Towards a second generation Topic Maps engine

Xuân Baldauf[1] and Robert Amor[2]

[1] University of Auckland, New Zealand
xuan--tm4j2--2008--tmra.de@academia.baldauf.org

[2] Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand
trebor@cs.auckland.ac.nz

Abstract. The core of the second generation Topic Maps standards (TMDM, XTM2.0) has been finalized, yet the uptake is still slow. In this paper, we highlight engineering considerations for a novel backend for the TM4J open source topic maps engine, which is currently in development, but already usable for some purposes. As the name suggests, the "TMDM" backend is designed to reflect the TMDM specification closely. In fact, it is much closer to the TMDM than to the internal legacy TM4J data model (which is based on the XTM 1.0 data model). This motivates a bridging layer between the TMDM and the XTM 1.0 data model. We emphasize how merging is implemented in the "TMDM" backend and conclude with some synthetic merging benchmarks of the current "TMDM" backend prototype.

Keywords: TM4J, TMDM, Topic Maps engine, Merging, Instant Merging, Dynamic Merging

1 Introduction

A new generation of Topic Maps standards (the Topic Maps Data Model [ISO13250-2], XTM 2.0 [ISO13250-3]) was finalized in 2006, yet adoption in the community remains slow. TM4J[1] is an open source Topic Maps engine written in Java, mainly by Kal Ahmed. The most recent release (TM4J 0.9.7, published in 2004) is based on the older XTM 1.0 [XTM1.0] standard. While development activity on TM4J slowed after 2004, TM4J is still the most comprehensive open source Java Topic Maps Engine, and several projects build on TM4J. Thus, TM4J clearly needs an update to support the new Topic Maps standards. Updating TM4J is preferable to designing a completely new and

[1] See http://tm4j.org/

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

independent Topic Maps engine, as, in the case of an ideal update, all TM4J legacy application can build on an updated TM4J without need for modification on their part. This in turn leverages the existing TM4J applications and allows them a smooth migration path to the new Topic Maps standards.

In this paper, we show the design principles of a novel backend for TM4J, which is anticipated to lead TM4J to version 2.0. We will use the term "TM4J1" for the branch of TM4J which keeps the architecture of TM4J 0.9.7 and will, in particular, not support the TMDM. We will use the term "TM4J2" for the branch of TM4J which undergoes the major architectural changes we are describing here.

1.1 Assessment

When starting to work with TM4J1, we were in need of a topic maps engine which would be able to consume many small automatically generated XTM 2.0 files and merge them into a large XTM file. However, TM4J1 supported neither the syntax of XTM 2.0 nor the semantics of XTM 2.0 (which is specified by the TMDM). Internally, merging is only done on request, not instantly, as the TMDM mandates in section 6.1: "*Any change to a topic map* [...] *shall be followed by* [...] *merging*". Each such merging request would apply to the whole topic map, making "simulated instant merging" (by requesting such merging after every small change) infeasible with respect to performance. Furthermore, the TM4J1 API is outdated in multiple ways. First, the TM4J1 API is based on an older version of the Java language (e.g. it lacks support for Java Generics). Second, the TM4J1 API is slightly, but still significantly incompatible with the TMAPI [TMAPI1.0SP1], giving rise to a need for a wrapping-layer around TM4J1 objects to make them appear as TMAPI objects. The TMAPI itself (as of version 1.0) has not yet been updated to the TMDM, thus the names of the classes and methods in the TMAPI 1.0 do not exactly match the names of classes and properties of the TMDM.

For these and other reasons, the following desired features have been identified:

1. Internal support for the TMDM,
2. Support for XTM 2.0,
3. Instant merging,
4. Dynamic merging (where the individual components can still be identified),
5. Support for modern Java language features, such as Java Generics,

6. Let TMDM guide the naming of classes, methods and fields,[2]
7. Translation between the TM4J2 data model and the TM4J1 data model[3].

2 The TMDM backend

The novel "TMDM" backend for TM4J is designed upon the well known principle of separation of concerns. This principle guides

1. that storage of Topic Maps (in RAM) should be separated from a merged view of Topic Maps,
2. that a TM4J1 data model view should be separated from a TM4J2 data model view,
3. that interfaces should be separated from implementations,
4. that interfaces themselves should be separated by concerns,
5. that event handling should be separated.

This is why there is not only one set of classes (or interfaces), but five:

1. The interfaces for TMDM data with read-write access.
2. The interfaces for TMDM data with read-only access.
3. The classes to store TMDM data.
4. The classes to view TMDM data in merged form.
5. The classes to access TMDM data through the TM4J1 data model.

Additionally, a new handling system to efficiently communicate events between the different layers of objects has been devised.

In the following sections, these layers and subsystems are described. Figure 1 (below) provides an overview over all these layers.

2.1 TMDM interfaces layer (read-write access)

This layer contains interfaces for representing TMDM objects, all within the package `org.tm4j.topicmap.tmdm`:

---

[2] If the TMDM guides the naming of classes, methods and fields for the "TMDM" backend as well as for the upcoming TMAPI 2 standard, then the "TMDM" backend may be automatically compatible with the upcoming TMAPI 2 standard, making a separate wrapping layer (as in TM4J1) unnecessary.

[3] When supporting the TMDM but, at the same time, serving as a backend for TM4J1 applications, there is a need for translating between the TM4J1 data model (which is the data model of XTM 1.0) and the TM4J2 data model (which is the TMDM).

1. `TopicMap` `extends Reifiable`
2. `Topic` `extends TopicMapConstruct`
3. `TopicName` `extends Scopeable`
4. `Variant` `extends Scopeable`
5. `Occurrence` `extends Scopeable`
6. `Association` `extends Scopeable`
7. `AssociationRole` `extends Reifiable`
8. `Scopeable` `extends Reifiable`
9. `Scope`
10. `Reifiable` `extends TopicMapConstruct`
11. `TopicMapConstruct`

Each of these interfaces contains methods to read and write properties of the TMDM item type they represent. For example, the `Topic` interface contains, among others, the following declarations:

```
public boolean  addSubjectIdentifier(Locator subjectIdentifier);
public boolean  removeSubjectIdentifier(Locator subjectIdentifier);
public Set<Locator> getSubjectIdentifiers();
```

As another example, the `TopicName` interface contains, among others, the following declarations:

```
public void         setType(Topic type);
public Topic        getType();
public void         setValue(String value);
public String       getValue();
```

Scopeable and Scope. The interface hierarchy here differs from the TMDM class hierarchy in that the interfaces `Scopeable` and `Scope` are introduced. While in the TMDM specification, scope is defined verbally ("*All statements have a scope.*"), a reflection of this definition is lacking in the original TMDM class hierarchy: scope is left to remain an arbitrary set of topics in each statement without a unique identity. This is changed in TM4J2. The rationale behind this is the reasonable assumption that the set of distinct scopes in a typical topic map is much smaller than the set of scopeables (that is, the set of statements). If this assumption is true, then instead of storing a mutable set of topics for each `Scopeable` (which typically consumes at least a Java array object header and pointers to each of the topic objects), it is more memory-efficient to just store a mutable pointer to an immutable `Scope` object. It is also assumed that, at query time, this compression increases cache-locality, as the number of distinct scope objects (`Scope` objects vs. sets of topics) to be traversed is much smaller. Furthermore, in case two topics of the same scope merge, changing the affected

`Scope` object[4] is much cheaper than changing, or even just keeping track of, all affected `Scopeable` objects. However, as TM4J2 is not fully implemented yet, and also because there is, to date, no well-agreed Topic Maps benchmark suite (consisting of demo topic-maps in various serialization formats and demo queries in the yet to be finalized Topic Maps Query Language), all these considerations are merely theoretical and are still in need of performance evaluation.

2.2 TMDM interfaces layer (read-only access)

This layer contains interfaces for representing TMDM objects which are only to be read, but not to be written, all within the package `org.tm4j.topicmap.tmdm`:

1. `ReadableTopicMap`
2. `ReadableTopic`
3. `ReadableTopicName`
4. `ReadableVariant`
5. `ReadableOccurrence`
6. `ReadableAssociation`
7. `ReadableAssociationRole`
8. `ReadableScopeable`
9. `ReadableScope`
10. `ReadableReifiable`
11. `ReadableTopicMapConstruct`

Each of these interfaces contains methods to just read properties of the TMDM item type they represent. They are stripped-down versions of their read-write counterparts. For example, the `ReadableTopic` interface contains, among others, the following declarations:

```
public Set<Locator> getSubjectIdentifiers();
```

As another example, the `ReadableTopicName` interface contains, among others, the following declarations:

```
public ReadableTopic getType();
public String        getValue();
```

[4] Giving up immutability of `Scope` objects leaves opportunity for two `Scope` objects being equal. While avoiding this repetition is the very reason to have `Scope` objects in the first place, actually having such repetition just in rare cases has only a tiny effect on the ratio between actual memory savings and possible memory savings by this method.

The rationale for having a layer of TMDM interfaces which just allow read-only access, separate from TMDM interfaces which allow read-write access, is the case of Virtual Topic Maps. A Virtual Topic Map[5] is a view[6] on something which looks like a topic map, but may not actually be a (modifiable) topic map itself. As one objective of topic maps is to be able to represent the structure of almost any type of information, it is only consequential to reformulate about almost any information source[7] as a topic map. However, changing such a topic map is often (unless it is materialized) not possible directly; however, changing the information source, and having this change reflected in the topic map view, is possible. If the translation between the information source and the topic map view is a one-way-process (i.e. only from the source to the topic map view and not the other way around) for theoretical or practical reasons, then there is no sensible way of implementing the setter methods which modify topic maps. If, on the caller side, only getter methods are needed (for example, if a GUI view or another topic maps view is built on the topic maps view), then the more adequate interface between these two sides is the set of TMDM interfaces which just allow read-only access.

Each read-write TMDM interface extends the corresponding read-only TMDM interface. Note that e.g. the return type of `ReadableTopicName.getType()` is not `Topic` but `ReadableTopic`, while the return type of `TopicName.getType()` is `Topic`. The reason is that the read-only TMDM interfaces have to be closed within themselves, i.e. they should not point into the world of read-write TMDM interfaces. Note also that narrowing the return type when overriding (from `ReadableTopic` to `Topic`) is a feature of Java 1.5, thus unavailable at the times the original TM4J1 architecture was designed.

---

5 It is unclear to whom to trace the term "Virtual Topic Maps". However, the earliest instance of explaining this term, which we could find, is following mailing-list post of Steve Pepper:
http://www.infoloom.com/pipermail/topicmapmail/2001q3/003190.html

6 A view is something which depends on, and its contents are defined by, what is viewed.

7 "Any information source" does not preclude topic maps themselves as information sources. For example, as Robert Barta points out in his talks about TMQL, it may be perfectly reasonable that a topic map is an information source for an inference engine which takes that topic map as input, infers new facts from existing facts, and exports a topic map view as output. Note that the topic map constructs of the output may be generated on demand, i.e. only when a query is active. This way, the memory requirements for such an inference engine can be much smaller than the memory requirements if the exported topic map view was materialized.

2.3 TMDM Basic implementation layer (read-write access)

This layer contains classes for representing TMDM objects, all within the package `org.tm4j.topicmap.tmdm.basic`:

1. `BasicTopicMap`
2. `BasicTopic`
3. `BasicTopicName`
4. `BasicVariant`
5. `BasicOccurrence`
6. `BasicAssociation`
7. `BasicAssociationRole`
8. `BasicScopeable` (abstract class)
9. `BasicScope`
10. `BasicReifiable` (abstract class)
11. `BasicTopicMapConstruct` (abstract class)

Each of these classes implements the appropriate read-write TMDM interface.

In a Model-View-Controller design, this layer contains the model. That means that all actions to modify a topic map are actions on objects in the Basic layer, the objects in the Basic layer act as mere storage. Thus, questions about whether two `BasicTopicMapConstruct`s are to be merged, or not, are not answered here. For example, even if two `BasicTopic` objects are to be merged, it is not possible to query the merged set of the merged topic's `BasicTopicName`s (directly) if only a reference to one of these `BasicTopic` objects is available. Effectively, the Basic layer represents topic maps as if the merging rules did not exist. However, actions on `BasicTopicMapConstruct`s induce events, which are typically forwarded to the Merged layer.

2.4 TMDM Merged implementation layer (read-only access)

This layer contains classes for representing TMDM objects, all within the package `org.tm4j.topicmap.tmdm.merged`:

1. `MergedTopicMap`
2. `MergedTopic`
3. `MergedTopicName`
4. `MergedVariant` (currently not implemented)
5. `MergedOccurrence`

6. `MergedAssociation`
7. `MergedAssociationRole`
8. `MergedScopeable` (abstract class)
9. `MergedScope`
10. `MergedReifiable` (abstract class)
11. `MergedTopicMapConstruct` (abstract class)

Each of these classes implements the appropriate read-only TMDM interface.

In a Model-View-Controller design, this layer contains an internal view on (a set of) other topic maps, each allowed to ignore the merging rules. Each time a viewed topic map (e.g. a `BasicTopicMap`) changes in some aspect, an event is fired and the merged topic map is updated accordingly.[8]

During the update, the merged topic map itself may fire events to its downstream event listener. For example, it may fire an event stating that two formerly separate `MergedTopicMapConstructs` have now been merged. An application may use these notifications to update its user interface accordingly.

The Merged layer is only a view. Consequently, it does not need to modify its upstream `TopicMapConstructs`. Thus, it only needs to operate on a read-only version of a topic map, and consequently it requires the objects it is operating on only to implement the read-only TMDM interfaces layer, not necessarily the read-write TMDM interfaces layer. As the Merged layer is a view, it also only implements the read-only TMDM interfaces layer itself.

Representation. Each `MergedTopicMapConstruct` is internally represented as a list of the individual upstream `ReadableTopicMapConstructs` (this list is called `components`), together with the reference to the `MergedTopicMapView` (see below) and the key (see below) of the `MergedTopicMapConstruct`.

Merging topics. Most of the supplementing indexing information for a particular `MergedTopicMap` is stored in a `MergedTopicMapView` object, which is attached to every `MergedTopicMapConstruct` of that `MergedTopicMap`. One of the indexes is `itemIdentifierOrSubjectIdentifierToMergedTopicMapConstruct`, containing a mapping from `Locators` to `MergedTopicMapConstructs`. Each time an upstream `ReadableTopicMapConstruct` receives an additional item identifier and, similarly, each time an upstream `ReadableTopic` receives an additional subject identifier, the corresponding `MergedTopicMapConstruct` is registered in this index under the additional identifier. If, for this additional identifier, there already exists an entry, then merging is triggered. Equality of subject locators is

[8] For example, consider that a new upstream `ReadableTopic` is created. Then, an event is fired to the downstream `MergedTopicMap`. Then, a new `MergedTopic` is created.

handled in the same way. Currently, merging of topics due to equality in the "`reified`" property is not implemented.

Merging statements. For each statement, there is a key object which represents that statement's equivalence class as defined by the TMDM. If two key objects are equal in each field, then these key objects themselves are equal. The choice of the fields of the key classes is guided by the TMDM's equality rules. For example, the data structure for the key for a `MergedOccurrence` is defined as follows:

```
public class MergedOccurrenceKey extends MergedScopeableKey {
  protected MergedTopic parent;
  protected MergedTopic type;
  protected Locator     datatype;
  protected String      value;
}
```

Whenever a statement is created or modified, an appropriate key object is entered in an appropriate index within the `MergedTopicMapView` object. If there is already an existing key object in the index which equals the new key object, then the statements of both keys are equal, and merging is triggered.

Dependent merging. If a `MergedTopic` is merged, then all the objects which are referencing this topic have to be updated. Thus, each `MergedTopic` maintains inverted indices about themself, that is, sets of `MergedTopicMapConstructs` which, for some property, point to that `MergedTopic`; each set for one particular property. In case of merging, these sets are traversed and the values for that property for the dependent `MergedTopicMapConstructs` are updated accordingly. (This also means that their keys are changed to reflect the new value for that property, which in turn can lead to more merging.)

In the current implementation, these sets are not complete: They are only implemented for the properties `Association.type`, `AssociationRole.type`, `AssociationRole.player`, `TopicName.parent`, `Occurrence.parent`. Thus, such sets are missing for example for `TopicName.type`, `Occurrence.type` as well as for scope. Note that it is reasonable to assume that most of these sets are empty for most topics, as most topics are never used as an association type, association role type, occurrence type or topic name type. Thus, it should be more memory efficient to replace these sets, currently 4 (and later 7) per `MergedTopic`, either by appropriate indices in the `MergedTopicMapView` object or by a unified full inverted index (that is, exactly one set per `MergedTopic`, where each entry is a pair of a particular `MergedTopicMapConstruct` and the property within that particular `MergedTopicMapConstruct` which points to that `MergedTopic`). Implementing and evaluating this is left for future work.

Merging complexity. Merging two `MergedTopic`s into one is quite similar to the union-find class of algorithms (employed for example in some implementations of Kruskal's algorithm): in both cases, two connected components are to be merged into one. The choice of what to merge with what may have a remarkable effect on the performance. Consider a list of n `MergedTopic`s, each initially representing only 1 `BasicTopic`. Consider that, for some reason (e.g. adding subject indicators), all topics are being merged with each other, one after another, such that each time, the last two topics are merged. What if, at each step, both `MergedTopic` objects are deleted and a new `MergedTopic` representing the two is created instead[9]? Then both lists of individual upstream `BasicTopic`s of both old `MergedTopic`s have to be copied into a unified list of the new `MergedTopic`, yielding $O(n^2)$ copy operations. What if one `MergedTopic` object is reused and the other `MergedTopic` object is merged into it? If, at each step, the last topic is merged into the second but last topic, then still the number of copy operations is in $O(n^2)$. However, if at each step, the second but last topic is merged into the last topic, the number of copy operations is in $O(n)$. Thus, choosing the order of what to merge into what is important.

The weighted-union heuristic [Galler1964][Hopcroft1971][Fischer1972] teaches to always merge the smaller `MergedTopic` (the smaller connected component) into the larger one. Then, the number of copy operations is in $O(n \cdot \log(n))$, regardless of the initial number of `BasicTopic`s in each `MergedTopic`. The proof is similarly straightforward: There are at most n initial `BasicTopic`s, and each `BasicTopic` undergoes only about $\log_2(n)$ copy operations. Let $c(m)$ be the number of `BasicTopic`s that a `MergedTopic` m contains. Suppose a `BasicTopic` b, directly before undergoing a copy operation, belongs to a `MergedTopic` m_0, which is going to be merged with `MergedTopic` m_1. This results in a new `MergedTopic` m_2. Then, the equation $c(m_2) \geq 2 \cdot c(m_0)$ holds. The reason is that $c(m_2) = c(m_0) + c(m_1)$ and $c(m_1) \geq c(m_0)$. (If this was not the case, then the `BasicTopic`s of m_0 would not be copied, but the `BasicTopic`s of m_1 would be copied instead, which contradicts the assumption.) Thus, after each copy-operation of a `BasicTopic`, the size of the `MergedTopic`, which the `BasicTopic` is member of, has at least doubled. After k such steps, the `BasicTopic` belongs to a `MergedTopic` which has at least 2^k `BasicTopic`s. Let $k_0 = min(\{k \mid 2^k \geq n\})$. After k_0 steps (possibly earlier), the `BasicTopic` belongs to a `MergedTopic` which has at least n `BasicTopic`s. At this stage, no further merging is possible (because there is only one `MergedTopic` left, which contains each of the n `BasicTopic`s). Thus, after about $\log_2(n)$ copy operations ($k_0 \leq ceil(\log_2(n))$) for each `BasicTopic`, the merging process is finished. ∎

[9] as suggested by the TMDM

The merging complexity considerations for other `MergedTopicMapConstructs` are similar.

Note that the conceptually simpler implementation may not always be the faster implementation. When merging `MergedTopic` m_0 with `MergedTopic` m_1 into a new `MergedTopic` m_2, then, conceptually, m_0 and m_1 have to be removed from the indices and m_2 has to be inserted into the indices (see [ISO13250-2], section 6.2). However, now that we know that merging *into* an existing `MergedTopic` m_1 is faster, we also know that the address of m_2 equals to the address of m_1 (although the state at the address changes from m_1 to m_2). Thus, we do not need to remove (the address of) m_1 from the indices, because all pointers to the state m_1 later point to the state m_2. We just have to remove everything pointing to the address of m_0 from the indices and insert new index entries such that they now point to the address of m_2. If removing and inserting can be combined into one update operation, this is even better. The TMDM backend was initially implemented without that optimization, but is now available with this optimization – with tremendous speedups (see section 3). The reason for this speedup is the change of the complexity class: Consider the merging of n topic maps into a big topic map, one at a time, and consider a subject which is represented in every such topic map (e.g. by a topic which serves as a type topic for some common type of association or occurrence). Then, the corresponding `MergedTopic` has many `BasicTopics` as components. Each time a component is added to the `MergedTopic`, without the merging optimization, all components are considered (e.g. to list all subject identifiers) when unindexing and reindexing. Thus, the number of consideration operations is in $O(n^2)$. With the merging optimization, only the newest `BasicTopic` is considered at a time, thus the number of considerations is in $O(n)$.

Unmerging complexity. The design of the Merged layer, as a view to process whichever input it is confronted with, allows not only for merging, but also for unmerging. Consider that a property of a `BasicReifiable` is changed. In this case, the corresponding `MergedReifiableKey` changes, the `BasicReifiable` may be removed from the set of components of one `MergedReifiable` and it may be added to the set of components of another `MergedReifiable`.

However, this simple response to change may be more complicated for topics. Consider a bipartite graph, where `BasicTopics` and locators are vertices and where there is an edge between a locator and a `BasicTopic` if, and only if, the locator is a subject identifier of the `BasicTopic`. Then, for each connected component in the graph, all the `BasicTopics` which are member of that component should be merged. Now, consider that an application removes such an edge of the connected component (for example by executing something like

`basicTopic.removeSubjectIdentifier(someSubjectIdentifier)`). Then, the `MergedTopic` of that connected component should split iff the connected component splits. However, there is no straightforward way to determine (locally) whether a removal of just one edge makes a connected component split. For example, if the connected component is a large cycle, removing one edge does not make the connected component split. However, if the connected component is nearly a large cycle with just one remote segment missing, removing one edge makes the connected component split.

Thus, until a more thorough analysis of this problem is performed, the current implementation for splitting is to reduce splitting to merging by *atomicizing* the connected component. That is, all edges are removed and then all edges, except the one to be initially removed, are re-added, eventually resulting in either one connected component, or two. As this is an $O(m+n\cdot\log(n))$ operation (where m is the number of edges and n is the number of vertices of the connected component), this operation is quite expensive. It remains a question of future research whether this operation needs more optimization (like an aggregated "remove all locators at once" operation), as removing locators may not be a very common operation.

2.5 TM4J1 compatibility layer

This layer contains interfaces for wrapping TMDM objects into TM4J1 objects, all within the package `org.tm4j.topicmap.tm4j1`:

1. `TopicMapImpl`
2. `TopicImpl`
3. `BaseNameImpl`
4. `VariantImpl` (currently not implemented)
5. `OccurrenceImpl`
6. `AssociationImpl`
7. `MemberImpl`
8. `ScopedObjectImpl` (abstract class)
9. `TopicMapObjectImpl` (abstract class)

At the core of the wrapping layer, there is a `BasicTopicMap` together with a `MergedTopicMap` within each `TopicMapImpl`. All read accesses which should also take into account merged topics (which is the default) are forwarded to objects of the Merged layer, while all write accesses are forwarded to the objects to the Basic layer.

All necessarily translations between the different models (TMDM vs. TM4J1 data model) are done on the fly, on a best effort basis. In particular, multiple players per role (allowed in TM4J1) are not supported, nesting of variants (allowed in TM4J1 due to a misunderstanding of XTM 1.0) will not be supported (variants are not yet implemented), topic name types and occurrence data types (allowed in the TMDM) are inaccessible via the TM4J1 layer.

Instances of the compatibility layer (e.g. instances of `TopicImpl`) are created on demand. This may result in two different `TopicImpl` objects representing the same `BasicTopic`. But because they also represent the same `MergedTopic` and topic maps explicitly are designed for merging, this did not have an apparent negative effect so far.

2.6 TopicMapEventListener

The event handling model has been changed radically. In TM4J1, a JavaBeans `PropertyChangeListener` or a JavaBeans `VetoableChangeListener` was registered against a particular property of interest of a particular object of interest. The property of interest was indicated by supplying the name of the property as a string.

This has multiple disadvantages. First, using strings, instead of compiler-checked literals (e.g. Java enums), is error prone, as accidental misspellings do not result in compile errors, but are silently accepted. Second, using strings leads to string comparison at runtime, which is slow compared to mere pointer dereferencing which would be employed by the runtime environment if compiler-supported language constructs would be used. Third, the list of event listeners (or even just the pointer to this list) in every single `TopicMapObject` contributes to the memory footprint of these `TopicMapObject`s. Fourth, an event handling model based on `PropertyChangeListener` requires the event source to provide an old and a new value of the property to the listener. In case the property is an array and the event is to add an element to the array, this means that an array representing the old list of elements and an array representing the new list of elements have to be provided to the listener, at the same time. This does not only mean that the listener has to do a side-by-side comparison of both supplied arrays (which is at least an $O(n)$ operation) in order to find out which element has been added. This also means that just preparing for sending an event is not an $O(1)$ operation anymore, but an $O(n)$ operation, involving creating a copy of an array.

For example, the relevant code to add an association role to an association in TM4J1 looks like (“association role” is called “member” in the TM4J1 data model):

```
Collection oldMembers = (m_members == null)? Collections.EMPTY_LIST
                                           : m_members;

// Fire vetoable change notification
Collection newMembers = new ArrayList(oldMembers);

newMembers.add(member);
fireVetoableChange("members",
    Collections.unmodifiableCollection(oldMembers),
    Collections.unmodifiableCollection(newMembers));

m_members = newMembers;
((MemberImpl) member).setParent(this);

firePropertyChange("members", oldMembers, m_members);
```

whereas the relevant code to add an association role to an association in TM4J2 looks like:

```
roles.add(role);
getEventListener().notifyAssociationRoleCreated(
                        getContainingTopicMap(),this,role);
```

It is clear that the latter code is not only shorter, but reasonably expected to be faster, too.

Thus, the event handling model of TM4J2 has been redesigned radically compared to TM4J1.

First, the new event handling model does not use string constants. It also does not use enum constants. Instead, it models every event as an action, that is, in the Java language, a method call. Using method calls instead of event objects has the advantages that there is no need to create, write to, read from, or delete event objects. Instead, all of these actions happen implicitly on the stack. Another advantage is that there is no need to define different event classes for different types of events, which would have led to an event class zoo. A further advantage of the implicitness of method calls is the good support for optimization by current Java virtual machines: If it can be determined (by the JavaVM) that the only possible implementation of a method call is the empty implementation, then the method call itself, along with all preparations to calculate the arguments of the method, can be optimized away.

Second, there is only one, and exactly one event listener per topic map, and there is no event listener for any of the other `TopicMapConstructs`. First, this saves main memory in all but one objects of each topic map. Second, by settling for exactly one event listener (and not for zero or one, or, respectively, a variable number of event listeners), an explicit check for `null` pointers, or, respectively, a

`for`-loop, is avoided, making the code more readable and less complex and thus more amenable to automatic optimization. Third, the default event listener is the no-operation event listener. If the JavaVM detects the no-operation-nature of the event listener (and chances are that current JavaVMs do so), then, as mentioned above, event handling is a no-operation also on the caller side, and thus with zero cost with respect to CPU cycles.

This does not mean that multiple event listeners are not supported. The concern for multiple listeners has just been separated from the concern of calling "the event listeners": If multiple event listeners are desired, it is easy to create a multiplexer event listener which forwards each event it receives itself to multiple event listeners.

2.7 Architectural overview

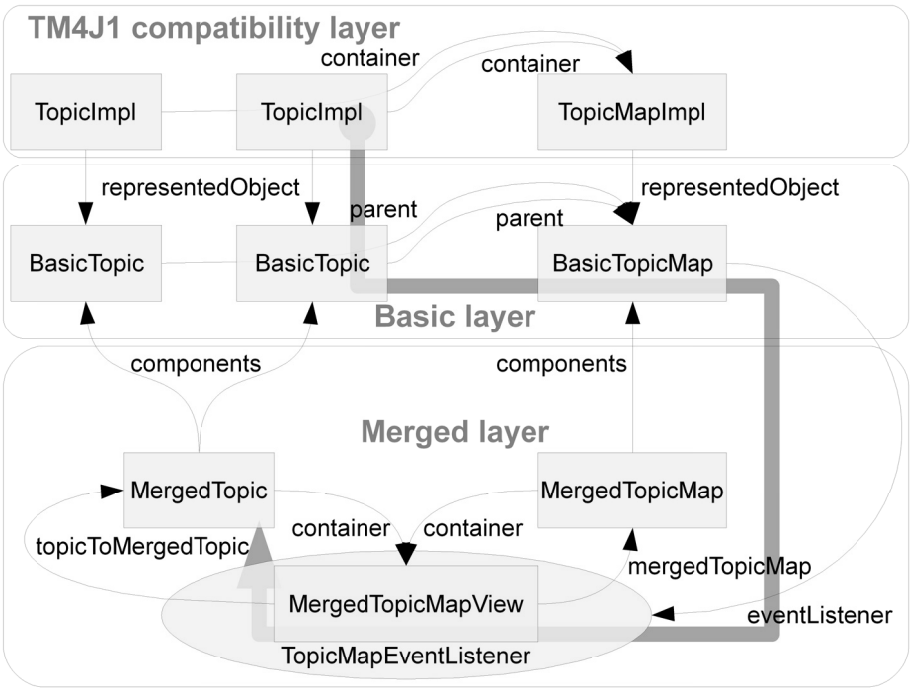

Fig. 1. Architectural overview

Putting everything together, Figure 1 shows a simplified example object graph and the three layer. The path throughout the graph shows an example event handling path.

2.8 XTM 2.0 reading support

XTM 2.0 reading support has been implemented partially, currently as what is called colloquially as a "hack". That is, element names of XTM 2.0 are handled in the same way as the corresponding element names of XTM 1.0 are handled. Features of XTM 2.0 which are not available in XTM 1.0 are thus silently ignored, even if the final backend supports the TMDM, because up to now, there are no "feature" holes punched into the curtain of the TM4J1 API, even if both sides of the curtain support more modern topic maps processing. A proper XTM 2.0 reading and writing support, building directly on TMDM backends, is part of the future work.

3 Evaluation

The contribution to the TM4J project has a size of more than 9000 lines of code, and it is available in the current CVS tree of TM4J, open for public review. We have benchmarked the current "TMDM" backend prototype with merging optimization and the current "TMDM" backend prototype without merging optimization against the old "memory" backend on a testing machine, containing 8 Intel Xeon E5335 cores and 16GiB of RAM, running Linux 2.6.25 in 64-bit mode. We take a set of between 1 and 1024 small XTM 2.0 files (on average 111 topic map constructs (among them about 24.6 topics and 22.5 binary associations) per file) generated by yet-to-be-published software out of the DBLP[10] dataset, and let all these backends merge these files into one merged XTM. Some topics are present in all files, most topics are only present in one or two files. Most associations are present in one or two files. Most topics have 2 or 3 subject indicators. We measure the processing time as well as the maximum used memory (by using statistical output of the garbage collector). We use the "`Java HotSpot(TM) 64-Bit Server VM (build 10.0-b22, mixed mode)`", the Java command line options "`-Xmx8G -da`" for processing time tests and the additional command line options "`-verbose:gc -XX:MinHeapFreeRatio=2 -XX:MaxHeapFreeRatio=4 -XX:MaxNewSize=2048k` " for RAM tests (to enable frequent garbage collector statistical output). All tests have been performed with pre-warmed caches to minimize influences of e.g. disk latency.

[10] DBLP is one of Computer Science's large bibliographic databases. One XTM 2.0 file corresponds to the author-paper relationship shown for one particular author, for example http://dblp.uni-trier.de/db/indices/a-tree/k/Knuth:Donald_E=.html. (Thanks to Michael Ley for providing access to the software which generates all DBLP author's pages.)

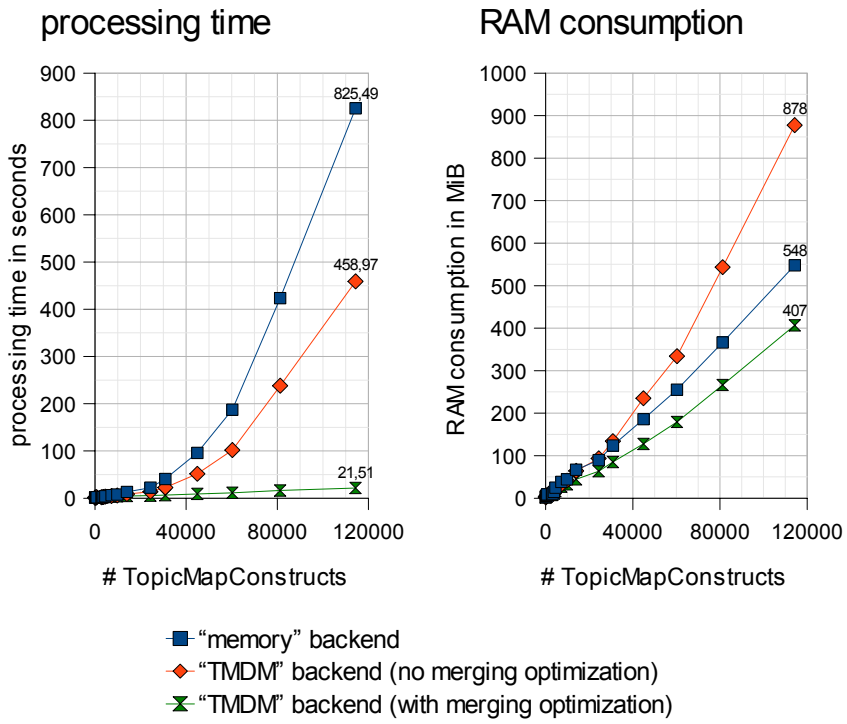

Fig. 2. Synthetic benchmark of merging over different input sizes

While the unoptimized "TMDM" backend prototype already outperforms the "memory" backend, it is evident that the merging optimization has a crucial impact, providing an about 20-fold increase in merging performance. The parabola shape of the processing time graph of the "TMDM" backend (without merging optimization) is well in line with the theoretic considerations that, without the merging optimizations, the merging complexity for n topics to be merged into one is in $O(n^2)$. The shape of the respective graph of the "memory" backend suggests that this backend, too, could profit from the merge optimization.

The "TMDM" backend prototype with merging optimization consumes less memory than the "memory" backend. However, this may also partly be because the "TMDM" backend prototype does not yet implement all desired features (e.g. support for variants, and more indices).

Interestingly, the memory footprints of both the "TMDM" backend prototype variants differ considerably, which should not happen for a memory-neutral

optimization. This could be an indicator of a memory leak in the "TMDM" backend prototype, and needs further investigation.

In general, the memory usage for an implementation employing standard Java techniques is still disappointing: about 3730 bytes per TopicMapConstruct. For a comparison, the corresponding XTM 2.0 input files consume just about 163 bytes per TopicMapConstruct. A quick analysis using the "`jmap`" tool[11] revealed that not only a big part of the memory consumption is due to storing characters in UTF-16 format (each character consuming 16 bits), but also that an even bigger part of memory consumption is due to instances of `java.util.HashMap$Entry` and arrays thereof. Thus, not only changing the internal string encoding to more space efficient encodings (like UTF-8) or employing string compression techniques (as most locators happen to have common prefixes), but also changing the implementation of `java.util.Map` from `java.util.HashMap` to more space-efficient ones as well as replacing small maps (e.g. those with only one or 2 entries) by specialized, compact data structures, seem to be promising improvements.

4 Future Work

BOCK raised in [Bock2008] the issue of using a Domain Specific Language to represent the TMDM, and of using interpreters of this language to generate each of the sets of classes or interfaces of the TMDM backend of TM4J2. This is a very interesting approach, as it not only helps to avoid coding errors, but also helps to change deep architectural decisions on a whim. For example, we expect that instead of implementing locators as `Locator` objects, but as UTF-8-encoded `byte[]` strings, the resulting topic map engine would both have a smaller memory footprint and be faster. (At the same time, the source code would lose object oriented elegance, which, however, is acceptable if the source code is automatically generated.) By tuning different architectural decisions, one can create instance specific Topic Maps engines, i.e. Topic Maps engines which are suited for a very specific type of topic map[12], to the point where finding the fastest Topic Maps engine for a particular topic map is a classical optimization problem. To make this possible in the first place, as many parts of TM4J2 as possible have to be reformulated in terms of a DSL. (BOCK has already completed the formulation of the TMDM and the automatic generation of the read-only TMDM interfaces, the read-write TMDM interfaces, the Basic layer and the event listener interface.)

[11] See http://java.sun.com/javase/6/docs/technotes/tools/share/jmap.html

[12] For example, a topic map containing Chinese text data would suffer from UTF-8 encoding.

Some statistical assumptions about "typical topic maps" need to be verified empirically. Among them are the following:

1. The set of distinct scopes is much smaller than the set of scopeables.
2. The set of topics used as association type is small.
3. The set of topics used as association role type is small.
4. The set of topics used as topic name type is small.
5. The set of topics used as occurrence type is small.

MAICHER suggested (in private conversation at the TMRA2007) that late, on demand merging in federated Topic Map databases may be a good thing, because it allows individual member Topic Maps to be virtual. This idea has merit, as those Topic Maps implementations also do not need to support an update notification (event generation mechanism). It also avoids the unmerging performance problem, because on demand merging does not require unmerging at all. Furthermore, supporting late merging is actually a quite natural way for supporting federated Topic Map databases in the first place. Thus, a separate `org.tm4j.topicmap.tmdm.merged.`*`ondemand`* view is part of future work.

Apart from dynamic early merging and dynamic on-demand (late) merging, support for static merging (that is, directly between `BasicTopicMapConstructs`) may be implemented. However, if the dynamic early merging layer is efficient enough, there may not be much incentive to implement static merging functionality.

The backend shown here is for RAM-only storage.[13] Exploring the extension of the backend to disk-based storage (e.g. using JDO) has been started by the authors, but is not part of this paper.

Acknowledgements

Thanks go to the Topic Maps Lab[14] at the University of Leipzig, Germany, for kindly providing access to the testing machine.

[13] Due to the gap between RAM latency and disk latency widening (and RAM having a lower price per (random) access than disk) and due to the price of RAM declining, we believe that distributed RAM-only Topic Maps engines will play an important role in the future.

[14] See http://www.topicmapslab.de/

References

[ISO13250-2]: International Organization for Standardization/International Electrotechnical Commission — Joint Technical Committee 1 — Subcommittee 34 — Working Group 3: "ISO/IEC IS 13250-2:2006: Information Technology — Document Description and Processing Languages — Topic Maps — Data Model" International Organization for Standardization, Geneva, Switzerland (August 2006) http://www.isotopicmaps.org/sam/sam-model/

[ISO13250-3]: International Organization for Standardization/International Electrotechnical Commission — Joint Technical Committee 1 — Subcommittee 34 — Working Group 3: "ISO/IEC IS 13250-3:2007: Information Technology — Document Description and Processing Languages — Topic Maps — XML Syntax" International Organization for Standardization, Geneva, Switzerland (August 2006) http://www.isotopicmaps.org/sam/sam-model/

[XTM1.0]: TopicMaps.Org Authoring Group: "XML Topic Maps (XTM) 1.0" International Organization for Standardization, Geneva, Switzerland (August 2001) http://www.topicmaps.org/xtm/1.0/

[TMAPI1.0SP1]: Kal Ahmed, Lars Marius Garshol, Geir Ove Grønmo, Stefan Lischke, Lars Heuer, Graham Moore: "TMAPI — Common Topic Map Application Programming Interface — 1.0 SP1" (February 2005) http://www.tmapi.org/

[Galler1964]: Bernard A. Galler, Michael J. Fisher: "An improvide equivalence algorithm" in *Communications of the ACM*, volume 7, issue #5, pages 301..304 (May 1964) http://portal.acm.org/citation.cfm?doid=364099.364331

[Hopcroft1971]: John E. Hopcroft, Jeffrey D. Ullman: "A linear list merging algorithm", Technical Report number CS-CSD-71-111, Cornell University, Ithaca, New York, USA (November 1971) http://ecommons.library.cornell.edu/bitstream/1813/10810/2/TR71-111.pdf

[Fischer1972]: Michael J. Fischer: "Efficiency of equivalence Algorithms", Artificial Intelligence Memo number 256, A. I. Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA (April 1972) http://dspace.mit.edu/bitstream/1721.1/6201/2/AIM-256.pdf

[Bock2008]: Benjamin Bock: "Topic Maps Middleware". Master's thesis, University of Leipzig, Germany (May 2008) http://academia.bock.be/publications/Bock2008_Topic-Maps-Middleware.pdf