Efficient Model Repository for Web Applications

Sergejs Kozlovičs(✉)

Institute of Mathematics and Computer Science,
University of Latvia, Raina blvd. 29, Riga 1459, Latvia
sergejs.kozlovics@lumii.lv

Abstract. Many model-based applications have been developed with standalone usage in mind. When migrating such applications to the web, we have to think about multiple users competing for limited server resources. In addition, we encounter the need to synchronize models via the network for client-side access. Thus, there is the risk that the model storage could become a bottleneck.

We propose a model repository that deals with these issues by using an efficient encoding of the model that resembles its Kolmogorov complexity. The encoding is suitable for direct sending over the network (with almost no overhead); it can also be used "as-is" in memory-mapped files, thus, utilizing the OS paging mechanism. By adding just 3 automatic indices, all traverse and query operations can be implemented efficiently. Our tests show that the proposed model repository outperforms other repositories concerning both CPU and memory and is able to hold 10,000 and more instances at the same time on a single server.

Keywords: Models · Model repository · Web applications

1 Introduction

In 2018, the growth in internet users has reached 4 billion people constituting over half of the world's population [11]. The wide availability of the internet combined with the development of cloud technologies made it much easier and cheaper to deploy web applications than it was 20 years ago. The obvious benefit of web applications is that they are available right away, without the need to install them. In addition, they are always up-to-date and accessible throughout the globe from different devices and operating systems. That is why many standalone applications are now being moved to the web environment. However, developers of web applications face additional issues such as the presence of network overhead and competition between multiple connected users for limited server resources.

The scope of this paper lies within classical model-based applications that have to be moved to the web. By *model-based application* we mean an application

© Springer Nature Switzerland AG 2018
A. Lupeikiene et al. (Eds.): DB&IS 2018, CCIS 838, pp. 216–230, 2018.
https://doi.org/10.1007/978-3-319-97571-9_18

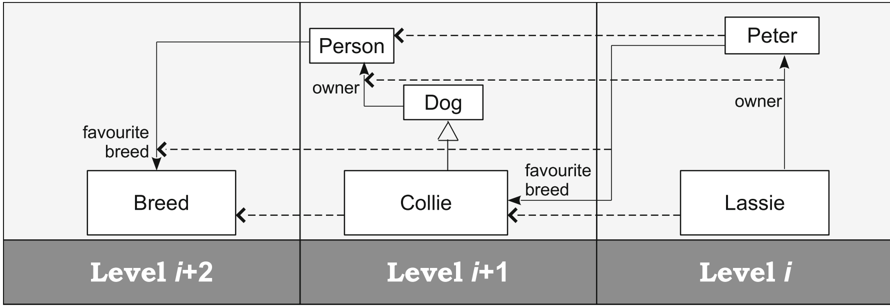

Fig. 1. An example of multiple mixed meta-levels (dashed lines represent the "instance-of" relation). The relation "favourite breed" between Person and Breed as well as the link between Peter and Collie cross two adjacent meta-levels.

that stores data in MOF[1]-like models and processes these data by corresponding model transformations [15,17]. We say "MOF-like" models, since in practice alternative implementations such as Java-based EMF/ECore are used [1,20]. By *model transformations* we mean not only specific programs written in some model transformation language (like MOLA, Lx, Epsilon, ATL, VIATRA, etc.), but also programs written in traditional programming languages (like Java or C++) and that are able to access MOF-like models via some API (e.g., ECore API) [7–9,13,21].

In model-based applications models are saved in a storage that we call *model repository*. This concept differs from the database concept in the following main points:

– the repository does not need to be able to perform complex queries- that is the task of model transformations; the repository just has to implement simple model traversal and update operations;
– in a repository, the model is usually loaded into memory, since transformations use the model intensively for both read and write operations; that differs from databases (not only SQL, but also graph and document databases), which are optimized for performing queries, while update operations are much slower (usually involving re-arranging indices)[2];
– model repository internal structures and APIs are tailored for storing models, i.e., both object-level data (objects, their attributes, and links) as well as meta-data (object types and their properties) can be stored. In some use cases multiple meta-levels are required (see Fig. 1), thus, model repository should be able to store them all. Although existing SQL and no-SQL databases can be tamed for these purposes (e.g., via object-relational mapping, ORM), such approach is more comprehensive and less efficient than using a true model repository directly [5,6].

[1] MOF (Meta-Object Facility) is a standard developed by OMG (Object Management Group) for describing formal models [15].

[2] For example, MongoDB write operations can be up to 10 times slower than read operations.

While migrating model-based applications to the web, there is the risk that the model repository could become a bottleneck, since model transformations use it intensively to implement business logic of model-based applications. Existing model repositories revealed two extremes: either a repository was memory-efficient, but not CPU-efficient (like ECore), or vice-versa (like the "New Repository" JR presented in 2010 [18]). Moreover, the internal encoding of model repositories was usually concealed, thus, sending the whole repository content via the network required to rely on the repository API, which was slower than if we had access to internal data structures. Thus, there remained a need for a fast repository that could be used in the web environment.

In this paper we propose a new model repository that is *both* CPU- and memory-efficient. It is also designed for fast synchronization via the network. Besides, the proposed repository has all the necessary functionality for storing and traversing models at different meta-levels. Our approach relies on a specific encoding of models.

The next section presents our idea. Section 3 reveals some interesting implementation details. In Sect. 4 we provide quantitative test results that confirm the feasibility of our approach. Finally, we discuss the potential of the proposed repository and conclude the paper (Sects. 5–6).

2 The Main Idea

When deciding which API the upcoming model repository has to implement, we aimed for an API that would be compatible with existing repositories. However, we tried to avoid high-level APIs (like *Epsilon Model Connectivity Layer* or *ATL Model Handler Abstraction Layer*), since they are not efficient (e.g., linked objects can only be set as a list, even when we need to include/exclude just one object), they conceal internal data structures too much, and they are hard to use with mixed meta-levels [8,13]. Instead, we focused on a low-level Repository Access API (RAAPI)[3]. RAAPI can be viewed as repository assembler, thus, the sequence of RAAPI calls can be treated as an assembly program for creating the content of the repository from scratch. This resembles how a sequence of low-level Turing machine operations results in the given string. We show below that the sequence of RAAPI operations can be kept short enough, thus, it can be used as an efficient encoding of a model (this resembles the Kolmogorov complexity concept with the difference that the sequence of operations results in a model instead of a string).

An interesting feature of RAAPI is that it was developed with Šostaks' conjecture in mind [14]:

> *It is difficult for a human to think at more than two meta-levels at a time. Still, it is fairly easy for a human to focus on any two adjacent meta-levels.*

[3] We proposed RAAPI in 2013 by combining the best from existing repository APIs. RAAPI can be mapped to virtually any model repository. The actual version can be found at http://webappos.org/dev/raapi/.

Table 1. Some modificating RAAPI functions (actions) and their encodings.

Modificating action	Code	Encoding (action code+arguments)
`Reference createClass(String name)`	0x01	2 numbers (1 code + 1 class reference) + 1 string (name)
`createGeneralization (Reference rSubClass, Reference rSuperClass)`	0x11	3 numbers (1 code + 2 class references)
`Reference createObject(Reference rClass)`	0x02	3 numbers (1 code + 1 class reference + 1 object reference)
`includeObjectInClass(Reference rObject, Reference rClass)`	0x12	3 numbers (1 code + 1 object reference + 1 class reference)
`Reference createAttribute(Reference rClass, String name, Reference rPrimitiveType)`	0x03	3 numbers (1 code + 1 class reference + 1 type reference) + 1 string (name)
`setAttributeValue(Reference rObject, Reference rAttribute, String value)`	0x04	3 numbers (1 code + 1 object reference + 1 attribute reference) + 1 string (value)
`Reference createAssociation( Reference rSourceClass, Reference rTargetClass, String sourceRole, String targetRole, boolean isComposition)`	0x05	6 numbers (1 code + 2 class references + 1 boolean as number + 2 references for direct and inverse association ends) + 1 string (sourceRole+'/'+targetRole)
`createLink(Reference rSourceObject, Reference rTargetObject, Reference rAssociationEnd)`	0x06	4 numbers (1 code + 2 object reference + 1 association end reference); for bi-directional links only one direction is stored (the opposite link can be derived)

To comply with this conjecture, RAAPI operations are defined for two adjacent meta-levels (the model and the meta-model level). However, all repository elements (objects, classes, attributes, and associations) are identified by 64-bit references (e.g., numbers or memory pointers) regardless of their meta-level. Thus, while working with levels i and $i+1$ we can obtain some element reference and then use it when working with levels $i+1$ and $i+2$. We can even mix references from different levels (e.g., linking an object "Peter" to a class "Collie" as in Fig. 1).

Certain RAAPI operations modify the state of the repository. We call them *modificating actions*. Some of them are mentioned in Table 1, Column 1 (besides create-actions there are also corresponding delete-actions, which are not mentioned). Other operations are read-only operations for querying/traversing the repository (see Table 2, Column 1).

Now, to encode the model we use a sequence of RAAPI modificating actions. Each modificating action is assigned an integer code. Action code as well as other non-string values (numbers, references, and booleans from action arguments as

Table 2. A representative set of RAAPI operations for querying/traversing the repository

Read-only operation	Indices and keys used	Search criteria
`findClass(String name)`	s2a(name)	the first action with code 0x01 and a2s(action) equal to *name*
`findAttribute(Reference rClass, String name)`	s2a(name)	the first action with code 0x03 and the first argument equal to rClass, and a2s(action) equal to *name* or recurse into superclasses (via getIteratorForDirectSuperClasses) for derived attributes
`isDirectSubClass(Reference rSubClass, Reference rSuperClass)`	r2a(rSubClass) r2a(rSuperClass)	the first action with code 0x11 and arguments rSubClass, rSuperClass
`isDerivedClass(Reference rSubClass, Reference rSuperClass)`	r2a(rSubClass) r2a(rSuperClass)	the first action with code 0x11 and arguments [rSubClass, rSuperClass] or recurse into subclasses of rSuperClasses or superclasses of rSubClass
`linkExists(Reference rSourceObject, Reference rTargetObject, Reference rAssociationEnd)`	r2a(rSourceObject) r2a(rTargetObject) r2a(rAssociationEnd) or r2a(rTargetObject) r2a(rSourceObject) r2a(inverse(rAssociationEnd))	first, we use r2a(rSourceObject), r2a(rTargetObject), and r2a(rAssociationEnd) to look for the first action with code 0x06 and arguments [rSourceObject, rTargetObject, rAssociationEnd]; if the action not found: — we use r2a(rAssociationEnd) to find the first action with code 0x05 to obtain the inverse association end; — we use r2a(rTargetObject), r2a(rSourceObject), r2a(inverse(rAssociationEnd)) to look for the first action with code 0x06 and arguments [rTargetObject, rSourceObject, inverse(rAssociationEnd)];
`getIteratorForDirectClassObjects (Reference rClass)`	r2a(rClass)	all actions with code 0x02 or 0x12 and the first argument equal to rClass
`getIteratorForDirectSuperClasses (Reference rSubClass)`	r2a(rClass)	all actions with code 0x11 and the first argument equal to rSubClass
`getIteratorForLinkedObjects (Reference rObject, Reference rAssociationEnd)`	r2a(rObject) r2a(rAssociationEnd) or r2a(rObject) r2a(inverse(rAssociationEnd))	all actions with code 0x06 and arguments 1 and 3 equal to [rObject, rAssociationEnd] and all actions with code 0x06 and arguments 2 and 3 equal to [rObject, inverse(rAssociationEnd)] (we use r2a(rAssociationEnd) to find the first action with code 0x05 to obtain the inverse association end)
`getIteratorForObjectsByAttributeValue (Reference rAttribute, String value)`	s2a(value)	all actions with code 0x04, the second argument equal to Attribute, and a2s(action) equal to *value*

well as the return value) are encoded as numbers stored as 64-bit IEEE doubles (see Table 1, Columns 2 and 3). The two main reasons for such encoding are:

- IEEE double is the only type for numbers supported by JavaScript in most browsers, thus, when using doubles, we can synchronize these numbers with the browser directly, without the conversion;
- the whole sequence of actions can then be stored in a single **actions** array, where each action occupies from 2 to 6 elements (thus, the *actions* array is in fact an array of variable-length mini-arrays).

Some of the modificating actions take also strings as arguments. We can assume that there is at most one string for each action (2 strings can be concatenated into one by using a delimiter, e.g., '/', see `createAssociation` in Table 1). All such strings are stored in the **strings** array in the same order as string-containing-actions (string-actions) from the *actions* array, thus, we can infer which string is associated with each actions just from the order of elements. When synchronizing, all the strings from the *strings* array are concatenated by some other delimiter and sent as one string.

An interesting fact is that our encoding stores only create-actions. When some repository element is deleted, instead of adding a new delete-action, we just delete the corresponding create-action from the *actions* arrays (and the corresponding string from the *strings* array, if any). Thus, the length of the sequence always corresponds to the size of the model.

> **Note.** Of course, this is a simplified view on the encoding. In fact, appending elements to and deleting them from an array is not trivial. Moreover, we also need some indexing to be able to iterate throughout these arrays while skipping unnecessary actions. As the next section shows, all these operations can be efficiently implemented (and the memory increases just linearly).

The server-side repository works directly with the *actions* and *strings* arrays (using a few helper arrays for efficient iterating), thus, minimizing memory consumption. The client-side repository (running in the browser) can convert the received *actions* and *strings* arrays to less efficient, but more convenient encoding using native JavaScript objects, since there is only one user at the client-side, controlling all the browser resources.

3 Implementation

The *actions* and *strings* arrays are implemented as classical resizable arrays with the amortized constant-time add and delete operations. Delete-actions are not deleted right away (which could result in shifting the arrays) – they are marked as deleted instead. When too many actions have been marked as deleted, or when there is no space for storing a new create-action, one or both arrays are re-arranged (this operation is rare compared to the cumulative number of add

and delete operations). The re-arrange operation compacts the given array by shifting the elements and eliminating delete marks. Then the array length is multiplied by 0.5, 1, or 2 depending on the number of free elements in the end of the re-arranged array.

Our experiments with RAAPI show that the length of the actions array is approximately 10 times the length of the strings array. We have chosen initial lengths of 10,000 and 1,000. The arrays can grow independently up to 1,310,720,000 and 131,072,000, respectively, unless lower limits are specified[4].

3.1 Additional Data Structures

To be able to traverse the model, we introduce 3 indexing data structures (indices). The first 2 are:

– the action-to-string map **a2s** (one action can have at most one associated string);
– the inverse string-to-action multimap **s2a** (the same string can be found in multiple actions, e.g., different objects can have the same attribute value).

They allow us to implement read-only RAAPI operations that return strings (e.g., *getClassName*, *getAttributeValue*) or look up for a reference given a string (e.g., *findClass*, *findAttribute*, or *getIteratorForObjectsByAttributeValue*).

Each action is identified by a corresponding index in the actions array. Each string is identified by an index in the strings array (however, string comparison is performed not on indices, but on the actual string values from the strings array).

The third indexing structure is the reference-to-action multimap **r2a** (the same reference, e.g., object reference, can be found within multiple actions). This map allows us to traverse only actions where the given reference is used. We do not need the inverse map, since, given an index in actions array, we can instantly access the corresponding mini-array containing the action code along with all references used as action arguments[5].

Notice that all 3 indices increase memory consumption just linearly (*a2s* and *s2a* sizes are comparable to the length of the *strings* array; *r2a* size is comparable to the *actions* length). However, when re-arranging actions and strings, we have to rebuild the indices (but that still keeps the amortized time for add and delete operations constant, since re-arrange is rare operation).

[4] The first number is the maximum length of actions that does not cause integer overflow ($2^{31} - 1$), which allows us to use 4-byte integers to encode positions in the actions array. The actions array then can occupy up to 10 GB (not counting strings), which we consider quite liberal for a single model accessed by a single user via a web application.

[5] We could also embed the *a2s* map into the *actions* array by appending a string index to mini-arrays of string-actions. However, that would mix the *actions* array with the auxiliary *a2s* array. In addition, the length of the *actions* array and, hence, the amount of data synchronized with the client would increase.

Having just these 3 maps/multimaps we can implement efficiently all read-only RAAPI operations as well as certain auxiliary internal operations such as cascade delete. The following subsections provide more detail.

3.2 Querying/Iteration

Table 2 mentions a representative subset of read-only RAAPI operations and reveals which indices and keys are used to implement them. Each key is used to obtain a list of actions from some index (*r2a* or *s2a*). Then these actions are checked against the conditions mentioned in Column 3 (sometimes *a2s* is used there to check equality of strings).

As Column 2 shows, sometimes we have to look at multiple lists of actions at the same time. For some RAAPI operations (e.g., *isDirectSubClass*) we just need to get the first action that belongs to all the given lists and meets the criteria, while for other (e.g., *getIteratorForLinkedObjects*) we have to iterate through all such actions.

Good news is that all lists of actions turn out to be sorted, since each time a new action is added, it is appended to the end of the actions array (perhaps, after re-arrange), where the index of the new action is greater than the index of all previous actions. Then this action and its arguments are added to the corresponding indices *r2a*, *a2s*, and *s2a*. Thus, we can use the "merge" approach when traversing actions that must belong to multiple lists (see the listing below). To make the search within multiple lists more efficient, we implemented the *nextGreaterOrEqual* operation via binary search.

```
findFirstActionWithin(lists) {
  // initializing iterators and getting first elements of
  // the lists (iterators return INFINITY, if there are no
  // more elements)
  for (i=0; i<lists.length; i++) {
   iterators[i] = lists[i].iterator();
   values[i] = iterators[i].first(); // INFINITY, if empty
  }
  m = max(values);
  while (m<INFINITY and not all values equal m) {
    // moving forward all iterators until each of them
    // points to an element >=m or to the end of the list
    for(i=0; i<lists.length; i++)
      values[i] = iterators[i].nextGreaterOrEqual(m);
    m = max(values);
  }
  return m; // INFINITY, if at least one list ended
}
```

The indices are used not only for queries/iterations, but also in modificating actions for validating the arguments. For example, in *setAttributeValue* we have to check that the given object exists and the given attribute reference is legitime, i.e., the object belongs to a class that has that attribute defined. In addition, we have to find and delete the previous attribute value, if any.

3.3 Cascade Delete

When a delete-operation is called, we find the corresponding create-operation in the actions array and mark it as deleted (for string actions we also mark *strings*[*a2s(action)*] as deleted). However, in certain cases cascade delete is required. For example, when deleting a class, all its objects have to be deleted as well. Thus, not only the *createClass* action (0x01) has to be marked as deleted, but also all subsequent *createObject* operations (0x02) having the same class reference as the first argument. When deleting an object, all corresponding attribute values (0x04) and links (0x06) have also to be deleted (marked). All such marked actions will be cleaned up during re-arrange.

To implement cascade delete we use the same *r2a* multimap as for querying/iteration. For instance, when we mark the *createClass* operation as deleted, we obtain the class reference *rClass*. Then we obtain the list *r2a(rClass)* and iterate through it to find actions with code 0x02 (*createObject*) and reference *rClass*. For each such action we obtain the object argument *rObject* and then iterate though the *r2a(rObject)* list and mark all its elements as deleted (since *rObject* is being deleted, any action that was stored after this *createObject* and referencing the same *rObject* must be marked as deleted; this will delete 0x04, 0x06, and, perhaps, other actions having *rObject* somewhere as an argument).

3.4 Memory-Mapped Files

Although we can use standard data structures (such as Java arrays and hashmaps) for the *actions* and *strings* arrays as well as for the indices, such implementation quickly leads to high dynamic memory consumption (and even to out-of-memory exception, if more than 110 middle-sized repositories are open, see below). This is undesirable for the web server. Our approach is to rely on memory-mapped files, a mechanism, which is available in most operating systems. The OS automatically swaps memory pages, while the programmer can access the data via a single pointer as if the data were always loaded into memory. With memory-mapped files, server memory is not limited to the size of the physical RAM, and the OS does all the low-level job automatically and efficiently (for instance, files are loaded into memory in lazy manner, thus attaching a file as a pointer is fast). The shortcoming is that memory-mapped files, in essence, are arrays. While the actions array can be mapped directly to a file, other data structures (indices and strings) have to be mapped to arrays manually.

To be able to store strings in a memory-mapped file, we use 2 arrays: **chars** and $\mathbf{strings_2}$. The first one is for appending characters of each new string (we use UTF-8 character encoding); the second one stores the start index in the *chars* array and the string length (in bytes). The re-arrange functions works only on the relative short $strings_2$ array, thus, characters are not moved[6].

[6] The *chars* array is much longer than $strings_2$. Thus, to save time during re-arrange, the *chars* array can be left "as is", without removing characters of deleted strings (still, it can be compacted occasionally, e.g., during save).

The *r2a*, *a2s*, and *s2a* indices are implemented as arrays of keys and values. The lengths of these arrays are prime numbers that depend on the lengths of the *actions* and $strings_2$ arrays. Prime lengths allow us to use these arrays as hash tables with open addressing and double hashing [7] [12]. Since *r2a* and *s2a* are multimaps, we modify traditional hashing approach: for multi-valued keys we store a negative number $-(k+1)$ in a hash table, where k is the number of values already stored for this key (including the collisions). Thus, to append a new value for the given key, we first skip $(k+1)$ elements and try to append the value as usual. Our experiments show that the number of collisions for such multimaps (when working on a repository containing data from a real use case) is 2.28 in average.

While deleting elements from a hash table may be non-trivial, our approach is simple: we just mark elements as deleted (when the corresponding actions are marked as deleted). During re-arrange, hash tables are rebuilt from scratch. However, this approach introduces a new issue: when traversing the values of a multimap, we can encounter such marked-as-deleted elements. If we need to iterate through all elements, we can just ignore these marked elements. However, the function *nextGreaterOrEqual* mentioned above won't work any more, since the sorted list of values now can contain deleted (marked) values, and the binary search algorithm won't work as expected. Generally speaking, the binary search has to be replaced with linear search[8]. However, since the number of marked elements is small (otherwise, the array is re-arranged), we introduce the following modification of the binary search operation: when we encounter a marked-as-deleted element that should become a new middle element, we look for the next non-marked element linearly. Then the search continues as ordinary binary search. This modification proved to be very fast in practice (it boosted model transformations by 60.56% compared to fully linear implementation of *nextGreaterOrEqual*).

4 Feasibility

In this section we provide details on CPU and memory benchmarks. We also give some notes on synchronization overhead.

4.1 CPU Benchmark Tests

Table 3 provides averaged CPU benchmark data for the proposed repository AR (acronym for "Actions Repository") in comparison with Ecore and JR [18,20].

[7] The first hash is modulo p, the second is modulo $(p-2)+1$, which is always co-prime with p. For strings we use Java built-in *hashCode* function. However, since it returns 0 on empty strings, and since the second hash calculates to 1, all empty strings would be stored in the beginning of the hash table, thus, drastically increasing the number of collisions (up to 2.85x in our experiments). We avoid such inefficient hash values by appending a constant dummy text to every string before calculating its hash.

[8] Grover's algorithm on a real quantum computer could take sub-linear time, but the proposed repository is intended for classical computers.

Table 3. CPU benchmark (all values are in milliseconds per repository)

	ECore	**JR**	**AR (Java hash maps)**	**AR (hash tables)**	**AR (memory-mapped files)**
Repository time	1016	857	423	618	760
Overhead time	20,015*	436	106	93	85

Processor Intel i7-2600, 3.40 GHz, repository running within a single thread, 64-bit Java Virtual Machine 1.8.0 on Windows, no heavyweight parallel processes running.
Profiler: Java VisualVM 1.8.0 in CPU profiling mode.
Repository time accuracy w.r.t. the mean value is 10%, overhead time accuracy is 25%.

*Due to ECore internal design, implementation of some RAAPI operations required significant overhead in order to avoid ECore exceptions.

AR and ECore are implemented in Java, while JR—in plain C (until now, JR proved to be the fastest repository we ever used in our model-based tools).

In our tests we were interested in 3 variations of AR: using Java standard data structures (Java arrays, HashMaps, and ArrayLists), using hash tables implemented manually via in-memory arrays, and using hash tables stored in a memory-mapped file. In all cases we used a transformation borrowed from the ontology editor OWLGrEd (http://owlgred.lumii.lv). The transformation we chose performs a set of actions (such as creating a dialog window from a model, storing the input in the repository, and refreshing the diagram from the updated model) that represent a real usage step of a graphical model-based tool. We measured not only CPU clock for each of the repositories, but also the overhead added by wrappers, which map universal RAAPI to native repository APIs (operations not provided by native APIs were implemented in wrappers). We can infer from Table 3 that AR outperforms both ECore and JR. Logically, memory-mapped files are a bit slower than direct in-memory hash tables. Java built-in data structures show the best CPU benchmark rates (but not the best memory rates, as is shown below).

4.2 Memory Benchmark Tests

Table 4 provides averaged memory benchmark for the repository, on which the transformation mentioned above was executed.

We measured not only memory consumption, but also repository load time. For memory-mapped files we split our tests into 2 groups: tests from the first group were executed, when there were no memory-mapped files on disk (thus, they had to be created by the OS and filled with data by AR); tests from the second group just opened existing memory-mapped files. As Table 4 shows, AR with memory mapped files was the only repository that could handle 10,000 models at the same time with significant room for scaling (and the OS reserved just 54 MiB of RAM for all of them, if we do not count the files on disk amounting to 122 GiB in total). We have to admit that our tests did not include heavyweight

Table 4. Repository memory usage (MiB/repository) and open time (ms/repository)

Number of repositories		**JR**	**ECore**	**AR (Java hash maps)**	**AR (hash tables)**	**AR (creating memory-mapped files); 12.2MiB on disk per repository**	**AR (opening memory-mapped files); 12.2MiB on disk per repository**
100	**memory**	33.41[⋈]	12.77	13.98	11.25	1.67	0.013
	time	1047[⋈]	849	153	63	61 (SSD) or 69 (HDD)	26 (SSD) or 9 (HDD)
1,000	**memory**	n/a	9.45[◇] or 9.57[◇◇]	13.74[*] or 13.55[**]	10.72[□]	0.071	0.013
	time		4763[◇] or 5414[◇◇]	189[*] or 714[**]	83[□]	68 (SSD) or 148 (HDD); 420 (LAN)[♮]	27 (SSD) or 33 (HDD); 280 (LAN)[♮]
10,000	**memory**	n/a	n/a	n/a	n/a	0.012	0.005
	time					226 (HDD)	41 (HDD)

Profiler: Java VisualVM 1.8.0 in CPU and memory profiling modes.

⋈ One JR instance; it is impossible to open multiple JR instances in the same process.
◇ 173 repositories before out of memory; automatic garbage collection
◇◇ 180 repositories before out of memory; forced garbage collection
* ≈111 repositories before freeze; automatic garbage collection
** 101 repository before out of memory; forced garbage collection
□ ≈142 repositories before out of memory; automatic garbage collection
♮ Windows share (samba) over a 100 Mbit/s local area network

parallel processes or intensive usage of memory by multiple users with inevitable competition for processor cache (these tests are subject to additional research). Nevertheless, current results look promising.

4.3 Synchronization

CPU and memory efficiency is not enough for a repository with web-based usage in mind. We have to be able to synchronize the repository efficiently between the client and the server. Our solution relies on using web sockets, a standardized protocol with low overhead (when set up properly, web sockets can hold 1,000,000 connections, and even more). Since web sockets can be used to transmit both binary and string data, the actions and strings can be synchronized efficiently. The client and the server can modify the repository independently, each on its side. We use the following trick to avoid collisions in references: when new repository elements (objects, classes, associations...) are created, the server assigns even references for them, but the client assigns odd. Modifications are exchanged asynchronously

(both at the server and at the client side), thus, a separate thread is busy with that, while the original thread running a model transformation continues without any delay (delay can only be caused by network buffers overflow). To optimize the synchronization process, we collect several modifications within a small time interval and then send them in bulk. Modifications are sent using the encoding of the *actions* and *strings* arrays with an exception that modifications can contain also delete-actions. When received, modifications are re-executed on the receiving side as if they occurred right there.

5 Discussion

Currently, AR iterators over repository elements are not thread-safe internally. We deal with this issue by synchronizing public RAAPI calls and copying the required elements each time an iterator is returned via RAAPI. In the future, to boost iterators, we could switch to the copy-on-write pattern (where copying is done only if a parallel modification is performed).

Since all elements (classes, associations, etc.) in AR are identified by 64-bit references, we can create classes and objects at different meta-levels and even mix them. Thus, AR can be used for storing models corresponding to virtually any meta-modelling standard (e.g., MOF, EMOF, or SMOF) or ontology language (e.g., OWL or OWL2) [2,3,15,16]. This can lead to interesting use cases. For instance, we can create meta-meta-level classes corresponding to the OWL2 standard (*OWL:Class*, *OWL:Property*, etc.). Then we can create ordinary metamodel classes and call *includeObjectInClass* to make them instances of the meta-meta-level classes (e.g., class *Person* would become an instance of *OWL:Class*). All these operations are legitime and are just added to the actions array. Then, by using AR indices, we can infer which classes are instances of OWL2 meta-metamodel, and forward them to a semantic reasoner.

AR can also be used in a NoSQL-manner, where the metamodel is not defined in advance. This can be implemented in 2 ways:

- by skipping metamodel checks (i.e., not validating action arguments);
- by introducing a wrapper. When some action requiring a metamodel element is performed, the wrapper creates a missing metamodel element on-the-fly. This, however, requires advanced techniques for guessing metamodel elements (e.g., guessing types of attributes or inheritance relations) and, perhaps, modifying them dynamically, if eventually we find that the initial guess was incorrect.

Our tests showed that AR is more efficient than JR. The JR authors showed that their repository outperforms popular *OpenLink Virtuoso*. Pacaci et al. showed that *Virtuoso* outperforms other graph databases, and Hellerstein et al. showed that graph databases outperform relational ones [4,19]. While these facts may seem to be in favor to the proposed repository, we have to admit that the performance depends on a particular usage scenario. For instance, Pacaci et al. showed

that traditional relational *Postgres* database outperformed *Virtuoso* in several specific tests [19].

It is hard to compare AR to linked data and their query mechanisms (like *Linked Data Fragments*, linkeddatafragments.org), since they are optimized for single-query usage (where each query can be quite complex), while AR is designed to serve multiple, but simple queries performed by model transformations.

6 Conclusion

We presented a model repository that outperforms existing repositories regarding both CPU and memory. The main idea was to use an efficient encoding of the model by storing a list of actions (and corresponding strings) that create the content of the repository (which resembles the Kolmogorov complexity concept). We added just 3 indexing arrays for implementing RAAPI query and iteration operations. The proposed encoding, combined with memory-mapped files, can hold 10,000 repositories (and even more) on a single server. That resembles the C10K problem (10,000 concurrent connections; that is considered a reasonable target for web-based applications[9]) [10]. However, stress tests concerning CPU cache and context switches still have to be performed.

The proposed repository encoding is used "as is", when synchronizing the repository via the network (the encoding even uses the IEEE double as the only JavaScript-compatible type for numbers). Since we use asynchronous web sockets, synchronization overhead is negligible (unless the network becomes a bottleneck).

The repository implements universal RAAPI, where the developer thinks at two adjacent meta-levels (the model and the meta-model level), but can use any number of meta-level and even mix them.

We hope the repository will find wide adoption, thus, we release it under an open-source license[10]. The repository is written in Java, but a dynamic-link library for accessing it from native code is available (32-bit and 64-bit versions for Windows, Linux, and MacOS platforms).

We are working on developing a model-based infrastructure for web applications (webAppOS), where the proposed repository will be a central component implementing memory abstraction. A webAppOS-based version of our graphical ontology editor OWLGrEd is coming soon. OWLGrEd diagrams will be stored using AR, making the proposed repository a part of the new OWLGrEd file format for both desktop and web-based versions of OWLGrEd.

Acknowledgments. The work has been supported by European Regional Development Fund within the project #1.1.1.2/16/I/001, application #1.1.1.2/VI-AA/1/16/214 "Model-Based Web Application Infrastructure with Cloud Technology Support".

[9] For more connections or during peek loads, one can borrow virtual cloud servers, e.g., from Amazon Elastic Cloud.

[10] The repository can be downloaded at http://webappos.org/dev/ar.

References

1. Eclipse Modeling Framework (EMF, Eclipse Modeling subproject). http://www.eclipse.org/emf
2. OWL 2 Web Ontology Language document overview (second edition). http://www.w3.org/TR/owl2-overview/
3. OWL Web Ontology Language reference. http://www.w3.org/TR/owl-ref/
4. Hellerstein, J.M., et al.: Ground: a data context service. In: Proceedings of CIDR (2017)
5. Ambler, S.: Mapping objects to relational databases: O/R mapping in detail. http://www.agiledata.org/essays/mappingObjects.html
6. Anuja, K.: Object Relational Mapping. Ph.D. thesis, Cochin University of Science and Technology (2007)
7. Barzdins, J., Kalnins, A., Rencis, E., Rikacovs, S.: Model transformation languages and their implementation by bootstrapping method. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 130–145. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78127-1_8
8. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MODELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006). https://doi.org/10.1007/11663430_14
9. Kalnins, A., Barzdins, J., Celms, E.: Model transformation language MOLA. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003-2004. LNCS, vol. 3599, pp. 62–76. Springer, Heidelberg (2005). https://doi.org/10.1007/11538097_5
10. Kegel, D.: The C10K problem. http://www.kegel.com/c10k.html
11. Kemp, S.: Digital in 2018: World's internet users pass the 4 billion mark. wearesocial.com blog. https://wearesocial.com/blog/2018/01/global-digital-report-2018
12. Knuth, D.E.: The Art of Computer Programming, Sorting and Searching, 2nd edn., vol. 3. Addison Wesley Longman Publishing Co., Inc., Redwood City (1998)
13. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book. http://www.eclipse.org/epsilon/doc/book/
14. Kozlovics, S.: The orchestra of multiple model repositories. In: van Emde Boas, P., Groen, F.C.A., Italiano, G.F., Nawrocki, J., Sack, H. (eds.) SOFSEM 2013. LNCS, vol. 7741, pp. 503–514. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35843-2_43
15. Object Management Group: OMG Meta Object Facility (MOF) Core Specification Version 2.4.1 (2011)
16. Object Management Group: MOF Support For Semantic Structures (SMOF) (2012). http://www.omg.org/spec/SMOF/
17. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3. formal/16-06-03 (2016)
18. Opmanis, M., Čerāns, K.: Multilevel data repository for ontological and metamodeling. In: Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010 (2011)
19. Pacaci, A., Zhou, A., Lin, J., Özsu, M.T.: Do we need specialized graph databases?: benchmarking real-time social networking applications. In: Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES 2017, pp. 12:1–12:7. ACM, New York (2017)
20. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Upper Saddle River (2008)
21. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3), 187–207 (2007)