



HAL
open science

Advanced Prefetching and Caching of Models with PrefetchML

Gwendal Daniel, Gerson Sunyé, Jordi Cabot

► **To cite this version:**

Gwendal Daniel, Gerson Sunyé, Jordi Cabot. Advanced Prefetching and Caching of Models with PrefetchML. Software and Systems Modeling, 2018, pp.1-35. 10.1007/s10270-018-0671-8. hal-01725030

HAL Id: hal-01725030

<https://inria.hal.science/hal-01725030v1>

Submitted on 7 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Advanced Prefetching and Caching of Models with PrefetchML

Gwendal Daniel · Gerson Sunyé · Jordi Cabot

Received: date / Accepted: date

Abstract Caching and prefetching techniques have been used for decades in database engines and file systems to improve the performance of I/O intensive application. A prefetching algorithm typically benefits from the system's latencies by loading into main memory elements that will be needed in the future, speeding-up data access. While these solutions can bring a significant improvement in terms of execution time, prefetching rules are often defined at the data-level, making them hard to understand, maintain, and optimize. In addition, low-level prefetching and caching components are difficult to align with scalable model persistence frameworks because they are unaware of potential optimizations relying on the analysis of metamodel-level information, and are less present in NoSQL databases, a common solution to store large models. To overcome this situation we propose PrefetchML, a framework that executes prefetching and caching strategies over models. Our solution embeds a DSL to configure precisely the prefetching rules to follow, and a monitoring component providing insights on how the prefetching execution is working to help designers optimize his performance plans. Our experiments show that PrefetchML is a suitable solution to improve query execution time on top of scalable model persistence frameworks. Tool support is fully available online as an open-source Eclipse plugin.

Gwendal Daniel
AtlanMod Team - Inria, IMT-Atlantique & LS2N
4, Rue Alfred Kastler
Nantes, France
E-mail: gwendal.daniel@inria.fr

Gerson Sunyé
AtlanMod Team - Inria, IMT-Atlantique & LS2N
4, Rue Alfred Kastler
Nantes, France
E-mail: gerson.sunye@inria.fr

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

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 [25, 28]. 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 work by adding load instructions (according to prefetching rules derived by static [18] 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.

Given that model-driven engineering (MDE) is progressively adopted in the industry [17, 23], we believe that the support of prefetching and caching techniques at the modeling level is required to raise the scalability of MDE tools dealing with large models where storing, editing, transforming, and querying operations are major issues [21, 32]. These large models typically appear in various engineering fields, such as civil engineering [1], automotive industry [4], product lines [26], 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 [13, 15, 19, 24]. These frameworks use lazy-loading techniques to load into main memory the 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. Existing frameworks typically rely on the database prefetching and caching capabilities (when they exist) to speed-up query computation in a generic way (i. e. regardless the context of the performed model manipulation). This facilitates their use in a variety of scenarios but prevent them from providing model-specific optimizations that would require understanding the type of the model (i.e. its metamodel) to come up with accurate loading strategies.

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 run-time 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. The DSL itself is generic and could be part of any modeling stack but 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.

This paper is an extension of our previous work introducing PrefetchML [11]. It provides three major new contributions: (i) a *monitoring component* that provides insights into the execution performance to guide developers on improving their prefetching plans, (ii) a global shared cache and a set of *cache consistency policies* extending the cache component to ensure cached elements are always consistent with the model when update operations are performed, and (iii) a new set of experiments based on the well-known Train Benchmark [29] including new model usage scenarios when comparing PrefetchML’s performance on top of NeoEMF/Map and NeoEMF/Graph.

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, its basic rule execution algorithm, and the consistency policies we have implemented. Section 5 presents our new monitoring component and its integration within the framework. Section 6 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 7 presents the benchmarks used to evaluate our prefetching tool and associated results. Section 8 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 [28] in order to improve query computation time. Their presence in NoSQL databases is much more limited, which contrasts with the increasing popularity of this type of databases as model storage solution. Existing work typically rely on learning techniques to dynamically optimize the loading of NoSQL database records based on their locality [12, 34], or improve memory transfer performance of in-memory key-value stores [33]. However, these solutions are strongly connected to the data representation (i. e. how the information is represented by the database primitives), and do not use conceptual schema information (that can be partially extracted automatically using schema inference techniques [27]) that could raise the abstraction level of prefetching and caching rules, and improve their reusability across multiple data sources. In addition, 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— that have different access patterns that should be optimized specifically.

Scalable modeling frameworks are built on top of relational or NoSQL databases to store and access large models [3, 13]. 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, 24], they perform a lot of fragmented queries on the database, thus adding a signifi-

cant execution time overhead. For the reasons described above, these frameworks cannot benefit from database prefetching solutions nor do they implement their own mechanism, with the partial exception of CDO [13] that provides some basic prefetching and caching capabilities¹. For instance, CDO is able to bring into memory all the elements of a list at the same time, or load 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.

Caching is a common solution used in current scalable persistence frameworks to improve query execution involving repeated accesses of model elements. However, these caches are tailored to a specific solution, and they typically lack advanced configurations such as the replacement policy to use, the maximum size of the cache, or the number of elements to drop when the cache is full. In addition, persistence framework caches are usually defined as internal components, and do not allow client applications to access the content of the cache.

Hartmann et al. [16] 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 when the model is modified 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 [20] query language. While Hawk provides an efficient backend-independent query language built on top of the Epsilon platform, it does not allow the definition of prefetching plans for the indexed models.

In summary, we believe that 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.

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

6. A monitoring/quality component providing feedback on the prefetching and caching plan for a given execution.

In the following sections, 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 fits the specific navigation path of the query.

Beyond a set of prefetching rules, each plan defines a cache that can be parametrized, and a consistency policy that defines the strategy to use to manage the life-cycle of cached element when the model is updated.

In what follows, we formalize the abstract and concrete syntax of the PrefetchML DSL and introduce them through a running example. The next section will introduce how these rules are executed as part of the prefetching engine.

3.1 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 1 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.

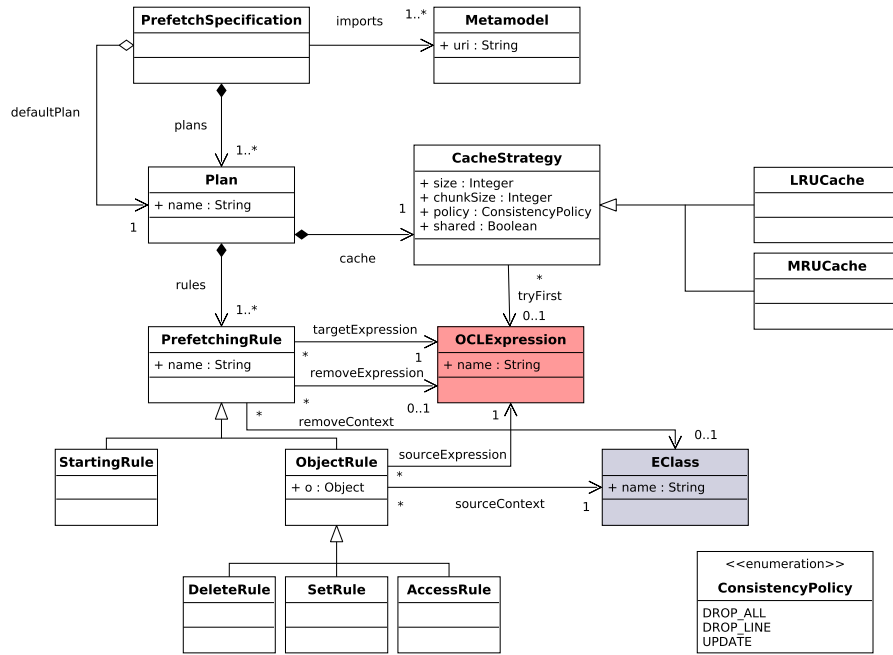


Fig. 1: Prefetch Abstract Syntax Metamodel

Each *Plan* contains a *CacheStrategy*, which represents the information about the cache policy the prefetcher applies to keep loaded objects in memory. Currently, available cache strategies are *LRUCache* (Least Recently Used) and *MRUCache* (Most Recently Used). These *Caches* define four parameters: (i) the maximum number of objects they can store (*size*), (ii) the number of elements to free when the cache is full (*chunkSize*), (iii) the *consistency policy* used to manage model modifications, and (iv) the integration of the cache with the running application (details on cache consistency/integration are provided in Section 4). 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 (from the *imported* metamodels), 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 *OCLEExpression* 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. A *remove* expressions can be coupled with the *tryFirst* expression contained in the *CacheStrategy* to tune the default replacement policy of the cache.

3.2 Concrete Syntax

We introduce now the concrete syntax of the PrefetchML language, which is derived from the abstract syntax metamodel presented in Figure 1. Listing 1 presents the grammar of the PrefetchML language expressed using XText [14], 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 *OCLEExpressions* are parsed as *Strings*, the model representation of the queries presented in Figure 1 is computed by parsing it using the Eclipse MDT OCL toolkit².

```

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=
        OCLEExpression)?
;

```

² <http://www.eclipse.org/modeling/mdt/?project=ocl>

```

LRUCache:
  'use_cache' 'LRU'
;

MRUCache:
  'use_cache' 'MRU'
;

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

enum ConsistencyPolicy:
  DROP_ALL = 'drop_all' |
  DROP_LINE = 'drop_line' |
  UPDATE = 'update'
;

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

StartingRule:
  'rule' name=ID ':' 'on_starting'
  'fetch' targetPatternExp=OCLEExpression
  ('remove_' 'type' removeType=ClassifierExpression removePatternExp=
    OCLEExpression)?
;

AccessRule:
  'rule' name=ID ':' 'on_access'
  'type' sourceType=ClassifierExpression (sourcePatternExp=OCLEExpression)
  ?
  'fetch' targetPatternExp=OCLEExpression
  ('remove_' 'type' removeType=ClassifierExpression removePatternExp=
    OCLEExpression)?
;

DeleteRule:
  'rule' name=ID ':' 'on_delete'
  'type' sourceType=ClassifierExpression (sourcePatternExp=OCLEExpression)
  ?
  'fetch' targetPatternExp=OCLEExpression
  ('remove_' 'type' removeType=ClassifierExpression removePatternExp=
    OCLEExpression)?
;

SetRule:
  'rule' name=ID ':' 'on_set'
  'type' sourceType=ClassifierExpression (sourcePatternExp=OCLEExpression)
  ?
  'fetch' targetPatternExp=OCLEExpression
  ('remove_' 'type' removeType=ClassifierExpression removePatternExp=
    OCLEExpression)?
;

OCLEExpression: STRING ;

ClassifierExpression: ID;

```

Listing 1: PrefetchML Language Grammar

3.3 Running Example

In order to better illustrate the features of PrefetchML, we introduce a simple example model. Figure 2 shows a small excerpt of the *Java* metamodel provided by MoDisco [5]. 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).

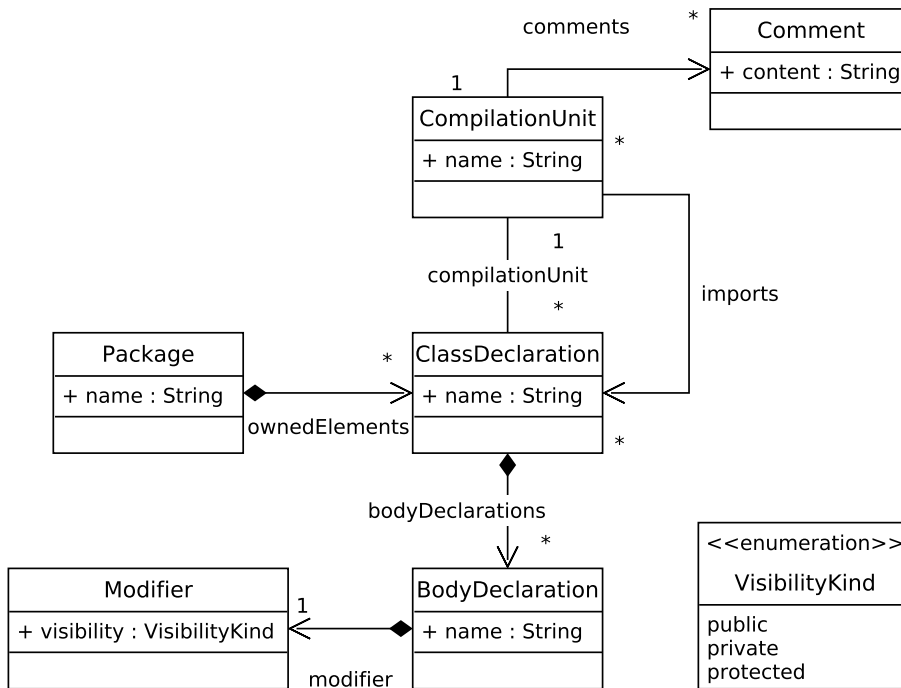


Fig. 2: Excerpt of Java Metamodel

Listing 2 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 : isEmptyPackage : 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 2: Sample OCL Queries

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 a scenario where all the queries expressed in Listing 2 are executed in the order they are defined.

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). The cache also defines the *shared* property, meaning that elements computed by the prefetching rules and the running application will be cached together. Finally, the cache uses the *drop_line* consistency policy, that removes lines from the cache corresponding to updated elements. Note that the consistency policy is not important in this example, because OCL expressions are side-effect free and do not generate update notifications.

The *plan* also defines 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, shared, policy=drop_line]
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 into 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, and present the different cache consistency policy and integration level that can be defined to tune the prefetching algorithm.

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 **User** uses a model-based tool that 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). The elements in this modeling stack are typically set-up by a **Modeling engineer** who configures them according to the application’s workload (for example by selecting a scalable persistence framework if the application aims to handle large models).

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 6 discusses two specific implementations of this component. The PrefetchML framework also provides *monitoring information* that gives useful insights into the execution to help the **Modeling Engineer** to customize the persistence framework and prefetching plans. The monitoring component is detailed in the next section.

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. Model modification events are also intercepted by the framework to update/invalidate cached values in order to keep the cache content consistent with the model state.

Figure 4 describes the internal structure of the PrefetchML Framework. As explained in Section 3, a PrefetchML specification *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 core component ensures cache consistency, by invalidating part or all cached records when update, create, or delete events are received. The **Rule Store** is a data structure that stores all the object rules (access, update, delete) contained in the input *PrefetchML description*.

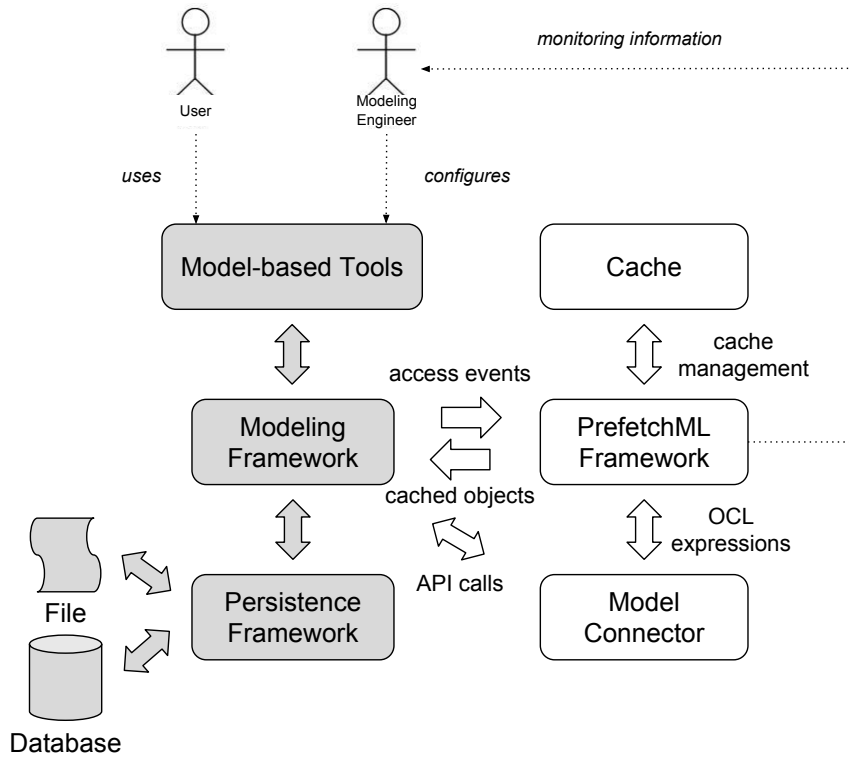


Fig. 3: PrefetchML Integration in MDE Ecosystem

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. By default, the core component manages its own cache where only prefetched elements are stored, providing a fine-grain control of the cache content. 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 receives events from the client application. It provides an API to send access, delete, and update events. These events

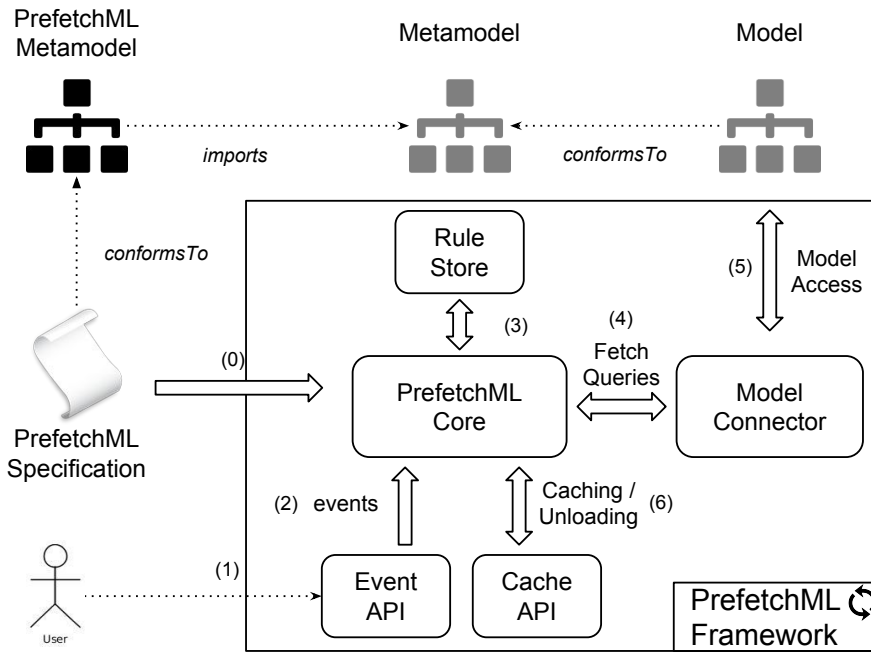


Fig. 4: Prefetch Framework Infrastructure

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. This information is 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 correspond to the event and the object that triggered it (3). Each *OCLEExpression* 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 handed back by the *PrefetchML Core*, which is in charge of caching them and freeing the cache from previously stored objects (6).

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 with the application execution. Model elements are cached asynchronously and are available to the client application through the *CacheAPI*.

Note that this infrastructure is not tailored to any particular data representation and can be plugged into 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 [15], CDO [13], and Morsa [24]. However, the efficiency of PrefetchML

(in particular the prefetcher throughput) can vary from one persistence solution to another because of synchronization feature and the persistence framework/-database ability to handle multiple queries at the same time. These differences are highlighted in the experiments we discuss in Section 7.

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, its `chunkSize` to 10, and the *drop line* consistency policy.

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) which 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.

4.3 Cache Consistency

The PrefetchML DSL presented in Section 3 allows to define prefetching rules that are triggered when an element in the model is *Accessed*, *Set*, and *Deleted*. However, these events are simply used to trigger prefetching rules, and updating the model may result in inconsistencies between the PrefetchML cache and the actual model state. While this is not a problem for side-effect free query computation such as OCL (where no element is modified), it becomes an issue when using PrefetchML

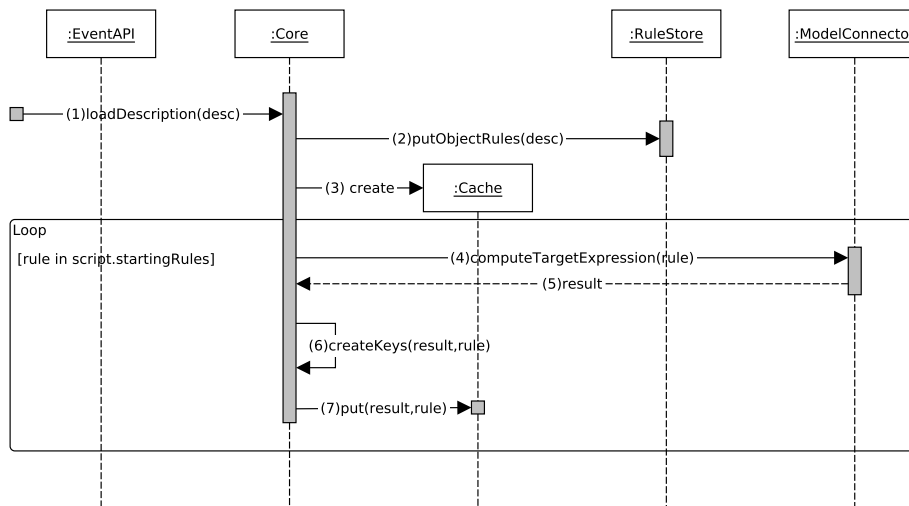


Fig. 5: PrefetchML Initialization Sequence Diagram

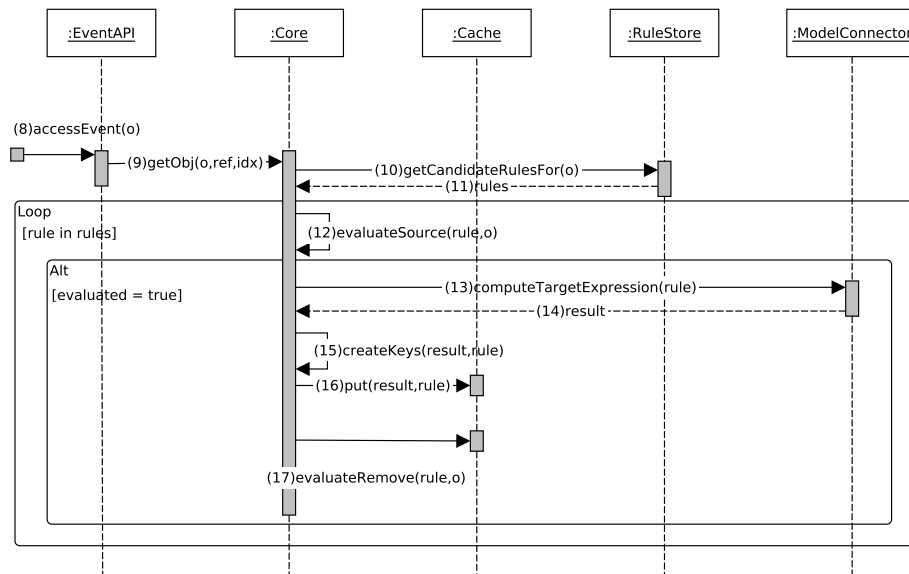


Fig. 6: PrefetchML Access Event Handling Sequence Diagram

on top of model-to-model transformation frameworks, or EMF-API based applications.

To overcome this limitation we have defined a set of cache consistency policies that are embedded in PrefetchML. They all ensure that the content of the cache is

consistent w.r.t the model, by handling updates with different strategies in order to limit execution overhead or increase cache hits. Available policies include:

- **Drop all:** drop the entire cache every time the model is updated
- **Drop line:** drop the cache lines corresponding to the updated element and all its references
- **Update:** update the cache lines corresponding to the updated element with the new value, including references

Drop all is the simplest cache consistency policy: it drops the entire cache each time a model update event is received. Dropping the entire cache is fast and does not have a significant impact on the prefetcher throughput. However, this policy drops elements that are still consistent with the model, and have an important impact on the prefetcher hit score. Full drop policy is typically used when model modifications are localized at a specific point of the execution, and concern an important part of the model. This consistency strategy can be specified in the cache parameters of a prefetching plan with the keyword *drop-all*

Drop line removes from the cache the updated element and all the elements referencing it. This approach is interesting if few model modifications are performed at multiple steps of the execution, and dropping the entire cache would have an important impact on the number of hits. However, dropping multiple lines is more expensive in terms of execution time because the framework has to inspect the cache to find all the elements to remove. This policy is used by default if no consistency policy is defined in the prefetching plan.

Update policy keeps the cache consistent with the model by updating all the lines corresponding to the modified object. This policy is interesting if a very small amount of model modifications are performed, and the updated objects are reused later and should stay in the cache. Updating the cache requires to find the cache lines to update, and navigate the model to find the updated values. This operation is costly, and may have a significant impact on the prefetcher performance if too many objects are updated during the query execution. Note that indexing techniques could be used to reduce this performance issue, but they also require to keep the index up-to-date with both the cache and the model content.

These different cache policies can be selected by the modeling engineer to tune PrefetchML according to its application workload. For example, an interactive model editor can benefit from the *Update* policy, because these kind of application usually has a low workload, with localized model modifications. On the other hand, in the context of a model-to-model transformation that typically creates and updates a lot of model elements, using a lightweight policy such as *drop line* is more appropriated.

Figure 7 shows the sequence diagram presenting how PrefetchML handles model modifications. When an element is updated, an *updateEvent* describing the old (*o*) and new (*n*) versions of the updated element is sent to the *EventAPI* (8). This event is forwarded to the *Core* component (9) that retrieves the consistency policy to use (10), and tells the *Cache* to update its content according to it (11). Depending on the policy use, the *Cache* will drop all its content, invalidate the lines corresponding to the updated element, or update its content. The rest of the sequence diagram is similar to the one presented in Figure 6, with the particularity

that rules are found and computed from the new version of the element instead of the old one.

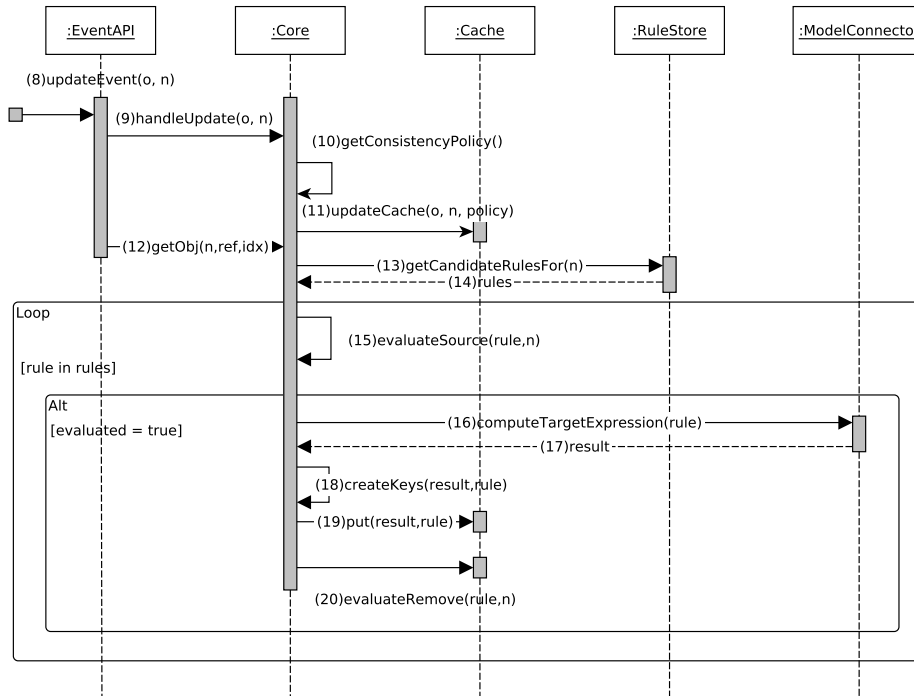


Fig. 7: PrefetchML Update Event Handling Sequence Diagram

4.4 Shared cache

As we have explained throughout this section, the PrefetchML framework embeds a cache dedicated to keep prefetched elements. The modeler has precise control over the cache content and can be assured that every object stored in the cache has been loaded by a PrefetchML rule. This approach is interesting when designers want to choose a cache size that perfectly fits their needs, and are not concerned by the other caches in the application environment (such as the ones embedded in the model persistence framework or the underlying tools they rely on). However, this isolation of the PrefetchML cache has a cost: as Figure 3 shows, this cache is accessed every time a model access operation is captured, and thus, the framework searches for the accessed element in the PrefetchML cache first, and only when it is not found it delegates to the persistence framework the retrieval of the requested object. The performance of this preliminary lookup strongly relies on the correctness of the prefetching plan: if the plan is good, the cache will have a decent chance to contain the element and this will improve the computation

time, if not, the program may waste a significant amount of time scanning the PrefetchML cache.

To overcome this limitation, we have defined a *shared cache* that contains both the elements loaded by prefetching rules, and by the persistence framework itself. To do so, every call to an element accessor that is captured by our framework (even if it does not trigger any prefetching rule) will create a cache entry at the PrefetchML (shared) cache level (i. e. the first cache that is accessed when searching for an object). This architecture provides two benefits: (i) it caches elements that are accessed multiple times even if they are not part of a prefetching rule, improving the PrefetchML cache accuracy and reducing the time spent in unnecessary lookups, and (ii) it improves the prefetcher throughput when both prefetching rules and application-level queries (not triggering any rule) are loading the same elements. In this last scenario the PrefetchML algorithm will be notified of every element being accessed allowing it to avoid duplicated work and move on the next rule to compute.

In addition, this shared cache can be seen as a default caching mechanism on top of model persistence frameworks that do not define their own cache: the PrefetchML framework keeps in memory the elements that have been accessed (when the persistence framework does not support this feature), and automatically retrieves them from the cache when they are accessed. In this context, a PrefetchML plan containing a simple *shared cache* declaration can be used to enhance the persistence framework with a simple caching strategy, and be complemented with additional prefetching and caching rules if needed. We show in our experiments (Section 7) that sharing the cache between the PrefetchML layer and the model persistence framework has a positive impact on query execution time.

Note that in this first version of the framework we only consider the integration of PrefetchML cache with the ones defined at the model persistence level. Studying the impact of low-level caches such as database caches, operating system optimizations, and hardware caches, and their integration into a global caching mechanism is left for future work. Note that designers can easily disable/enable this shared cache option using the *shared cache* parameter in their PrefetchML plans.

5 Plan Monitoring

This section details the monitoring component we have integrated to the PrefetchML framework. We first introduce the new language constructs and framework updates, then we present an example of the information a modeler can get from the framework and how it can be used to customize the prefetching plan. Finally, we show how this same monitoring information can be employed to dynamically adapt the PrefetchML algorithm and an appropriate cache integration.

5.1 Language Extensions for Plan Monitoring

Prefetching and caching can significantly improve model query computation, but this improvement is tightly coupled to the quality of the plan to execute. Intuitively, a *good* prefetching plan is a plan that loads elements before they are needed

by the application, and keeps them in memory for a sufficiently long time to make later accesses faster, without polluting cache content with irrelevant objects.

While this intuitive approach is easy to conceptualize, it can be hard to apply in real-life scenarios: the modeler does not know the exact content of the cache, and multiple rules may interact with each other, filling/freeing the cache with different expressions at the same time. Moreover, comparing the quality of two prefetching plans and/or the impact of an update on a specific rule is not a straightforward task, and requires to have a close look at the cache content. To help designers to evaluate the quality of their prefetching plans we have defined a monitoring component that presents execution information allowing them to detect problematic and missing rules, guards, and interaction between prefetching instructions.

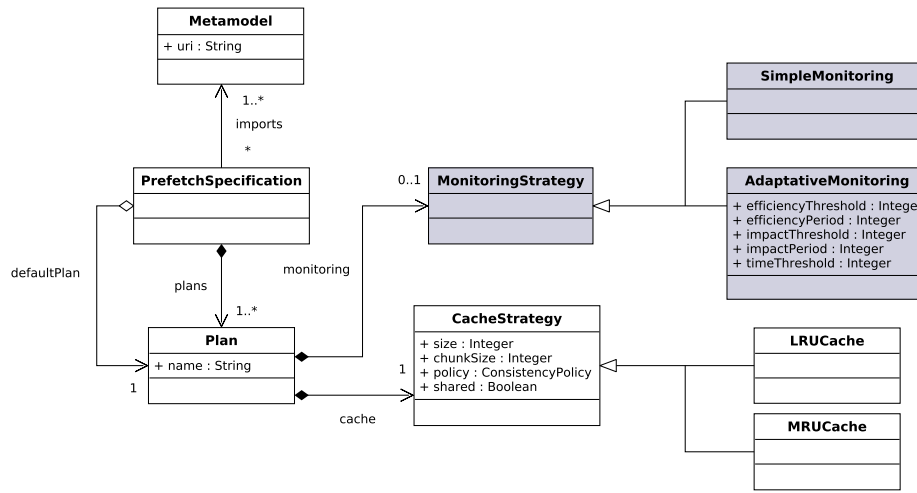


Fig. 8: Prefetch Abstract Syntax Metamodel with Monitoring Extensions

Figure 8 shows the extended abstract syntax of the PrefetchML DSL with the new constructs dedicated to monitoring (grey nodes). In addition to its *CacheStrategy*, now a PrefetchML *Plan* can define an optional *MonitoringStrategy* that collects execution information such as the number of hits and misses for each rule. Current available monitoring strategies are *SimpleMonitoring* that provides these metrics to the model designer under request (Section 5.2), and *AdaptiveMonitoring* that uses them together with a set of user-defined thresholds to optimize the prefetching algorithm at runtime (Section 5.3).

These new language constructs are used to initialize the monitoring component through the **MonitorAPI** shown in Figure 9. This API defines a set of methods to instantiate and parameterize a monitor, and access computed metrics. These metrics are updated each time an element is loaded by a prefetching rule or accessed from the cache. Monitoring information can be displayed to end-users to

help them improve their PrefetchML plans, or used at runtime by the framework itself to adapt the plan dynamically.

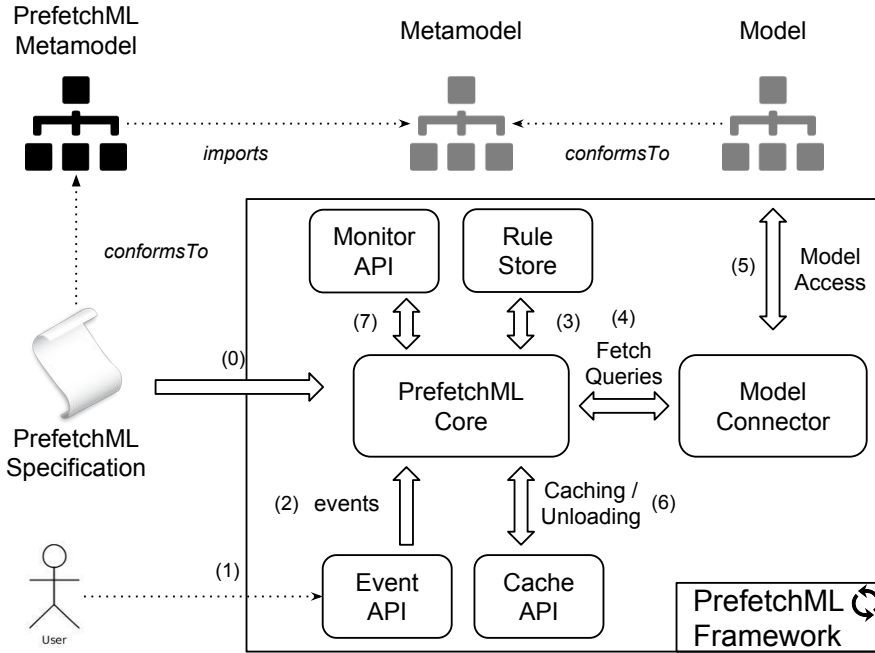


Fig. 9: Prefetch Framework Infrastructure with Monitoring

In the following, we detail the metrics computed by the monitoring component and how they can be used by a modeler to improve prefetching plans.

5.2 Simple Monitoring

SimpleMonitoring is the first monitoring strategy we have added to the PrefetchML grammar (Figure 8). It can be added to a PrefetchML plan by using the keywords `use simple monitoring`. Once activated, the framework will collect information during the execution, and compute a set of metrics that will be presented to the modeler to help in the quality evaluation of the plan. The metrics are the following:

1. **HitScore**: the total number of elements accessed from the cache
2. **MissScore**: the number of elements the persistence framework had to load because of cache misses
3. **MissPerFeature**: categorize the cache misses score per accessed element feature
4. **CachedByRule**: the number of elements cached by each prefetching rule
5. **HitPerRule**: the number of cache hits generated by each prefetching rule
6. **CachedTimestampPerRule**: the list of caching instruction timestamps for each prefetching rule

7. **HitTimestampPerRule:** the list of cache hit timestamps for each prefetching rule
8. **TotalPrefetchingTime:** the total time spent on prefetching/caching actions

Metrics 1-3 correspond to *global* accuracy informations that represent the entire prefetching plan usefulness. A good plan will typically generate a high *HitScore* and a low *MissScore*. Misses are categorized by feature (attribute or reference), providing insights on a potential new rule to add to the plan. Metrics 4 and 5 provide fine information for each rule within the PrefetchML plan: the number of cached elements per rule and the number of hits generated by each rule. This information can be used to evaluate the usefulness of a specific rule (for example by comparing the ratio *HitPerRule/CachedByRule* to a given threshold). Finally, metrics 6-8 provide time-based information, showing the impact of a given rule over time. This information can be used to find rules that are applied at some point of the computation where they should not which lets the designer tune the OCL conditions to control when they are triggered. The total prefetching time allows determination of which part of the computation was dedicated to prefetching and caching instructions. This information is particularly interesting when PrefetchML is applied on top of a backend that does not handle multi-threaded accesses, emphasizing execution time bottlenecks.

Listing 4 shows a possible output of the monitoring component after the execution of the queries presented in the running example (Listing 2) with the PrefetchML plan presented in Listing 3 enabled over a sample model. The table shows, for each rule, the number of executions, the total and average computation time, the number of cached elements, and the number of generated hits. This output format is the default one provided by PrefetchML, additional information such as time-based metrics are available through the monitor API.

The table shows that three rules were executed: *r1*, *r2*, and *r3*. Rule *r1* was executed one time, which is the expected behavior for starting rules, that are executed when the prefetching plan is loaded. The table also shows that *r1* cached 45000 elements, but only generated 3000 hits which is low compared to the total hit score (around 1%). Loading these 45000 elements required 6900 milliseconds (15% of the total execution time), which is high compared to the benefit. Removing the rule from the plan would allow the framework to use this execution time to increase the throughput of the other rules. Compared to *r1*, rules *r2* and *r3* cached fewer elements, but generated most of the global hit score (respectively 52% and 47%).

The last part of the presented listing shows the features that generated cache misses. In our example, there is only one feature (`Package.ownedElement`) that generated all the misses. This information shows that adding a prefetching rule for this feature would improve the global hit score and thus improve the efficiency of the prefetching plan.

Based on the monitoring informations, we were able to detect that *r1* should be removed, and a new rule *r4* should be added to prefetch the feature that generated the misses. Listing 5 shows the updated plan.

```

1  == PrefetchML Monitoring ==
2  Monitoring started at 12:30:34:145
3  #Hits: 234 000
4  #Misses: 125000
5  #Total Prefetching Time: 45000 ms
6
7  == Rule → #Execution | Tot. Time | Avg. Time | #Cached | #Hits
8  ==
9  r1 → 1 | 6900 | 6900 | 45000 | 3000
10 r2 → 1493 | 14500 | 10 | 12500 | 120000
11 r3 → 5890 | 23600 | 4 | 30456 | 111000
12
13 == Feature → #Misses ==
14 Package.ownedElements → 125000

```

Listing 4: PrefetchML Monitoring Example

```

1  plan samplePlan {
2    use cache LRU[ size=100, chunk=10]
3    rule r2 : on access type ClassDeclaration fetch
4      self.bodyDeclarations.modifier
5    rule r3 : on access type ClassDeclaration fetch
6      self.compilationUnit.imports.compilationUnit.comments.
7        content
8      remove type Package
9    rule r4 : on access type Package fetch
10   self.ownedElements
11 }

```

Listing 5: Tuned PrefetchML Plan

5.3 Adaptive Monitoring

AdaptiveMonitoring is the second monitoring strategy we have added to the PrefetchML language (Figure 8). It can be activated within a PrefetchML plan using the keywords `use adaptive monitoring`. When this strategy is set, the framework collects runtime information (as for the *SimpleMonitoring* strategy) and uses a set of heuristics to dynamically adapt prefetching plans to the query computation.

We have defined five heuristics that are used by the framework to disable prefetching rules that are not beneficial for the application. We consider that a rule is harmful if it pollutes the cache content with useless objects and/or if it reduces the throughput of the prefetcher by spending execution time computing loading instructions that are not caching relevant elements. These heuristics can be parametrized by setting the *threshold values* of the *AdaptiveMonitoring* component, and retrieve:

1. **RuleEfficiency:** $Hit_r / Cache_r < threshold \rightarrow disable(r)$
2. **Time-based RuleEfficiency:** $Hit_r / Cache_r < threshold$ during a period of time $t \rightarrow disable(r)$
3. **RuleImpact:** $Hit_r / HitScore < threshold \rightarrow disable(r)$
4. **Time-based RuleImpact:** $Hit_r / HitScore < threshold$ during a period of time $t \rightarrow disable(r)$
5. **TimeImpact:** $TotalTime > threshold \rightarrow \forall r, disable(r)$

RuleEfficiency evaluates the rule efficiency by comparing the number of hits it has generated with the number of cached objects. The rule is disabled when this value goes under a given threshold, meaning that the rule cached too many objects compared to the number of hits it generated. While this strategy can be interesting for simple prefetching plan, it may disable useful rules for more complex plans that cache elements that are accessed late in the query computation (typically *starting rules*). To handle this kind of rules we have defined **Time-based RuleEfficiency**, that extends **RuleEfficiency** by disabling a rule if its computed ratio is below a threshold for a given period of time t . **RuleImpact** computes the impact of a rule by comparing the number of hits it generates w.r.t the global *HitScore*, and disables the rule if this value goes below a given threshold. This strategy disables low-impact rules, giving more execution time to other rules that are generating more hits. **Time-based RuleImpact** is similar, but it only disables a rule if its computed ratio is below a threshold for a given period of time. Finally, **TimeImpact** is a plan-level strategy that disables all rules if the prefetching time increases over a given threshold.

All the thresholds and time intervals used to define the presented heuristics can be configured in PrefetchML plans using their corresponding keywords: `efficiencyThreshold`, `efficiencyPeriod`, `impactThreshold`, etc.

Note that in this first version of the *adaptive monitoring* component rules can only be disabled, re-enabling rules is a more complicated task, because computed ratios do not evolve once rules have been disabled. To allow rules re-enabling, we plan to add another monitoring layer that keeps traces of accessed elements and computes which rules would have prefetched them. Monitoring information could also be used to create new rules based on the feature misses. While creating a rule for a single feature is simple, the key point is to find the optimal rule(s) to reduce the number of misses, without polluting the cache content and the prefetcher throughput. This could be done by using constraint solvers in order to find the optimal set of rules to create from a set of misses.

6 Tool Support

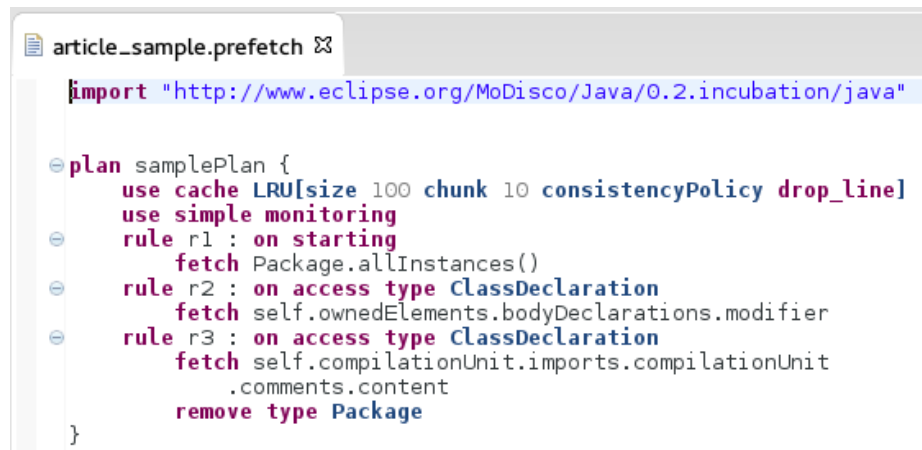
In this Section we present the tool support for the PrefetchML framework. It is composed of two main components: a language editor (presented in Section 6.1) that supports the definition of prefetching and caching rules, and an execution engine with two different integration options: the EMF API and the NeoEMF/Graph persistence framework (presented in Sections 6.2 and 6.3). The presented components are part of a set of open source Eclipse plugins available at https://github.com/atlanmod/Prefetching_Caching_DSL.

6.1 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 3.2 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. Note that monitoring constructs defined in Section 5 are available in the editor, allowing to choose a monitoring strategy and define its thresholds.

Figure 10 shows an example of the PrefetchML editor that contains the prefetching and caching plan defined in the running example of Section 3. The plan contains an addition `use simple monitoring` line that enables simple monitoring capabilities, providing execution information to the modeler.



```

import "http://www.eclipse.org/ModelDisco/Java/0.2.incubation/java"

plan samplePlan {
  use cache LRU[size 100 chunk 10 consistencyPolicy drop_line]
  use simple monitoring
  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
}

```

Fig. 10: PrefetchML Rule Editor

6.2 EMF Integration

Figure 11 shows the integration of PrefetchML within the EMF framework. Note that only two components must 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` is 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.

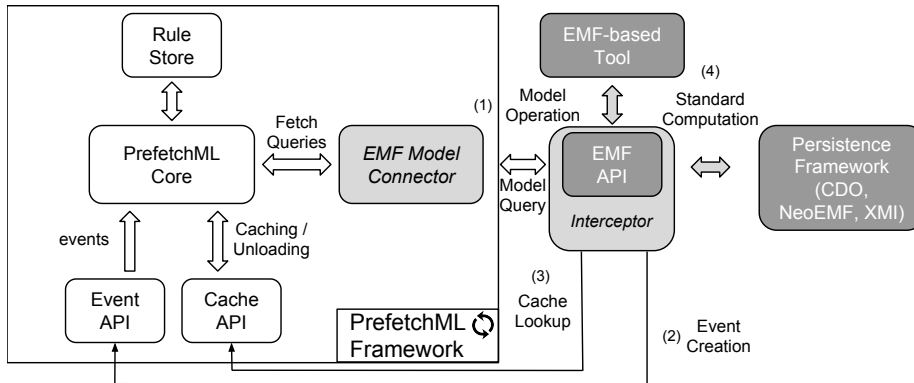


Fig. 11: Overview of EMF-Based Prefetcher

6.3 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 [30], 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 [31]. 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 (Figure 12) 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 7), 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.

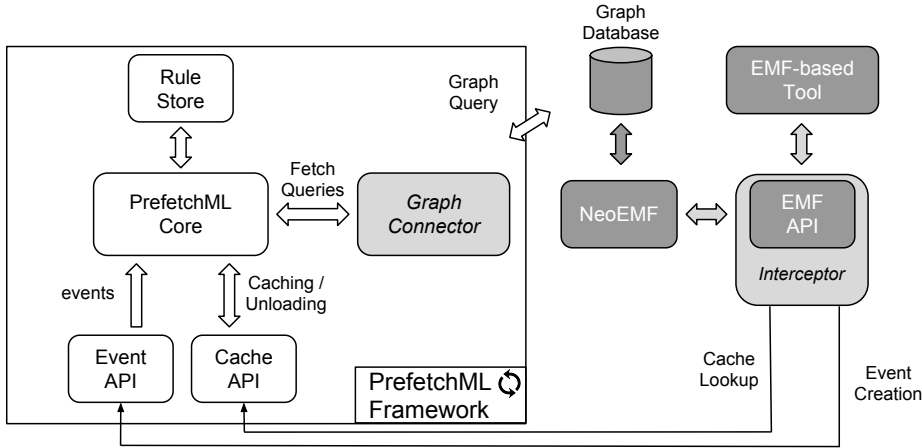


Fig. 12: Overview of NeoEMF-Based Prefetcher

7 Evaluation

In this Section, we evaluate the performance of our PrefetchML Framework by comparing the performance of executing a set of OCL queries on top of two different backends: NeoEMF/Graph and NeoEMF/Map when (i) no prefetching is used and (ii) EMF-based prefetching is active. Models stored in NeoEMF/Graph are also evaluated with a third strategy using the dedicated graph-based prefetching presented in section 6.3.

Queries are executed in two modeling scenarios: *single query execution* where queries are evaluated individually on the models, and *multiple query execution* where queries are computed sequentially on the models. The first one corresponds to the worst case scenario where the prefetcher and the query itself compete to access the database and retrieve the model elements. The second scenario corresponds to the optimal prefetching context: rules target all the queries at once, and the workflow contains idling intervals between each evaluation, giving more execution time to the prefetcher to load elements from the database.

Note that we do not compare the performance of our solution with existing tools that can be considered related to ours because we could not envision a fair

comparison scenario. For instance, Moogle [22] 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 different trade-offs.

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 1.8.

7.1 Benchmark Presentation

The executed queries are adapted from the Train Benchmark [29], which is a benchmark used to evaluate the performance of model transformation tools. It defines the *railway* metamodel, which describes classes to represent railway networks, such as *Route*, *Semaphore*, and *Switch*. A complete description of this metamodel can be found on the benchmark repository³ and in the associated publication [29]. In this experiment we use four queries adapted from the ones defined in the benchmark:

- **RouteSensors**: to compute a subset of the sensors contained in a route.
- **RegionSensors**: to access all the sensors contained in a region.
- **ConnectedSegments**: to navigate all the track elements connected to a sensor.
- **SwitchSet**: to retrieve for each entry of a route its corresponding switch elements.

The first query navigates multiple references from a *Route* element in order to retrieve the *Sensors* it directly and indirectly contains. The second one performs a simple navigation to retrieve all the *Sensor* elements contained in a *Region*. The third query performs a long navigation sequence to retrieve all the *TrackElements* connected to a *Route* element. The last query retrieves the *Semaphores* associated to a given *Route*, and then navigates them to find the *Switch* elements satisfying a condition⁴.

The prefetching plans used in this benchmark 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. The validity of the resulting plans has been checked via a preliminary execution of the queries with the *Simple Monitoring* component enabled. Note that this plan creation strategy can be reused for any OCL query computation. Additionally, and based on our experience on defining PrefetchML plans, we have applied a merging strategy to create a single PrefetchML rule for common segments in navigation paths, in order to minimize the number of rules that have to be executed by the framework.

In this evaluation we do not consider the definition of bad PrefetchML plans (i.e. plans that contains rules that are never triggered and/or rules that load elements that are not needed), but our previous experiments [11] have shown

³ <https://github.com/FTSRG/trainbenchmark>

⁴ Details of the queries can be found at https://github.com/atlanmod/Prefetching_Caching_DSL

Table 1: Experimental Set Details

Query	#Input	#Traversed	#Res
RouteSensors	320	28 493	1296
RegionSensors	320	25 431	15 805
ConnectedSegments	15 805	98 922	67 245
SwitchSet	320	14 957	252

that these plans have a limited impact on the execution compared to the default execution without PrefetchML⁵.

The queries have been executed with a MRU cache that can contain up to 20% of the input model. 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. Another benchmark presenting a different cache configuration is available in our previous work [11]. 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.

Prefetching plans are evaluated in two cache settings: a first one with an embedded cache dedicated to prefetched objects, meaning that only elements that have been loaded by the framework are in the cache, and a second one using a *shared* cache storing elements of both the prefetcher and the persistence framework. In addition, we evaluate for each cache configuration the impact of the *Adaptive Monitoring* component presented in Section 5.3. We manually set the *RuleEfficiency* threshold to 0.5 and the *RuleImpact* one to 0.25, based on the information provided by the *Simple Monitoring* component used in our preliminary run⁶. Note that the monitoring component is reset between each execution of the query in order to keep traces of the prefetching and caching instructions that are specific to a given execution.

The experiments are run over one of the models provided with the benchmark, which contains 102 875 elements. The associated XMI file is 19 MB large. Queries are evaluated over all the instances of the model that conform to the context of the query. In order to give an idea of the complexity of the queries, we present in Table 1 the number of input elements for each query (**#Input**), the number of traversed elements during the query computation (**#Traversed**) and the size of the result set for each model (**#Res**).

7.2 Results

This section describes the results we obtained by running the experiment presented above. We first introduce the results for the *single query execution* scenario, then we present the results for the *multiple query execution* scenario. Note that the correctness of query results in both scenarios has been validated by comparing the results of the different configurations with the ones of the queries executed

⁵ Note that this previous experiment embedded a primitive version of the *Adaptive Monitoring* component that was able to disable a rule if it did not generate any hit

⁶ Time-based monitoring is not considered in this evaluation

without any prefetching enabled using EMFCompare.⁷ Presented results have been obtained by using Eclipse MDT OCL to run the OCL queries on the different persistence frameworks.

Tables 2 and 3 present the average execution time (in milliseconds) of 10 executions of the presented queries over the benchmarked model stored in NeoEMF/Graph and NeoEMF/Map, using the *single query execution* scenario. Each line presents the result for the kind of prefetching that has been used: no prefetching (*NoPref.*), EMF-Prefetching with dedicated cache (*EMF Pref.*) and adaptative monitoring (*EMF Pref. (Adaptative)*), and EMF-Prefetching with shared cache (*EMF-Pref. (Shared)*) and adaptative monitoring (*EMF-Pref. (Shared + Adaptative)*). Note that Table 2 contains an additional line corresponding to the graph specific prefetcher.

Table 4 shows the average execution time (in milliseconds) of 10 executions of all the queries over NeoEMF/Graph and NeoEMF/Map in the *multiple query execution* scenario.

In the first part of the result tables, the cells show 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). In the second part, results show the execution time of a second execution of the query when part of the loaded elements has been cached during the first computation (Warmed Execution).

Table 2: NeoEMF/Graph Query Execution Time in milliseconds

(a) Cold Execution				
	Route Sensors	Region Sensors	Connected Segments	Switch Set
No Pref.	30 294	1633	14 801	915
EMF Pref.	30 028	1982	14 625	1047
EMF Pref. (Adaptative)	30 190	1851	14 834	1012
EMF Pref. (Shared)	28 902	1803	13 850	998
EMF Pref. (Shared + Adaptative)	29 251	1712	14 025	951
Graph Pref.	25 143	1477	11 811	830
(b) Warmed Execution				
	Route Sensors	Region Sensors	Connected Segments	Switch Set
No Pref.	16 087	908	8887	528
EMF Pref.	259	183	874	130
EMF Pref. (Adaptative)	242	175	851	118
EMF Pref. (Shared)	236	179	877	130
EMF Pref. (Shared + Adaptative)	225	171	849	122
Graph Pref.	1140	445	2081	264

⁷ <https://www.eclipse.org/emf/compare/>

Table 3: NeoEMF/Map Query Execution Time in milliseconds

(a) Cold Execution				
	Route Sensors	Region Sensors	Connected Segments	Switch Set
No Pref.	33 770	1307	11 935	499
EMF Pref.	2515	1210	10 166	410
EMF Pref. (Adaptative)	2611	1150	10 234	378
EMF Pref. (Shared)	1640	1090	7488	353
EMF Pref. (Shared + Adaptative)	1712	1024	7592	331
(b) Warmed Execution				
	Route Sensors	Region Sensors	Connected Segments	Switch Set
No Pref.	33 279	1129	11 389	271
EMF Pref.	203	167	783	105
EMF Pref. (Adaptative)	192	159	775	92
EMF Pref. (Shared)	221	161	837	105
EMF Pref. (Shared + Adaptative)	201	155	786	101

Table 4: Multiple Query Execution Time in milliseconds

(a) Cold Execution		
	NeoEMF/Graph	NeoEMF/Map
No Pref.	47 312	45 965
EMF Pref.	38 597	16 897
EMF Pref. (Adaptative)	36 438	15 785
EMF Pref. (Shared)	34 522	13 742
EMF Pref. (Shared + Adaptative)	32 458	13 005
Graph Pref.	31 479	–
(b) Warmed Execution		
	NeoEMF/Graph	NeoEMF/Map
No Pref.	23 471	47 823
EMF Pref.	1698	1896
EMF Pref. (Adaptative)	1583	1812
EMF Pref. (Shared)	1681	1793
EMF Pref. (Shared + Adaptative)	1598	1714
Graph Pref.	3489	–

7.3 Discussion

The main conclusions we can draw from these results (Tables 2 to 4) are:

- PrefetchML improves the execution time of all the queries on top of NeoEMF/Map for both scenarios. Execution time is improved by around 16% for *RegionSensors*, and up to 95% for *RouteSensors*. These results can be explained by the concurrent nature of the backend, that can be accessed by the query computation and the PrefetchML framework at the same time without execution time bottleneck. In addition, NeoEMF/Map does not contain any model element cache, and the second execution of the queries directly benefit from

the PrefetchML cache, showing execution time improvement up to 99% for the *RouteSensor* query.

- EMF-based prefetcher improves the execution time of first time computation of queries that perform complex and multiple navigations (*RouteSensors* and *ConnectedSegments* queries) on top of NeoEMF/Graph. EMF-Prefetcher also drastically improves the performance of the second execution of the queries: an important part of the navigated objects is contained in the cache, limiting the database overhead. However, when the query is simple such as *RegionSensors* or only contains independent navigations such as *SwitchSet*, 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.
- Graph-based prefetcher is faster than the EMF one on the first execution of the queries in both scenarios because prefetching queries can benefit from the database query optimizations (such as indexes), to quickly load 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 this improves the overall execution time. On the other hand, the Graph-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 vertices using a low-level query framework such as Mogwai [10].
- Sharing the cache between the PrefetchML framework and the running application globally improves the performances for all the queries, w.r.t the performances without sharing the cache. This is particularly true for simple queries such as *RegionSensors*, where the prefetcher and the query are computing the same information at the same time, and sharing the fetched elements reduces the concurrency bottlenecks.
- Enabling the *Adaptative Monitoring* component improves the performance of the first execution of simple queries such as *RegionSensors* and *SwitchSet*. The performance bottleneck of the EMF prefetcher for these kind of queries is detected by our heuristics (especially the *RuleEfficiency* one) and the component disables the corresponding rules to save time. On the other hand, our monitoring component adds a small execution overhead for queries that are more complex and where no rules can be disabled, because the monitoring instructions have a performance impact despite whether they are able to detect rules to disable or not. The second execution of the query shows that our monitoring component is now able to detect that the queried elements are already cached (due to the reset performed between the two executions), and thus capable of disabling the prefetching rules to avoid concurrent access of the underlying database, improving the query computation performance.

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 always positive for NeoEMF-based implementations. PrefetchML on top of standard EMF is also always better on a warmed execution but for ad hoc scenarios where most queries may be executed a single time and may not be related to each other, PrefetchML adds sometimes a small overhead to

the overall the query computation. A tuning process, taking into account the kind of ad hoc queries typically executed (e.g. their likely footprint), may be needed to come up with an optimal prefetching strategy.

In addition, we have shown that the *Adaptive Monitoring* component can improve the performance of simple queries by detecting and disabling rules that are harmful in terms of execution time. Nevertheless, it's also true that the definition of the monitoring thresholds is a complex task. We leave as further work to provide some assisting mechanism to semi-automatically define such thresholds based on the structure of the query at hand. A first step in this direction would be to use first the *Simple Monitoring* component to spot the bottlenecks in the execution.

8 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 back-end 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 usage scenario. Note that 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. PrefetchML embeds a monitoring component that partially addresses this issue by helping modelers to detect those undesired scenarios and optimize their existing plans. The PrefetchML framework has been implemented on top of the EMF as well as on NeoEMF/Graph, and experimental results 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 a 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). The PrefetchML DSL presented in Section 3 is a readable language that eases designers' tasks of 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. Finally, we have integrated a monitoring component in our framework that can provide a set of metrics allowing modelers to finely optimize their PrefetchML plans (6). This monitoring component is also used to automatically disable harmful rules during the execution.

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 possible optimization source would

be the logs of past runs of tools reading/updating models conforming to that metamodel. Process mining on those logs could suggest a first version of potentially good prefetching plans. Finally, we would also like to work on the adaptative monitoring component to allow dynamic rule creation/re-enabling based on the runtime discovery of frequent model access patterns.

Aknowledgement

This work has been partially funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No. 737494 (MegaM@Rt2 project) and the Spanish government (TIN2016-75944-R project).

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. Mogwai: a framework to handle complex queries on large models. In *Proc. of the 10th RCIS Conference*, pages 1–12. IEEE, 2016.
11. G. Daniel, G. Sunyé, and J. Cabot. PrefetchML: a framework for prefetching and caching models. In *Proc. of the 19th MoDELS Conference*, pages 318–328. ACM/IEEE, 2016.
12. M. Dimitrov, K. Kumar, P. Lu, V. Viswanathan, and T. Willhalm. Memory system characterization of big data workloads. In *Proc. of the 1st Big Data Conference*, pages 15–22. IEEE, 2013.
13. Eclipse Foundation. The CDO Model Repository (CDO), 2016. URL: <http://www.eclipse.org/cdo/>.
14. 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.
15. 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.
16. 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.
17. J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *Proc of the 33rd ICSE*, pages 633–642. IEEE, 2011.
18. 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.
19. M. Koegel and J. Helming. EMFStore: a model repository for EMF models. In *Proc. of the 32nd ICSE*, pages 307–308. ACM, 2010.
20. 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.
21. 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.
22. D. Lucrédio, R. P. d. M. Fortes, and J. Whittle. Moogles: A model search engine. In *Proc. of the 11th MoDELS Conference*, pages 296–310. Springer, 2008.
23. 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.
24. 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.
25. R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. *Informed prefetching and caching*. ACM, 1995.

26. R. Pohjonen and J.-P. Tolvanen. Automated production of family members: Lessons learned. In *Proc. of PLEES'02*, pages 49–57. IESE, 2002.
27. D. S. Ruiz, S. F. Morales, and J. G. Molina. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Proc. of the 34th ER Conference*, pages 467–480. Springer, 2015.
28. A. J. Smith. Sequentiality and prefetching in database systems. *TODS*, pages 223–247, 1978.
29. G. Szárnyas, B. Izsó, I. Ráth, and D. Varró. The train benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling*, pages 1–29, 2017.
30. Tinkerpop. Blueprints API, 2016. URL: blueprints.tinkerpop.com.
31. Tinkerpop. The Gremlin Language, 2016. URL: gremlin.tinkerpop.com.
32. 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 Jyväskylä, 2006.
33. K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proc. of the VLDB Endowment*, 8(11):1226–1237, 2015.
34. P. Zhu, G. Sun, P. Wang, and M. Chen. Improving memory access performance of in-memory key-value store using data prefetching techniques. In *Proc. of the 11th APPT Workshop*, pages 1–17. Springer, 2015.