An Efficient Computation Strategy for allInstances()

Ran Wei and Dimitrios S. Kolovos

Department of Computer Science,
University of York, United Kingdom
{ran.wei, dimitris.kolovos}@york.ac.uk

Abstract. Contemporary model query and transformation engines typically provide built-in facilities for retrieving all instances of a particular type/kind regardless of their location in a model (i.e. OCL's allInstances()). When implemented in a naive manner, such facilities can be computationally expensive for large models. We contribute a novel approach for implementing allInstances()-like facilities for EMF models, which makes use of static analysis and metamodel introspection and we report on the results of extensive benchmarking against alternative approaches.

1 Introduction

As models involved in MDE processes get larger and more complex[1, 2], model query and transformation languages are being stressed to their limits [3, 4]. One of the most computationally-expensive operations that model query and transformation engines support is the ability to retrieve collections of instances of a particular type/kind regardless of their location in a model (i.e. OCL's *allInstances()*). In this paper we discuss existing strategies for computing such collections of instances and we highlight their advantages and shortcomings. We then contribute a novel computation strategy that makes use of static analysis and metamodel introspection to pre-compute and cache all such collections needed in the context of a query in one pass. We present an implementation of the proposed strategy on top of an existing model query language (Epsilon's EOL [5]) and benchmark it against alternative computation strategies.

2 Background and Motivation

The majority of contemporary model query and transformation languages provide support for retrieving collections of all model elements that are instances of a particular type/kind. For example, OCL, QVTr, ATL, and Acceleo provide the built-in *allInstances()* operation which can be invoked on a type to return a set containing all its instances (e.g. *Person.allInstances()*), Epsilon's EOL provides the *getAllOfType()* and *getAllOfKind()* operations, and QVTo the *objects(type : Type)* and *objectsOfType(type : Type)* operations that operate in a similar way.

We collectively refer to all such operations as *allInstances()* in the remainder of the paper.

For file-based EMF models, a naive strategy to implement *allInstances()* is to navigate the in-memory model element containment tree upon invocation, and collect and return all instances of the requested type. Repeatedly traversing the containment tree to fetch all instances of the same type for multiple invocations of the operation on that type is clearly inefficient, so the majority of model query and transformation engines provide support for caching and reusing the results of previous invocations of the operation (this is straightforward for side-effect free languages but requires some additional book-keeping for languages that can mutate the state of a model).

When a query (or a transformation) contains a large number of calls to *allInstances()* for different types, instead of traversing the containment tree for each of these calls/types on demand, it can be more efficient for the execution engine to pre-compute and cache all these collections in one pass at start-up instead (*greedy caching*). This can incur a higher upfront cost and increase the memory footprint, however, for a sufficiently high number of invocations on different types, it is very likely to pay off eventually – particularly as models grow in size.

Overall, when more than one calls to *allInstances()* are made for different types in the context of a query, the on-demand approach is sub-optimal in terms of performance. On the other hand, if a query only calls *allInstances()* on a small number of types (compared to the total number of types in the metamodel), greedy caching is wasteful.

3 Program- and Metamodel-Aware Instance Collection

Given in-advance knowledge of the metamodel of a model, and the types on which *allInstances()* is likely to be invoked in the context of a query (e.g. obtained through static analysis of the query itself) operating on that model, in this section we demonstrate how a query execution engine can efficiently pre-compute and cache the results of only these invocations by traversing the contents of the model only once.

We demonstrate the proposed algorithms and their supporting data structures with reference to a concrete OCL-like query language (Epsilon's EOL). For conciseness, we also restrict the discussion to EOL queries operating on a single EMF-based model which conforms to an Ecore metamodel comprising exactly one EPackage. However, the proposed approach is trivially portable to other query and transformation languages of a similar nature, and to queries that involve more than one models conforming to multi-EPackage metamodels.

3.1 Cache Configuration Model

Figure 1 demonstrates a data structure (in the form of a metamodel), an instance of which needs to be populated at compile-time (e.g. by statically analysing the

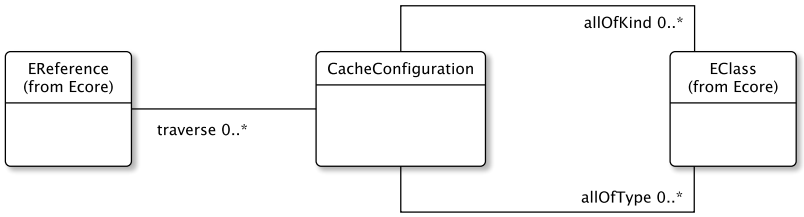

Fig. 1. Cache Configuration Metamodel

query of interest and by introspecting the metamodel of models on which it will be executed) in order to facilitate efficient execution of *allInstances()* at runtime.

CacheConfiguration acts as a *container* for the *EClasses* of the model's metamodel that the engine may need to retrieve all instances of in the context of the query of interest. *EClasses* of interest can be linked to a *CacheConfiguration* through the latter's *allOfKind* and *allOfType* references (EOL, like QVTo, support distinct operations for computing all direct and indirect instances of a given type). We intentionally refrain from discussion the *traverse* reference in Figure 1 for now.

3.2 Query Static Analysis

The first step of the process is to generate an initial version of the cache configuration model by statically analysing the query of interest. Figure 2 demonstrates the type-resolved abstract syntax graph of the example EOL program illustrated in Listing 1.1, which operates on models conforming to the metamodel of Figure 3. To compute the initial version of the cache configuration model we need to iterate through the abstract syntax graph and locate instances of:

- *MethodCallExpression* for which the name of the method called is *allOfKind*, *allOfType*, *allInstances* (alias of *allOfKind()*), the resolved type of their target expression is *ModelElementType*, and which have no parameter values;
- *PropertyCallExpression* for which the name of the property is *all* (alias of *allOfKind()*), and the resolved type of their target expression is *ModelElementType*.

Listing 1.1. An example EOL Program

```
1 WebPage.allOfType().println();
2 Member.allOfKind().println();
```

Having identified the calls of interest, we construct a new *CacheConfiguration* and for each call to *allOfType()* we create an *allOfType* link to its respective *EClass*. Similarly, for all other calls of interest we link the respective *EClasses* to the cache configuration via its *allOfKind* reference. The initial extracted cache configuration model for our running example is illustrated in Figure 5.

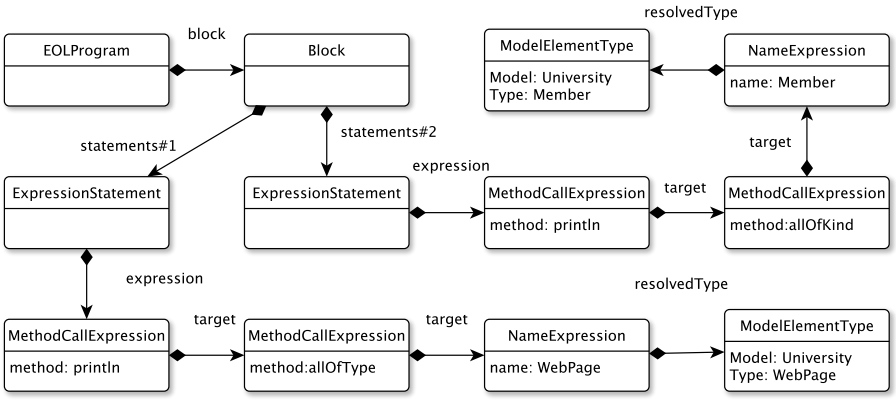

Fig. 2. The Abstract Syntax Graph of the EOL program of Listing 1.1

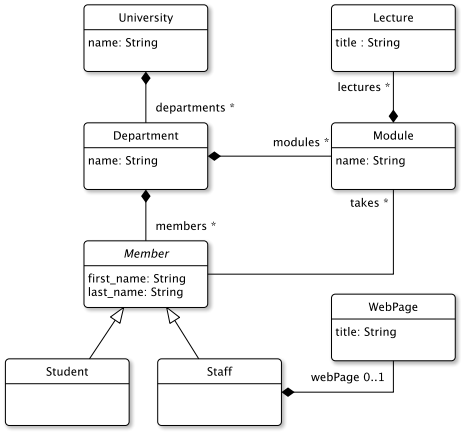

Fig. 3. The University Metamodel

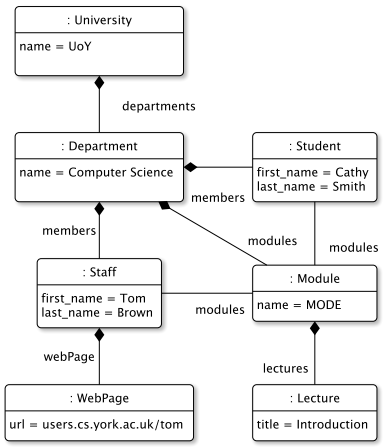

Fig. 4. The University Model

3.3 Containment Reference Pruning

Following the process discussed above, the execution engine can now be aware of all the *allInstances()* collections it needs to pre-compute and cache (*Web-*

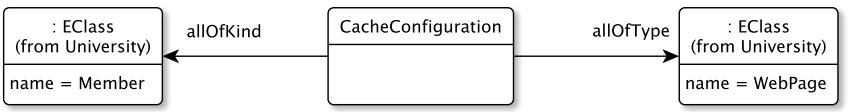

Fig. 5. Initial Extracted Cache Configuration Model

Page.allOfType() and *Member.allOfKind()* in our running example). The next step is to collect the model elements of interest in one pass and as efficiently as possible. A straightforward collection strategy would involve navigating the entire model containment tree, assessing whether each model element is of one of the types of interest and, if so, adding it to the appropriate cache(es).

However, by inspecting the example model in Figure 4, we observe that traversing the containment closure of the *modules* reference of the "Computer Science" Department model element is guaranteed not to reveal any model elements of interest (according to the metamodel of Figure 3 modules can only contain lectures and neither of these types of elements are of interest to the query). This observation can be generalised and exploited to prune the subset of the containment tree that the engine will need to visit in order to populate the caches of interest.

To achieve this we need to analyse the metamodel and compute the subset of containment references that can potentially lead to elements of interest. The proposed algorithm is illustrated in Algorithm 1. Please note that the algorithm has been simplified for presentation purposes and that implementations of the algorithm need to make use of memoisation to avoid infinite recursion that can be caused by circular containment references of no interest. Adding the computed containment references that need to be traversed at runtime to the (incomplete) cache configuration model of Figure 5, produces the (complete) configuration model of Figure 6.

3.4 Instance Collection and Caching

Having computed the cache configuration model, the final step includes traversing only the identified containment references of the in-memory model at runtime in a top-down recursive manner to collect and cache the elements of interest.

For example, with reference to the example model of Figure 4, the instance collection process starts at the top-level *:University* element. The element's EClass is not linked to the cache configuration via one of its *allOfType* or *allOfKind* references, and as such the element is not cached. Navigating the university's *departments* reference reveals a *:Department* element, which also does not need to be cached. The process does not need to navigate the department's *modules* reference as it is not linked to the cache configuration via the latter's *traverse* reference, and as such it proceeds with its *members* reference. Traversing the *members* reference reveals an instance of *Student* and an instance of *Staff*, both of which are cached in preparation for the *Member.allOfKind()* invocation. Similarly, the *webpage* reference of :Staff is traversed and reveals a *:WebPage*, which is also cached in preparation for the *WebPage.allOfType()* invocation.

```
let cm = the initial version of the configuration cache model;
let p = the EPackage that the model conforms to;
let refs = empty list of EReferences;

foreach non-abstract EClass c in p do
    foreach containment EReference r of c do
        call shouldBeTraversed(r);
    end
end

function shouldBeTraversed(r : EReference) : Boolean
    let types = transitive closure of r's type and all its sub-types;
    if types includes any of the EClasses in cm then
        add r to refs;
        return true;
    end
    else
        foreach containment EReference tr of each of the types do
            if shouldBeTraversed(tr) then
                return true;
            end
        end
        return false;
    end
end
```

Algorithm 1: Containment Reference Selection Algorithm

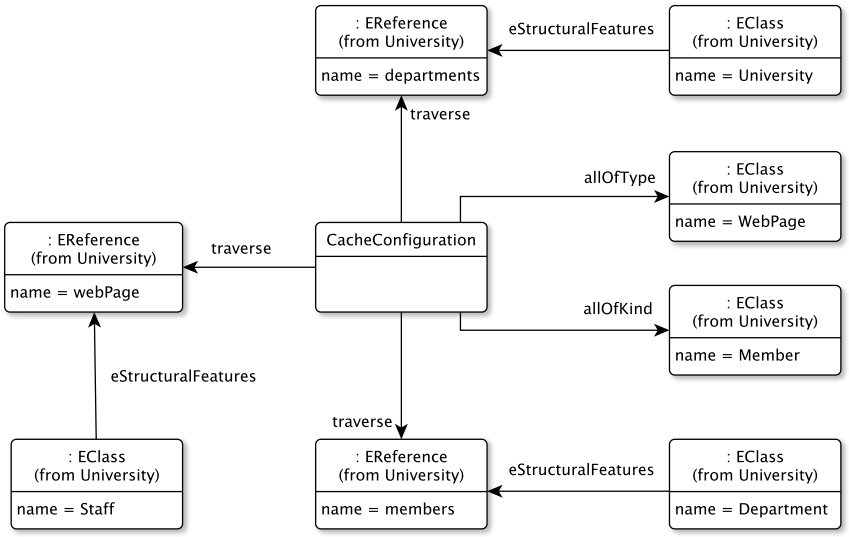

Fig. 6. Complete Cache Configuration Model

4 Evaluation

In this section we report on the results of benchmarks performed on four different strategies for computing *allInstances()*.

1. Lazy (on-demand) computation (L)
2. Greedy pre-caching (G)[1]
3. Type-aware pre-caching (T)[2]
4. Type-and-reference-aware pre-caching (TR)

Benchmarks were performed on a computer with Intel(R) Core(TM) i7 CPU @ 2.3GHz, with 8GB of physical memory, running OS X Yosemite. The version of the Java Virtual Machine used was 1.8.0_31-b13. Results are in seconds.

For our benchmarks, models of varying sizes obtained from reverse engineered Java code in the 2009 GraBaTs contest[3] are used. These models, named set0, set1, set2, set3 and set4 (9.2MB, 27.9MB, 283.2MB, 626.7MB, 676.9MB respectively) are stored in XMI 2.0 format and have been used for various benchmarks for different tools [6, 7].

4.1 Model Element Coverage

To quantify model coverage in our benchmarks, we counted the number of elements in each data set and then automatically generated EOL programs which exercise 20%, 40%, 60%, 80% and 100% of the total elements for each data set. An example generated EOL program is provided in Listing 1.2.

We then executed all the generated EOL programs and measured performance in terms of the time taken to load the models with the four different strategies and the time taken to execute the programs.

Listing 1.2. An example of generated EOL program for model element coverage

```
1 var size = 0;
2 var methodInvocation = MethodInvocation.all.first();
3 size = size + MethodInvocation.all.size();
4 var qualifiedName = QualifiedName.all.first();
5 size = size + QualifiedName.all.size();
6 ...
7 size.println();
```

[1] As discussed in Section 2, this approach naively pre-computes all possible *allOfType* and *allOfKind* caches.

[2] This approach makes use of static analysis as discussed in Section 3.2 but does not prune containment references and as such it needs to visit the entire containment tree at runtime. It is included in this benchmark only to assess the additional benefits of containment reference pruning.

[3] GraBaTs2009: 5th Int. Workshop on Graph-Based Tools, `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/`

4.2 Results

The obtained results are presented in Table 1. Initials *L*, *G*, *T* and *TR* represents the approaches aforementioned (Lazy, Greedy, Type-Aware and Type-and-Reference-Aware). Since the execution time of the EOL programs for *G*, *T* and *TR* is practically the same[4], we only present one result for all three of them under the * columns. *Imp.* represents the performance improvement of a certain approach, *Load* represents the time it takes to load the models, whereas *Exec.* represents the time it takes to execute the EOL programs. Finally, *Total* represents the time it takes to load the model and execute an EOL program for a single experiment.

From the benchmarks we observe that with the *Greedy*, *Type-Aware* and *Type-and-Reference-Aware* approaches, programs execute significantly faster than with the *Lazy* approach. These approaches require more time to load the models due to the overhead incurred by their respective caching logic; such overhead affects the performance for small data sets (set 0 in this case). However, as the size of models gets larger, these approaches provide marginal benefits in terms of the time it takes to load a model and to execute an EOL program (total time). In general, *TR* provides better performance but for some cases in which *TR* needs to visit elements deep in the containment tree, *T* and *G* marginally outperform it. In terms of memory footprint, the three approaches behave very similarly and incur a small linear overhead compared to *L*.

5 Related Work

Several database-based model persistence prototypes have been proposed for persisting and loading large models, including Morsa [8], Neo4EMF [7], MongoEMF [9], EMF Fragments [10] and Hawk [6]. The general idea behind these prototypes is that they are able to load only the parts of a model that are needed for the task at hand (e.g. to compute particular queries), so that large models can be accessed efficiently both in terms of loading time and memory consumption.

Computing *allInstances()* in such systems typically does not require traversing the entire model and can be achieved through efficient internal queries expressed in the underpinning database's native query language (e.g. SQL, Cypher). Despite the clear technical advantages of database-based technologies, there are still valid reasons for using file-based formats (e.g. XMI) for model persistence in certain contexts, such as standards-compliance, tool interoperability, and compatibility with existing file-based version control systems such as Git and Subversion.

[4] This is expected as all three strategies populate all caches required before the EOL program executes.

Table 1. Benchmark results for Lazy, Greedy, Type-Aware, Type-and-Reference-Aware caching (* in the table represents the results for G,T and TR collectively).

Perc.	L		G	T	TR	*	Imp.G	Imp.T	Imp.TR	Imp.*
	Load sec.	Exec.	Load sec.	Load sec.	Load sec.	Exec. sec.	Total %	Total %	Total %	Exec. %
					Set0					
20%	0.552	0.015	0.652	0.554	0.572	0.001	-15.17%	2.12%	-1.06%	93.33%
40%	0.555	0.007	0.631	0.572	0.561	0.002	-12.63%	-2.14%	-0.18%	71.43%
60%	0.549	0.012	0.645	0.571	0.573	0.003	-15.51%	-2.32%	-2.67%	75.00%
80%	0.543	0.026	0.652	0.573	0.576	0.005	-15.47%	-1.58%	-2.11%	80.77%
100%	0.552	0.141	0.638	0.623	0.619	0.013	6.06%	8.23%	8.80%	90.78%
Perc.					**Set1**					
20%	1.643	0.606	1.856	1.653	1.672	0.01	17.03%	26.06%	25.21%	98.35%
40%	1.596	0.595	1.875	1.736	1.711	0.011	13.92%	20.26%	21.41%	98.15%
60%	1.587	0.556	1.843	1.786	1.773	0.013	13.39%	16.05%	16.66%	97.66%
80%	1.611	0.571	1.86	1.787	1.788	0.017	13.98%	17.32%	17.28%	97.02%
100%	1.606	0.626	1.866	1.852	1.852	0.021	15.46%	16.08%	16.08%	96.65%
Perc.					**Set2**					
20%	14.159	2.244	17.169	14.802	14.809	0.007	-4.71%	9.72%	9.68%	99.69%
40%	14.061	4.402	17.979	16.587	16.613	0.015	2.54%	10.08%	9.94%	99.66%
60%	14.456	3.305	16.96	16.276	15.851	0.02	4.40%	8.25%	10.64%	99.39%
80%	15.151	5.685	18.145	17.724	18.217	0.03	12.77%	14.79%	12.43%	99.47%
100%	15.223	6.2	17.32	17.769	17.839	0.036	18.98%	16.89%	16.56%	99.42%
Perc.					**Set3**					
20%	34.199	8.706	38.096	34.17	33.753	0.017	11.17%	20.32%	21.29%	99.80%
40%	31.786	9.756	37.552	35.086	34.809	0.028	9.54%	15.47%	16.14%	99.71%
60%	31.835	12.222	37.528	36.516	35.662	0.045	14.72%	17.01%	18.95%	99.63%
80%	32.417	11.456	39.301	39.302	37.795	0.068	10.27%	10.26%	13.70%	99.41%
100%	35.872	13.7	38.659	40.779	40.513	0.071	21.87%	17.59%	18.13%	99.48%
Perc.					**Set4**					
20%	36.133	7.586	43.745	39.477	37.278	0.018	-0.10%	9.66%	14.69%	99.76%
40%	37.99	12.973	43.515	41.044	41.01	0.039	14.54%	19.39%	19.45%	99.70%
60%	36.457	14.131	44.883	42.348	41.055	0.05	11.18%	16.19%	18.75%	99.65%
80%	37.782	11.762	41.932	44.038	45.168	0.065	15.23%	10.98%	8.70%	99.45%
100%	37.617	14.563	44.813	46.914	43.406	0.078	13.97%	9.94%	16.67%	99.46%

6 Conclusion and Future Work

In this paper we have proposed a novel approach for computation and caching of *allInstances()*-like operations (e.g. used in declarative model transformation rules) on in-memory EMF models. We have compared the proposed approach against three alternative approaches via extensive benchmarking and demonstrated the benefits it delivers in terms of aggregate model loading and query execution time. Such an approach brings benefits only to model management programs which trigger multiple calls to *allInstances()*.

In future iterations of this work, we wish to investigate how static analysis and metamodel introspection can be used to further improve performance of computationally-expensive queries at runtime (e.g. by constructing and maintaining in-memory indexes that can improve the performance of collection filtering operations applied to the results of *allInstances()*).

References

1. Parastoo Mohagheghi, Miguel A Fernandez, Juan A Martell, Mathias Fritzsche, and Wasif Gilani. MDE adoption in industry: challenges and success criteria. In *Models in Software Engineering*, pages 54–59. Springer, 2009.
2. Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context. In *Model Driven Engineering Languages and Systems*, pages 476–491. Springer, 2005.
3. Dimitrios S. Kolovos, Richard F. Paige, and Fiona AC Polack. Scalability: The holy grail of model driven engineering. In *ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering*, pages 10–14, 2008.
4. Marcel Van Amstel, Steven Bosems, Ivan Kurtev, and Luís Ferreira Pires. Performance in model transformations: experiments with ATL and QVT. In *Theory and Practice of Model Transformations*, pages 198–212. Springer, 2011.
5. Dimitrios S. Kolovos, Richard F Paige, and Fiona AC Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture–Foundations and Applications*, pages 128–142. Springer, 2006.
6. Konstantinos Barmpis and Dimitris Kolovos. Hawk: Towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 6. ACM, 2013.
7. Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4EMF, a scalable persistence layer for EMF models. In *Modelling Foundations and Applications*, pages 230–241. Springer, 2014.
8. Javier Espinazo Pagán, Jesúss Sánchez Cuadrado, and Jesús García Molina. Morsa: A scalable approach for persisting and accessing large models. In *Model Driven Engineering Languages and Systems*, pages 77–92. Springer, 2011.
9. Bryan Hunt. MongoEMF, 2014, https://github.com/BryanHunt/mongo-emf/wiki.
10. Markus Scheidgen. Reference representation techniques for large models. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 5. ACM, 2013.