PrefetchML: a Framework for Prefetching and Caching Models

Gwendal Daniel
AtlanMod Team - Inria, Mines Nantes & Lina
4, Rue Alfred Kastler
Nantes, France
gwendal.daniel@inria.fr

Gerson Sunyé
AtlanMod Team - Inria, Mines Nantes & Lina
4, Rue Alfred Kastler
Nantes, France
gerson.sunye@inria.fr

Jordi Cabot
ICREA
UOC
Av. Carl Friedrich Gauss, 5
Castelldefels, Spain
jordi.cabot@icrea.cat

ABSTRACT

Prefetching and caching are well-known techniques integrated in database engines and file systems in order to speed-up data access. They have been studied for decades and have proven their efficiency to improve the performance of I/O intensive applications. Existing solutions do not fit well with scalable model persistence frameworks because the prefetcher operates at the data level, ignoring potential optimizations based on the information available at the metamodel level. Furthermore, prefetching components are common in relational databases but typically missing (or rather limited) in NoSQL databases, a common option for model storage nowadays. To overcome this situation we propose PrefetchML, a framework that executes prefetching and caching strategies over models. Our solution embeds a DSL to precisely configure the prefetching rules to follow. Our experiments show that PrefetchML provides a significant execution time speedup. Tool support is fully available online.

Keywords

Prefetching; MDE; DSL; Scalability; Persistence Framework; NoSQL

1. INTRODUCTION

Prefetching and caching are two well-known approaches to improve performance of applications that rely intensively on I/O accesses. Prefetching consists in bringing objects into memory before they are actually requested by the application to reduce performance issues due to the latency of I/O accesses. Fetched objects are then stored in memory to speed-up their (possible) access later on. In contrast, caching aims at speeding up the access by keeping in memory objects that have been already loaded.

Prefetching and caching have been part of database management systems and file systems for a long time and have proved their efficiency in several use cases [23, 25]. P. Cao et al. [6] showed that integrating prefetching and caching strategies dramatically improves the performance of I/O-intensive applications. In short, prefetching mechanisms works by adding load instructions (according to prefetching rules derived by static [16] or execution trace analysis [8]) into an existing program. Global policies, (e.g., LRU - least recently used, MRU - most recently used, etc.) control the cache contents.

Currently, there is lack of support for prefetching and caching at the model level. Given that model-driven engineering (MDE) is progressively adopted in the industry [15, 21] such support is required to raise the scalability of MDE tools dealing with large models where storing, editing, transforming, and querying operations are major issues [19, 28]. These large models typically appear in various engineering fields, such as civil engineering [1], automotive industry [4], product lines [24], and in software maintenance and evolution tasks such as reverse engineering [5].

Existing approaches have proposed scalable model persistence frameworks on top of SQL and NoSQL databases [11, 13, 17, 22]. These frameworks use lazy-loading techniques to load into main memory those parts of the model that need to be accessed. This helps dealing with large models that would otherwise not fit in memory but adds an execution time overhead due to the latency of I/O accesses to load model excerpts from the database, specially when executed in a distributed environment.

In this sense, this paper proposes a new prefetching and caching framework for models. We present *PrefetchML*, a domain specific language and execution engine, to specify prefetching and caching policies and execute them at runtime in order to optimize model access operations. This DSL allows designers to customize the prefetching rules to the specific needs of model manipulation scenarios, even providing several execution plans for different use cases. Our framework is built on top of the Eclipse Modeling Framework (EMF) infrastructure and therefore it is compatible with existing scalable model persistence approaches, regardless whether those backends also offer some kind of internal prefetching mechanism. A special version tailored to the NeoEMF/Graph [3] engine is also provided for further performance improvements. The empirical evaluation of PrefetchML highlights the significant time benefits it achieves.

The paper is organized as follows: Section 2 introduces further the background of prefetching and caching in the modeling ecosystem while Section 3 introduces the PrefetchML DSL. Section 4 describes the framework infrastructure

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MODELS '16, October 02-07, 2016, Saint-Malo, France
© 2016 ACM. ISBN 978-1-4503-4321-3/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2976767.2976775

and its rule execution algorithm and Section 5 introduces the editor that allows the designer to define prefetching and caching rules, and the implementation of our tool and its integration with the EMF environment. Finally, Section 6 presents the benchmarks used to evaluate our prefetching tool and associated results. Section 7 ends the paper by summarizing the key points and presenting our future work.

2. STATE OF THE ART

Prefetching and caching techniques are common in relational and object databases [25] in order to improve query computation time. Their presence in NoSQL databases is much more limited, which is problematic due to the increasing popularity of this type of databases as model storage solution. Moreover, database-level prefetching and caching strategies do not provide fine-grained configuration of the elements to load according to a given usage scenario–such as model-to-model transformation, interactive editing, or model validation–and are often strongly connected to the data representation, making them hard to evolve and reuse.

Scalable modeling frameworks are built on top of relational or NoSQL databases to store and access large models [3, 11]. These approaches are often based on lazy-loading strategies to optimize memory consumption by loading only the accessed objects from the database. While lazy-loading approaches have proven their efficiency in terms of memory consumption to load and query very large models [9,22], they perform a lot of fragmented queries on the database, thus adding a significant execution time overhead. For the reasons described above, these frameworks cannot benefit from database prefetching solutions nor they implement their own mechanism, with the partial exception of CDO [11] that provides some basic prefetching and caching capabilities[1]. For instance, CDO is able to bring into memory several objects of a list at the same time, or loading nested/related elements up to a given depth. Nevertheless, alternative prefetching rules cannot be defined to adapt model access to different contexts nor it is possible to define rules with complex prefetching conditions.

Hartmann et al. [14] propose a solution to tackle scalability issues in the context of models@run.time by splitting models into chunks that are distributed across multiple nodes in a cluster. A lazy-loading mechanism allows to virtually access the entire model from each node. However, to the best of our knowledge the proposed solution does not provide prefetching mechanism, which could improve the performance when remote chunks are retrieved and fetched among nodes.

Optimization of query execution has also been targeted by other approaches not relying on prefetching but using a variety of other techniques. EMF-IncQuery [4] is an incremental evaluation engine that computes graph patterns over an EMF model. It relies on an adaptation of the RETE algorithm, and results of the queries are cached and incrementally updated using the EMF notification framework. While EMF-IncQuery can be seen as an efficient EMF cache, it does not aim to provide prefetching support, and cache management cannot be tuned by the designer. Hawk [2] is a model indexer framework that provides a query API. It stores models in an index and allows to query them using the EOL [18] query language. While Hawk provides an efficient backend-independent query language, it does not allow the definition of prefetching plans for the indexed models.

In summary, we believe no existing solution provides the following desired characteristics of an efficient and configurable prefetching and caching solution for models:

1. Ability to define/execute prefetching rules independently of the database backend.
2. Ability to define/execute prefetching rules transparently from the persistence framework layered on top of the database backend.
3. A prefetching language expressive enough to define rules involving conditions at the type and instance level (i.e. loading all instances of a class A that are linked to a specific object of a class B).
4. A context-dependent prefetching language allowing the definition of alternative prefetching and caching plans for specific use cases.
5. A readable prefetching language enabling designers to easily tune the prefetching and caching rules.

In the following, we present PrefetchML, our prefetching and caching framework that tackles these challenges.

3. THE PREFETCHML DSL

PrefetchML is a DSL that describes prefetching and caching rules over models. Rules are triggered when an event satisfying a particular condition is received. These events can be the initial model loading, an access to a specific model element, the setting of a value or the deletion of a model element. Event conditions are expressed using OCL guards.

Loading instructions are also defined in OCL. The set of elements to be loaded as a response to an event are characterized by means of OCL expressions that navigate the model and select the elements to fetch and store in the cache. Not only loading requests can be defined, the language also provides an additional construct to control the cache content by removing cache elements when a certain event is received. Using OCL helps us to be independent of any specific persistence framework.

Prefetching and caching rules are organized in plans, that are sets of rules that should be used together to optimize a specific usage scenario for the model since different kinds of model accesses may require different prefetching strategies. For example, a good strategy for an interactive model browsing scenario is to fetch and cache the containment structure of the model, whereas for a complex query execution scenario it is better to have a plan that fit the specific navigation path of the query.

Beyond a set of prefetching rules, each plan defines a cache that can be parametrized, and a caching policy that manages the life-cycle of the cached elements.

In what follows, we first introduce a running example and then we formalize the abstract and concrete syntax of the PrefetchML DSL. Next Section will introduce how these rules are executed as part of the prefetching engine.

3.1 Running Example

In order to better illustrate the features of PrefetchML, we introduce a simple example model. Figure 1 shows a small excerpt of the *Java* metamodel provided by MoDisco [5].

[1]https://wiki.eclipse.org/CDO/Tweaking_Performance

A *Java* program is described in terms of *Packages* that are named containers that group *ClassDeclarations* through their *ownedElements* reference. A *ClassDeclaration* contains a *name* and a set of *BodyDeclarations*. *BodyDeclarations* are also named, and its *visibility* is described by a single *Modifier*. *ClassDeclarations* maintain a reference to their *CompilationUnit* (the physical file that stores the source code of the class). This *CompilationUnit* has a *name*, a list of *Comments*, and a list of *imported ClassDeclarations* (corresponding to the **import** clauses in Java programs).

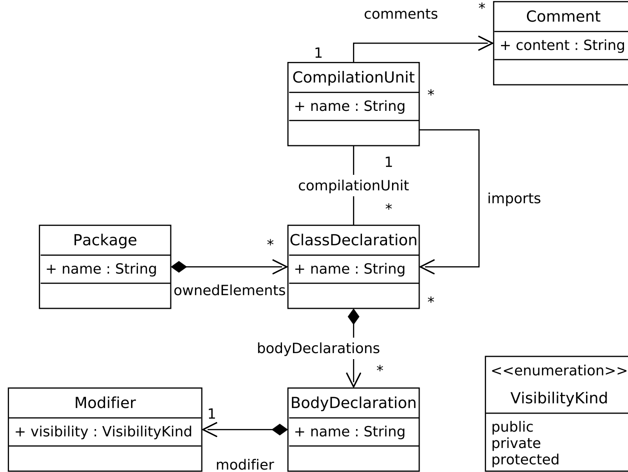

Figure 1: Excerpt of Java Metamodel

Listing 1 presents three sample OCL queries that can be computed over an instance of the previous metamodel: the first one returns the *Package* elements that do not contain any *ClassDeclaration* through their *ownedElements* reference. The second one returns from a given *ClassDeclaration* all its contained *BodyDeclarations* that have a private *Modifier*, and the third one returns from a *ClassDeclaration* a sequence containing the *return Comment* elements in the *ClassDeclarations* that are imported by the *CompilationUnit* associated to the current element.

```
context Package
def : isAnEmptyPackage : Boolean =
     self.ownedElements→isEmpty()

context ClassDeclaration
def : privateBodyDeclarations : Sequence(
     BodyDeclaration) =
     self.bodyDeclarations
       →select(bd | bd.modifier = VisibilityKind::
           Private)

context ClassDeclaration
def : importedComments : Sequence(Comment) =
     self.compilationUnit.imports.compilationUnit.
         comments
       →select(c | c.content.contains('@return')))
```

Listing 1: Sample OCL Query

3.2 Abstract Syntax

This section describes the main concepts of PrefetchML focusing on the different types of rules it offers and how they can be combined to create a complete prefetch specification.

Figure 2 depicts the metamodel corresponding to the abstract syntax of the PrefetchML language. A *PrefetchSpecification* is a top-level container that *imports* several *Metamodels*. These metamodels represent the domain on which prefetching and caching rules are described, and are defined by their *Unified Resource Identifier (URI)*.

The imported *Metamodels* concepts (classes, references, attributes) are used in prefetching *Plans*, which are named entities that group rules that are applied in a given execution context. A *Plan* can be the *default* plan to execute in a *PrefetchSpecification* if no execution information is provided.

Each *Plan* contains a *CacheStrategy*, which represents the information about the cache policy the prefetcher applies to keep loaded objects into memory. Currently, available cache strategies are *LRUCache* (Least Recently Used) and *MRUCache* (Most Recently Used). These *Caches* define two parameters: the maximum number of objects they can store (*size*), and the number of elements to free when it is full (*chunkSize*). In addition, a *CacheStrategy* can contain a *tryFirst OCL expression*. This expression is used to customize the default cache replacement strategy: it returns a set of model elements that should be removed from the cache if it is full, overriding the selected caching policy.

Plans also contain the core components of the PrefetchML language: *PrefetchingRules* that describe tracked model events and the loading and caching instructions. We distinguish two kinds of *PrefetchingRules*:

- *StartingRules* that are prefetching instructions triggered only when the prefetching plan is loaded
- *ObjectRules* that are triggered when an element satisfying a given condition is accessed, deleted, or updated.

ObjectRules can be categorized in three different types: *Access* rules, that are triggered when a particular model element is accessed, *Set* rules that correspond to the setting of an attribute or a reference, and *Delete* rules, that are triggered when an element is deleted or simply removed from its parent. When to fire the trigger is also controlled by the *sourceContext* class, that represents the type of the elements that could trigger the rule. This is combined with the *sourceExpression* (i.e. the guard for the event) to decide whether an object matches the rule.

All kinds of *PrefetchingRules* contain a *targetExpression*, that represents the elements to load when the rule is triggered. This expression is an *OCLExpression* that navigates the model and returns the elements to load and cache. Note that if *self* is used as the *targetExpression* of an *AccessRule* the framework will behave as a standard cache, keeping in memory the accessed element without fetching any additional object.

It is also possible to define *removeExpressions* in *PrefetchingRules*. When a *removeExpression* is evaluated, the prefetcher marks as free all the elements it returns from its cache. Each *removeExpression* is associated to a *removeContext Class*, that represents the context of the OCL expression. *remove* expressions can be coupled with the *tryFirst* expression contained in the *CacheStrategy* to tune the default replacement policy of the cache.

3.3 Concrete Syntax

We introduce now the concrete syntax of the PrefetchML language, which is derived from the abstract syntax metamodel presented in Figure 2. Listing 2 presents the grammar

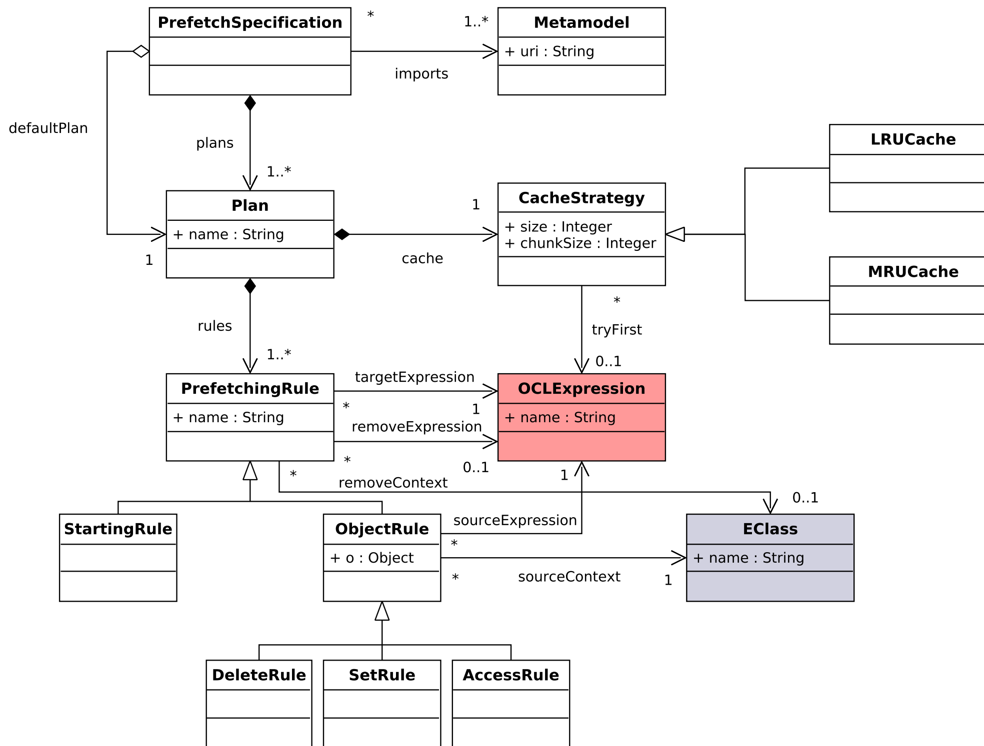

Figure 2: Prefetch Abstract Syntax Metamodel

of the PrefetchML language expressed using XText [12], an EBNF-based language used to specify grammars and generate an associated toolkit containing a metamodel of the language, a parser, and a basic editor. The grammar defines the keywords associated with the constructs presented in the PrefetchML metamodel. Note that *OCLExpressions* are parsed as *Strings*, the model representation of the queries presented in Figure 2 is computed by parsing it using the Eclipse MDT OCL toolkit[2]

```
grammar fr.inria.atlanmod.Prefetching
with org.eclipse.xtext.common.Terminals
import "http://www.inria.fr/atlanmod/Prefetching"

PrefetchSpecification:
  metamodel=Metamodel
  plans+=Plan+
;

Metamodel:
  'import' nsURI=STRING
;

Plan:
  'plan' name=ID (default?='default')? '{'
    cache=CacheStrategy
    rules+=(StartingRule | AccessRule)*
  '}'
;

CacheStrategy:
  (LRUCache{LRUCache} | MRUCache{MRUCache})
    (properties=CacheProperties)? ('when␣full␣
        remove␣' tryFirstExp=OCLExpression)?
;

LRUCache:
  'use␣cache' 'LRU'
;

MRUCache:
  'use␣cache' 'MRU'
;

CacheProperties:
  '[' 'size'size=INT ('chunk'chunk=INT)? ']'
;

PrefetchingRule:
  (StartingRule | AccessRule | DeleteRule |
      SetRule)
;

StartingRule:
  'rule' name=ID ':' 'on␣starting'
  'fetch' targetPatternExp=OCLExpression
  ('remove␣' 'type'removeType=ClassifierExpression
      removePatternExp=OCLExpression)?
;

AccessRule:
  'rule' name=ID ':' 'on␣access'
  'type'sourceType=ClassifierExpression (
      sourcePatternExp=OCLExpression)?
  'fetch' targetPatternExp=OCLExpression
  ('remove␣' 'type'removeType=ClassifierExpression
      removePatternExp=OCLExpression)?
;

DeleteRule:
  'rule' name=ID ':' 'on␣delete'
  'type'sourceType=ClassifierExpression (
      sourcePatternExp=OCLExpression)?
  'fetch' targetPatternExp=OCLExpression
  ('remove␣' 'type'removeType=ClassifierExpression
      removePatternExp=OCLExpression)?
;

SetRule:
  'rule' name=ID ':' 'on␣set'
```

[2]http://www.eclipse.org/modeling/mdt/?project=ocl

```
'type'sourceType=ClassifierExpression (
      sourcePatternExp=OCLExpression)?
  'fetch' targetPatternExp=OCLExpression
  ('remove␣' 'type'removeType=ClassifierExpression
        removePatternExp=OCLExpression)?
;

OCLExpression: STRING ;

ClassifierExpression: ID;
```

Listing 2: PrefetchML Language Grammar

Listing 3 provides an example of a *PrefetchSpecification* written in PrefetchML. To continue with our running example, the listing displays prefetching and caching rules suitable for the queries expressed in the running example (Listing 1).

The *PrefetchSpecification* imports the Java *Metamodel* (line 1). This *PrefetchSpecification* contains a *Plan* named *samplePlan* that uses a *LRUCache* that can contain up to 100 elements and removes them by chunks of 10 (line 4). It is also composed of three *PrefetchingRules*: the first one, `r1` (5-6), is a starting rule that is executed when the plan is activated, and loads and caches all the *Package* classes. The rule `r2` (7-8) is an access rule that corresponds to the prefetching and caching actions associated to the query *PrivateBodyDeclarations*. It is triggered when a *ClassDeclaration* is accessed, and loads and caches all the *BodyDeclarations* and *Modifiers* it contains. The rule `r3` (9-11) corresponds to the query *ImportedComments*: it is also triggered when a *ClassDeclaration* is accessed, and loads the associated *CompilationUnit*, and the *Comment contents* of its *imported ClassDeclarations*. The rule also defines a `remove` expression, that removes all the *Package* elements from the cache when the load instruction is completed.

```
1   import "http://www.example.org/Java"
2
3   plan samplePlan {
4     use cache LRU[size=100,chunk=10]
5     rule r1 : on starting fetch
6       Package.allInstances()
7     rule r2 : on access type ClassDeclaration fetch
8       self.bodyDeclarations.modifier
9     rule r3 : on access type ClassDeclaration fetch
10      self.compilationUnit.imports.compilationUnit.
          comments.content
11      remove type Package
12  }
```

Listing 3: Sample Prefetching Plan

4. PREFETCHML FRAMEWORK INFRASTRUCTURE

In this Section we present the infrastructure of the PrefetchML framework and its integration in the modeling ecosystem (details on its integration on specific modeling frameworks are provided in the next section). We also detail how prefetching rules are handled and executed using the running example presented in the previous Section.

4.1 Architecture

Figure 3 shows the integration of the **PrefetchML** framework in a typical modeling framework infrastructure: grey nodes represent standard model access components: a model-based tool accesses a model through a modeling API, which delegates to a persistence framework in charge of handling the physical storage of the model (for example in XML files, or in a database).

In contrast, the PrefetchML framework (white nodes) receives events from the modeling framework. When the events trigger a prefetching rule, it delegates the actual computation to its **Model Connector**. This component interacts with the modeling framework to retrieve the requested object, typically by translating the OCL expressions in the prefetching rules into lower level calls to the framework API. Section 5 discusses two specific implementations of this component.

The PrefetchML framework also intercepts model elements accesses, in order to search first in its **Cache** component if the requested objects are already available. If the cache contains the requested information, it is returned to the modeling framework, bypassing the persistence framework and improving execution time.

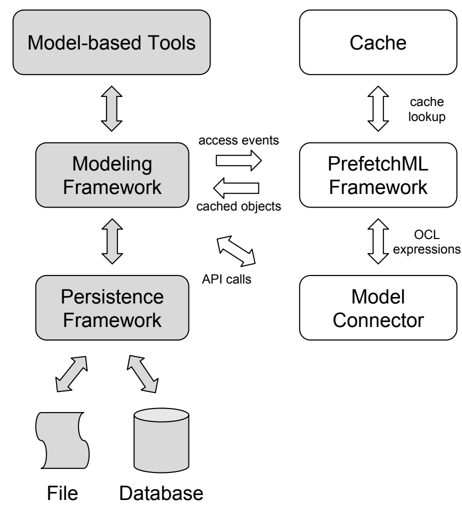

Figure 3: PrefetchML Integration in MDE Ecosystem

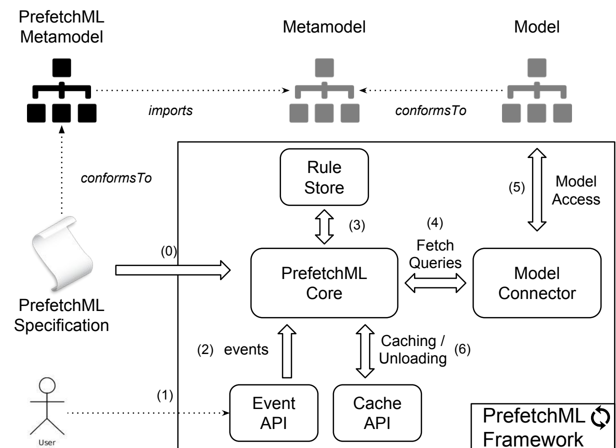

Figure 4: Prefetch Framework Infrastructure

Figure 4 describes the internal structure of the PrefetchML Framework. As explained in Section 3, a PrefetchML specifi-

cation *conforms to* the PrefetchML metamodel. This specification *imports* also the metamodel/s for which we are building the prefetching plans.

The **Core** component of the PrefetchML framework is in charge of loading, parsing and storing these *PrefetchML specifications* and then use them to find and retrieve the prefetching / caching rules associated with an incoming event, and, when necessary, execute them. This component also contains the internal *cache* that retains fetched model elements in memory. The **Rule Store** is a data structure that stores all the object rules (access, update, delete) contained in the input *PrefetchML description*. The **Model Connector** component is in charge of the translation and the execution of *OCLExpressions* in the prefetching rules. This connector can work at the modeling framework level, meaning that it executes fetch queries using the modeling API itself, or at the database level, translating directly OCL expressions into database queries.

The **CacheAPI** component gives access to the cache contents to client applications. It allows manual caching and unloading operations, and provides configuration facilities. This API is an abstraction layer that unifies access to the different cache types that can be instantiated by the Core component. Note that in our architecture we integrated prefetching and caching solutions in the sense that the core component manages its own cache, where only prefetched elements are stored. While this may result in keeping in the cache objects that are not going to be recurrently used, using a LRU cache strategy allows the framework to get rid off them when memory is needed. In addition, the grammar allows to define a minimal cache that would act only as a storage mechanism for the immediate prefetched objects.

The **EventAPI** is the component that is in charge of receiving events from the client application. It provides an API to send access, delete, and update events. These events are defined at the object level, and contain contextual information of their encapsulated model element, such as its identifier, the reference or attribute that is accessed, and the index of the accessed element. These informations are then used by the Core Component to find the rules that match the event.

In particular, when an object event is sent to the PrefetchML framework (1), the *Event API* handles it and forwards it to the *Core Component*, which is in charge of triggering the associated prefetching and caching rule. To do that, the *Core Component* searches in the *Rule Store* the rules that corresponds to the event and the object that triggered it (3). Each *OCLExpression* in the rules is translated into fetch queries sent to the *Model Connector* (4), which is in charge of the actual query computation over the model (5). Query results are handled back by the *PrefetchML Core*, which is in charge of caching them and freeing the cache from previously stored objects.

As prefetching operations can be expensive to compute, the PrefetchML Framework runs in the background, and contains a pool of working threads that performs the fetch operations in parallel of the application execution. Model elements are cached asynchronously and available to the client application through the *CacheAPI*. Prefetching queries are automatically aborted if they take too much time and/or if their results are not relevant (according to the number of cache hits) in order to keep the PrefetchML Framework synchronized with the client application, e.g. preventing it from loading elements that are not needed anymore.

The PrefetchML framework infrastructure is not tailored to a particular data representation and can be plugged in any kind of model persistence framework that stores models conforming to the Ecore metamodel and provides an API rich enough to evaluate OCL queries. This includes for example EMF storage implementations such as XMI, but also scalable persistence layers built on top of the EMF, like NeoEMF [13], CDO [11], and Morsa [22].

4.2 Rule Processing

We now look at the PrefetchML engine from a dynamic point of view. Figure 5 presents the sequence diagram associated with the initialization of the PrefetchML framework. When initializing, the prefetcher starts by loading the *PrefetchDescription* to execute (1). To do so, it iterates through the set of plans and stores the rules in the *RuleStore* according to their type (2). In the example provided in Listing 3 this process saves in the store the rules `r2` and `r3`, both associated with the *ClassDeclaration* type. Then, the framework creates the cache (3) instance corresponding to the active prefetching plan (or the default one if no active plan is provided). This creates the `LRU` cache of the example, setting its `size` to 100 and its `chunkSize` to 10.

Next, the PrefetchML framework iterates over the *StartingRules* of the description and computes their *targetExpression* using the *Model Connector* (4). Via this component, the OCL expression is evaluated (in the example the target expression is `Package.allInstances()`) and the resulting elements are returned to the *Core* component (5) that creates the associated identifying keys (6) and stores them in the cache (7). Note that starting rules are not stored in the Rule Store, because they are executed only once when the plan is activated, and are no longer needed afterwards.

Once this initial step has been performed, the framework awaits object events. Figure 6 shows the sequence diagram presenting how the PrefetchML handles incoming events. When an object event is received (8), it is encapsulated into a working task which contains contextual information of the event (object accessed, feature navigated, and index of the accessed feature) and asynchronously sent to the prefetcher (9) that searches in the *RuleStore* the object rules that have the same type as the event (10). In the example, if a *ClassDeclaration* element is accessed, the prefetcher searches associated rules and returns `r2` and `r3`. As for the diagram above, the next calls involve the execution of the target expressions for the matched rules and saving the retrieved objects in the cache for future calls. Finally, the framework evaluates the remove OCL expressions (17) and frees the matching objects from the memory. In the example, this last step removes from the cache all the instances of the *Package* type.

5. TOOL SUPPORT

In this Section we present the tool support for the PrefetchML framework. It is composed of two main components: a language editor (Section 5.1) that supports the definition of prefetching and caching rules, and a execution engine with two different integration options: the EMF API and the NeoEMF/Graph persistence framework (Sections 5.2 and 5.3). The presented components are part of a set of open source Eclipse plugins available at https://github.com/atlanmod/Prefetching_Caching_DSL.

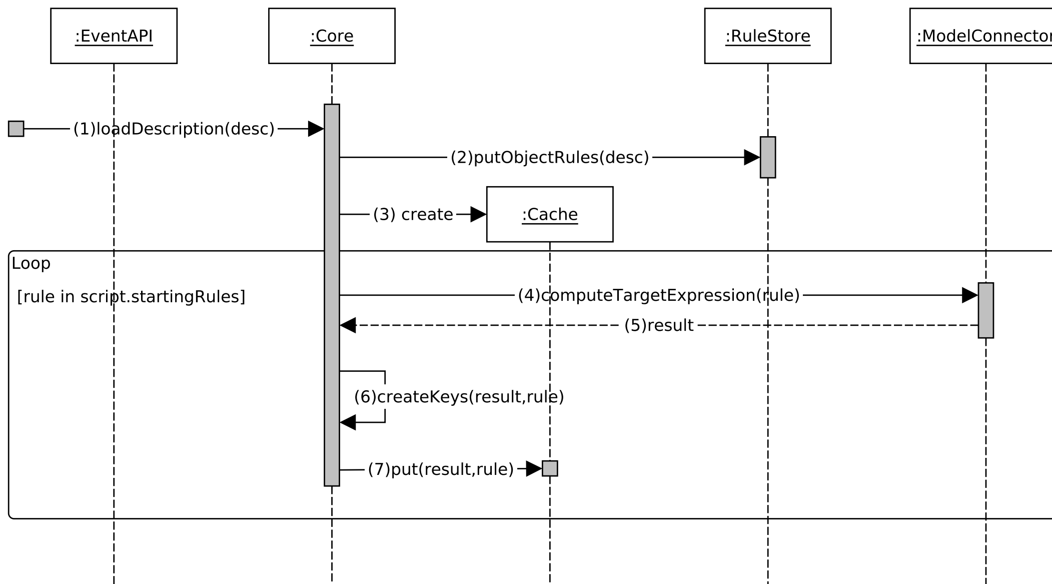

Figure 5: PrefetchML Initialization Sequence Diagram

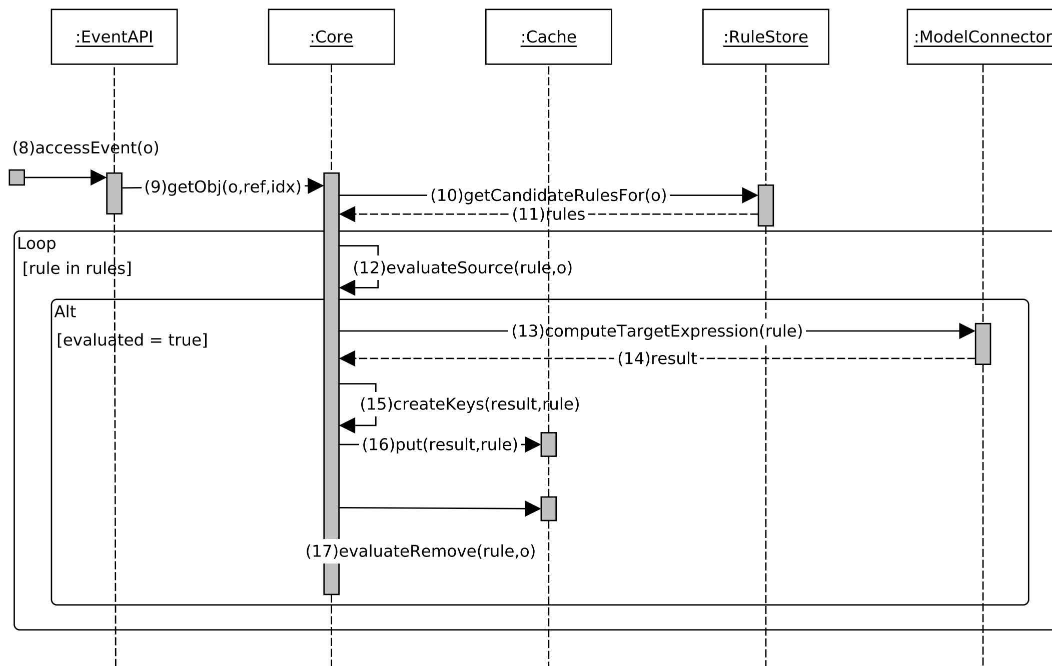

Figure 6: PrefetchML Event Handling Sequence Diagram

5.1 Language Editor

The PrefetchML language editor is an Eclipse-based editor that allows the creation and the definition of prefetching and caching rules. It is partly generated from the XText grammar presented in Section 3.3 and defines utility helpers to validate and navigate the imported metamodel. The editor supports navigation auto-completion by inspecting imported metamodels, and visual validation of prefetching and caching rules by checking reference and attribute existence.

Figure 7 shows an example of the PrefetchML editor that contains the prefetching and caching plan defined in the running example of Section 3.

5.2 EMF Integration

Figure 8 shows the integration of PrefetchML within the EMF framework. Note that only two components must

```
article_sample.prefetch

import "http://www.eclipse.org/MoDisco/Java/0.2.incubation/java-neoemf"

plan SamplePlan {
    use cache LRU[size 100 chunk 10]
    rule r1 : on starting
        fetch Package.allInstances()
    rule r2 : on access type ClassDeclaration
        fetch self.ownedElements.bodyDeclarations.modifier
    rule r3 : on access type ClassDeclaration
        fetch self.compilationUnit.imports.compilationUnit
            .comments.content
        remove type Package
}
```

Figure 7: PrefetchML Rule Editor

be adapted (light grey boxes). The rest are either generic PrefetchML components or standard EMF modules.

In particular, dark grey boxes represent the standard EMF-based model access architecture: an *EMF-based* tool accesses the model elements through the *EMF API*, that delegates the calls to the *PersistenceFramework* of choice (XMI, CDO, NeoEMF,...), which is finally responsible for the model storage.

The two added/adapted components are:

- An *Interceptor* that wraps the EMF API and captures the calls (1) to the EMF API (such as `eGet`, `eSet`, or `eUnset`). EMF calls are then transformed into *EventAPI* calls (2) by deriving the appropriate event object from the EMF API call. For example, an `eGet` call will be translated into the *accessEvent* method call (8) in Figure 6. Once the event has been processed, the *Interceptor* also searches in the cache the requested elements as indicated by the Model Connector (3). If they are available in the cache, they are directly returned to the EMF-based tool. Otherwise, the Interceptor passes on the control to the EMF API to continue the normal process.

- An *EMF Model Connector* that translates the OCL expressions in the prefetching and caching rules into lower-level EMF API calls. The results of those queries are stored in the cache, ready for the Interceptor to request them when necessary.

This integration makes event creation and cache accesses totally transparent to the client application. In addition, it does not make any assumptions about the mechanism used to store the models, and therefore, it can be plugged on top of any EMF-based persistence solution.

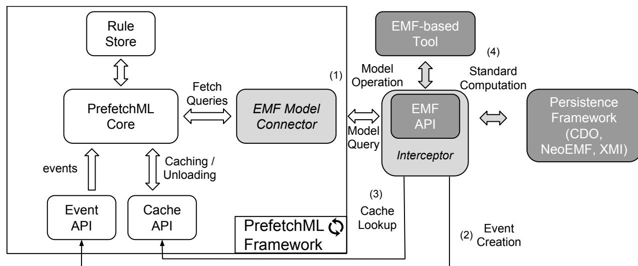

Figure 8: Overview of EMF-Based Prefetcher

5.3 NeoEMF/Graph Integration

To take advantage of the query facilities of graph databases (a proven good alternative to store large models) and make sure PrefetchML optimizes as much as possible the rule execution time in this context, we designed a Model Connector dedicated to NeoEMF/Graph, a persistence solution that stores EMF models into graph databases. Note that, PrefetchML can work with NeoEMF without this dedicated support by processing calls through the EMF API as explained in the previous section. Still, offering a native support allows for better optimizations.

NeoEMF/Graph is a scalable model persistence framework built on top of the EMF that aims at handling large models in graph databases [3]. It relies on the Blueprints API [26], which aims to unify graph database accesses through a common interface. Blueprints is the basis of a stack of tools that stores and serializes graphs, and provides a powerful query language called Gremlin [27]. NeoEMF/Graph relies on a *lazy-loading* mechanism that allows the manipulation of large models in a reduced amount of memory by loading only accessed objects.

The prefetcher implementation integrated in NeoEMF/-Graph uses the same mechanisms as the standard EMF one: it defines an *Interceptor* that captures the calls to the EMF API, and a dedicated *Graph Connector*. While the EMF Connector computes loading instructions at the EMF API level, the *Graph Connector* performs a direct translation from OCL into Gremlin, and delegates the computation to the database, enabling back-end optimizations such as uses of indexes, or query optimizers. The *Graph Connector* caches the results of the queries (i.e. database *vertices*) instead of the EMF objects, limiting execution overhead implied by object reifications. Since this implementation does not rely on the EMF API, it is able to evaluate queries significantly faster than the standard EMF prefetcher (as shown in our experimental results in Section 6), thus improving the throughput of the prefetching rule computation. Database vertices are reified into EMF objects when they are accessed from the cache, limiting the initial execution overhead implied by unnecessary reifications.

6. EVALUATION

In this Section, we evaluate the performance of our PrefetchML Framework by comparing the performance of executing a set of OCL queries when (i) no prefetching is used, (ii) EMF-based prefetching is active, or (iii) NeoEMF/Graph dedicated prefetching is active. In all three cases, the back-end to store the models to be queried is NeoEMF/Graph. This allows to test all three combinations with the same base configuration. To have a better overview of the performance gains, each prefetching mechanism is tested on two different models and with two cache sizes. We also repeat the process using a good and a bad prefetching plan. The latter aims to evaluate whether non-expert users choosing a wrong prefetching plan (e.g. one that for instance prefetches objects that will never be used since they are not involved in any of the queries / operations in the scenario) could harm tool efficiency a lot. Each query is executed twice on each combination to evaluate the benefit of the cache of PrefetchML in subsequent query executions.

Note that we do not compare our solution with existing tools that can be related to our one because we could not envision a fair comparison scenario. For instance, Moogle [20] is a model search approach that creates an index to retrieve

full models from a repository, where our solution aims to improve performances of queries at the model level. IncQuery [4] is also not considered as a direct competitor because it does not provide a prefetch mechanism. In addition, IncQuery was primarily designed to execute queries against models already in the memory which is a different scenario with a different trade-off.

Experiments are executed on a computer running Fedora 20 64 bits. Relevant hardware elements are: an Intel Core I7 processor (2.7 GHz), 16 GB of DDR3 SDRAM (1600 MHz) and a SSD hard-disk. Experiments are executed on Eclipse 4.5.2 (Mars) running Java SE Runtime Environment 1.7. To run our queries, we set the Java virtual machine parameters `-server` and `-XX:+UseConcMarkSweepGC` that are recommended in the Neo4j documentation.

6.1 Benchmark Presentation

The experiments are run over two large models automatically constructed by the MoDisco [5] Java Discoverer, which is a reverse engineering tool that computes low-level models from Java code. The two models used in the experiments are the result of applying MoDisco discovery tool over two Java projects: the MoDisco plugin itself, and the Java Development Tools (JDT) core plugin. Resulting models contain respectively 80 664 and 1 557 006 elements, and associated XMI files are respectively 20 MB and 420 MB large.

As sample queries on those models to use in the experiment we choose three query excerpts extracted from real MoDisco software modernization use cases:

- **BlockStatement**: To access all statements contained in a block
- **TypeToUnit**: To access a type declaration and navigate to all its imports and comments
- **ClassToUnit**: To extend the TypeToUnit query by navigating the body and field declarations of the input class

The first query performs a simple reference navigation from the input *Block* element and retrieves all its *Statements*. The second query navigates multiple references from the input *Type* element in order to retrieve the *Imports* and *Comments* contained in its *CompilationUnit*, and the third query extends it by adding filtering using the `select` expression, and by navigating the *BodyDeclarations* of the input *Class* element in order to collect the declared variables and fields[3].

Good prefetching plans have been created by inspecting the navigation path of the queries. The context type of each expression constitutes the source of *AccessRules*, and navigations are mapped to target patterns. *Bad* plans contain two types of prefetching and caching instructions: the first ones have a source pattern that is never matched during the execution (and thus should never be triggered), and the second ones are matched but loads and caches objects that are not used in the query.

The queries have been executed using two different cache configurations. The first one is a large MRU cache that can contain up to 20% of the input model (*C1*), and the second is a smaller MRU cache that can store up to 10% of the input model (*C2*). We choose this cache replacement policy according to Chou and DeWitt [7] who state MRU is the best replacement algorithm when a file is being accessed in a looping sequential reference pattern. In addition, we compare execution time of the queries when they are executed for the first time and after a warm-up execution to consider the impact of the cache on the performance.

Queries are evaluated over all the instances of the models that conform to the context of the query. In order to give an idea of the complexity of the queries, we present in Tables 1 and 2 the number of input elements for each query (**#Input**), the number of traversed element during the query computation (**#Traversed**) and the size of the result set for each model (**#Res**).

[3]Details of the queries can be found at https://github.com/atlanmod/Prefetching_Caching_DSL

Table 1: Experimental Set Details (MoDisco)

Query	#Input	#Traversed	#Res
BlockStatement	1837	4688	2851
TypeToUnit	348	1895	1409
ClassToUnit	166	3393	2953

Table 2: Experimental Set Details (JDT)

Query	#Input	#Traversed	#Res
BlockStatement	58484	199228	140744
TypeToUnit	1663	16387	13496
ClassToUnit	1347	48961	41925

6.2 Results

Table 3 presents the average execution time (in milliseconds) of 100 executions of the presented queries using Eclipse MDT OCL over the JDT and MoDisco models stored in the NeoEMF/Graph persistence framework. Columns are grouped according to the kind of prefetching that has been used. For each group we show the time when using the good plan with first cache size, the good plan with the second cache size and the bad plan with the first cache.

Each cell shows the execution time in milliseconds of the query the first time it is executed (Cold Execution). In this configuration the cache is initially empty, and benefits of prefetching depend only on the accuracy of the plan (to maximize the cache hits) and the complexity of the prefetching instructions (the more complex they are the more time the background process has to advance on the prefetching of the next objects to access). The second result shows the execution time of a second execution of the query (Warmed Execution), when part of the loaded elements has been cached during the first computation.

The correctness of query results has been validated by comparing the results of the different configurations with the ones of the queries executed without any prefetching enabled.

6.3 Discussion

The main conclusions we can draw from these results (Table 3) are

- EMF-based prefetcher improves the execution time of first time computations of queries that perform complex and multiple navigations for both JDT and MoDisco models (*ClassToUnit* query). However, when the query is simple such as *BlockStatement* or only contains in-

Table 3: Query Execution Time in milliseconds (Cold Execution / Warmed Execution)

Model	OCL Query	No Pref.	EMF Pref.			Graph Pref.		
			C1 (20%)	C2 (10%)	Inv	C1 (20%)	C2 (10%)	Inv
MoDisco	BlockStatement	2057/696	2193/169	2145/187	2134/722	1687/238	1688/233	2155/734
	TypeToUnit	1999/598	2065/83	2072/97	2045/615	1337/155	1418/165	2038/643
	ClassToUnit	2703/722	2588/169	2618/188	2798/758	1616/218	1664/233	2787/753
JDT	BlockStatement	15170/8521	16235/318	16700/3044	16180/8868	11688/1288	12446/4256	16234/8638
	TypeToUnit	6624/2267	6822/269	6768/336	6832/2347	5270/685	5338/685	6877/2331
	ClassToUnit	11637/5421	10780/229	10526/250	11946/5687	7716/873	7817/884	11789/5556

dependent navigations such as *TypeToUnit*, the EMF prefetcher results in a small execution overhead since the prefetch takes time to execute and with simple queries it cannot save time by fetching elements in the background while the query is processed.

- EMF-based prefetcher drastically improves the performance of the second execution of the query: an important part of the navigated objects is contained in the cache, limiting the database overhead.
- NeoEMF-based prefetcher is faster than the EMF one on the first execution because queries can benefit from the database query optimizations (such as indexes), to quickly prefetch objects to be used in the query when initial parts of the query are still being executed, i.e. the prefetcher is able to run faster than the computed query. This increases the number of cache hits in a cold setup (and thus the execution time)
- NeoEMF-based prefetcher is slower than the EMF-based one on later executions because it stores in the cache the vertices corresponding to the requested objects and not the objects themselves, therefore extra time is needed to reify those objects using a low-level query framework such as the Mogwaï [10]
- Wrong prefetcher plans are not dangerous. Prefetching does not add a significant execution time overhead and therefore results are in the same order of magnitude as when there is no prefetching at all.
- Too small caches reduce the benefits of Prefetching since we waste time checking for the existence of many objects that due to the cache size are not there any longer generating a lot of cache misses. Nevertheless, even with a small cache we improve efficiency after the initial object load.

To summarize our results, the PrefetchML framework is an interesting solution to improve execution time of model queries over EMF models. The gains in terms of execution time are positive, but results also show that the EMF prefetcher is not able to provide first-time improvement for each kind of query, and additional information has to be taken into account to provide an optimal prefetching strategy, such as the reuse of navigated elements inside a query, or the size of the cache.

7. CONCLUSIONS AND FUTURE WORK

We presented the PrefetchML DSL, an event-based language that describes prefetching and caching rules over models. Prefetching rules are defined at the metamodel level and allow designers to describe the event conditions to activate the prefetch, the objects to prefetch, and the customization of the cache policy. Since OCL is used to write the rule conditions, PrefetchML definitions are independent from the underlying persistence backend and storage mechanism.

Rules are grouped into plans and several plans can be loaded/unloaded for the same model, to represent fetching and caching instructions specially suited for a given specific usage scenario. Some automation/guidelines could be added to help on defining a good plan for a specific use-case in order to make the approach more user-friendly. However, our experiments have shown that even if users choose a bad plan the overhead is really small. The execution framework has been implemented on top of the EMF as well as NeoEMF/Graph, and results of the experiments show a significant execution time improvement compared to non-prefetching use cases.

PrefetchML satisfies all the requirements listed in Section 2. Prefetching and caching rules are defined using a high-level DSL embedding the OCL, hiding the underlying database used to store the model (1). The EMF integration also provides a generic way to define prefetching rules for every EMF-based persistence framework (2), like NeoEMF and CDO. Note that an implementation tailored to NeoEMF is also provided to enhance performance. Prefetching rules are defined at the metamodel level, but the expressiveness of OCL allows to refer to specific subset of model elements if needed (3). In Section 3 we presented the grammar of the language, and emphasized that several plans can be created to optimize different usage scenario (4). Finally, the PrefetchML DSL is a readable language that eases designers' task on writing and updating their prefetching and caching plan (5). Since the rules are defined at the metamodel level, created plans do not contain low-level details that would make plan definition and maintenance difficult.

As future work we plan to work on the automatic generation of PrefetchML scripts based on static analysis of available queries and transformations for the metamodel we are trying to optimize. Another information source to come up with prefetching plans is the dynamic discovery of frequent access patterns at the model level (e.g. adapting process mining techniques). This is a second direction we plan to explore since it could automatically enhance existing applications working on those models even if we do not have access to their source code and/or no prefetching plans have been created for them. Adding an adaptive behavior to PrefetchML may also allows to detect if a plan is relevant for a given scenario, and switch-on/off specific rules according to the context of the execution.

8. REFERENCES

[1] S. Azhar. Building information modeling (BIM): Trends, benefits, risks, and challenges for the AEC industry. *Leadership and Management in Engineering*, pages 241–252, 2011.

[2] K. Barmpis and D. Kolovos. Hawk: Towards a scalable model indexing architecture. In *Proc. of BigMDE'13*, pages 6–9. ACM, 2013.

[3] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4EMF, a Scalable Persistence Layer for EMF Models. In *Proc. of the 10th ECMFA*, pages 230–241. Springer, 2014.

[4] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental evaluation of model queries over EMF models. In *Proc. of the 13th MoDELS Conference*, pages 76–90. Springer, 2010.

[5] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A model driven reverse engineering framework. *IST*, pages 1012 – 1032, 2014.

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Performance Evaluation Review*, pages 188–197, 1995.

[7] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, pages 311–336, 1986.

[8] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *ACM SIGMOD Record*, pages 257–266. ACM, 1993.

[9] G. Daniel, G. Sunyé, A. Benelallam, and M. Tisi. Improving memory efficiency for processing large-scale models. In *Proc. of BigMDE'14*, pages 31–39. CEUR Workshop Proceedings, 2014.

[10] G. Daniel, G. Sunyé, and J. Cabot. Mogwaï: a framework to handle complex queries on large models. In *Proc. of the 10th RCIS Conference*. IEEE, 2016.

[11] Eclipse Foundation. The CDO Model Repository (CDO), 2016. URL: http://www.eclipse.org/cdo/.

[12] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proc. of OOPSLA'10*, pages 307–309, New York, NY, USA, 2010. ACM.

[13] A. Gómez, G. Sunyé, M. Tisi, and J. Cabot. Map-based transparent persistence for very large models. In *Proc. of the 18th FASE Conference*. Springer, 2015.

[14] T. Hartmann, A. Moawad, F. Fouquet, G. Nain, J. Klein, and Y. Le Traon. Stream my models: reactive peer-to-peer distributed models@ run. time. In *Proc. of the 18th MoDELS Conference*, pages 80–89. IEEE, 2015.

[15] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *Proc of the 33rd ICSE*, pages 633–642. IEEE, 2011.

[16] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *ACM SIGARCH Computer Architecture News*, pages 43–53. ACM, 1991.

[17] M. Koegel and J. Helming. EMFStore: a model repository for EMF models. In *Proc. of the 32nd ICSE*, pages 307–308. ACM, 2010.

[18] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon object language (EOL). In *Proc. of the 2nd ECMDA-FA*, pages 128–142. Springer, 2006.

[19] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, et al. A research roadmap towards achieving scalability in model driven engineering. In *Proc. of BigMDE'13*, pages 1–10. ACM, 2013.

[20] D. Lucrédio, R. P. d. M. Fortes, and J. Whittle. Moogle: A model search engine. In *Proc. of the 11th MoDELS Conference*, pages 296–310. Springer, 2008.

[21] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, and W. Gilani. MDE adoption in industry: challenges and success criteria. In *Proc. of Workshops at MoDELS 2008*, pages 54–59. Springer, 2009.

[22] J. E. Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A scalable approach for persisting and accessing large models. In *Proc. of the 14th MoDELS Conference*, pages 77–92. Springer, 2011.

[23] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. *Informed prefetching and caching*. ACM, 1995.

[24] R. Pohjonen and J.-P. Tolvanen. Automated production of family members: Lessons learned. In *Proc. of PLEES'02*, pages 49–57. IESE, 2002.

[25] A. J. Smith. Sequentiality and prefetching in database systems. *TODS*, pages 223–247, 1978.

[26] Tinkerpop. Blueprints API, 2016. URL: blueprints.tinkerpop.com.

[27] Tinkerpop. The Gremlin Language, 2016. URL: gremlin.tinkerpop.com.

[28] J. Warmer and A. Kleppe. Building a flexible software factory using partial domain specific models. In *Proc. of the 6th DSM Workshop*, pages 15–22. University of Jyvaskyla, 2006.