Code-First Model-Driven Engineering: On the Agile Adoption of MDE Tooling

Artur Boronat
School of Informatics, University of Leicester, UK
artur.boronat@leicester.ac.uk

***Abstract*—Domain models are the most important asset in widely accepted software development approaches, like Domain-Driven Design (DDD), yet those models are still implicitly represented in programs. Model-Driven Engineering (MDE) regards those models as representable entities that are amenable to automated analysis and processing, facilitating quality assurance while increasing productivity in software development processes. Although this connection is not new, very few approaches facilitate adoption of MDE tooling without compromising existing value, their data. Moreover, switching to MDE tooling usually involves re-engineering core parts of an application, hindering backward compatibility and, thereby, continuous integration, while requiring an up-front investment in training in specialized modeling frameworks. In those approaches that overcome the previous problem, there is no clear indication – from a quantitative point of view – of the extent to which adopting state-of-the-art MDE practices and tooling is feasible or advantageous.**

In this work, we advocate a code-first approach to modeling through an approach for applying MDE techniques and tools to existing object-oriented software applications that fully preserves the semantics of the original application, which need not be modified. Our approach consists both of a semi-automated method for specifying explicit view models out of existing object-oriented applications and of a conservative extension mechanism that enables the use of such view models at run time, where view model queries are resolved on demand and view model updates are propagated incrementally to the original application. This mechanism enables an iterative, flexible application of MDE tooling to software applications, where metamodels and models do not exist explicitly. An evaluation of this extension mechanism, implemented for Java applications and for view models atop the Eclipse Modeling Framework (EMF), has been conducted with an industry-targeted benchmark for decision support systems, analyzing performance and scalability of the synchronization mechanism. Backward propagation of large updates over very large views is *instant*.

***Keywords*-Domain model, MDE, EMF, roundtrip synchronization, algebraic specification, performance analysis.**

I. Introduction

Agile methods do not provide guidance on design, and naïvely assume that it emerges from each iteration, relying on re-factoring to embed design decisions [1]. Such an approach results in a possibly rather diffuse implementation of the domain model. Domain-Driven Design (DDD) [2], widely accepted approaches for building complex software systems, regards application domain knowledge as the most important asset for their design, relying on a model of the domain for clarifying requirements with domain experts and with developers. Uludag et al. showed how DDD facilitates the integration of design practices, including the definition of a domain model, in agile software development for large systems [3]. However, it is difficult to systematize the design of the domain model according to DDD [4].

Model-Driven Engineering (MDE) encompasses the application both of modeling notation for representing such domain and of model management tools for their automated analysis and processing [5]–[7]. A number of studies have investigated the state of practice of MDE [8]–[14], identifying core challenges that affect the adoption of MDE, highlighting poor documentation, tool maturity, synchronization of models with code. In addition, Seybold et al. [13] remark a steep learning curve to use MDE tools, even for well-seasoned Java developers. To adopt modeling practices, Hutchinson et al. [8], [12] recommend a change of mentality in the organization as a progressive and iterative process. Whittle et al. [15] recommend more research on support for facilitating the creative process of modeling and a stronger emphasis on (simple) tools that can be embedded in software development processes.

We propose an approach to adopt non-intrusive MDE tooling in agile practices by cherry picking the most suitable tool, either MDE-agnostic or MDE-aware, depending on the task at hand. Developers can keep using the tools (and programming languages) that they are familiar with. MDE tooling can be applied for specific problems without requiring a strong commitment, thereby fostering a gradual application of modeling practices and, as a consequence, learning MDE. Our approach facilitates flexible modeling in technical spaces that are MDE agnostic, where a notion of metamodel does not exist. We are thus approaching flexibility from the point of view of interoperability. Building domain models can be done using programming languages familiar to developers, e.g., Java, facilitating the development of executable prototypes that need not be thrown away. While a model-first approach has undoubtedly many benefits for software development, practitioners usually prefer a code-first approach that delivers feedback quickly without having to resource to heavyweight modeling frameworks.

In our work, we consider the Eclipse Modeling Framework (EMF) [16], which is a mature Java-based metamodeling framework that unifies the use of Java, UML and XML that can be used both for developing data-driven systems and for designing domain-specific languages. Accord-

ing to [17], EMF is still by far the most popular MDE tool, within the Eclipse ecosystem, used in open source projects. EMF augments Java with the possibility of exploiting the meta-represented model through reflection facilities, XML serialization, and with non-trivial facilities, like ensuring systematic consistency of bidirectional associations, which are common in many applications and usually implemented ad hoc. Unfortunately, its adoption in the development of existing applications breaks backward compatibility, which is a highly regarded practice in agile software development as its absence breaks continuous integration (and delivery). Moreover, model-driven reverse engineering processes [18] may also require non-trivial data migration processes, making EMF rather a heavyweight technology.

While MDE-agnostic applications can be reverse engineered [18], [19] or adapted [20], a few approaches [21], [22] build interfaces with MDE technology for MDE-agnostic systems at run time. These approaches have been evaluated with a number of case studies used in industry. A quantitative analysis of performance and scalability has nonetheless not been performed with very large data sets. Therefore, it is unclear to what extent such interfaces can be used with state-of-the-art techniques. In this work, we have implemented a bidirectional synchronization mechanism that bridges MDE-agnostic (Java programs) and MDE-aware (EMF-based) systems at the data level using techniques from modern model transformation, including transformation of feature values on demand and incremental propagation of updates. This mechanism has been evaluated, considering performance and scalability perspectives, by using an industry-targeted benchmark for decision support systems, TPC-DS [23], that involves very large data sets. The conclusions of this research should help inform decisions when adopting MDE technology, depending on the scalability requirements of the system to be developed. In [21], such decisions could only be taken for model cardinalities of up to 116K elements, which are now raised to the millions.

The contributions of this work are summarized in this paragraph. In section II, an algebraic presentation of domain models (and view models) and of model instances, or system states, (and views) is developed, describing the view update problem at stake. The formalization is novel, revolving around the notion of feature value (as opposed to object) as unit of data, and relies on standard mathematical entities. This formalization scaffolds the conceptualization of our contribution independently of the chosen object-oriented programming language and metamodeling framework, fostering reuse of the proposed code-first MDE approach in a variety of use cases within the object-oriented paradigm, and in their implementation. In section III, we discuss a solution to the view update problem by presenting a synchronization model that is independent of technical spaces and then discuss how it has been realized in EMF-SYNCER, using state-of-the-art transformation techniques, including deferred execution and bidirectional, incremental propagation of updates, for syncing Java programs with EMF views at run time. EMF-SYNCER does not require modifying existing systems and guarantees backward compatibility. The formal representation of the synchronization model adopts a declarative stance, focusing on preconditions and key operation properties that facilitate their correct behavior, without having to deal with implementation intricacies. In section IV, EMF-SYNCER is analysed, in terms of performance and scalability by using the TPC-DS benchmark. The results of this empirical evaluation are then used to discuss to what extent, in terms of size of data sets, it is feasible to use EMF-based tooling. In section V, related work is discussed, concluding with some final remarks and future research directions.

II. VIEW UPDATE PROBLEM

Our approach is inspired in the view update problem [24] as we consider *view models* of existing domain models that are implicit, and possibly scattered, in object-oriented programs. A view model can be a faithful reflection of the domain model, or a projection of it, but extensions of view model classes involved in the synchronization are not supported. This design decision is justified by the fact that we are interested in a pragmatic application of MDE technology to non-MDE object-oriented programs, without losing information during synchronization. This does not preclude a user from using additional classes to store auxiliary information during the MDE process, whose existence will be forgotten by the synchronizer.

In the following subsections, we present an algebraic formalization of models, of their instances and of updates, concluding with an informal description of the synchronization problem. This formalization lays the foundations for presenting synchonization model in section III, independently of specific programming/modeling languages (and accompanying tool support).

A. Domain Models and View Models

An excerpt of a domain model representation in a Java program is given in Listing 1, which declares the domain class `StoreReturns` and decorates its fields with JPA annotations[1] that link the class to a table in a relational database, used in the experiments in section IV. The corresponding class in the view model, where only the fields of interest have been captured, is represented in Figure 1 and, in section III-C, the definition of a view model from a domain model is explained in more detail.

In this work, we are interested in capturing updates at the granularity level of (structural) features, within objects,

[1]The meaning of the used JPA annotations is not relevant for the presentation of the work and they have been included to show that there is non-MDE technology that is made available to MDE tools through the EMF-SYNCER.

and they are treated as independent units. Therefore, object-oriented domain models are represented as a set of (structural) feature values that are well typed, from which specific objects can be derived. We start by defining the notion of feature type and the notion of feature value

Definition 1 (Feature Type): Let t be a value type indexed by the set $\{0\}\cup\mathbb{N}$ of natural numbers as defined by the following grammar $t ::= dt \mid e \mid c \mid t \to t'$, where: dt denotes any primitive data type, such as String, Integer, Boolean, etc., available in Ecore (and their counterparts in Java) [16, pg 124]; e denotes any user-defined enumeration type; and c denotes any class type, representing sets of object identifiers o; and $t \to t'$ denotes sets of entries mapping a key of type t to a value of type t'. A feature type $f_m : c \to t$ where m is a record $(l, u, ordered, unique, cont, op)$ describing the usual properties of a feature: a lower bound l, an upper bound u, whether the feature is $ordered$ or $unique$, whether it is a $cont$ainment, and the opposite reference – if any.

Note that a feature type uses value types t that are indexed by $\{0\}\cup\mathbb{N}$, i.e. $\{0\}\cup\mathbb{N} \to t$. However, notation is abused and we represent them by t, for the sake of simplicity, but their values v_i are represented with an explicit indexed $i \in \mathbb{N}$.

A view model $v(\mathcal{M})$ is represented by its set of feature types. These can be either attribute values, when either $t = dt$ or $t = e$, or reference values, when $t = c$. Each feature type denotes a (possibly infinite) set of feature values representing the constituents of a model instance. A domain model $\mathcal{M}$ can be regarded as a partial specification of a view model, where some design decisions – e.g. multiplicity constraints – are not captured nor documented, probably not intendedly. Therefore the same representation can be used both for domain models, where the m component is optional, and for view models.

```
@Entity
public class StoreReturns {
  @OneToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "sr_returned_date_sk",
  referencedColumnName = "d_date_sk")
  private DateDim srReturnedDateSk;

  @Column(name = "sr_return_time_sk")
  private Long srReturnTimeSk;

  @EmbeddedId
  private StoreReturnsId srId;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "sr_customer_sk")
  private Customer srCustomerSk;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "sr_store_sk",
  referencedColumnName = "s_store_sk")
  private Store srStoreSk;

  @Column(name = "sr_return_amt")
  private Double srReturnAmt;
  ...
}
```

Listing 1. Excerpt of Java Domain Class

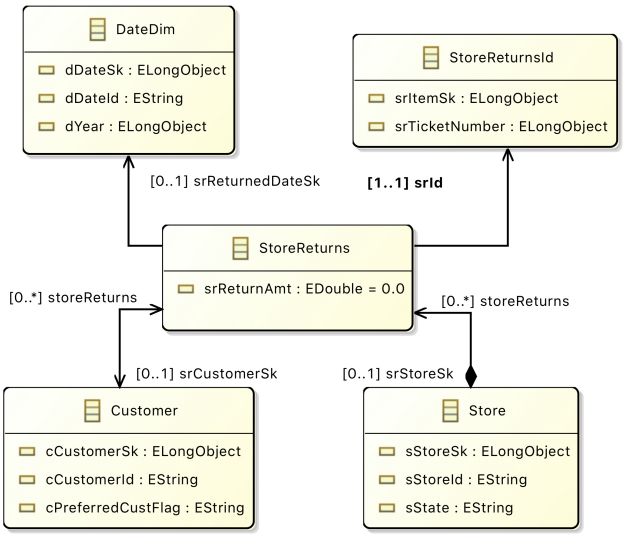

Figure 1. Excerpt of View Model used for the TPC-DS Benchmark

For example, considering the abbreviations `StoreReturns` as `SR`, `StoreReturnsId` as `SRId`, `Customer` as `C`, `Store` as `S`, `DateDim` as `D`, the class `StoreReturns` of Figure 1 is represented using the following feature types:

$$\texttt{srReturnAmt} : \texttt{SR} \mapsto \texttt{EDouble}$$

$$\texttt{srReturnDateSk}_{(l=0,u=1)} : \texttt{SR} \mapsto \texttt{D}$$

$$\texttt{srId}_{(l=1,u=1)} : \texttt{SR} \mapsto \texttt{SRId}$$

$$\texttt{srCustomerSk}_{(l=0,u=1,op=\texttt{storeReturns})} : \texttt{SR} \mapsto \texttt{C}$$

$$\texttt{srStoreSk}_{(l=0,u=1,op=\texttt{storeReturns})} : \texttt{SR} \mapsto \texttt{S}$$

Inheritance can also be considered in this representation. A naïve, and verbose, strategy consists in defining feature types for the class in which it is declared and for each of its subclasses. More precisely, when a feature f is declared for a class with name c with a type t and multiplicity constraints m, a feature type $f_m : c \to t$ is in the model together with each feature type of the form $f_m : c' \to t$, where c' is the name of a subclass of the class with name c. However, such formal representation is only used at a conceptual level in our framework and end users can keep using the corresponding notation in their technical space, Java programs or EMF models in this work.

B. Domain Model Instances and Views

We use the notation $v : t$ for denoting that a value v belongs to the domain of a type t, i.e. $v \in [\![t]\!]$. For representing values, we use families of values v_i, denoting sets $\{i \mapsto v\}$ for $i \in \{0\} \cup \mathbb{N}$. The notation for families of values v_i is abused and we also use it for representing a specific value v, indexed by a given natural number i.

This confusion is disambiguated from the context where the expression is used. When the context mentions *one* feature value then v_i refers to an indexed value, and it refers to a family of values v_i otherwise. Therefore, we will use the predicate $v_i : t$ to denote that all the values in the family v_i are typed with t.

Definition 2 (Feature Value): Given a feature type $f_m : c \to t$, a feature value is defined as a mapping $f : o \mapsto v_i$, where $o : c$ and $v_i : t$.

Given a model $\mathcal{M}$, a model instance m is represented as a set m of feature values $f : o \mapsto v_i$ that are typed with the feature types in $\mathcal{M}$, and the *instance of* relation, at model instance level, is denoted by $m : \mathcal{M}$. The notion of model instance may refer either to a domain model instance, or system state, or to a specific *view*. When the component m in a feature type $f_m : c \to t$ specifies constraints, the notion of feature value $f : o \mapsto v_i$ is enriched as follows:

- *cardinality constraints:* the cardinality of a value family v_i is constrained as follows: $m.l \leq |v_i| \leq m.u$;[2]
- *order* is captured by the index i of a value family v_i;
- *unique:* if a feature has the unique constraint, $m.unique$, then, for any i, the set $\{i \mapsto v\}$ for a particular feature value defines an injective mapping;
- *containment integrity:* if a feature $f : o \mapsto v_i$ in m is a reference and has the containment constraint, then for any object o' referred by f, there cannot exist another containment f' in m, which also contains the object o';
- *bidirectional associations:* if a feature f in m is a reference belonging to object o points to object o' and has an opposite reference f', then the feature value $f' : o' \mapsto o_i$, for a some i, also belongs to m.

C. View Updates

In our approach, MDE views, which are synchronized with a domain model instance, can be materialized through a persistence API in the MDE technical space, or it can be used as a virtual view at run time. In both cases, views can be updated and such updates are incrementally propagated to the underlying domain model instance on demand. There are two main cases of view updates: when root objects, w.r.t. the containment hierarchy, are added or deleted, and when feature values are updated. In what follows, we develop the notion of update that is used in our synchronization model and explain how an update is applied to a model instance, which can be a domain model instance or its associated view.

Definition 3 (Atomic Updates): Given a model $\mathcal{M}$ and a model instance m, such that $m : \mathcal{M}$, an atomic update is represented as a pair $(\mu, f : o \mapsto v_i) \in (\{upsert, delete\} \times f_m : c \to t)$ for a specific value v indexed by some i.

According to this definition, an atomic update can only affect *one* value in the family of values bound to the object o through f.

[2]In EMF, the value -1 is used to denote the UML value many, $*$, but we simply consider that $*$ is the infinite natural number.

The set Δ of updates for a particular model $\mathcal{M}$ comes equipped with a binary operation $\cdot : \Delta \times \Delta \to \Delta$ so that, for all given $\delta_1, \delta_2, \delta_3 \in \Delta$:

- $\cdot$ is associative, i.e., $(\delta_1 \cdot \delta_2) \cdot \delta_3 = \delta_1 \cdot (\delta_2 \cdot \delta_3)$,
- 1 is the identity element, i.e., $\delta_1 \cdot 1 = 1 \cdot \delta_1 = \delta_1$
- for any update $\delta \in \Delta$, there is an inverse delta δ^{-1} that cancels the former, i.e., $\delta \cdot \delta^{-1} = 1$.

Hence, $(\Delta, \cdot)$ represents the set of composite updates that can be defined as sequences of atomic updates $\delta \in \Delta$, including empty and singleton sequences.

Definition 4 (Update Application): Given a model $\mathcal{M}$ and a model instance m, such that $m : \mathcal{M}$, a delta $\delta \in (\Delta, \cdot)$ is applied to m, denoted by the expression $\delta(m)$, using the operator $\circ : (\Delta, \cdot) \times \mathcal{M} \to \mathcal{M}$ as follows:

$$(upsert, f : o \mapsto \{i \mapsto v\})(m \uplus \{f : o \mapsto (v_i \uplus \{i \mapsto v'\})\}) = (m \uplus \{f : o \mapsto (v_i \uplus \{i \mapsto v\})\}) \quad \text{(update)}$$

$$(upsert, f : o \mapsto \{j \mapsto v\})(m \uplus \{f : o \mapsto v_i\}) = (m \uplus \{f : o \mapsto v_i \uplus \{j \mapsto v\}\}) \ when \ \forall i, i \neq j \quad \text{(insert)}$$

$$(delete, f : o \mapsto \{j \mapsto v\})((m \uplus \{f : o \mapsto (v_i \uplus \{i \mapsto v\})\})) = (m \uplus \{f : o \mapsto v_i\}) \quad \text{(delete)}$$

$$1(m) = m \quad \text{(identity)}$$

$$(\delta_1 \cdot \delta_2)(m) = \delta_2(\delta_1(m)) \quad \text{(composition)}$$

where $\uplus$ represents disjoint union of sets.

In the equational presentation above, terms – with variables – are used in the left-hand side of the equations to denote a search of the relevant feature values in given model instance, through pattern matching. An upsert update is characterized with the equations *(update)*, which updates the value v' at position i with the new value v in the feature f for object o, and *(insert)*, which inserts a new value at an index j that is not used. A delete update is characterized with the equation *(delete)*, where the expression $\{i \mapsto v\}$ characterizes the set of indexed values with the value v, in case there are duplicates. Hence, deletion has set semantics and all entries containing the value v are deleted, irrespectively of their indexes. Equations *(identity)* and *(composition)* define the base case of the inductive definition of the application operator over the structure of updates and that update application is compositional.

When considering updates, the following assumptions are taken into account: an atomic update can be applied if it refers to a valid feature type – i.e., updates are type preserving – and to an existing object *(type preservation)*; a feature value whose cardinality is the lower bound cannot be subject to a deletion *(lower bound)*; a feature value $f : o \mapsto v_i$ with the unique constraint cannot be subject to an insert of a value that is already contained in v_i; a feature value whose cardinality has met the upper bound cannot be subject to an insertion, although it may be updated *(upper bound)*; when a reference value is updated and this reference has an opposite, the update of the opposite reference is included in the update

(bidirectional reference integrity); an update describing a move of an object from one container to another one is defined in terms of an atomic deletion of the object from the source container, *followed by* a upsert of the object to the target container *(containment integrity)*; when an object is removed, all of its containments are removed recursively and resulting dangling references are also removed – such deletions are part of the update *(delete cascade semantics)*. These pre-conditions guarantee the correct behaviour of the update application operator.

In this work, the consistency relation $\mathcal{R}$ between domain models $\mathcal{M}$ and their view models $v(\mathcal{M})$ is defined by an isomorphism that maps feature types by name, taking into account the name of their class as well. A domain model instance m, $m : \mathcal{M}$, and a view v, $v : v(\mathcal{M})$ are consistent when $(m, v) \in \mathcal{R}$. An update δ on m introduces an inconsistency that needs to be repaired by propagating δ to the view v. Symmetrically, when an update δ is applied to v, consistency needs to be restored by propagating it to the model instance m. With this problem in mind, we proceed to discuss a solution in the following section.

III. Synchronization of Feature Values

In this section, we present our solution to the view update problem for domain model instances in the form of a conceptual synchronization model that is independent from technical spaces. The synchronization model employs a synchronization policy for specifying of the consistency relation between domain models and view models, to be taken into account when their instances are synchronized.

This synchronization model has been implemented in EMF-Syncer for obtaining EMF views from Java domain model instances at run time, and for keeping them in sync incrementally. EMF-Syncer relies on the reification of Java domain models as EMF view models, which can abstract away implementation details from the Java source code while making the domain model more stringent using multiplicity and structural constraints.

In what follows, the default synchronization policy is described, then the tool-agnostic synchronization model is presented together with a brief description of its realization in EMF-Syncer. At the end, a method to obtain view models, when these are not available, from (DDD) domain models is discussed embedding MDE practices in agile environments.

A. Default Synchronization Policy

The (default) domain-independent policy maps model instances by relying on a one-to-one mapping between their corresponding feature types, based on the name of the class for which they are defined and on their name. This policy is useful when the original domain model was clearly defined and the view model could be extracted faithfully, either from the whole domain model or from an excerpt of it.

The main synchronization policy is declared as a map

$$P : \Omega \times (\Delta_{\mathcal{M}}, \cdot) \to \Omega \times (\Delta_{\mathcal{M}'}, \cdot)$$

where Ω is the domain of synced links between object identifiers $\mathcal{O} \to \mathcal{O}$, which are used to map objects in the source technical space to their counterparts in the target technical space. P is an overloaded function, for updates and for values, that takes a store of synced links and a value in a source technical space and obtains its counterpart in the target technical space, while tracking new synced links.

For mapping values, the definition of P proceeds by cases, depending on the type of the value:

$$\begin{aligned} &P(\omega, v) = (\omega, v) && \textit{when } v : dt \textit{ or } v : e \\ &P(\omega, v) = (\textit{let } o' = \textit{fresh}(R(c)) \textit{ in} \\ &\quad ((v \in \omega)\,?\,\omega : \omega \cup (v \mapsto o'), \\ &\quad (v \in \omega)\,?\,\omega[v] : o')) && \textit{when } v : c \end{aligned}$$

The most interesting case is when the value is an object identifier. If it has been synchronized already, and it exists in the store $v \in \omega$, the target object identifier is simply retrieved from the store with the expression $\omega[v]$. When the object identifier is not synchronized, a fresh identifier of the target class $R(c)$ is inserted in the store ω of synced links with the expression $\omega \cup (v \mapsto o')$ and returned.

The propagation P of atomic updates is defined in equation *(syncing)* below, which translates values using the expression $P(\omega, v)$ and then updates the two components of the co-domain, the store of synced links and the view.

$$\begin{aligned} &P(\omega, (\mu, f : o \mapsto v_i)) = \\ &\quad (\omega \cup P(\omega, o).\omega \cup P(\omega, v).\omega, \\ &\quad (\mu, (R(o.\textit{type}, f).f : P(\omega, o).v \mapsto P(\omega, v).v))) && \text{(syncing)} \\ &P(\omega, 1) = (\omega, 1) && \text{(identity)} \\ &P(\omega, \delta_1 \cdot \delta_2) = (P(\omega, \delta_1).\omega \cup P(\omega, \delta_2).\omega, \\ &\quad P(\omega, \delta_1).v \cdot P(\omega, \delta_2).v) && \text{(composition)} \end{aligned}$$

In equation *(syncing)*, dot notation is used to access the components of $P_\omega(v)$, using $\_.\omega$ for projecting the store of synced links and $\_.v$ for projecting the value. In addition, the expression $o.\textit{type}$ is used to obtain the class name of the object type. Equations *(identity)* and *(composition)* define the extension of P to a function on sequences of atomic updates.

Renaming of feature types and of classes are defined with the R function using the identities $R(c) = c$ for class names c and $R(c, f) = (R(c), f)$ for names f of features belonging to class c, where the expression $R(c, f).f$ obtains the target feature name.

The operations P for updates and values have unique inverse maps P^{-1}. For updates, P^{-1} is declared as

$$P^{-1} : \Omega^{-1} \times (\Delta_{\mathcal{M}'}, \cdot) \to \Omega^{-1} \times (\Delta_{\mathcal{M}}, \cdot)$$

Its definition proceeds analogously, by replacing P with P^{-1}, ω with ω^{-1}, and R with R^{-1}.

B. EMF-SYNCER

The synchronization model has been implemented both for domain models embedded in Java programs and for EMF view models in EMF-SYNCER, available at https://emf-syncer.github.io. A syncing session in EMF-SYNCER has two main stages: an initial stage where feature values are synchronized on demand, and a (backward) incremental propagation stage where updates are propagated from the view to the underlying domain model instance.

In the initial stage, the domain model instance is described as a big composite update and the result of this initial propagation is that the source non-EMF model instance and the EMF view are synchronized in the store ω of synced links. Two synchronization strategies have been implemented, providing support for push-based initial synchronization, in which all of the feature values are mapped to target feature values, and for pull-based initial synchronization, in which feature values are only initialized when they are accessed in the target EMF application.

In the second stage, updates to either non-EMF or EMF views are incrementally propagated in a push-based fashion. The push-based strategy consists in applying the synchronization policy to the view v, which is initially empty, with $(P(\omega, \delta_s).v)(v)$, where ω is empty initially and reuses the initialized store of synced links for subsequent incremental propagations. For propagating updates δ_t back to Java domain model instances m, the inverse synchronization policies are applied with $(P^{-1}(\omega^{-1}, \delta_t).v)(m)$.

For the initial synchronization of model instances, the tool takes advantage of the fact that there are no updates of existing feature values or deletions of objects, and the representation of model instances as deltas is circumvented, as the type of the delta, μ, is known. The computational cost of the push-based synchronization strategy is linear in the size of feature values, which can be undesirable for the initialization phase for very large model instances. Subsequent incremental propagation of updates, once the model instance has been synchronized with its view, is usually *instant*, as shown in the experimental results, because the size of updated feature values tends to be a fraction of the original model instance size. That is, updates can be propagated in *real-time* – in less than 1 $ms.$ for updates affecting about 10^3 objects and in $ms.$ for updates affecting about 10^5 objects – where objects may contain several feature values.

The implementation of this synchronization policy makes use of the fact that feature values can be indexed by the object identifier and by the feature name so that the search for the feature value to be synchronized is obtained in constant time. The pull-based synchronization strategy, only available for the initial stage at the moment, is also linear but only in the size of the feature values that are *accessed* in the actual computation in the EMF application, thus avoiding unnecessary propagations for executing the task at hand. We refer the reader to section IV, where we justify the pragmatism of the pull-based strategy when dealing with benchmark model queries and updates.

C. Domain Model Reification

MDE-agnostic domain models can be explicitly represented using an appropriate modeling language. This process is called reification[3] and, in this work, is also used to concretize design decisions, that may have been deferred or not taken inadvertently. In this section, we discuss advantages of using MDE for developing applications, a number of use cases where MDE tooling can be adopted and a method for building EMF view models from Java source code.

Modeling and Abstraction: Regarding the base (source) code, reification can be regarded as the means to embed modeling practices in agile software development processes in projects where modeling is used at different levels of formality: when there is no clear domain model in the application, and when there is a domain model. In the first case, reification becomes a modeling process that may help either to capture the main concepts and structural constraints intermingled in the source code, to refactor the application, to abstract the existing domain model, or to document the design (architecture) of the system. Abstraction is supported by selecting classes and structural features that need to belong to a view model. In the second case, reification brings advantages that stem from the level of expressiveness in the target modeling notation, refining a domain model by adding multiplicity (e.g. in structural features, uniqueness, ordering, optionality) and structural constraints (e.g. in references, bidirectional references, containments). That is, reification helps in refining design decisions that could not be captured earlier due to a lack of appropriate notation for capturing the domain model in the source code.

Use Cases: Depending on the existence (or lack) of a view model, there are two main types of use cases in which this reification process can take place: when the view model does not exist, and when the view model already exists. In the first case the view model has to be defined explicitly, which involves using a modeling tool. Use cases that fall under this category correspond to re-engineering of an application as a model-driven application or to the application of MDE technology to implement parts of the business logic of the application. In the second case, reification of the domain model has already happened. Use cases that fall under this category correspond to development of executable interfaces between MDE-agnostic and MDE-aware systems, re-engineering of an application as a model-driven application, or even model versioning when the source domain model is already explicitly represented.

[3]https://en.wikipedia.org/wiki/Reification_(computer_science)

A Reification Method Using EMF: In what follows, we describe a reification method that captures the static part of a domain model, implicitly available in the source code, using the technologies that we have used to implement our synchronization model. On the one hand, we use Java programs as the domain model and EMF as the target modeling framework, where view models are represented as Ecore models. EMF provides an import facility that reifies Java programs as Ecore models. Such facility assumes that object types are defined as interfaces, which can then be annotated with modeling information (e.g. multiplicity constraints and structural constraints mentioned above) [16] that is not present in the source code. On the other hand, the implementation of the domain model may include domain-specific abstractions, e.g. queues, of the base domain model that do not have a direct representation in EMF. Such abstractions need to be modelled in the view model explicitly, possibly requiring a domain-specific synchronization policy.

IV. Empirical Evaluation

In this section, we investigate whether the proposed synchronization model is a pragmatic approach to adopt MDE tooling by checking to what extent EMF-Syncer allows us to apply EMF-based tools to a representative Java program at run time. Two main research questions are considered:

RQ1: *Does the time overhead imposed by the synchronization algorithms, both during the initial phase and during the incremental propagation phase, hinder the pragmatism of the approach?* For the analysis of this question, two dimensions are considered: the time taken for syncing feature values, and the time taken to perform tasks (queries and updates) with them.

Regarding the first dimension, an implementation of a basic algorithm that maps Java objects to EMF objects is considered. This algorithm is domain-specific – that is, specific to a domain model, – can only be executed in batch mode and does not offer synchronization. Therefore its raw performance sets a baseline for comparing the overhead incurred by additional functionality.

Regarding the second dimension, we are using the TPC Benchmark DS [23]. The purpose of TPC benchmarks is to provide relevant, objective performance data to industry users and the current version of the TPC-DS benchmark considers emerging technologies, such as big data systems. This benchmark allows us to generalize our findings to common functionality found in decision support systems used in industry and that may involve very large data sets.

Five tasks have been analyzed: a selection of the store returns corresponding to a given customer (**Q1**) together with an update that adds one store return to the customer (**B1**); an invariant that checks that store returns have valid identifiers (**Q2**); and a typical big data query, extracted from the benchmark [23, B.1], which finds customers who have returned items more than 20% more often than the average customer returns for a store in the state 'TN' for the year 2000 (**Q3**), together with an update for all of the retrieved customers that deselects them as preferred customers (**B3**), by setting a feature value in each customer instance.

These five tasks have been implemented in SQL, using the query template facilitated in the benchmark specification for **Q3**, in Java and in EMF. The SQL implementation executes SQL native queries via the JPA interface considering that they only return lists of strings, minimizing fetching time. The Java and the EMF implementations are exactly the same but for the implementation of the domain classes, which are declared in a different namespace. In addition, the Java domain classes are linked to a MariaDB relational database[4] via JPA and the EMF domain classes only contain the feature values, thus forming a view model, that are relevant for the tasks to be computed. The correctness of the Java/EMF queries is cross-checked with the results obtained with the corresponding SQL queries for each database every time a query is executed.

As a collateral effect, the experimental setup for the benchmark is also illustrating the use case where data stored in a relational database is made available as EMF model instances, reusing JPA providers widely used in industry, e.g. Hibernate, without having to rely on EMF persistence solutions, such as CDO, which requires knowing both EMF and JPA [13]. On the other hand, Java objects are assumed to be in memory at run time because the problem of fetching model instances from disk or from an external store is out of the scope of this work. Therefore, loading time is not relevant for the research questions at hand and has been excluded from the analysis.

RQ2: *Does the synchronization algorithm scale well with respect to the size of the given data set in the Java application?* For the analysis of this question, databases of different sizes have been generated using the Java port of the TPC-DS benchmark[5], which allows for no sexism in the generated data set[6] and this has been the preferred option. Table I shows a summary of model instance sizes involved in the experiments. A size factor on a logarithmic scale (with base 2) has been used to increase the size of the factor in order to be able to experiment with a number of databases in a standard computer. The baseline object mapper traverses the graph of Java objects in memory and has been used to obtain the cardinality of the model instances involved, noting that the actual number of objects in memory is twice the size mentioned in the table, given to the Java/EMF duality that EMF-Syncer maintains. The other columns show the cardinality of the model instance that has been synchronized, after the corresponding query was executed (**Q1I**, **Q2I** and **Q3I**), and the cardinality of the set of feature values that

[4]Version 10.2.11-MariaDB with INNODB 5.7.20.

[5]https://github.com/Teradata/tpcds

[6]In the official TPC-DS benchmark, only males are chosen as managers.

is propagated incrementally after the update was performed (**Q1B** and **Q3B**).

A. Experiment Setup

The experiments were performed on a computer with a quad-core processor 2.5 GHz Intel Core i7 and 16 GB RAM, using Java SRE 18.9, using Java HotSpot 64-Bit Server VM 18.9, and the default G1 Garbage Collector was configured to start concurrency GC cycles at 70% heap occupancy with a maximum GC pause of $200ms$. These parameters were coupled with a maximum heap size limit of 12GB and an aggressive heap usage strategy, minimizing interferences with processes external to the experiment. Pauses due to garbage collection should be avoided for small databases where response time is of a few ms, or even μs.

Factor	Mapper	Q1I	Q1B	Q2I	Q3I	Q3B
2^{-4}	55463	35	3	47850	31538	883
2^{-3}	84166	13	3	71648	48342	1338
2^{-2}	167242	5	3	143510	95487	2708
2^{-1}	332987	5	3	287466	189253	5654
2^{0}	664036	9	3	575028	376522	11727
2^{1}	1288030	11	3	1150570	712745	23231
2^{2}	2527000	11	3	2299130	1377435	46098
2^{3}	5006596	15	3	4601328	2705932	54806
2^{4}	9408198	61	3	9206178	4805109	124447

Table I
SIZES (NUMBER OF OBJECTS)

Figure 2 shows the time taken (in ms) by the three queries in SQL, Java and EMF. The Java application is using lazy (JPA) associations between domain classes and these are initialized when loading data from the database. In addition, the Java queries have been executed twice minimizing the impact of lazy object loading (**Java 1st**) and only the time taken by the queries in the second iteration (**Java 2nd**), where objects are already loaded in memory, has been considered. In EMF, each query has been performed twice to measure the impact of deferred initialization in the synchronizer: the first time (**EMF 1st**) corresponds to query time mixed with feature value initialization while the second time (**EMF 2nd**) is mostly query time. In the second iteration, there are still feature values whose value corresponds to null values (in the case of one-bounded references) or to default values and EMF-SYNCER does not distinguish them from EMF feature values that have been initialized with default values already. Performing a check to avoid such cases turned out to be more expensive than performing the translation.

Figure 3 shows the time taken (in ms) by the baseline object mapper and by the synchronizer for each query. The time taken by the baseline mapper reflects the time taken by the translation of the whole domain model instance into EMF. For each query, the time taken by the synchronizer has been monitored in two stages: initial synchronization together with deferred initialization of features of values when executing the correponding query (**Initialization**), excluding the time taken by proper query execution; and during backward propagation (**Backward**). In **Q2**, we have not considered updates and, consequently, there is no **Backward** component.

The times reported are average times (in ms) extracted from 1000 iterations for experiments for size factors up to 2^{-1}, and from 100 iterations for the rest of cases, but for size factor 2^4, where 10 iterations were run. The first 10% of results have been considered as warm up exercise for the JVM and have been excluded from the analysis. The raw data together with the tools to replicate the experiment are available at https://github.com/emf-syncer/emf-syncer.tpcds.

B. Analysis and Interpretation of Results for RQ1

The initial synchronization algorithm is linear in the size of the list of objects given and the propagation algorithm (both from Java to EMF and from EMF to Java) is linear in the size of the update. From the results obtained, plotted in Figure 2, Java queries (e.g. **Q3**), including queries over synchronized EMF views, are more efficient than the SQL queries run on MariaDB via JPA, once the excerpt of the database relevant for queries is available in memory as object instances. The reason for this being that contextual information is already available in memory for evaluating the query whereas SQL queries have to traverse very large tables in the database to find that initial context.

Figure 2 shows that EMF-SYNCER introduces some overhead when computing queries over EMF objects when comparing the same queries over Java objects, even when deferred synchronization is resolved the second execution in **Q1**, **Q2** and **Q3**. This is due to the instrumentation in EMF code performed by EMF-SYNCER, which, for example, needs to check whether an EMF feature value has been initialized or not in the presence (or lack) of default values in a Java feature value. Time used in EMF queries (**EMF 2nd**) should thus be understood as the performance of queries run on synchronized EMF views.

Synchronization time is split in the time used in the initial phase and in deferred synchronization of feature values once they are requested. EMF-SYNCER thus balances the synchronization workload and tailors it to the computation task to be performed. Both times are aggregated as **Initialization** time and, in Figure 3, it can be seen that the initial synchronization of 664K objects takes about one second. In practice, the end user application will experience a split of this time in the initial phase and the rest when feature values are requested for the first time in a query. Deferred synchronization resolution occurs *once and for all*, that is, a feature value is synchronized the first time it is accessed, and stays synchronized for subsequent accesses. EMF feature values corresponding to Java feature values

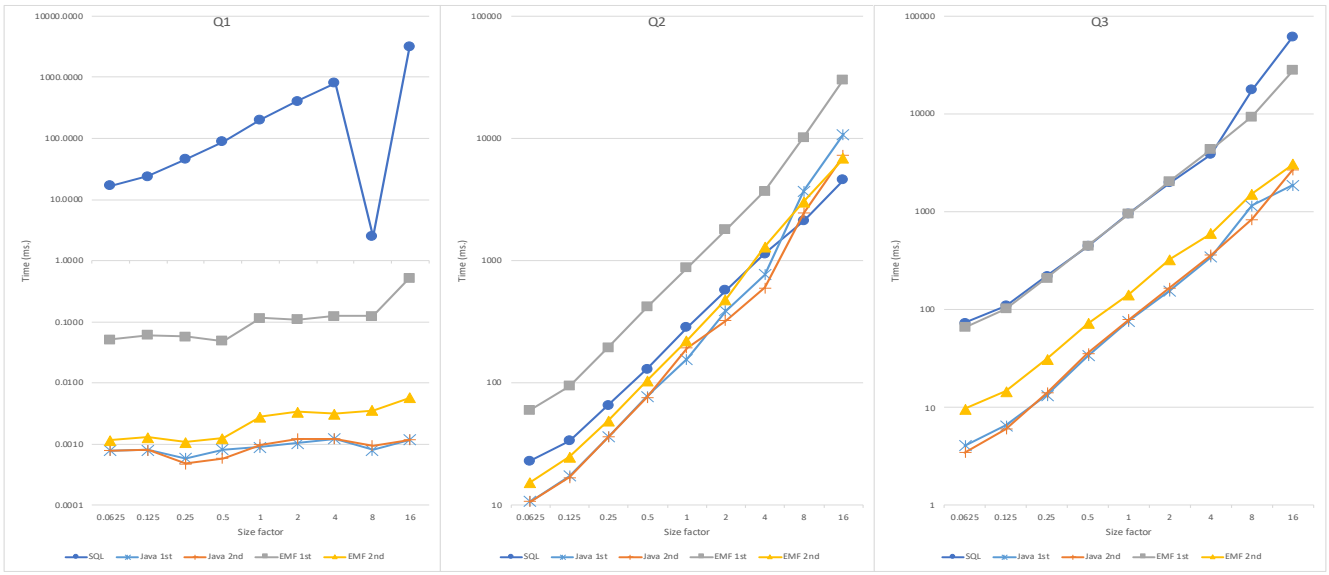

Figure 2. Query Times and Scalability (**Q1** left, **Q2** middle, **Q3** right)

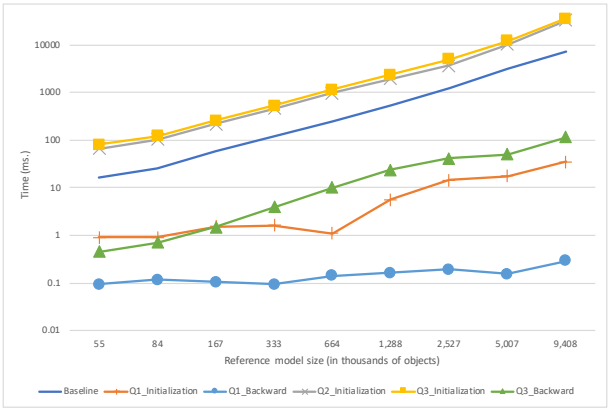

Figure 3. Scalability of Synchronization Mechanism

that are not initialized are a special case as EMF-SYNCER cannot distinguish them from initialized feature values.

Regarding synchronization times, in Figure 3, while the baseline mapping of objects (**Baseline**) is more efficient when comparing it with the initial synchronization of large models, subsequent incremental synchronizations (**Backward**) offer a much better performance (more than one order of magnitude) than re-executing the baseline mapping from scratch, especially for large model instances.

Therefore, when adopting EMF tooling at run time in a EMF-agnostic JVM application, while there is an initial synchronization overhead, considering both the initial phase and deferred synchronization of feature values, this overhead is reasonable and balanced between initial synchronization and deferred synchronization. EMF-SYNCER tailors the synchronization effort depending on the computation task being performed, depending on the access to feature values.

C. Analysis and Interpretation of Results for RQ2

We consider two cases, when the resulting model instance is small and when the initial data set is very large.

Table I shows that EMF-SYNCER only syncs those feature values that are accessed depending on the task being performed. For example, while the batch object mapper traverses 9.5M objects for a size factor of 2^4, EMF-SYNCER only needed to sync 4.8M objects for **Q3**.

Figure 3 shows that EMF-SYNCER can synchronize small excerpts of a model instance (in **Q1**) in a few ms independently of the total size of the model instance. Moreover, initial synchronization time together with deferred synchronization time exhibits a reasonably linear growth w.r.t. the size of model instance, which scales relatively well, when compared with the performance of the baseline mapper.

Incremental propagation of updates is efficient, even for large updates. For example, updating 124K objects takes about 114 ms, in **Q3_Backward**, for a size factor of 2^4. Forward incremental propagation (from Java to EMF, once model instances are synchronized) is based on the same algorithm and the same data structures and it, therefore, exhibits a similar performance.

For large model instances, e.g. 2^3, in **Q3**, it can be observed that initialization time (the sum of initial synchronization time and deferred synchronization time) together with query time **Q3 (EMF 1st)**, excluding loading time, is actually better than the time used to computer the same query with SQL. This shows that the scalability of EMF-SYNCER performance can be better than the scala-

bility of query evaluation in industry database management systems in some scenarios. The reason is the availability of contextual information in memory that is readily exploited when evaluating navigation expressions, which need to be computed when using SQL. From the current experiment results we cannot generalize this statement, however, as there are situations (e.g., **Q2**) where relational operations, together with appropriate indexing, lead to more efficient results, subject to availability of expertise in both database design and business domain.

D. Threats to Validity

Our synchronization model, and EMF-SYNCER, works at the level of feature values. In Table I, the size of a model instance has been presented in terms of the cardinality of its constituents, without decomposing the basic unit of data in the object oriented paradigm, the object. In addition, using this granularity also facilitates comparison with other benchmarks that use EMF for processing very large models. One must take into consideration however, that performance results may vary depending on the density of feature values in objects and in how many of them are used.

The size of a database is generated according to a size factor but the actual data is randomly generated and the cardinality of the results obtained by the queries may not be correlated for each database. For example, the time **Q3B** for the size factor 2^3 corresponds to the update of 54K objects and the time **Q3B** for the size factor 2^2 is for 46K objects, and the actual size increase does not correspond to the logarithmic scale used for the overall database size. Object sizes together with an explanation of each task help in scrutinizing the obtained results objectively.

V. RELATED WORK

The adoption of MDE tooling at run time was investigated in [22], [25], [26] where given a *system meta-model*, providing the types of the runtime data, and an *access model*, specifying how to manipulate the data through the API of the system, the tool Sm@rt automatically generates a synchronizer, which maintains the consistency of the model (view in our work) and the system state (model instances in our work) in the presence of concurrent updates, either to the model or to the system state. Sm@rt has been used to implement the execution model of QVT for synchronizing *models@runtime* in [27], [28].

Sm@rt and EMF-SYNCER facilitate the adoption of MDE tooling in MDE-agnostic systems at run time. However, in Sm@rt, concurrent updates are allowed, while synchronization is controlled by the client program in EMF-SYNCER. More importantly, Sm@rt assumes the existence of an *access model* that adapts the existing system, in order to inspect its state at run time while EMF-SYNCER can inspect the state of Java programs out of the box. Song et al. studied the feasibility of adopting MDE tooling with a number of case studies and performance was assessed with models containing less than 1000 elements [28]. We have taken this research one step further in order to scrutinize, from a quantitative point of view, the extent to which it is pragmatic to use MDE tooling in representative industrial scenarios, which may involve large data sets.

A number of works, see survey by Bruneliere et al. [29], use views over models, or *model views*, to facilitate interoperability between different modelling languages. Such approaches rely on an explicit representation of models, which are considered views, i.e. data extracted from a base model and that conforms to a viewtype given as a metamodel, and usually offer expressive view type definition languages or mechanisms to infer views based on algorithms that rely on the existence of metamodels.

Our work focuses on *view models* instead, where domain models that are used implicitly in software development and that form the core part of the executable system are reified as models, that can coexist with the system at run time. On the other hand, relying on the observation that an EMF model can be realized as a Java program, our approach can also be applied to define model views.

Our approach is related to reverse engineering processes [30] that extract designs from source code. This is normally done with two purposes: program comprehension and design recovery. Program comprehension processes normally involve three phases: extraction of data from the source program, their representation as models, and their visualization and exploration for obtaining information. Tilley [31] presents a layered modeling to approach hyperstructure understanding by using the reverse engineering environment Rigi, where a data model is represented using a relational model, a conceptual model is represented using the Telos language and a semantic network helps in visualizing and exploring information in the domain model. Freitas et al. [32] describe the use of DSLs for building reverse engineering tools. In particular, an extraction language is used to specify the extraction and storage of model data from a source program, and a model language that is used for generating tools that represent models graphically, facilitating their visualization and exploration. Building specifications in those DSLs is a manual process that speeds up the reverse engineering process. From a conceptual point of view, our approach assumes that there is already an object-oriented domain model implemented in the source code. While reifying the domain model may help in improving it, as explained in section III-C, the main objective is both to reuse the design knowledge implicit in an executable program and to enable the application of MDE techniques, and their associated tooling, while maintaining backward compatibility. For example, using MDE tooling, e.g. Sirius [7], graphical editors can be developed for the domain model in

[7] https://www.eclipse.org/sirius/overview.html

order to visualize and explore views – and even edit them – with the added advantage of updating the domain model instance in the underlying Java program at run time.

Design recovery is combined with recognition of architectural patterns and of component-based systems in [33]–[37]. Our approach does not consider any special notion of module or component and it can only infer architectural aspects that are already present together with the domain model in the source code. Going beyond design recovery, DeBaud and Rugaber [38], [39] related domain analysis to re-engineering by proposing a method that constructed executable object-oriented domain models that both capture the program purpose and record it using abstract classes, which can then be refined to implement the new program. This method can be applied for building object-oriented programs from legacy systems implemented in languages that are not necessarily object oriented, improving productivity in the re-engineering process. By relying on standard object oriented constructs, such as abstract classes, their resulting domain models are eligible to be mapped to view models using our approach, thereby, enriching domain engineering with the application of MDE techniques and tooling.

In model-driven reverse engineering [18], the abstract syntax of a program is represented in a model (code can be regarded as models), which can then be subject to automated analysis, including code generation. For example, Modisco [40] extracts models from Java programs that conform to a Java metamodel using a batch process, and the Epsilon JDT driver [41] performs this extraction process incrementally. In our approach, a form model extraction occurs during reification. However the objective is different, while reification aims at defining a view model of the Java program, which abstracts implementation details and refines modeling decisions, in reverse engineering model extractors produce an abstract syntax graph, abstracting away concrete syntax, that contains implementation details in full. While the second approach is fully automated, it cannot be informed by designers and domain experts.

Ecoreification [20] is an approach for re-engineering Java applications atop EMF, which provides backward compatibility at the service level. The approach extracts Ecore-conforming metamodels from Java code, from which code that unifies Java classes and EMF classes is obtained. While this work is motivated at the metamodel level, to enable the application of model transformation languages, like ATL, or graphical metamodeling editors, like Sirius, it can also be applied for extracting domain models as motivated in our work and is, therefore, the most relevant related work.

The key idea behind the Ecoreification approach is to convert the original Java classes into adapters that delegate accessors and mutators to their counterpart, newly-generated EMF classes. In this way, the responsibility for defining state is shifted to EMF classes, creating a strong commitment with EMF, which becomes the interface for the persistence layer of the application. While application state can still be accessed via the original Java interface, thus ensuring backward compatibility at the service level, the implementation of the persistence layer needs to be changed, which is a drastic decision requiring migration of data for existing applications.

VI. Conclusions and Future Work

We have focused on the pragmatic aspects of applying MDE tooling in agile environments that adopt a *code-first* stance, prioritizing software over documentation. A roundrip synchronization model has been presented for extending MDE-agnostic software applications with MDE-aware functionality, while preserving their original semantics, at run time. Data synchronization occurs at the level of feature values by using views that need not be materialized.

This synchronization model has been implemented in EMF-Syncer, providing deferred initialization of feature values and incremental propagation of updates, whose performance and scalability has been analyzed from a quantitative point of view. EMF-Syncer can be used as an infrastructure tool to embed modeling practices in agile software project, which fosters experimentation with MDE tooling in a non-intrusive manner. The tool, available as an IDE-independent library at https://emf-syncer.github.io, is easily reusable as a Maven dependency.

The results from our experiments with EMF-Syncer show that the propagation of large updates over models instances consisting of millions of objects is *instant*. Moreover, EMF-Syncer can perform an initial synchronization with millions of objects in reasonable time and that, for small cardinalities, it is performed in instant time up to 500K elements. In our experiments, the cardinality of elements considered has been increased by one order of magnitude with respect to the cardinality of the specimens considered in previous experiments, e.g., [21], thus providing a better insight on the capabilities of state-of-the-art MDE technology for scenarios involving large data sets.

While preconditions required to facilitate the correctness of the synchronization policy have been enumerated, formal properties of the synchronization model, which relies on an isomorphic consistency relation between domain models and view models, have not been elaborated as part of the results. In future work, we would like to explore more flexible ways of specifying consistency relations, and their formal properties, in order to implement effective interoperability bridges across technical spaces for applying MDE tooling practices wherever they add value.

Acknowledgment

The author would like to thank the anonymous reviewers for helping in improving the article.

References

[1] R. L. Nord, I. Ozkaya, and P. Kruchten, "Agile in distress: Architecture to the rescue," in *Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation*. Springer, 2014, pp. 43–57.

[2] Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] Ö. Uludag, M. Hauder, M. Kleehaus, C. Schimpfle, and F. Matthes, "Supporting large-scale agile development with domain-driven design," in *Agile Processes in Software Engineering and Extreme Programming - 19th International Conference, XP 2018*, ser. LNBIP, vol. 314. Springer, 2018, pp. 232–247. [Online]. Available: https://doi.org/10.1007/978-3-319-91602-6_16

[4] E. Landre, H. Wesenberg, and J. Olmheim, "Agile enterprise software development using domain-driven design and test first," in *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. ACM, 2007, pp. 983–993.

[5] S. Kent, "Model driven engineering," in *Integrated Formal Methods*. Springer, 2002, pp. 286–298.

[6] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006. [Online]. Available: http://dx.doi.org/10.1109/MC.2006.58

[7] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice, Second Edition*, ser. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017. [Online]. Available: https://doi.org/10.2200/S00751ED2V01Y201701SWE004

[8] J. E. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE*. ACM, 2011, pp. 633–642. [Online]. Available: https://doi.org/10.1145/1985793.1985882

[9] J. E. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE*. ACM, 2011, pp. 471–480. [Online]. Available: https://doi.org/10.1145/1985793.1985858

[10] J. Whittle, J. E. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "Industrial adoption of model-driven engineering: Are the tools really the problem?" in *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS*, ser. LNCS, vol. 8107. Springer, 2013, pp. 1–17. [Online]. Available: https://doi.org/10.1007/978-3-642-41533-3_1

[11] J. Whittle, J. E. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014. [Online]. Available: https://doi.org/10.1109/MS.2013.65

[12] J. E. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Sci. Comput. Program.*, vol. 89, pp. 144–161, 2014. [Online]. Available: https://doi.org/10.1016/j.scico.2013.03.017

[13] D. Seybold, J. Domaschka, A. Rossini, C. B. Hauser, F. Griesinger, and A. Tsitsipas, "Experiences of models@runtime with EMF and CDO," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. ACM, 2016, pp. 46–56.

[14] N. Kahani, M. Bagherzadeh, J. Dingel, and J. R. Cordy, "The problems with eclipse modeling tools: a topic analysis of eclipse forums," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, B. Baudry and B. Combemale, Eds. ACM, 2016, pp. 227–237. [Online]. Available: http://dl.acm.org/citation.cfm?id=2976773

[15] J. Whittle, J. E. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "Industrial adoption of model-driven engineering: Are the tools really the problem?" in *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS*, ser. LNCS, vol. 8107. Springer, 2013, pp. 1–17.

[16] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[17] D. S. Kolovos, N. D. Matragkas, I. Korkontzelos, S. Ananiadou, and R. F. Paige, "Assessing the Use of Eclipse MDE Technologies in Open-Source Software Projects," in *Proceedings of the International Workshop on Open Source Software for Model Driven Engineering co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 29, 2015.*, ser. CEUR Workshop Proceedings, vol. 1541. CEUR-WS.org, 2015, pp. 20–29.

[18] S. Rugaber and K. Stirewalt, "Model-driven reverse engineering," *IEEE Software*, vol. 21, no. 4, pp. 45–53, 2004. [Online]. Available: https://doi.org/10.1109/MS.2004.23

[19] A. F. Iosif-Lazar, A. S. Al-Sibahi, A. S. Dimovski, J. E. Savolainen, K. Sierszecki, and A. Wasowski, "Experiences from designing and validating a software modernization transformation (E)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 597–607. [Online]. Available: https://doi.org/10.1109/ASE.2015.84

[20] H. Klare, E. Burger, M. Kramer, M. Langhammer, T. Saglam, and R. Reussner, "Ecoreification: Making Arbitrary Java Code Accessible to Metamodel-Based Tools," in *2017 ACM/IEEE MODELS*, Sep. 2017, pp. 221–228.

[21] A. Zolotas, H. H. Rodriguez, D. S. Kolovos, R. F. Paige, and S. Hutchesson, "Bridging proprietary modelling and open-source model management tools: The case of PTC integrity modeller and epsilon," in *20th ACM/IEEE International*

Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017. IEEE Computer Society, 2017, pp. 237–247. [Online]. Available: https://doi.org/10.1109/MODELS.2017.18

[22] H. Song, G. Huang, F. Chauvel, and Y. Sun, "Applying MDE tools at runtime: Experiments upon runtime models," in *Proceedings of the 5th Workshop on Models@run.time, Oslo, Norway, October 5th, 2010*, ser. CEUR Workshop Proceedings, N. Bencomo, G. S. Blair, F. Fleurey, and C. Jeanneret, Eds., vol. 641. CEUR-WS.org, 2010, pp. 25–36. [Online]. Available: http://ceur-ws.org/Vol-641/paper_02.pdf

[23] TPC, "Decision Support Benchmark (version 2)," 2019. [Online]. Available: http://www.tpc.org/tpcds/

[24] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for Bi-directional Tree Transformations: a Linguistic Approach to the View Update Problem," in *POPL*. ACM, 2005, pp. 233–246. [Online]. Available: http://doi.acm.org/10.1145/1232420.1232424

[25] H. Song, Y. Xiong, F. Chauvel, G. Huang, Z. Hu, and H. Mei, "Generating synchronization engines between running systems and their model-based views," in *Models in Software Engineering, Workshops and Symposia at MODELS*, ser. LNCS, vol. 6002. Springer, 2009, pp. 140–154. [Online]. Available: https://doi.org/10.1007/978-3-642-12261-3_14

[26] H. Song, G. Huang, F. Chauvel, Y. Sun, and H. Mei, "Sm@rt: representing run-time system data as mof-compliant models," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE*. ACM, 2010, pp. 303–304. [Online]. Available: https://doi.org/10.1145/1810295.1810362

[27] H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei, "Supporting runtime software architecture: A bidirectional-transformation-based approach," *Journal of Systems and Software*, vol. 84, no. 5, pp. 711–723, 2011. [Online]. Available: https://doi.org/10.1016/j.jss.2010.12.009

[28] H. Song, G. Huang, F. Chauvel, W. Zhang, Y. Sun, W. Shao, and H. Mei, "Instant and incremental QVT transformation for runtime models," in *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS*, ser. LNCS, vol. 6981. Springer, 2011, pp. 273–288. [Online]. Available: https://doi.org/10.1007/978-3-642-24485-8_20

[29] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer, "A feature-based survey of model view approaches," *Software & Systems Modeling*, Sep 2017. [Online]. Available: https://doi.org/10.1007/s10270-017-0622-9

[30] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990. [Online]. Available: https://doi.org/10.1109/52.43044

[31] S. R. Tilley, "Domain-retargetable reverse engineering. III. layered modeling," in *Proceedings of the International Conference on Software Maintenance, ICSM 1995, Opio (Nice), France, October 17-20, 1995*. IEEE Computer Society, 1995, p. 52. [Online]. Available: https://doi.org/10.1109/ICSM.1995.526527

[32] F. Gouveia de Freitas and J. C. Sampaio do Prado Leite, "Reusing domains for the construction of reverse engineering tools," in *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, Oct 1999, pp. 24–34.

[33] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse-engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.4360050402

[34] D. R. Harris, A. S. Yeh, and H. B. Reubenstein, "Extracting architectural features from source code," *Autom. Softw. Eng.*, vol. 3, no. 1/2, pp. 109–138, 1996. [Online]. Available: https://doi.org/10.1007/BF00126961

[35] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé, "Pattern-based reverse-engineering of design components," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 226–235. [Online]. Available: http://doi.acm.org/10.1145/302405.302622

[36] J. . Favre, F. Duclos, J. Estublier, R. Sanlaville, and J. . Auffret, "Reverse engineering a large component-based software product," in *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, March 2001, pp. 95–104.

[37] L. Chouambe, B. Klatt, and K. Krogmann, "Reverse engineering software-models of component-based systems," in *12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece*. IEEE Computer Society, 2008, pp. 93–102. [Online]. Available: https://doi.org/10.1109/CSMR.2008.4493304

[38] J. DeBaud, B. Moopen, and S. Rugaber, "Domain analysis and reverse engineering," in *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994*. IEEE Computer Society, 1994, pp. 326–335.

[39] J. DeBaud and S. Rugaber, "A software re-engineering method using domain models," in *Proceedings of the International Conference on Software Maintenance, ICSM 1995, Opio (Nice), France, October 17-20, 1995*. IEEE Computer Society, 1995, pp. 204–213.

[40] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Information & Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014. [Online]. Available: https://doi.org/10.1016/j.infsof.2014.04.007

[41] A. García-Domínguez and D. S. Kolovos, "Models from code, or code as models?" in *Proceedings of the 16th International Workshop on OCL and Textual Modelling co-located with 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016.*, ser. CEUR Workshop Proceedings, vol. 1756. CEUR-WS.org, 2016, pp. 137–148.