. For ORM, there are several prevalent design patterns, most notable the Active Record pattern by Fowler. For Topic Maps, no such pattern is established. This paper introduces Active Topic Maps, a pattern for Topic Maps – object mapping, the domain-specific language ActiveTMML to define such a mapping, and a prototypical implementation, called ActiveTM. ActiveTM is based on Ruby Topic Maps and also supports the generation of web-forms based on ActiveTMML definitions. This full-featured software stack greatly improves the development productivity of Topic Maps based portals compared to other solutions.

1 Introduction

The Topic Maps Data Model (TMDM) [1] offers many liberties while designing an ontology. Many classes and methods are required to offer the full flexibility and functionality of the TMDM to a programmer using a Topic Maps engine with a generic application programming interface (API), e.g. Ruby Topic Maps (RTM) [2]. The parameters of many of these methods are manifold. This is because the Topic Maps constructs represented as instances of the classes have many properties to be retrieved and modified. All these methods are aligned to the TMDM and not optimized for the particular domain. The development of a domain application requires the programmer to either use the generic TMAPI-

Maicher, L.; Garshol, L. M. (eds.): *Subject-centric computing. Fourth International Conference on Topic Maps Research and Applications, TMRA 2008, Leipzig, Germany, October 16-17, 2008, Revised Selected Papers.* (Leipziger Beiträge zur Informatik: XII) - ISBN 978-3-941152-05-2

like methods directly or to encapsulate them in domain model classes. Here, TMAPI does not only refer to the Java- and PHP-based standardized APIs [3] [4] but to any Topic Maps API with a similar set of functions.

The encapsulation in domain model classes allows to use only the model objects in the other parts of the application because the program code to access the data store once resides solely in the model. This technique is commonly referred to as Model-View-Controller (MVC) pattern [13]. The creation of these model classes is straightforward from the definition of an ontology but still requires some amount of work. The idea presented here is to define the ontology in a domain-specific language (DSL) [6] and use this to generate the model classes, including the code to retrieve and persist the objects in a Topic Maps data store.

The prevalent functions of persistent storage are create, read, update, and delete (CRUD) [14]. Create and update include scrutinizing the constraints for the particular data objects. Using TMAPI directly does not allow to check particular constraints of the domain model. The consistency of a topic map can be verified afterwards using a custom constraint language or eventually with Topic Maps Constraint Language TMCL [12]. Deletion of data objects may be restricted due to other constraints or may entail other deletions or updates. In relational databases, these functions may trigger functions to preserve consistency. For Topic Maps, no standardized approach exists. Additionally, access control based on data in the topic map is needed in real world applications but not provided in a standardized way by current solutions. Access control is not covered in this paper.

2 Previous Approaches

The most commonly used technique to access Topic Maps data is the usage of a library with a TMAPI-like interface. A goal is to encapsulate and thus simplify the usage of a TMDM data store using a special API. The next subsections illustrate the usage of TMAPI and two previous approaches to optimize the way how Topic Maps data is accessed and how a domain can be modeled.

The technique object-relational mapping (ORM) is, besides using SQL, the predominant way to access relational databases. The popular Active Record design pattern implements ORM. The homonymous Ruby library offers a DSL to describe domain model classes and transparently implements the database access for these classes.

Bringing together these two techniques finally leads to the concept of Topic Maps – object mapping.

2.1 A Basic Read Operation using TMAPI

The following example illustrates the steps necessary using Java TMAPI 1.0 to read a certain name from a topic `t`. It does not even include handling of scopes but it is already quite lengthy.

```
// get typing topic (pseudo code)
Topic type = tm.getTopicBySubjectIdentifier
           ("http://psi.example.com/firstname");

// iterate over all topic names

Iterator i=t.getTopicNames().iterator();
while (i.hasNext()) {
    TopicName tn = (TopicName) i.next();
    // check type
    if (tn.type == type) {
        // use name, e.g. output it
        System.out.print( tn.getValue() );
        break;
    }
}
```

HEUER introduces the concept of accessing characteristics of a topic using a Hash-like syntax in the Topic Maps engine *Mappa* [5]. Transferring this concept to the Java language, the previous example would look like this[1]:

```
Set<TopicName> names = t.get
                  ("-http://psi.example.com/firstname");
for (TopicName tn : names ) {
    System.out.print( tn.getValue() );
    // break after the first one
    break;
}
```

This is significantly shorter than the first example. Using it in Python, the language Mappa is written in, is even shorter as Python's Syntax is more terse than Java's. In Ruby Topic Maps, the Hash-like access works the same way as in Mappa:

```
puts t["-http://psi.example.com/firstname"].first.value
```

Still, the way to access the data is not domain specific. The subject identifier in the string cannot be checked by a compiler nor by an interpreter at runtime. Assuming the topic `t` represents an object `p` of class `Person`. There should be two methods in `p`: one to get and one to set a single first name. Depending on the

[1] This implicitly assumes an API using Java generics

domain ontology (where multiple first names may be allowed), this could also be methods to get and set a list of first names and additionally to add and remove single first names from this list.

2.2 Topic Maps Objects

Moore, Ahmed, and Brodie demonstrated *Topic Map Objects* (TMO) at the TMRA 2006 conference [16]. TMO is a framework providing domain-specific classes to retrieve and update Topic Maps data in a distributed environment. Moore, Ahmed, and Brodie don't build upon TMAPI but on *Topic Map Webservices* (TMWS). TMWS provides access to a topic map using a SOAP interface. The goal of TMO is to unify the advantages of TMWS with the features of modern object-oriented languages like Java and C#. The resulting framework allows programmers to work with domain objects without knowing the TMDM in detail.

TMO consists of two components: The first component of TMO is TMWS. The TMWS framework used is functionally equivalent to TMAPI. In this component, no higher level of abstraction or domain-specificity is introduced. From the perspective of abstraction, the feature of transactional updates is not relevant, however this could be exploited to integrate constraint checking. The intention of this feature seems to be optimization of network traffic. The Object Manager Service (OMS) is the second component of TMO. This component can create domain-specific objects from Topic Maps data and provide an application with these objects in a serialized fashion. The topic maps ontology data is part of the class definition while the instance data resides in the object. The object manager contains the program code to read all properties of the domain object from the topic map and fill the private variables in the objects at the time of its construction. The updates in the objects can be transferred back to TMWS later. The domain classes contain the program code to update the objects, later read accesses see the current values.

TMO is written in the C# programming language for the .NET platform. It is part of the commercial product TMCore by Networked Planet Limited, Oxford, UK, to which the authors belong [at the time of writing]. A graphical user interface, based on Microsoft Visual Studio is planned[2] but not publicly available yet. It involves creation of an XML document which makes the annotation of domain classes unnecessary. The automatic creation of program classes seems not to be planned, so one has to assume that the code to update a topic map has to be written by hand. The following example shows the definition of a class `Person` with a property firstname.

[2] According to Moore, in a private conversation on 2008-04-06

```
[TopicTypeAttribute("http://www.networkedplanet.com/person")]
public class Person : TopicMapObjectBase {
    private string m_name;
    private string m_age;

        [TopicNameAttribute()]
        public string FirstName {
            get { return m_name; }
            set {
                OccurrenceSet(this, "FirstName", value);
                m_name = value ;
            }
        }
        [TopicOccurrenceAttribute
           ("http://www.networkedplanet.com/ octypes/age")]
        public string Age {
            get { return m_age; }
            set {
                OccurrenceSet(this ,"Age" , value);
                m_age = value ;
            }
        }
}
```

The example is derived from one of the examples given in [16] and allows to reason that TMO uses a pre-TMDM data model[3], topic names do not have a type yet. The occurrence age shows how the type would be specified. Another observation is that there is no clear distinction between `OccurrenceSet` and `TopicNameSet`. This might be a typo in the document, though.

A graphical user interface would clearly be an advantage of this solution, while the usage of a web service may lead to performance deficits compared to the usage of a local library's API. TMO objects can only be used asynchronously, a direct update of the underlying topic map is not possible with this architecture. The domain-specific information is given at (at least) two locations: the object manager (for reading) and the domain classes (for updating).

2.3 Bogachev's Subject-Centric Programming Language

In [8], BOGACHEV presents the similarities of Topic Maps and the COBOL programming language. The advantage of COBOL is the definition and manipulation of *business data* in the language. In many modern programming languages this domain specific information was outsourced to a relational

[3] At the time of writing of TMO, TMDM was not finalized, so in fact this is not a big surprise.

database, decreasing transparency and simplicity. With these assumptions in mind, BOGACHEV developed his subject-centric programming language [9].

He criticizes that object-oriented languages help to model things on a computer, but not to represent knowledge about these things. He questions what happens if information changes over time and how to deal with information from different sources. Furthermore, he asks how interference rules and calculated values can be respected in such a system and how visibility and update rights can be bound to specific user groups.

To address all this, he defines *metaproperties* which are classes derived from a specific property type. In the example, the property `firstname` is derived from `ActiveTopic::Name`[4].

```
class FirstName < ActiveTopic::Name
    psi           'http://psi.ontopedia.net/firstname'
    historical    true
    card_max      1
    domain        :person
end

class Person < ActiveTopic::Topic
    psi          'http://psi.ontopedia.net/Person'
    name         :firstname
end
```

For both, the definition and the usage, BOGACHEV orientates himself at the syntax of the Active Record Ruby library, but there is no implementation[5].

A continuative work is *Authoring topic maps using Ruby-based DSL: CTM, the way I like it* [10], a domain-specific language for defining Topic Maps ontologies and facts in a Ruby-based syntax. The emphasis here is not a programming framework but an alternative approach to the Compact Topic Maps Notation (CTM) [11].

2.4 The Active Record Design Pattern

FOWLER develops the design pattern *Active Record* [15] which implements the principle of object-relational mapping. The program code to access the storage layer (i.e. the relational database) is directly part of the model classes in Active Record. The objects are created or retrieved from the database using class

[4] The choice of the namespace prefix `Active` for the class and the usage of a Ruby-based syntax make it obvious that he has the Ruby on Rails component Active Record in mind.

[5] Private conversation, 2008-04-02

methods from the same class. They are stored using instance methods of the concrete objects.

The Ruby library *Active Record*[6] is part of the web application framework *Ruby on Rails*[7]. It implements the homonymous design pattern. In Active Record, the names of the getters and setters for simple properties are derived from the column names in the database schema. They cannot be defined in the model classes and cannot be retrieved from the model classes without an active database connection. Thus, the complete model definition is available at runtime only. Associations between objects are defined using a DSL and not automatically derived from the database schema using e.g. the foreign keys. For a model class without associations, a class extending `ActiveRecord::Base` is sufficient. The statements to define an association to another class are called `has_one`, `has_many`, and `has_and_belongs_to_many`. The opposite table (or the join table respectively) holding the foreign key needs to use the statement `belongs_to`. For all statements, there are parameters to refine the definition if the schema does not match the naming convention exactly. The example shows the definition of a class `Person` with some associations and the creation of a single instance. The exact usage can be found in the Active Record API documentation[8].

```
class Person < ActiveRecord::Base
    belongs_to   :home_country, :class_name => "Country"
    has_many     :cars, :foreign_key => "owner_id"
end
p = Person.create
p.firstname = "Benjamin"
p.save
```

When calling the method `save` the library executes the following statement[9].

```
INSERT INTO 'people' (first_name) VALUES ('Benjamin');
```

The Active Record library provides a second, separate DSL called *Migrations* to describe the database schema. Changes to the ontology always require changes to the database schema and must be reflected there. Thus, a restart of an application is needed whenever the ontology and consequently the schema changes.

---

6 http://wiki.rubyonrails.org/rails/pages/ActiveRecord

7 http://www.rubyonrails.org

8 http://api.rubyonrails.org

9 Please note the table name is “people” but the class name is “Person”. This is a convention used in the Ruby library Active Record, which contains a pluralization module. In the class definition this convention can be overwritten

3 Domain Modeling

The definition of an ontology is part of the modeling of the domain to which the application should be specific. Nowadays, software developers are used to model their problem using object oriented techniques. There are many tools available to aid such a development process, ranging from a sheet of paper and a pen to sophisticated UML [17] editors. Integrated development environments like Netbeans[10] or Eclipse[11] directly assist writing code in a particular programming language. The result of a development process is a formal specification of the model, covering all relevant aspects to address the problem of the given domain.

Defining model classes using UML results in a class diagram from which domain specific code can be generated. The resulting code is self-contained and does not include a mapping to a Topic Maps ontology. Our goal is to model a Topic Maps ontology and the corresponding model classes at the same time. Generally, to allow an efficient workflow, it is essential to do exactly the things necessary and avoid everything else. Applied to modeling the ontology of a domain problem, this includes the description of the relevant entities, their characteristics and associations.

Using Topic Maps technology, the ontology is part of the topic map itself. The upcoming Topic Maps Constraint Language (TMCL) [12] strives to standardize the definition of ontologies in Topic Maps. However, this does not include naming of classes nor methods. For an ontology to model both, object-oriented and Topic-Maps-oriented aspects, TMCL has to be augmented or a new language has to be created. TMCL is not finalized at the time of writing and, following a pragmatic approach, the creation of a new language was chosen with ActiveTMML. As a later step, a formal mapping between ActiveTMML and TMCL should be defined. This could be done using a small ontology which defines the basic information necessary to create ActiveTMML code out of a Topic Maps ontology defined in TMCL.

Alternatively, TMCL fragments could be used as parameters to ActiveTMML statements. The benefit would be a single source for a complete ontology definition. The downside would be that this would presumably not integrate well with graphical TMCL editors.

[10] http://www.netbeans.org
[11] http://www.eclipse.org

4 ActiveTM

ActiveTM is a Ruby library implementing ActiveTMML, the Active Topic Maps Modeling Language. In this section, firstly ActiveTMML will be introduced, then the library itself is presented. The library ActiveTM is not the only use case for ActiveTMML, as it can also be used as a basis for code generation in other languages.

4.1 ActiveTMML

ActiveTMML is a ontology modeling language for both, Topic Maps and object-oriented models in a single language. It does not (yet) strive to be feature-complete regarding the flexibility of the TMDM but to be functional for the *common 80% of use cases*. As ActiveTMML is only suitable for ontology modeling, it is called a domain-specific language (DSL) [6] [7]. DSL are commonly divided into two types: internal and external DSL. While external DSL come with their own syntax, internal DSL borrow their syntax from a host programming language. Consequently, internal DSL are constrained by their host language's syntax but they also benefit from their toolchain, i.e. can be compiled or interpreted using the host language's tools. This liberates the developer of a internal DSL from developing a parser and leaves him with adding semantics to the given syntax. ActiveTMML is an internal DSL using the host language Ruby[12]. Compared to other popular programming languages like Java[13], C#[14], and Python[15], Ruby offers a comparatively free syntax.

There are two flavors of ActiveTMML. The standalone syntax uses just method calls like `model` and `occurrence` in special contexts. The in-class syntax is used as part of a class definition, as it is done in Active Record, and will be detailed in the ActiveTM section. The following example shows the standalone syntax:

```
model :Person do
  name :firstname
  name :lastname
  occurrence :age
end
```

[12] http://www.ruby-lang.org
[13] http://java.sun.com/
[14] http://www.ecma-international.org/publications/standards/Ecma-334.htm
[15] http://www.python.org/

This standalone ActiveTMML code technically calls a method `model` and passes two parameters: The symbol[16] `Person` and a block of code, introduced by `do` and ended by `end`. This is common Ruby syntax and can be executed by any Ruby interpreter. Consequently, ActiveTMML code can be seamlessly mixed with other Ruby code.

The method `model` uses a special context in which the definition of this particular model is evaluated. A context is a class, module or object which implements methods corresponding to the statements of ActiveTMML. During the evaluation of the block (which is the definition of a domain model class), the calls are delegated to the context class, module or object.

The code of the method `model` and the context can be *anything*, depending of the concrete implementation of the ActiveTMML language. Possibilities range from generating classes (as done in section 4.2) to generating files (e.g. source code for a particular language or library, as proposed in section 5). It is also possible to produce any other output, for example a relational database schema and ActiveRecord classes based on the model. There are prototypes fulfilling exactly this purpose. Additionally, the output of a TMCL file is an option.

The obvious object-oriented interpretation of the example above is to create a class called `Person` with getters and setters for the properties `firstname`, `lastname` and `age`. The Topic Maps interpretation of the same definition is a topic identified by the item identifier[17] "Person" typing other topics, its instances. The default identifier can be overwritten using the method `psi` in the code block. It is not possible to define PSIs of instances directly in ActiveTMML. An algorithm generating PSIs for instances depends on the concrete implementation of an ActiveTMML interpreter. Section 4.3 explains how this was solved in ActiveTM.

The example above defines three characteristics, two names and one occurrence. The argument to the methods `name` and `occurrence` is interpreted as an item identifier for the type of the characteristic. This can be overwritten using the keyword parameter `psi` in each method. The datatype for names is always string, for occurrences it defaults to string, too. The datatype of occurrences can be overwritten using the keyword parameter `datatype`.

A slightly bigger example shows the usage of keyword parameters in the Ruby syntax as well as the definition of a binary association. The definition of an association consists of a parameter for the name, the role type on this side of the

[16] Ruby symbols are similar to LISP symbols. In short, a Ruby symbol is a word preceded by a colon. It is commonly used as a constant. Technically it is comparable to an internalized `String` in Java.

[17] The item identifier is relative here. According to TMDM it must be resolved against a locator to make it absolute.

association, and the association type. The other role type is retrieved from the name of the association-property (in this case "country"), unless given as the fourth parameter. In natural languages, the type of the thing referred to, often[18] as is the type of the opposite role in a Topic Maps ontology.

```
model :Person do
  name :firstname, :psi => "http://psi.example.com/first"
  name :lastname,  :psi => "http://psi.example.com/last"
  occurrence :age, :datatype => "xsd:integer"
  has_one :country, "inhabitant", "country-inhabitant"
end
```

The statement `has_one` adds the constraint that there is only one country in the given association with this a particular person. The pendant to `has_one` is `has_many`. These two method names are inspired from the Active Record library, their parameters and interpretation differs due to the different intention of modeling.

An obvious feature to add to the ActiveTMML are model constraints, like defining cardinalities. This could be achieved using additional keyword parameters for example. The downside is the increasing complexity of both, the model code and the interpreter code. Once TMCL is finalized, it should be used to define finer granular constraints. For the implementation of ActiveTM, the same functionality is achieved using filters and validations as it is done in Active Record.

4.2 Definition of Model Classes in ActiveTM

Besides the standalone ActiveTMML code, a class definition in ActiveTM can be done using the standard Ruby syntax to define a class. Therefore, the class needs to be provided with the necessary code for the ActiveTMML statements. This can be done in two ways: by extending the superclass `ActiveTM::Base` (analogous to Active Record) or by including the module `ActiveTM::Topic` as a Mixin. The latter allows more liberties in terms of the class hierarchy.

```
class Person < ActiveTM::Base
  topic_map "http://psi.example.com/"
  psi "http://psi.example.com/ontology/person"

  name  :firstname
```

[18] An exception is for example the artificial language Lojban which was developed by the Logical Language Group in 1987. In this language, not the other role is addressed but the relation of the other thing to the current role. A child would refer to its "mother" (English term, natural referencing style) as "the one I am child of" (English term, Lojban referencing style), thus using the own role type.

```
    names :middlenames
    name  :lastname
    occurrence :age, :datatype => "xsd:integer"
    has_one :country, "inhabitant", "country-inhabitant",
                      :class => :Country

    def fullname
      "#{firstname} #{lastname}"
    end
  end
```

This creates a single class `Person`. The definition of the class itself is pure Ruby code. The statements `topic_map`, `psi`, `name` and so on are basically calls to methods in the class scope. The definitions of these methods are provided by the class `ActiveTM::Base` (or by the module `ActiveTM::Topic` respectively). Each instance of the class holds a single reference to a Topic in the underlying Topic Map.

The example above also introduces the method "topic_map". This method defines the base locator for this class. The statement "names" introduces a characteristic which may occur multiple times. Another addition is the usage of the keyword parameter "class" with the symbol "Country" in the "has_one" statement. This specifies that the retrieved object should be interpreted as an instance of class Country, no matter what other types it is instance of.

The definition of the method `fullname` in the previous example shows the mixture of of normal Ruby code with the ActiveTMML code, allowing to create virtual properties based on the definition of existing ones. The number-sign and the curly brackets are Ruby string interpolation syntax. This allows to embed the results of the methods called directly in the string.

4.3 Usage of ActiveTM Objects

As with the definition, the usage of ActiveTM objects is aligned with Active Record. Until now, only the ontology layer was covered. For the usage, the instance layer comes into play. In the instance layer, referencing instance topics a requirement. Referencing topics works using identifiers (internal, subject identifiers or subject locators) for creating and querying particular topics. The language-internal object references serve all other purposes.

The following example shows the creation of a new Person-object. A parameter can be passed to the create method to define an identifier. If not given, an identifier will be generated. The default algorithm to generate an identifier is to append a random fragment identifier to the type PSI. This can be overwritten by

providing the class with a instance method `generate_psi` which acts as a hook. This approach is similar to the `before_save` hook in Active Record.

```
p = Person.create("johndoe")
p.firstname = "John"
p.add_middlename "George"
p.lastname = "Doe"
p.save
```

As shown in the example, for single characteristics, a setter and a getter are created, for multiple characteristics an add method as well as a remove method and a getter are created. The same principle applies to `has_one` and `has_many`.

Similar to Active Record, there is a default finder as well as dynamic finder methods for the characteristics. The default finder takes an identifier, the dynamic finders accept an argument like the setter methods. The example shows several ways to retrieve the single topic created above. There are also finders to find multiple objects instead of only returning only the first one found. As always, the usage follows the Active Record example.

```
p = Person.find("johndoe")
p = Person.find_by_firstname("John")
ps = Person.find(:all)
p = Person.find_all_by_firstname("John")
```

5 Code Generation

The methods to access the characteristics and association of topics follow a common scheme which can be formalized in a code template. In ActiveTM these templates are evaluated at runtime. ActiveTMML can also be used to generate code or other output in any language, given templates of code to fill the domain-specific parts in. The following example shows a simplified but working implementation of the `name`-method which creates a getter for the firstname-property. The comment below shows the code which is actually send to `eval`.

```
def name(property_name, options={})
  name_type = options[:type] || property_name
  eval <<-EOD
    def #{property_name}
      @topic[\"#{name_type}\"].first.value
    end
  EOD
end
#def firstname
```

```
#  @topic["firstname"].first.value
#end
```

Additional to the generation of accessor methods, also meta information can be integrated into the definition of classes. Active Record creates a method called "columns" which introspects the schema and returns a list of database columns for this model object. The information about this columns can be used to generate so-called "scaffolds", complete CRUD users interfaces for the specific model objects. They provide the developer with a basic user interface for free. This basic interface can be used for administrative purposes as well as the basis for the interface for end users.

6 Ontology Introspection

Besides the accessor methods for the characteristics and associations and besides the introspection methods, the classes also provide a getter `topic` which returns the underlying Ruby Topic Maps topic object. Using this topic, all aspects of the TMDM can be addressed. Setting the flag `acts_as_topic` enables the methods directly in ActiveTM objects, for example the set of occurrences:

```
# standard way
p.topic.occurrences
# direct way
class Person
  acts_as_topic
end
p.occurrences
```

The another flag, called `acts_smart` enables ActiveTM objects to look into the Topic Map and find possible characteristics for properties which are not explicitly defined. Assuming that for class Person, no characteristic `shoesize` is defined yet, a *smart acting* ActiveTM object tries to find a name or an occurrence with an identifier matching "shoesize". This works through Ruby's `method_missing` which handles calls to non-existing methods. Analogously to the getter, using the not-yet-defined setter creates an occurrence[19]:

```
p.shoesize = 38 # creates occurrence,
                # type "shoesize", datatype "xsd:integer"
p.shoesize # returns 38
```

[19] Given a string, also a topic name could be created. This depends on the concrete implementation of this function.

Upon success, a set of getters and setters may be created to minimize the overhead of search names and occurrences another time.

Additionally, also the topic names of name types and occurrence types could be searched to find objects corresponding to the method name of the undefined methods.

By its nature, this kind of ontology introspection is highly experimental and may be suitable for programmatically exploring a topic map but not for productive use.

7 Conclusion and Outlook

ActiveTM augments the possibilities of Ruby Topic Maps in a productivity-enhancing way. It enables usage of domain-specific access while not constraining the generic Topic Maps API. ActiveTMML can be used to define ontologies and generate code, code snippets, and ontology documentation. The explicit definitions clearly define the resulting code and thus provide a predictable behavior independent of the data in the topic map. This enables productive usage of ActiveTMML definitions and ActiveTM classes. The intersection of the conceptual design between TMCL and ActiveTMML suggests quite a lot synergies and should be further exploited.

The introspection is rather experimental and not suitable for productive environments. Changes in the data can result in completely different code generated and render the application unusable. Still, it may be interesting to experiment with the introspection, to develop more sophisticated algorithms to look into the topic maps or interpret commonly used modeling patterns to aid writing code for productive usage.

References

[1] ISO/IEC IS 13250-2:2006: Information Technology – Document Description and Processing Languages – Topic Maps – Data Model. International Organization for Standardization, Geneva, Switzerland.
http://www.isotopicmaps.org/sam/sam-model/

[2] Bock, B.: Ruby Topic Maps, In: Maicher, L., Garshol, L.M.: Scaling Topic Maps. LNAI 4999, Springer, Berlin (2008)

[3] Ahmed, K., Garshol, L.M., Grønmo, G.O., Heuer, L., Lischke, S., Moore, G.: *Common Topic Map Application Programming Interface, 2004*
http://www.tmapi.org/

[4] Holje, E., Schmidt, J.: *PHPTMAPI*. http://phptmapi.sf.net/, 2006-11-10

[5] Heuer, L.: *Semagia Mappa - The Python Topic Maps engine*. 19 April 2008. http://code.google.com/p/mappa/

[6] Fowler, M.: *Domain Specific Language*. http://www.martinfowler.com/bliki/DomainSpecificLanguage.html, 13 February 2004

[7] Fowler, M.: *Language Workbenches: The Killer-App for Domain Specific Languages?* http://www.martinfowler.com/articles/languageWorkbench.html, 12 June 2005

[8] Bogachev, D.: *COBOL and Topic Maps?* Open Session at TMRA 2007. http://homepage.mac.com/dmitryv/TopicMaps/TMRA2007/CobolAndTMs.pdf. 2007-10

[9] Bogachev, D.: *Subject-centric programming language or what was good about COBOL*. Blogentry. http://subjectcentric.com/post/Subject-centric_programming_language_or_%what_was_good_about_COBOL. 23. October 2007

[10] Bogachev, D.: *Authoring topic maps using Ruby-based DSL: CTM, the way I like it*. http://subjectcentric.com/post/Authoring_topic_maps_using_Ruby_based_DS%L_CTM_the_way_I_like_it, 28 February 2008

[11] ISO/IEC Draft 13250-6:2007: Information Technology – Document Description and Processing Languages – Topic Maps – Compact Syntax, 2007-11-16. International Organization for Standardization, Geneva, Switzerland. http://www.isotopicmaps.org/ctm/

[12] ISO/IEC FCD 19756: *Information Technology – Document Description and Processing Languages – Topic Maps – Constraint Language*. International Organization for Standardization, Geneva, Switzerland, 2008-08-12 http://www.isotopicmaps.org/tmcl/

[13] Burbeck, S.: *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)* 1987.

[14] Kilov, H.: *From semantic to object-oriented data modeling* Bell Commun. Res., Morristown, NJ. In: *Systems Integration '90., Proceedings of the First International Conference on Systems Integration*, 1990 pages: 385–393 ISBN 0818690275

[15] Fowler, M., Rice, D.: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003. – ISBN 0321127420

[16] Moore, G., Ahmed, K., Brodie, A.: Topic Map Objects. In: *Leveraging the Semantics of Topic Maps*, 2006

[17] OMG.org: Unified Modeling Language Specification. http://www.omg.org/technology/documents/formal/uml.html