Fast Model Repository
as Memory for Web Applications

Sergejs KOZLOVIČS [1,2]

Institute of Mathematics and Computer Science, University of Latvia

Abstract. We address the task of migrating standalone model-based applications to the web, where we face the need to synchronize models between the client and the server. Because of synchronization overhead and limited server memory that has to be shared among all connected users, there is the risk that the model storage could become a bottleneck. We propose a model repository that uses an efficient encoding of the model that resembles its Kolmogorov complexity and is suitable for direct sending over the network (with almost no serialization overhead). All traverse and query operations can also be implemented efficiently by introducing just 3 automatic indexes. By utilizing the OS paging mechanism, we are able to hold 10,000 and more repositories on a single server.

Keywords. models, model repository, web applications

1. Introduction

Many existing applications have been developed using the model-driven approach. However, usually, they are standalone applications. Since the web brings obvious benefits such as instant availability and the ability to use them from different devices and operating systems, developers consider the option to migrate their existing standalone applications to the web. In this paper we focus on model-based applications. By *model-based application* we mean an application that stores data in MOF[3]-like models and processes these data by corresponding model transformations [1,2]. We say "MOF-like" models, since in practice alternative implementations such as Java-based EMF/ECore are used [3,4]. 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++) that are able to access MOF-like models via some API (e.g., ECore API) [5,6,7,8,9].

---

[1]Corresponding Author: Sergejs Kozlovičs, Institute of Mathematics and Computer Science, Raina blvd. 29, LV-1459, Riga, Latvia; E-mail: sergejs.kozlovics@lumii.lv.

[2]Supported by European Regional Development Fund within the project #1.1.1.2/16/I/001, application #1.1.1.2/VIAA/1/16/214 "Model-Based Web Application Infrastructure with Cloud Technology Support".

[3]MOF (Meta-Object Facility) is a standard developed by OMG (Object Management Group) for describing formal models [1].
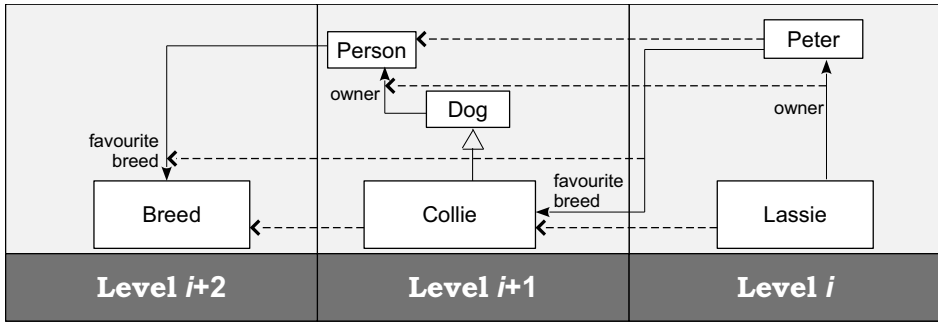
Figure 1. An example of multiple mixed meta-levels (dashed lines represent the "instance-of" relation). The relation "favourite breed" between Person and Breed as well as the link between Peter and Collie cross two adjacent meta-levels.

In model-based applications models are usually saved in an in-memory storage that we call *model repository*. Unlike databases, model repositories are simpler, but more efficient. The model 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 addition, model repository internal structures and APIs are tailored for storing models. Some can store models at multiple meta-levels (see Figure 1).

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 [10]). 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 able to utilize the OS paging mechanism and synchronize its content via the network with almost no serialization overhead.

Our idea is to use a specific encoding of models, which is introduced in the next section. Section 3 reveals some interesting implementation details. In Section 4 we provide quantitative test results that confirm the feasibility of our approach. In Section 5 we show how the proposed repository could serve as a memory analog for web applications. Finally, we discuss other potential use cases of the proposed repository and conclude the paper (Sections 6–7).

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 [7,8]. Instead, we focused on a low-level Repository Access API (RAAPI)[4]. 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 [11]:

> *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.*

To comply with this conjecture, RAAPI operations are defined for two adjacent meta-levels (the model and the meta-model level). However, 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 the object "Peter" to the class "Collie" in Figure 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).

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 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 concatenated 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-

---

[4]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 references + 1 association end reference); for bi-directional links only one direction is stored (the opposite link can be derived)

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.

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

Read-only operation	Indices and keys	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)];

`getIteratorForDirect-` `  ClassObjects(Reference rClass)`	r2a(rClass)	all actions with code 0x02 or 0x12 and the first argument equal to rClass
`getIteratorForDirect-` `  SuperClasses(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)

`getIteratorForObjects-`
`  ByAttributeValue`
`  (Reference rAttribute,`
`  String value)`
	s2a(value)	all actions with code 0x04, the second argument equal to Attribute, and a2s(action) equal to *value*

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[5].

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.

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).

---

[5]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 10GB (not counting strings), which we consider quite liberal for a single model accessed by a single user via a web application.

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 modifying 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. Class objects can be found by traversing the list *r2a(class_reference)*. When deleting an object, all corresponding attribute values (0x04) and links (0x06) have also to be marked (this can be done by traversing *r2a(object_reference)*). All marked actions will be cleaned up during re-arrange.

3.4. Memory-Mapped Files

To avoid out-of-memory exceptions when too many model repositories are open, we rely on memory-mapped files — a mechanism available in the majority of 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 **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].

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 is

---

[6]The *chars* arrays still can be compacted occasionally, e.g., during save.

[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.

2.28 in average (when working on a repository containing data from a real use case).

While deleting elements from hash tables is usually not 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*).

3.5. Synchronization and Avoiding Collisions

To synchronize the repository efficiently between the client and the server we use web sockets, a standardized protocol with low overhead for transmitting both binary and string data. Thus, 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 term *repository client* to denote each side). Later, they synchronize changes. To optimize the synchronization process, we collect several modifications within a small time interval and then send them in bulk. Changes are sent asynchronously, thus, the main process continues without any noticeable delay. 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. However, if no precautions are taken, there is the risk of collisions. Below we list possible types of collisions and provide their resolving mechanisms.

3.5.1. Reference Collisions

When the server and the client create new elements (objects, classes, associations, etc.), they could reserve the same reference for different entities. To avoid this type of collision, we split the set of references into even and odd: the server-side repository client assigns even references for new objects, while browser-side JavaScript repository client assigns odd. Thus, both repository clients will have a predefined last bit for new references. We can generalize this approach to support more than 2 repository clients: each repository client (the server, the browser, the debugger, etc.) has the number of predefined lower bits and their values.

---

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

Assume that we reserve 10 lower bits to support up to 1024 repository clients. We can approximate the number of new references each repository client reserves each second by 1000. Taking into a consideration that IEEE double has 52 bits of mantissa that can be used to store integer references, we come up with more than 136 years of repository usage without re-mapping references.

If the number of repository clients is not known in advance, but is quite small (up to 10), instead of reserving 10 lower bits, we can start with just 1 reserved bit for the client and the server. Then we can take the server-side repository and split its even references into two groups: ending with 00_2 and 10_2 (the client-side repository will continue to use odd references). However, in this case the reference sets are not equally distributed between all repository clients. Moreover, if the same repository client is constantly being split, its reference set will decrease exponentially with each new client.

3.5.2. Attribute Collisions

The server and the browser can alter object attribute values independently before they synchronize their changes. We must ensure that all repository clients converge to the same repository state after synchronization. However, we do not address the problem of synchronizing server and client threads,— it is the responsibility of the developer.[9]

Assume we have some object attribute with value v_0 (v_0 could be null, if no assignments were made yet, or if the value had been deleted). Now both repository clients set new attribute values: the server assigns v_1 and the browser assigns v_2 (we can assume $v_1 \neq v_0$ and $v_2 \neq v_0$; otherwise we can skip the idempotent assignment). After synchronization, the server gets value v_2, and the browser gets value v_1, resulting in inconsistency if $v_1 \neq v_2$. To avoid that, we synchronize the old value v_0 along with the new value (thus, the server sends both v_1 and v_0, and the browser sends both v_2 and v_0). Now, before the assignment, each client compares its current value against the received old value v_0. If the current value equals v_0, then the assignment is made as usual. If the current value differs, then we assign the value min(*current value, received synchronized value*). In our example, the server assigns $\min(v_1, v_2)$ and the browser assigns $\min(v_2, v_1)$, resulting in the same repository state.

3.5.3. Link Collisions

When dealing with inconsistent *createLink* and *deleteLink* operations, we can treat links as Boolean attribute values depending on whether the link exists. Thus, the same technique from Section 3.5.2 applies.

4. Feasibility

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

---

[9]Similarly, when two processes on the same machine modify a common variable simultaneously, they end up with the same memory state. However, the developer is responsible for further process behavior that might depend on the value of that shared variable.

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.

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 [3,10]. 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 54MiB 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 parallel processes or intensive usage

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

Number of reposi- tories		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

of memory by multiple users with inevitable competition for processor cache. Nevertheless, current results look promising.

5. Repository as Web Memory

Having the proposed model repository, both the web server and web browser can use it as shared "web memory", which is kept in sync automatically and transparently. This resembles classical multi-processor systems, where the common memory is shared between multiple processor cores, see Figure 2.

Nevertheless, "web memory" differs from classical RAM in the following points:

- Classical RAM, in essence, is an array; "web memory", in its turn, is a model, which is a graph-like structure.
- In classical RAM, it is impossible to create hooks for memory assignments. In "web memory", the corresponding repository API function can be overridden, thus, making it possible to track changes for further synchronization.
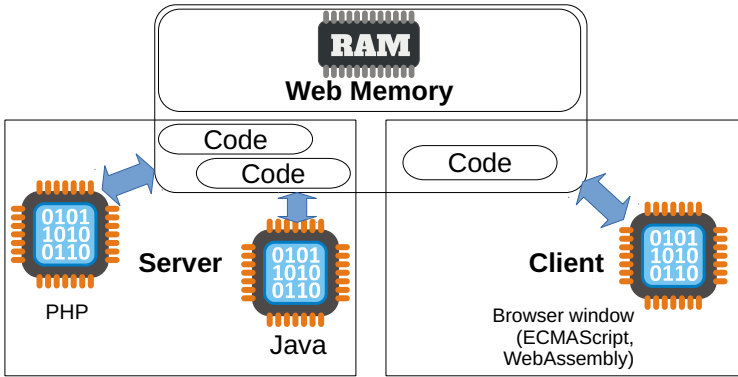
Figure 2. "Web memory": common memory shared between the server and the client.
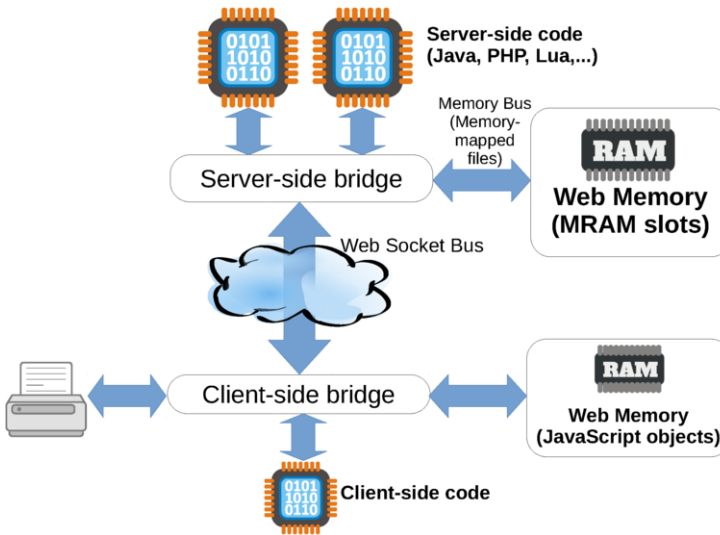

Figure 3. Bridges are responsible for synchronizing "web memory" and for providing access to memory from server-side and client-side code.

We can go even further by providing an indirection layer that factors out not only memory synchronization, but also access to server-side and client-side code, thus, making the illusion that web applications are being written for a single computer, without the need to think about network-specific aspects. We are working on implementing this approach in our infrastructure for web applications *webAppOS* (`http://webappos.org`). Figure 3 depicts a part of *webAppOS* architecture that resembles a typical chipset layout within the motherboard of a personal computer. The main repository content is created at the server side (each repository occupies one "MRAM[10] slot"). Each server-side repository is mapped to the OS virtual memory space by means of memory-mapped files. Thus, server-side code can

---

[10]Model Repository as Memory. The acronym was deliberately chosen to resemble RAM (Random Access Memory).

be distributed between processes, where each of them can access MRAM slots directly by addressing the same memory-mapped files. We call such direct access to repositories Memory Bus (this resembles how a northbridge accesses memory slots in a PC).

There is also Web Socket Bus for synchronizing the model over the network by means of web sockets. We call web socket end points the Server-side bridge and Client-side bridge (analogs of northbridge and southbridge on the motherboard). On certain model changes, the bridges will trigger execution of server-side code, client-side code, or even access to client-side devices (such as printers).

6. 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 has been 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) [1,13,14,15]. 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 meta-model 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 legitimate 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 [16,17]. 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 [16].

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.

7. 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[11]) [18]. 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[12]. 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

---

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

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

References

[1] Object Management Group, *OMG Meta Object Facility (MOF) Core Specification Version 2.4.1*, Object Management Group Std. formal/2011-08-07, 2011.

[2] Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3*, formal/16-06-03, Object Management Group Std. formal/2011-01-01, 2016.

[3] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework, 2nd Edition*, E. Gamma, L. Nackman, and J. Wiegand, Eds.   Addison-Wesley, 2008.

[4] Eclipse Modeling Framework (EMF, Eclipse Modeling subproject). http://www.eclipse.org/emf.

[5] A. Kalnins, J. Barzdins, and E. Celms, "Model transformation language MOLA," in *Model Driven Architecture*, Lecture Notes in Computer Science.   Springer Berlin Heidelberg, 2005, vol. 3599, pp. 62–76.

[6] J. Barzdins, A. Kalnins, E. Rencis, and S. Rikacovs, "Model transformation languages and their implementation by bootstrapping method," in *Pillars of computer science*, A. Avron, N. Dershowitz, and A. Rabinovich, Eds.   Berlin, Heidelberg: Springer-Verlag, 2008, pp. 130–145.

[7] D. Kolovos, L. Rose, and R. Paige, "The Epsilon Book," http://www.eclipse.org/epsilon/doc/book/.

[8] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Proceedings of the 2005 international conference on Satellite Events at the MoDELS*, MoDELS'05.   Berlin, Heidelberg: Springer-Verlag, 2006, pp. 128–138.

[9] D. Varró and A. Balogh, "The model transformation language of the VIATRA2 framework," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 187–207, Oct. 2007.

[10] M. Opmanis and K. Čerāns, "Multilevel data repository for ontological and meta-modeling," in *Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010*, 2011.

[11] S. Kozlovics, "The orchestra of multiple model repositories," in *SOFSEM 2013: Theory and Practice of Computer Science*, Lecture Notes in Computer Science, vol. 7741. Springer Berlin Heidelberg, 2013, pp. 503–514.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching.*   Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[13] Object Management Group, *MOF Support For Semantic Structures (SMOF)*, Object Management Group Std. ptc/2012-08-18, 2012, http://www.omg.org/spec/SMOF/.

[14] W3C, *OWL Web Ontology Language Reference*, http://www.w3.org/TR/owl-ref/, W3C Recommendation, 10 February 2004.

[15] W3C, *OWL 2 Web Ontology Language Document Overview (Second Edition)*, http://www.w3.org/TR/owl2-overview/, W3C Recommendation, 11 December 2012.

[16] A. Pacaci, A. Zhou, J. Lin, and M. T. Özsu, "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'17. New York, NY, USA: ACM, 2017, pp. 12:1–12:7.

[17] J. H. et al., "Ground: A data context service," *Proceedings of CIDR 2017*.

[18] D. Kegel, "The C10K problem," http://www.kegel.com/c10k.html.