Offline Delta-driven Model Transformation with Dependency Injection

Artur Boronat[0000−0003−2024−1736]

Department of Informatics, University of Leicester,
aboronat@le.ac.uk

Abstract. When model transformations are used to implement consistency relations between very large models (VLMs), incrementality plays a cornerstone role in the realization of practical consistency maintainers. State-of-the-art model transformation engines with support for incrementality normally rely on a publish-subscribe model for linking model updates − deltas − to the application of model transformation rules, in so called dependencies, at run time. These deltas can then be propagated along an already executed model transformation. A small number of such engines use domain-specific languages (DSLs) for representing model deltas offline in order to enable their use in asynchronous, event-based execution environments.
The principal contribution of this work is the design of a forward delta propagation mechanism for incremental execution of model transformations, which decouples dependency tracking from delta propagation using two innovations. First, the publish-subscribe model is replaced with dependency injection, physically decoupling domain models from consistency maintainers. Second, a standardized representation of model deltas is reused, facilitating interoperability with EMF-compliant tools, both for defining deltas and for processing them asynchronously. This procedure has been implemented in a model transformation engine, whose performance has been evaluated empirically using the VIATRA CPS benchmark. In the experiments performed, the new transformation engine shows gains in the form of several orders of magnitude in the initial phase of the incremental execution of the benchmark model transformation and delta propagation is performed in real time, independently of the size of the models involved, whereas the up-to-now best-performant approach is dependent.

Keywords: Mappings between languages, traceability, incremental model transformation, performance benchmark.

1 Introduction

Significant issues in the application of Model-Driven Engineering (MDE) in large-scale industrial problems stem from interoperability and scalability of current MDE tools [1,17,16]. Model transformation, widely accepted as the *heart and*

soul of MDE [23], deals with model manipulation either by translating models or by synchronizing them. Current tool support for model transformation is a key root cause for many of the bottlenecks hampering scalability in MDE [8,2]. This is particularly crucial when transformations are used to implement consistency maintainers between very large models (VLMs), consisting of milions of elements. In this context, incrementality ensures that only those parts of the model that are inconsistent or that have been modified − a model delta − are transformed or, more precisely, propagated along an already executed transformation [11,12].

Current state-of-the-art approaches that support incremental execution of model transformations share common features: the delta propagation mechanism is usually decoupled from the delta detection mechanism in order to facilitate maintainability of the consistency maintainer; and deltas are represented either in memory for synchronous notification or offline, with dedicated domain-specific languages, for asynchronous notification. The most mature tools rely on a publish/subscribe mechanism, where model deltas are notified at run time whenever a model is updated. This notification mechanism is synchronous and loosely couples model updates with the delta propagation mechanism, facilitating maintainability of the underlying transformation engine after fixing the type of notification. However, it usually requires an observer for each object that can be modified, with a consequent impact on performance, and the model transformation must be live, in memory, in order to listen for changes. These problems can be avoided by using offline deltas. The publish/subscribe mechanism can be extended to enable asynchronous delta notification but this is normally achieved by using dedicated domain-specific languages to represent deltas offline, which do not involve standardized formats, hindering the interoperability of those transformation engines in existing modeling tool ecosystems.

In this paper, the design of a forward delta propagation procedure is presented for executing model transformations in incremental mode that can handle documented change scenarios [4], i.e. documents representing a change to a given source model. Such documents are defined with the EMF change model [24], both conceptually and implementation-wise, guaranteeing interoperability with EMF-compliant tools. This design decision replaces a publish/subscribe notification with dependency injection: each notification is directly performed by the implementation of the domain model at run time by injecting the dependency corresponding to the model update that has been performed. Aspect-oriented programming is used to weave code into an already existing implementation of a domain model totally decoupling domain models from the consistency maintainer at design time. The proposed forward delta propagation procedure has been implemented in YAMTL [6], a model transformation engine for VLMs, enabling the execution of model transformations both in batch mode and in incremental mode without additional user specification overhead. This new extension dramatically improves the performance of the batch execution mode when dealing with sparse model deltas, which can be propagated in real time (i.e. in μs.).

This work is structured as follows: Section 2 provides a self-contained description of the class of model transformations supported using a class diagram to

relational schema model transformation; Section 3 presents the forward propagation procedure implemented in the model transformation engine together with the main innovations; Section 4 discusses the performance of the transformation engine with an adaptation of the VIATRA CPS benchmark; Section 5 discusses related work from reactive and bidirectional model transformation.

2 Model Transformation: A Running Example

The type of model transformations that are considered in this work are classified as unidirectional and out-place. For example, when considering the well-known example that maps class diagrams to relational schemas, a class diagram is used by queries to extract information and a relational schema is built from scratch. If we consider a graph transformation perspective, both models are considered to form part of the same graph in order to enable transformation by rewriting. In that case, we are only considering transformations where the two models are two clearly disjoint subgraphs and where rewriting is performed deterministically.

In this work, model transformations are represented using an implementation-agnostic graphical syntax, quite close to that used in the graph transformation literature. In this representation, metamodels are given as class diagrams, the abstract syntax of models is given as object diagrams and model transformations are represented as a collection of rules, where each rule is defined as a pair of model patterns, called left-hand side (LHS) and right-hand side (RHS). The notion of metamodel, model and model pattern correspond to those of type graph, attributed graph with containments and node inheritance, and graph pattern in the graph transformation literature [5,10]. For example, the rules `A->C` and `R->FK` of Figure 1 map attributes to columns. The $ before a variable denotes string interpolation.

Graph patterns in rules can be augmented with universally quantified variables (represented by an overlaid box). Moreover, rules are augmented with a `when` clause to express conditions that must be satisfied by the variables in LHS, and with a `where` clause to indicate how variables from LHS and from RHS are related via the application of other rules, expressed as two graph patterns. Formulas in a `when` clause may be expressed in conjunctive form, as all filter conditions must be satisfied in order for the rule to be applied, whereas formulas in a `where` clause may be expressed in disjunctive form (assuming mutually exclusive conditions), as all the side effects expressed in a `where` clause must be evaluated. The variables of RHS of the main rule must appear either in the LHS of the main rule or in the RHS of a `where` transformation step. The rule `C->T` of Figure 1 illustrates how to map a class to a table with a primary key column `PK_COL` and for each attribute `A` whose type is a `DataType`, the corresponding column is obtained by applying a rule, with the rule `A->C`, and for each attribute `OTHER` whose type is the class `C`, matched in LHS of the main rule, a new foreign key column is added to the table `T`, with the rule `R->FK`.

From an operational point of view, transformation rules are applied unidirectionally from LHS to RHS performing an out-place transformation following

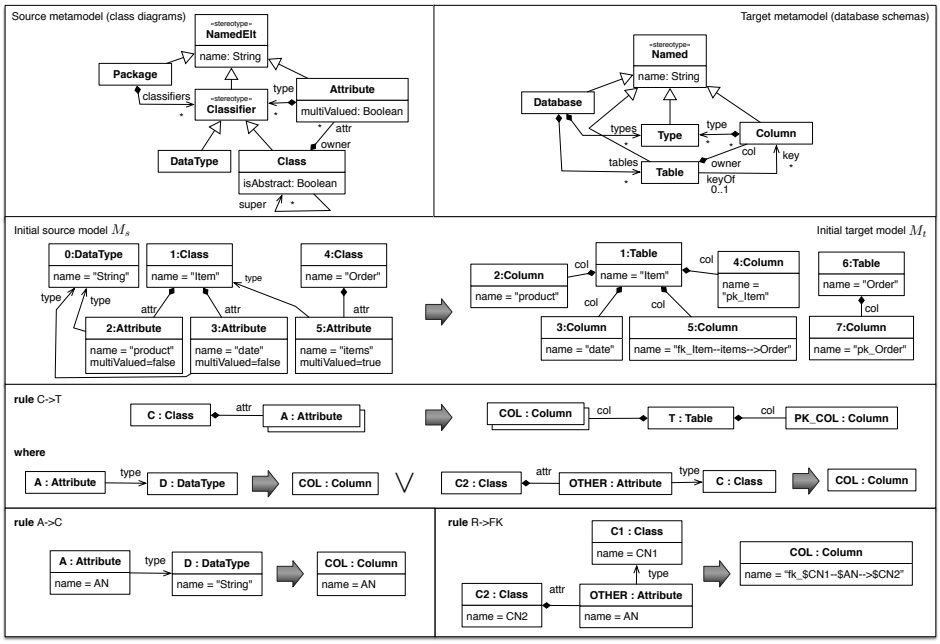

Fig. 1. Metamodels, example and transformation rules.

two steps. First, during the *matching phase*, matches for the rules in the model transformation are found as long as they are not shared by different rules and these are included in a set *matchPool*. A match is formally defined as a graph morphism from LHS to the source graph, which satisfies the `when` conditions, but it is represented as a map from variables to object identifiers for the sake of presentation in this paper.

Second, during the *execution phase*, each match is processed by triggering the application of a transformation rule, which is represented as a transformation step, denoted by $r : \overrightarrow{in \mapsto \varsigma} \rightarrow \overrightarrow{out \mapsto \varsigma}$, which consists of a labelled pair of two matches, the match for the input pattern of the rule, which enables its application, and the match for the output pattern of the rule, with the objects that result from applying the rule. When a rule is applied, the source model is only used for query purposes but the target model is constructed by adding the pattern of the RHS instantiated with values from the variables both in the LHS and in the RHS of `where` transformation steps. In addition, `where` transformation steps may further expand the structure of the target model. This execution model resembles the application of forward rules used in triple graph grammars (TGGs) [22], where the source graph is annotated as rules are applied and only the target graph is constructed together with a link in a correspondence graph, where each link denotes a transformation step.

3 Delta-driven Model Transformations

This section presents the mechanism to propagate documented deltas δ_t from a source model M_s to a target model M_t in an incremental way, when the (unidirectional) synchronization correspondence between these two models is represented with a model transformation t as described in the previous section. This has been implemented in the YAMTL transformation engine [6], which has been extended with two modes of execution: *initialization*, the transformation is executed in batch mode but, additionally, tracks those parts of the source model involved in transformation steps as *dependencies*; *propagation*, the transformation is executed incrementally for a given source delta.

In order for a model transformation to be executed in propagation mode, it first needs to be executed in initialization mode in order both to create transformation steps and to inject the dependencies that facilitate the analysis of the impact of changes in the already executed model transformation. Therefore, the transformation t is applied to M_s using the original batch semantics [6] while injecting dependencies in the transformation engine. Once the initialization is done, any number of source forward deltas δ_s can be propagated.

Given a source documented delta δ_s between a source model M_s, already synchronized with a target model M_t via a model transformation $t : M_s \xrightarrow{*} M_t$ (where $\xrightarrow{*}$ denotes a sequence of transformation steps), and an updated source model M'_s, the transformation engine propagates the model update δ_s along t. The effect of this forward propagation is the application of an update δ_t on the target model M_t.

In the following subsections, we explain the different phases of the new execution modes, initialization and propagation, in more detail. As the initialization mode faithfully corresponds to the batch execution of a model transformation, the discussion of this mode focuses on the type of dependencies that are injected in the transformation engine in Section 3.1. The discussion on the propagation mode focuses on how deltas are represented in Section 3.2. Then, the two main phases of the propagation execution mode, namely impact analysis and delta propagation, are explained in sections 3.3 and 3.4, respectively.

3.1 Dependency Injection

When running a model transformation in initialization mode, the engine monitors the source model and whenever an object ς is matched or a feature call, represented as a pair (ς, f) of an EMF object ς and a feature name f, is performed, a dependency is injected into the dependency registry. A dependency thereby links either an object ς or a feature call (ς, f) to transformation steps $r : \overrightarrow{in \mapsto \varsigma} \rightarrow \overrightarrow{out \mapsto \varsigma}$ in which it is used. Such dependencies are detected both during the matching phase and during the execution phase.

In the matching phase, while finding a match for a rule, the engine keeps track of all of the feature calls used in both element and rule `when` conditions. When a match is found to be valid, the collection of dependencies is injected into the dependency registry for the transformation step that uses that match. Otherwise,

Rule	Source Match	Target Match	Dependencies from M_s
C->T	c $\mapsto$ 1	t $\mapsto$ 1,	(1,name), (1,att),
		pk_col $\mapsto$ 4	(5, type), (5, multiValued)
C->T	c $\mapsto$ 4	t $\mapsto$ 6, pk_col $\mapsto$ 7	(4, name), (4, attr)
A->C	att $\mapsto$ 2	col $\mapsto$ 2	(2, name)
A->C	att $\mapsto$ 3	col $\mapsto$ 3	(3, name)
R->FK	ref $\mapsto$ 5	fk_col $\mapsto$ 5	(5,name), (5,type),
		fk_col $\mapsto$ 5	(1, name), (4,name)

Table 1. Analysis of dependencies for the initial MT $t : M_s \xrightarrow{*} M_t$ of Figure 2.

when the match is not valid, the collected dependencies are discarded. Additionally, when inserting a match in the *matchPool*, the transformation engine also records reverse matches as injected dependencies between matched objects ς and the transformation step in which they are matched.

Dependencies may also be found when executing a transformation step, e.g., while executing initialization expressions associated with attributes in model patterns in RHS and in `where` clauses. In such cases, the transformation engine injects a dependency for the transformation step every time a feature call in the source model is detected. As a result, note that several transformation steps may depend on the same object ς, when rules have more than one single input element, or on the same feature call (ς, f).

Table 1 shows the dependencies that are found when executing the transformation of Figure 1 in initialization mode from model M_s. Each row in the table represents a transformation step, where: the source match indicates where the rule has been applied, the target match indicates what objects were created, and dependencies refers to the set of feature calls associated with a transformation step. Reverse matches are extracted from source matches, by reading them in the opposite direction.

Dependency injection is configured with an aspect whose pointcut matches feature calls under a user-defined namespace. Hence, the model transformation engine is entirely decoupled from the domain model at design time. They become tightly coupled at compilation time and, hence, at run time.

3.2 Representable Deltas

The EMF change model [24] is used to represent deltas to an instance of any other EMF model. It is built-in in EMF and, therefore, available for any EMF-compliant tool. In this section, we describe how a documented delta is represented with the EMF change model and how it can be automatically defined given any potentially *live* atomic update.

A delta consists of a `ChangeDescription` which contains a map of `objectChanges`, which refer to those objects that are updated and, for each such object, it contains a list of `FeatureChanges`. A `FeatureChange` (`FC`) refers to the structural feature that needs to be updated and provides the new value. For single-valued

attributes, a `FeatureChange` contains the new `dataValue` if the feature is an attribute. For references and multi-valued attributes, a `FeatureChange` includes a containment reference `listChanges` pointing to `ListChange`. `ListChange`s are used to represent addition to, removal from, or movement *within* the given feature values. In particular, movement only captures when an object changes to a different index within the collection. However, it does not capture structural changes, e.g. change of container, which are represented as a removal from and an addition to the corresponding containment references. When a `FeatureChange` refers to a containment reference, objects to be added are pointed by `objectsToAttach` and objects to be removed are pointed by `objectsToDetach`.

`FeatureChange`s capture when a feature value is updated for an object but EMF also permits adding and removing root objects to a resource, representing the model in memory, which need not be contained by any other object. Such changes are considered to be performed on the resource itself and are represented with `ResourceChange`s, one for each changed resource. A `ResourceChange` (`RC`) contains the `ListChange`s for the root objects of the corresponding resource, similarly to multi-valued features. For a more detailed explanation of the EMF change model, we refer the reader to [24].

Table 2 shows a classification of atomic model updates that are representable with the EMF change model as explained above. Note that moving and object structurally, case 12 − *move (inter.)*, − is represented in a composite delta by two opposite actions, removing the object either from the root contents of the resource − if it is a root object (case 2) − or from a containment reference − if it is a contained object (case 10) − and adding it either to the root contents of the resource − if it is to become a root object (case 1) − or to another containment reference in another container object (case 9). This case is not captured by the EMF change model explicitly but the transformation engine is able to infer it, as explained in the following section.

Cases	Granularity	Level	Feature	Delta action	Delta representation	*DO*	*DFC*
1,2	atomic	root		add/remove	`RC::listChanges`	✓	
3	atomic	root		move (intra.)	`RC::listChanges`		
4,5	atomic	any	single-valued att	add/remove	`FC`		✓
6,7	atomic	any	multi-valued att	add/remove	`FC::listChanges`	✓	✓
8	atomic	any	multi-valued att	move (intra.)	`FC::listChanges`		✓
9,10	atomic	any	ref	add/remove	`FC::listChanges`		✓
11	atomic	any	ref	move (intra.)	`FC::listChanges`		✓
12	composite	any	containment ref	move (inter.)	opposite remove and add actions in cases $\{2, 10\}/\{1, 9\}$		✓

Table 2. Summary of model update types, with their representation in EMF.

A delta, which may represent atomic and composite changes, is defined as an instance of the EMF change model and can be serialized. EMF also provides

facilities for applying them and reversing them. Furthermore, EMF provides a change recorder, which enables recording *live updates* as a `ChangeDescription` for either a root object, a collection of root objects, a resource or a resource set. The resulting `ChangeDescription` is the representation of a *history scenario* [4], from the updated model to the original one, which is optimized. That is, atomic changes for the same feature of the same object may be discarded or merged, as long as the optimization process preserves reversibility. Hence, reversing the recorded delta may yield less changes than were originally made. Reversed deltas represent *documented scenarios* and can be propagated along a model transformation, as discussed in subsequent sections.

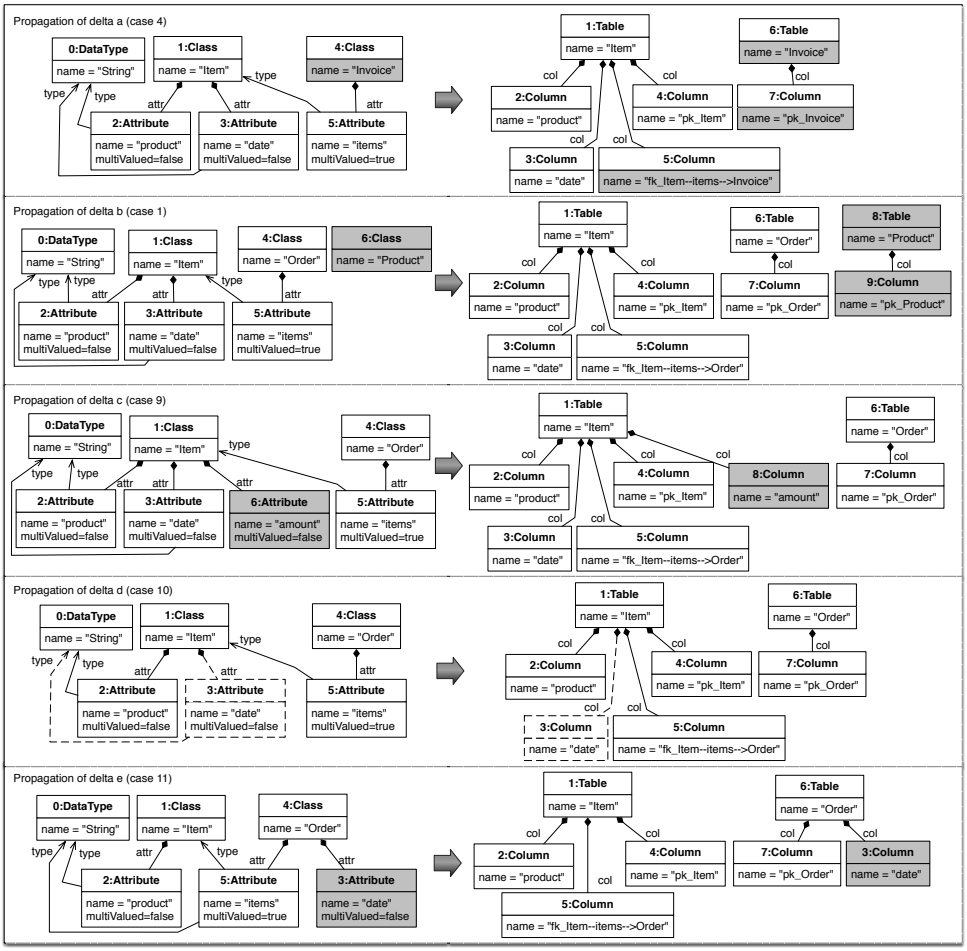

Fig. 2. Source/target metamodels, initial synchronized models and forward delta propagation (a-e).

The EMF change recorder enables the possibility of deferring the observation of updates to the point in which they occur, saving memory resources, and interoperability. Furthermore, recorded (history) deltas can be regarded as a rollback mechanism for implementing transactional model updates, which may be performed live.

Figure 2 shows examples of documented deltas, defined over the source model M_s of the running example. Such deltas are representable as EMF model changes, i.e. operationally, but are graphically depicted using the abstract syntax of M_s, using their state-based representation for the sake of presentation. Additions and updates, including moves, are highlighted in grey colour. Objects that are added, and thus created, have a new identifier. Objects that are updated and/or moved preserve their identifier. Removals are highlighted by using dashed lines for the contour lines of the corresponding shapes. The given deltas are instantiations of case 4 (delta `a`), changing the name of the class `Order` to `Invoice`; case 1 (delta `b`), adding a root class `Product`; case 9 (delta `c`), adding a single-valued attribute `amount` to class `Item`; case 10 (delta `d`), removing the attribute `date` from class `Item`; and case 11 (delta `e`), structurally moving the attribute `date` from class `Item` to class `Order`.

In the following subsections, the different phases of the procedure for forward propagation of source deltas is discussed and the aforementioned examples will be used for illustrating them.

3.3 Impact Analysis

In this subsection, we discuss how source documented deltas are analyzed in order to determine which transformation steps are affected by source changes. This analysis is comprised of three main steps: identification of atomic model updates from a documented delta, initialization of locations for newly enabled rules, and marking of transformation steps impacted by changes.

Identification of atomic model updates. In the first step, the transformation engine infers which objects and which feature calls have been impacted by changes. For objects, it also infers whether an object has been added or removed, ignoring if the object is moved, either within the same collection or structurally.

For affected objects, such information is recorded in the set DO of *dirty objects* of the form $(\varsigma, ctype)$, where ς is the affected object and *ctype* is the type of change from the set $\{\texttt{ADD}, \texttt{DEL}\}$. To obtain a dirty object from the delta, `FeatureChange`s and `ResourceChange`s are traversed considering two cases: when an object ς is added either to a containment feature (for a `FeatureChange`) or to the root contents of the resource (for a `ResourceChange`) and such object is not removed elsewhere in the delta, either from a containment reference or from the root contents of the resource; and, similarly, when an object is deleted and it is not added elsewhere in the delta. DO is augmented with $(\varsigma, \texttt{ADD})$ in the first case and with $(\varsigma, \texttt{DEL})$ in the second case.

For affected feature calls, such information is recorded in the set DFC of *dirty feature calls* of the form (ς, f), where ς is an object and f is a feature

	Case	*DO*	*DFC*	Rule	Source Match	Target Match	$matchPool_\Delta$	dirty?
a	4	–	(4, `name`)	`C->T`	$\texttt{c} \mapsto 4$	$\texttt{t} \mapsto 6, \texttt{pk\_col} \mapsto 7$	✓	✓
b	1	(6, *ADD*)	–	`C->T`	$\texttt{c} \mapsto 6$		✓	
c	9		(1, `attr`)	`C->T`	$\texttt{c} \mapsto 1$	$\texttt{t} \mapsto 1, \texttt{pk\_col} \mapsto 4$	✓	✓
		(6, *ADD*)		`A->C`	$\texttt{att} \mapsto 6$		✓	
d	10		(1, `attr`)	`C->T`	$\texttt{c} \mapsto 1$	$\texttt{t} \mapsto 1, \texttt{pk\_col} \mapsto 4$	✓	✓
		(3, *DEL*)		`A->C`	$\texttt{att} \mapsto 3$	$\texttt{col} \mapsto 3$		✓
e	11	–	(1, `attr`),	`C->T`	$\texttt{c} \mapsto 1$	$\texttt{t} \mapsto 1, \texttt{pk\_col} \mapsto 4$	✓	✓
			(4, `attr`)	`C->T`	$\texttt{c} \mapsto 4$	$\texttt{t} \mapsto 6, \texttt{pk\_col} \mapsto 7$	✓	✓

Table 3. Impact analysis of source deltas a-e.

name. For each `FeatureChange` of an `ObjectChange`, the dirty feature call (ς, f) with the object ς referred by the `ObjectChange` and the feature name f referred to by the `FeatureChange` is added to *DFC*.

Table 2 shows how atomic model update types are represented using the EMF change model (column *delta representation*), internally, using the sets *DO* and *DFC*. Table 3 shows the sets *DO* of dirty objects and *DFC* of dirty feature calls for the source deltas of Figure 2. Note that the sets *DO* and *DFC* decouple the transformation engine from the EMF change model and provide another entry point for defining deltas programmatically, which can be used for capturing atomic *live changes* received via EMF adapters.

Initialization of delta locations. For each dirty object $(\varsigma, \texttt{ADD})$, the object ς is added to the extent associated with $type(o)$ in the location map used for delta propagation. This potentially enables new matches when rules are matched during the delta propagation phase.

Marking of impacted transformation steps. In this step, transformation steps that are affected by the atomic changes in the source delta are marked as dirty. For each dirty object $(\varsigma, \texttt{ADD}) \in DO$, the extent of type $\texttt{type}(\varsigma)$ is augmented with ς. This will potentially enable new matches for some rule during the change propagation phase. For each dirty object $(\varsigma, \texttt{DEL}) \in DO$, we obtain the list of transformation steps that are affected from the map of reverse matches. Such transformation steps will then remain transient and the objects in their target match will not be linked to other objects in the target models. In particular, note that when processing root objects or a containment reference, an object that is removed in the delta is not present in the updated source model and, therefore, it does not trigger the transformation step that had been executed in the initial transformation.

For each dirty feature call $(\varsigma, f) \in DFC$ we obtain the list of transformation steps that are affected from the registry of dependencies. For each such transformation step, the satisfaction of its source match is checked. If such source match is still valid, then it is inserted into $matchPool_\Delta$, the pool of matches that are used to schedule rule applications during the change propagation phase.

For each atomic change in Figure 2, Table 3 shows the marking of transformation steps that are (re-)scheduled according to the dependencies of Table 1. In particular, if a transformation step is re-scheduled, its current source and target matches are included, it is marked as dirty and included in $matchPool_{\Delta}$. If a transformation step is not to be re-executed, it is simply marked as dirty. New transformation steps, with fresh matches due to new objects, are scheduled in $matchPool_{\Delta}$. This last step is actually achieved by augmenting the corresponding type extent with the new objects and the matches are scheduled during the change propagation phase, explained in the next subsection.

3.4 Change Propagation

After the impact analysis phase, delta propagation proceeds by executing a model transformation using the matching and execution phases, as outlined in Section 2. Figure 2 illustrates the propagation of source deltas according to the model transformation of Figure 1. We highlight how incrementality has been considered in these two phases below.

Matching Phase. During the matching phase (in batch/initialization execution mode), matches for a given rule are found by traversing objects from the extent of the types associated with the elements of the source pattern of the rule, with the constraints specified in the form of graphical patterns and `when` conditions. In propagation mode, the transformation engine employs the same pattern matching algorithm but it fetches objects from the location map used for delta propagation, initialized during the change impact analysis phase. Therefore, new matches may be found for objects that have been created by the source delta. Those matches are inserted both into $matchPool$ and $matchPool_{\Delta}$, scheduling new transformation steps. Table 3 shows that two new transformation steps are scheduled, one for rule `C->T` in delta `b`, and one for rule `A->C` in delta `c`.

Execution Phase. During the execution phase, transformation steps determined by the matches in $matchPool_{\Delta}$ are executed. Such matches originate from the impact analysis phase, corresponding to transformation steps that are *dirty* and need to be re-executed, and from the matching phase above, corresponding to new transformation steps.

The re-execution of a transformation step is performed as in the batch/initialization mode but for the creation of transformation steps. Whereas a newly scheduled transformation step needs to get its output objects initialized (instantiated for output elements), a dirty transformation step *reuses* the objects of the target match and unsets their features. This avoids loss of contextual information, which is not affected by changes, when re-executing a tranformation step. In particular, those references to output objects that emerge from the external context are preserved. On the other hand, references from those output objects are re-calculated by re-executing the transformation step. It is worth noting that the transformation engine uses `where` clauses to define references to objects that are created by other rules, which in turn uses a cache mechanism

to avoid re-executing the transformation step that produced it. Therefore, when a dirty transformation rule is re-executed, the initialization of output element bindings are performed again. However, those bindings that are initialized in a `where` clause are also initialized incrementally. That is, only those objects that belong to a match of a new scheduled transformation step will be transformed from scratch. References to already initialized objects will be simply fetched. Hence, the granularity of the target delta is as fine grained (at binding level) as the source delta for the underlying graph structure of the model.

4 Performance Analysis

For the empirical analysis of the incremental execution of model transformations in YAMTL using the propagation procedure presented above, we have used the VIATRA CPS benchmark [27]. The transformation *YAMTL-incr* implemented for our model transformation engine passes the sanity checks of the benchmark. The software artifacts used in this section and the results obtained are publicly available in a GitHub repository [7] and YAMTL is available at `https://yamtl.github.io/`.

This evaluation is an extension of the one performed for the batch component of the VIATRA CPS benchmark in [6]. From the original VIATRA CPS benchmark, two incremental variants of the transformation implemented with *EMF-IncQuery* have been selected: *ExplicitTraceability* (EXPL) [25] and *QueryResultTraceability* (QRT) [26], out of which the first one is the best performing solution up to now. These transformations have been extracted as independent Java projects. Classes implementing them have been kept intact in the new projects, including their namespaces, so that errors are not introduced due to lack of expertise. Although these two transformations produce results that are different from the other transformations, the main differences are due to reordering of multi-valued references and we have considered them valid for this evaluation. On the other hand, a benchmark measurement harness considering the best practices recommended by the VIATRA team [13] was developed in order both to fine-tune measurements and to crosscheck results. This harness removes dependencies to other components of the VIATRA CPS benchmark so that experiments can be run locally.

In the present work, we aimed at answering the following research questions: *(RQ1)* Does *YAMTL-incr* show any performance penalty w.r.t. its execution in batch mode (*YAMTL-batch*)? *(RQ2)* Does *YAMTL-incr* show any improvement in performance w.r.t *EXPL* or *QRT* during initialization phase? *(RQ3)* And during propagation phase?

From the scenarios provided in the original benchmark, the scenarios *client-server* and *statistic based* [29] were considered. The CPS model generator [28] was used to obtain the input models to be used for the analysis so that their size depends on a logarithmic factor. The biggest models considered, in the client server scenario, consist of millions of nodes (10.16M) and edges (27.53M) and are, hence, VLMs.

For each tool and scenario, the experiments are run in isolation, i.e. in a separate Java process. For each of the input models, an initial experiment is performed to warm up the JVM and, then, twelve more experiments to measure performance. Each experiment consists of four phases: model load and engine initialization, initial transformation, delta propagation and model storage. In between each execution phase, the harness sends hints to the JVM to run garbage collection and waits for one second before proceeding on to the next phase. The first phase includes the instantiation of a fresh engine instance, avoiding interference between experiments as caches are not reused. The delta propagation phase includes the application of the delta to the source model and its propagation. Only initial transformation and delta propagation times have been considered in the quantitative analysis. For the results the median obtained for each of these two phases out of ten experiments is used, after removing the minimum and the maximum results.

In both solutions *EXPL* [25] and *QRT* [26], the delta is applied to the source model by directly modifying the resource containing the model. In the solution with YAMTL such delta was recorded and persisted using the EMF change model as described in Section 3.2. To analyze whether this feature could become a threat to validity, a separate experiment was run by excluding the query part of the model update (searching for the objects to be updated) in the solution *EXPL* but this change did not affect performance results perceptibly and the original solutions provided by the authors of the VIATRA CPS benchmark were considered. Therefore, the actions performed during the propagation phase are equivalent in all of the evaluated solutions.

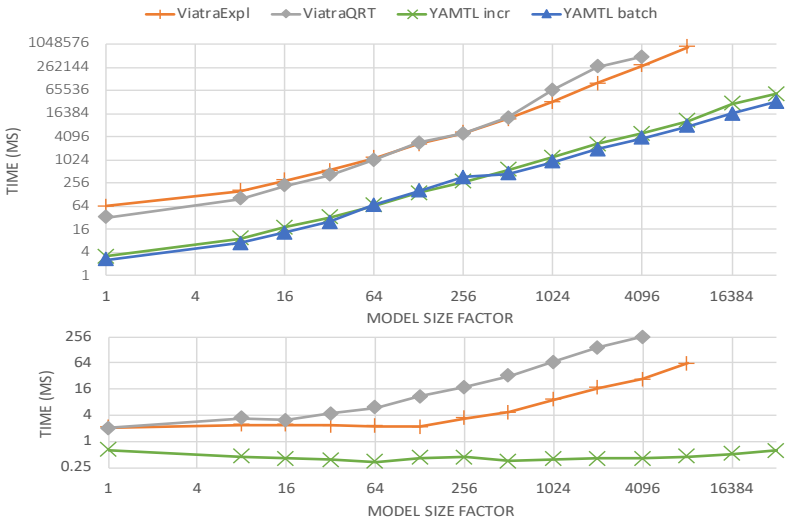

Fig. 3. Performance of initialization (top) and delta propagation (bottom).

Figure 3 shows the performance results obtained both for the initial model transformation and for forward delta propagation for the models generated for the client-server scenario. Scales both for time (ms.) along Y axis and for model size factors along X axis are logarithmic allowing us to compare the scalability of the different approaches. In the initialization phase, we have included the execution of YAMTL in batch mode (*YAMTL-batch*) over the source model, and it can be seen that tracking dependencies incurs a small penalty. However, the other two solutions (*EXPL* and *QRT*) operate several orders of magnitude slower. In the propagation phase, it can be observed that while *YAMTL-incr* exhibits a constant propagation time (in μs.) for the source delta, the cost of the other solutions depends on the size of the input model. Furthermore, for the other incremental approaches, when both initial and propagation time are combined their performance worsens due to their costly initialization phase.

5 Related Work

In this section, we discuss techniques used in related work for achieving incrementality in both reactive and bidirectional model transformation.

Reactive model transformation [21,3] enable the propagation of model updates from source models to target models on demand. State-of-the-art tool support relies on notification mechanisms, enabling live detection of source model updates either for immediate processing, as in VIATRA [3], or for deferred processing, as in ReactiveATL [21]. In these approaches, source model update notifications are usually fine-grained and kept in memory. Such notifications can only be detected when the transformation engine is in memory (live) as well. The use of a notification mechanism means that models are *loosely coupled* to the transformation engine. Working with offline model updates, as in the proposed delta propagation procedure, completely decouples detection of deltas from the transformation engine, freeing model update developers from the overhead of having the transformation infrastructure in memory. The latter is only needed for propagating changes but not for defining them. In reactive approaches, when an observer receives an update notification, information about the intent of the overall model delta, i.e. the contextual information relating different atomic updates, is lost. This problem is avoided using documented deltas, which may be serialized, enabling their processing − e.g. aggregating composite changes like the *move operation* − and optimization − reduction of atomic operations that are cancelled when composed. We refer the reader to [9] for an additional discussion of delta-based model updates against state-based model updates.

Among bidirectional model transformation approaches, Triple Graph Grammars (TGG), introduced in [22], are a declarative approach for specifying bidirectional consistency relations between models. Although our approach is not bidirectional, it is worth comparing how incrementality is supported in operational TGG rules. Incrementality was first introduced in TGG synchronization in [11,12]. Efficient approaches for TGG synchronization [18,20,19] avoid analyzing the whole model by relying on dependencies which hint at the im-

pact of a model update directly. Precedence-based approaches [18,20] keep a binary precedence relation over the set of model elements in order to determine when creation or deletion of a model element affects another one. While [18] overestimates the actual dependencies by defining them at the type level, others underestimate them relying on user feedback [20] or on special correspondences [12]. [19] decouples impact analysis of model updates from consistency restoration by delegating the former to VIATRA's incremental pattern matcher, which has a built-in dependency tracker, and by defining operational rules using a reactive model transformation approach. However, these two phases are still tightly coupled using a synchronous communication mechanism between the incremental pattern matcher and the synchronization procedure since the pattern matcher may trigger revocations/applications of forward marking rules after revoking/applying one of them. That is, the model synchronization procedure uses the pattern matcher to know when synchronization terminates. In the delta propagation mechanism proposed in the present work, either the revocation of applied transformation steps or the creation of new transformation steps cannot trigger further applications because rule matches are computed against the source model and they are unique, that is the same match cannot enable two different rules. A new transformation step may be found when new elements are inserted in the source model. On the other hand, when a transformation step is revoked, no other rule can be applied or a conflict would have been detected when the rule was applied the first time.

Some transformation engines with support for bidirectional transformations, like NMF [15,14], support the offline representation of model deltas. However, to the best of our knowledge, none of the aforementioned approaches uses a standardized notation for them, such as the EMF model change, which can be regarded as the de-facto standard for representing model deltas in the EMF modeling tool ecosystem.

6 Concluding Remarks

The main contribution of this work is the design of a delta propagation procedure for executing delta-driven model transformations, which has been implemented in YAMTL. The novelty of the approach consists in the use of a standardized representation of model deltas, which facilitiates interoperability with EMF-compliant tools, and in the use of dependency injection mechanism, which allows the transformation engine to be aware of model updates without having to rely on a publish-subscribe infrastructure. The VIATRA CPS benchmark has been used to justify that *(1)* the initialization transformation in YAMTL is several orders of magnitude faster than the up-to-now fastest incremental solutions and that *(2)* propagation of sparse deltas can be performed in real time for VLMs, independently of their size, whereas other solutions show a clear dependence on their size. Hence, YAMTL shows satisfactory scalability in incremental execution of model transformations on VLMs. Additional studies with larger classes of models will be considered in future work.

References

1. P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial context — motorola case study. In *MoDELS*, volume 3713, pages 476–491. LNCS, 2005.
2. A. Benelallam, A. Gómez, M. Tisi, and J. Cabot. Distributing relational model transformation on mapreduce. *Journal of Systems and Software*, 142:1 – 20, 2018.
3. G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. Viatra 3: A reactive model transformation platform. In *ICMT*, volume 9152, pages 101–110. LNCS, 2015.
4. G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations - change (in) the rule to rule the change. *Software and System Modeling*, 11(3):431–461, 2012.
5. E. Biermann, C. Ermel, and G. Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software and System Modeling*, 11(2):227–250, 2012.
6. A. Boronat. Expressive and efficient model transformation with an internal DSL of Xtend. In *MODELS 2018*, pages 78–88. ACM, 2018.
7. A. Boronat. YAMTL evaluation repository with the incremental component of the VIATRA CPS benchmark, 2018. https://github.com/yamtl/viatra-cps-incr-benchmark.
8. G. Daniel, F. Jouault, G. Sunyé, and J. Cabot. Gremlin-ATL: A scalable model transformation framework. In *ASE*, pages 462–472. IEEE Computer Society, 2017.
9. Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. In *MODELS*, volume 6981, pages 304–318. LNCS, 2011.
10. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, March 2006.
11. H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In *MoDELS*, volume 4199, pages 543–557. LNCS, 2006.
12. H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and System Modeling*, 8(1):21–43, 2009.
13. D. Harmath and I. Ráth. Viatra/query/faq: Performance optimization guidelines, 2016. https://wiki.eclipse.org/VIATRA/Query/FAQ#Performance_optimization_guidelines.
14. G. Hinkel. Change propagation in an internal model transformation language. In *ICMT*, volume 9152, pages 3–17. LNCS, 2015.
15. G. Hinkel and E. Burger. Change propagation and bidirectionality in internal transformation dsls. *Softw Syst Model*, 2017.
16. J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *ICSE*, pages 471–480. ACM, 2011.
17. D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The grand challenge of scalability for model driven engineering. In M. R. V. Chaudron, editor, *Models in Software Engineering (MiSE). Collocated with MODELS.*, volume 5421, pages 48–53. LNCS, 2008.
18. M. Lauder, A. Anjorin, G. Varró, and A. Schürr. Efficient model synchronization with precedence triple graph grammars. In *ICGT*, volume 7562, pages 401–415. LNCS, 2012.
19. E. Leblebici, A. Anjorin, L. Fritsche, G. Varró, and A. Schürr. Leveraging incremental pattern matching techniques for model synchronisation. In *ICGT*, volume 10373, pages 179–195. LNCS, 2017.

20. F. Orejas and E. Pino. Correctness of incremental model synchronization with triple graph grammars. In *ICMT*, volume 8568, pages 74–90. LNCS, 2014.
21. S. M. Perez, M. Tisi, and R. Douence. Reactive model transformation with ATL. *Sci. Comput. Program.*, 136:1–16, 2017.
22. A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *WG*, pages 151–163. LNCS 903, 1994.
23. S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.
24. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.
25. VIATRA Team. Explicit traceability m2m transformation, 2016. https://github.com/viatra/viatra-docs/blob/master/cps/Explicit-traceability-M2M-transformation.adoc.
26. VIATRA Team. Query result traceability m2m transformation, 2016. https://github.com/viatra/viatra-docs/blob/master/cps/Query-result-traceability-M2M-transformation.adoc.
27. VIATRA Team. VIATRA CPS benchmark (cps to deployment transformation), 2016. https://github.com/viatra/viatra-docs/blob/master/cps/CPS-to-Deployment-Transformation.adoc.
28. VIATRA Team. VIATRA CPS benchmark (model generator), 2016. https://github.com/viatra/viatra-docs/blob/master/cps/Model-Generator.adoc.
29. VIATRA Team. VIATRA CPS benchmark (scenario specification), 2016. https://github.com/viatra/viatra-cps-benchmark/wiki/Benchmark-specification#cases.