Noname manuscript No.
(will be inserted by the editor)

Incremental Execution of Rule-based Model Transformation

Using Dependency Injection and Standardized Model Changes

Artur Boronat

Received: date / Accepted: date

Abstract When model transformations are used to implement consistency relations between very large models, incrementality plays a cornerstone role in detecting and resolving inconsistencies efficiently when models are updated. Given a directed consistency relation between two models, the problem studied in this work consists in propagating model changes from a source model to a target model in order to ensure consistency while minimizing computational costs. The mechanism that enforces such consistency is called consistency maintainer and, in this context, its scalability is a required non-functional requirement.

State-of-the-art model transformation engines with support for incrementality normally rely on an observer pattern for linking model changes, also known as deltas, to the application of model transformation rules, in so called dependencies, at run time. These model changes can then be propagated along an already executed model transformation. Only a few approaches to model transformation provide domain-specific languages (DSLs) for representing and storing model changes in order to enable their use in asynchronous, event-based execution environments.

The principal contribution of this work is the design of a forward change propagation mechanism for incremental execution of model transformations, which decouples dependency tracking from change propagation using two innovations. First, the observer pattern-based model is replaced with dependency injection, decoupling domain models from consistency maintainers. Second, a standardized representation of model changes is reused, enabling interoperability with EMF-compliant tools, both for defining model changes and for processing them asynchronously. This procedure has been implemented in a model transformation engine, whose performance has been evaluated experimentally 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 change propagation is performed in real time for those model sizes that are processable by other tools and, in addition, is able to process much larger models.

Keywords Mappings between languages · traceability · incremental execution · performance benchmark.

Artur Boronat
School of Informatics, University of Leicester
E-mail: artur.boronat@le.ac.uk

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 [2,36,32]. The Eclipse Modeling Framework (EMF) [46] has been used as a de facto standard for implementing modeling tools, e.g., for AADL in the OSATE platform [21] or for the AUTOSAR tool platform [1]−both in the automotive domain,−fostering interoperability via a common exchange format. Model transformation, widely accepted as the *heart and soul* of MDE [45], 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 [17,4]. This is particularly crucial when transformations are used to implement consistency maintainers between very large models, consisting of milions of elements. In this context, incrementality en-

sures that only those parts of the model that have been modified−a model change−are propagated along an already executed transformation [23,24].

Current state-of-the-art approaches that support incremental execution of model transformations share common features: the change propagation mechanism is usually decoupled from the change detection mechanism in order to facilitate maintainability of the consistency maintainer; and model changes are represented either in memory for synchronous notification or offline, with dedicated domain-specific languages, for asynchronous notification. The most mature tools rely on the observer pattern [22], where model changes are regarded as events that are notified to observers[1] at run time whenever a model is changed. This notification mechanism is synchronous and loosely couples model changes with the change 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, running as a thread, in order to listen for changes. These problems can be avoided by using offline changes. The observer pattern can be extended to enable asynchronous change notification using a publish-subscribe mechanism but this is normally achieved by using dedicated domain-specific languages to represent changes 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 model change propagation procedure is presented for executing model transformations in incremental mode that can handle documented model changes, called change scenarios in [6], i.e., documents representing a change to a given source model. Such documents are defined with the EMF Change Model [46], both conceptually and implementation-wise, guaranteeing interoperability with EMF-compliant tools. This design decision replaces an observer pattern 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 change. 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 model change propagation procedure has been implemented in YAMTL [9], a model transformation engine for very large models, enabling the execution of model transformations both in batch mode and in incremental mode without additional user specification overhead. YAMTL is implemented in Xtend, which transpiles to Java, and uses EMF as the (meta-)modeling front end. EMF is a de facto implementation of the MOF standard [39] for meta-modeling, in particular of *essential MOF (eMOF)*, and most of the concepts referred to in the rest of this article are common to all object-oriented meta-modeling frameworks. Specifically, an EMF metamodel may consist of classes defining modeling concepts in terms of their structural features, which can be attributes, if typed with data types, or references, if typed with other classes. Classes can be related using multiple inheritance hierarchies, and (unidirectional or bidirectional) associations, which can be containments (corresponding to the notion of compositions in UML). Features can be defined with multiplicity and ordering constraints. Therefore, the contributions presented should be reusable in tools built on top of other meta-modeling frameworks. Importantly, a substantial subset of EMF metamodels and their models can be represented as type graphs and graphs, respectively, for graph transformation [7,8].

In YAMTL, model changes can be offline, represented with the standard EMF Change Model, or online, using a tool-specific change notification API. Offline changes help decouple change recording from the transformation engine, which can be physically distributed. For example, when recording changes to a model in a given modelling environment, the model transformation engine is not required at all. Nevertheless, online changes allow for a faster, synchronous notification of changes at a lower level of abstraction.

Our new tool that implements the proposed model change propagation mechanism has been evaluated with all of the cases in the VIATRA CPS benchmark, including the up-to-now most performant solutions (using VIATRA) and a new solution using ATL [16,15], based on the Active Operation Framework [34]. The new extension greatly improves the performance of the batch execution mode when propagating model changes. Sparse changes can be propagated in μs. Larger model changes, involving knock-on effects, whose extent depends on the model size, in ms, for those models that are processable by other tools. Moreover, the economical use of resources in the YAMTL implementation enables processing much larger models, which cannot sometimes be transformed by the other tools involved in the study.

The VIATRA CPS benchmark was also used to analyze the performance of ATL in a specialized environment with an abundance of computational resources [16], e.g., the publish-subscribe case was analyzed with a Java heap of 130GB. Our experiments provide a complementary view by limiting computational resources

[1] Called EMF adapters in EMF terminology.

to those available in a standard computer, e.g., a Java heap of 12GB, recreating the conditions under which such tools are normally operated.

This work is structured as follows: §2 provides a self-contained description of the class of model transformations supported using a class diagram to relational schema model transformation as a running example; §3 presents the forward propagation procedure implemented in the model transformation engine together with the main innovations; §4 describes the case study used in the VIATRA CPS benchmark and a solution that uses the advanced model transformation features presented in the previous section; §5 discusses the performance of the transformation engine with an adaptation of the VIATRA CPS benchmark; §6 discusses related work from reactive and bidirectional model transformation; and §7 wraps up conclusions from the study.

This article is an extended version of [13], which presented the core notions of the proposed forward model change propagation mechanism, relying on dependency injection when object features were used in a model transformation rule. In this extension, we look at dependencies between computation steps. Specifically at dependencies between model transformation steps−MT steps henceforth−resulting from the application of a model transformation rule, and between query steps, resulting from the execution of a helper query, and MT steps. Dependencies between MT steps enable sophisticated incremental execution of model transformations, and they may need to be declared explicitly to allow for the use of programming facilities that are external to the model transformation language (e.g., by using Java collections in a model transformation rule). In addition, the extension also enables undoing MT steps that use such external programming facilities in the presence of a deleting source model change. Dependencies between query steps and MT steps enable the incremental execution of the model transformation when the helper query needs to be re-evaluated. These extensions allow for a larger class of model transformations to be executed incrementally. In this extension, we also provide a more comprehensive experimentation using different cases from the VIATRA CPS benchmark.

2 Model Transformation: A Running Example

The type of model transformations that are considered in this work are classified as unidirectional and outplace. 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.

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. 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 [8,20]. Fig. 1 shows the class diagram metamodel and the relational schema metamodel used to define the model transformation.

Model transformations are represented as a collection of model transformation rules−MT rules henceforth−where each rule is defined as a pair of model patterns, called left-hand side (LHS) and right-hand side (RHS). For example, rules `A->C` and `R->FK` of Fig. 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), e.g., `A:Attribute` and `COL:Column` in rule `C->T` of Fig. 1. Moreover, rules are augmented with a `when` clause to express conditions that must be satisfied by the variables in their LHS, and with a `where` clause to indicate how variables from their LHS and from their 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. Formulas in a `where` clause may be expressed in disjunctive form (assuming mutually exclusive conditions), expressing side effects as MT steps that may occur or not. The variables of the RHS of the main rule must appear either in the LHS of the main rule or in the RHS of a `where` MT step.

Rule `C->T` of Fig. 1 illustrates how to map a class to a table with a primary key column `PK_COL`. For each attribute `A` whose type is a `DataType`, the corresponding column is obtained by applying a rule `A->C` via a `where` clause. For each attribute `OTHER` whose type is the class `C`, matched in LHS of rule `C->T`, a new foreign key column is added to the table `T`, with the rule `R->FK`. Given a source model, the rules of a model transformation are applied until no more rules can be applied, producing a sequence of rule applications, which is called model transformation sequence. The second row in Fig. 1 shows a model transformation sequence from the given source class diagram to a target relational schema, representing a sequence of rule applications of the rules mentioned above from the given source model, where the specific MT steps have been abstracted away using the notation $*$.

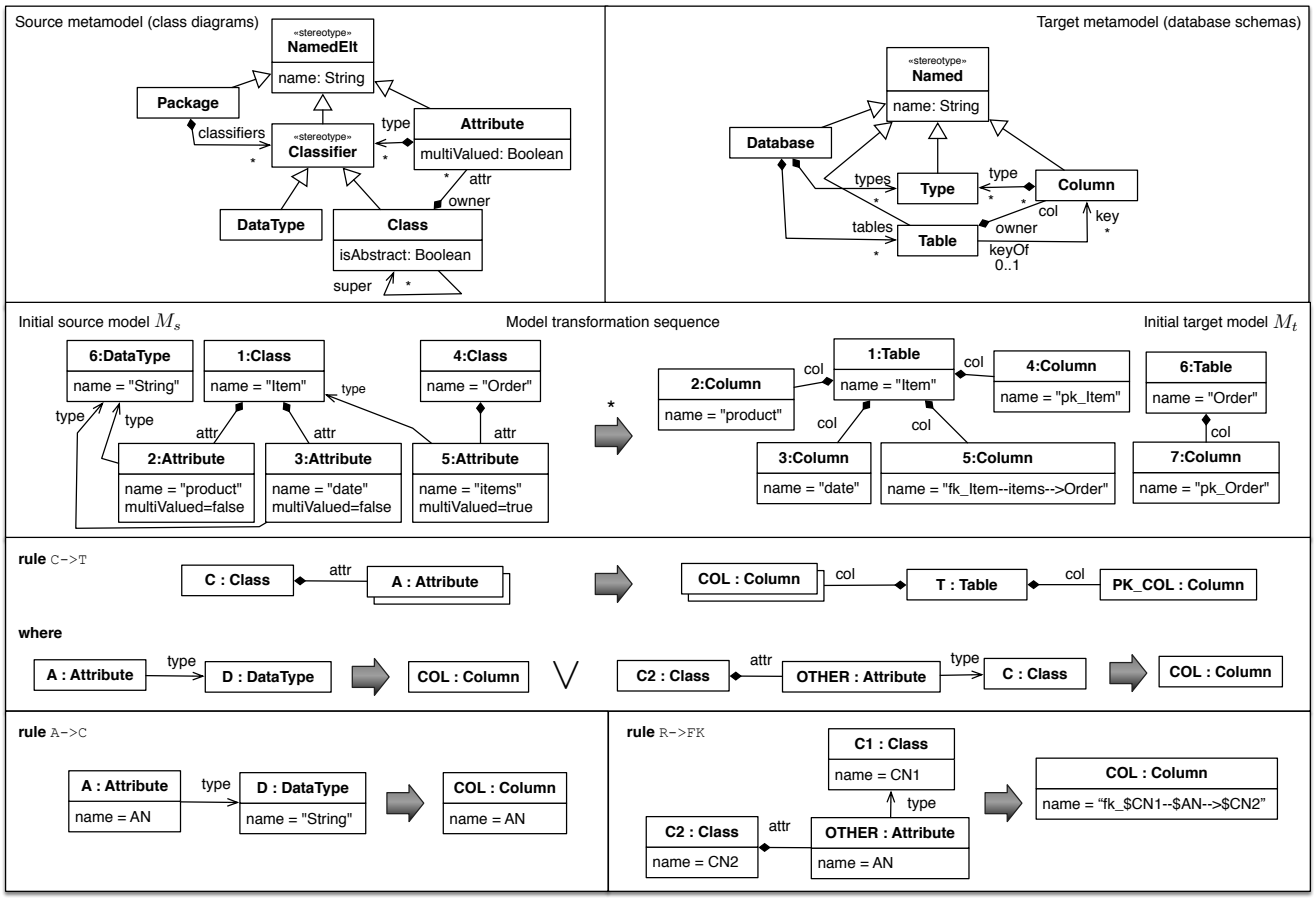

Fig. 1 Metamodels, example and MT rules.

Three different types of rules are considered in this work: matched, lazy and unique lazy. *Matched rules* are applied by the transformation engine and their application involve mapping objects matched by the LHS into objects as defined by the RHS in a one-to-one, injective relationship. Matched rules can be given a priority—a low number means high priority − which mandates the order in which the transformation engine is to match them. *Lazy rules* are applied on demand, from a `where` clause of another rule, but have a different semantics by yielding a one-to-many relationship between matched objects and objects that are created in the RHS, as each application of the rule creates fresh objects. Matched rules and lazy rules create an execution context, storing LHS matches so that these can be used during the evaluation of the corresponding MT step, where the objects of their RHS are created. *Unique lazy rules* combine aspects of the two previous types: they are applied on demand and they define a one-to-one relationship between matched objects and freshly created objects but they reuse the calling execution context. The three rules in Fig. 1 are matched rules. Examples of unique and lazy rules can be found in the case study of §4.

MT rules can be simplified by means of the use of helpers, which define queries using model navigation expressions, which may be represented either with graph patterns or with an OCL-like language. Helpers are often employed to initialize variables in a `where` clause. The case study of §4 provides examples of helpers.

From an operational point of view, MT rules are applied unidirectionally from LHS to RHS performing an out-place transformation following 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. Such matches 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, corresponding to nodes in the LHS pattern, to object identifiers, corresponding to nodes in the source graph, for the sake of presentation in this paper. A match is said to be valid when there is a one-to-one mapping between the variables in the source pattern and objects in a source model, and when conditions in `when` clauses are satisfied.

Second, during the *execution phase*, each match is processed by triggering the application of a MT rule, which is represented as a MT step, denoted by $r : \overrightarrow{in \mapsto \varsigma} \rightarrow \overrightarrow{out \mapsto \varsigma}$, which consists of a pair of two matches labelled with the rule name r, the match $\overrightarrow{in \mapsto \varsigma}$ for the input pattern of the rule with input variables in and matched objects ς, and the match $\overrightarrow{out \mapsto \varsigma}$ for the output pattern of the rule with output variables out and the object ς that result from applying the rule. Each match consists of a list of pairs, consisting of a variable name in (or out) and of an object reference ς. Vector notation is used to denote the list. 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` MT steps. In addition, `where` MT steps may further expand the structure of the target model. Whereas a `when` clause denotes a condition that needs to be satisfied to trigger the rule, a `where` clause expands the current transformation sequence by applying other MT rules implicitly. That is, `where` clause is used to transform some source elements into target elements, so that these can be used in the calling rule. This execution model resembles the application of forward rules used in triple graph grammars (TGGs) [44], 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 MT step.

The `where` clause of a MT rule can also contain a statement `insertDependency`(R, ς), with a rule name R and an input matched object ς, which is used to define explicit dependencies between the current MT step and MT steps of rule R involving ς in its match. This statement makes the transformation engine aware of user-defined dependencies between MT steps that may exist due to features that are external to the transformation language, e.g., when Java collections are used to share information between rules. The different types of dependencies will be explained in §3.6.

A MT rule can also be equipped with `undo` actions, which are used to express how to inverse the actions performed in the RHS. Undo actions enable the use of arbitrary data structures (e.g., Java collections), which are external to the model transformation language, in a MT rule while enabling incremental propagation of model changes that delete parts of the input model. Examples of how to manage explicit dependencies with `insertDependency` and of how to manage side effects using auxiliary data structures with `undo` are given in §4.2.

In YAMTL, a model transformation is specified textually. In a MT rule, the LHS pattern is written textually and `when` clauses are encoded as filter conditions. The RHS is writtern in terms of object templates where each feature (attribute or reference) is initialized with a lambda expression in Xtend, where Java collections can be used to either add or remove elements to a model. `where` clauses are encoded using the operator `fetch`, which operationalizes the semantics of a `where` clause, by finding a rule MT step that links the given source elements to target elements. Helpers, which are normally expressed using graph patterns in graphical concrete syntax, are encoded using queries in Xtend that traverse the model. A more detailed account of YAMTL's syntax and semantics can be found in [9].

3 Change-driven Model Transformations

This section presents the mechanism to propagate documented model changes δ_s, from a source model M_s to a target model M_t, for a given model transformation sequence t in an incremental way. Such propagation of source model changes corresponds to the incremental evaluation of a model transformation that has already been executed, and which is available in the form of a sequence of MT steps. For a model transformation to be executed incrementally only for those excerpts of a source model that have been modified, the MT steps that witness the execution of the model transformation need to be augmented with dependencies that track which parts of the source model are involved by it. The YAMTL transformation engine [9] has been extended with two modes of execution: *initialization*, a model transformation is executed in batch mode generating a model transformation sequence, using the original batch semantics [9], while tracking those parts of the source model involved in MT steps as *dependencies*; *propagation*, the model transformation is executed incrementally for a given source model change. Once the initialization is done, any number of source documented model changes δ_s can be propagated.

Given a source documented change 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 MT steps), and a changed source model M_s', the transformation engine propagates the model change δ_s along t. The effect of this forward propagation is the application of a model change 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. First, an explicit conceptualization of the relevant components of the propagation mechanism is presented in §3.1, segregating them from implementation details. As the initialization mode faithfully corresponds to the batch execution of a model

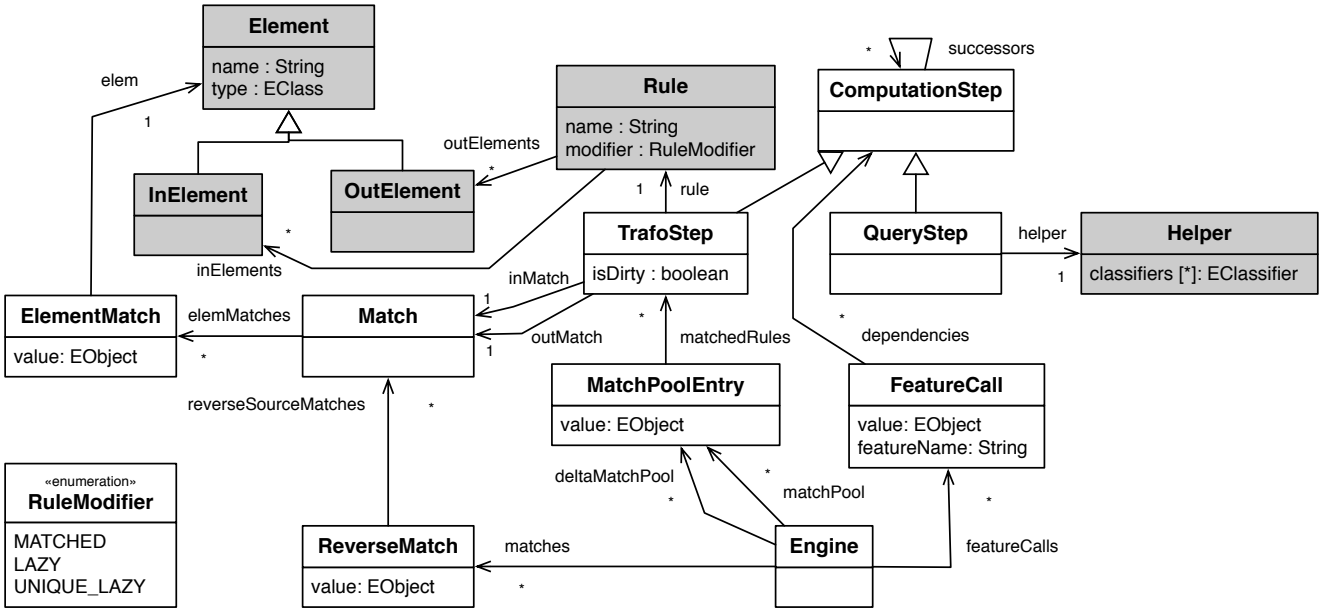

Fig. 2 Object execution model.

transformation, the discussion of this mode focuses on the type of dependencies that are implicitly injected in the transformation engine in §3.2. The standard EMF Change Model is summarized in §3.3 for the sake of self-containment. The discussion of the propagation mode focuses on its two main phases, namely impact analysis and change propagation, which are explained in sections 3.4 and 3.5, respectively. The different types of dependencies that are traced are presented in §3.6, explaining how explicit dependencies affect impact analysis and change propagation. The transformation used in the VIATRA CPS benchmark is described in §4.2.

3.1 Execution Model

This section presents an outline of the main concepts involved in the incremental propagation of source changes in YAMTL, to be used in the following subsections. Fig. 2 captures an excerpt of the syntactic concepts used to represent model transformations in memory, highlighted in grey colour, and the main semantic concepts used both to track dependencies and to execute model transformations incrementally. The model in Fig. 2 can be regarded as a loose contract defining the requirements that need to be considered in the design (and implementation) of a model transformation tool, in order to adapt the proposed change propagation mechanism.

A `rule` is defined with an input pattern consisting of several `inElements`, which define variables that need to be matched, together with `when` clauses imposing a local constraint on when the match is satisfiable, and with an output pattern, consisting of several `outElements`, which refer to binding expressions describing the side effects of the rule in the output model. A `ComputationStep` represents either the application of a rule to a given input match or the evaluation of a helper query. The rest of semantics concepts will be explained in the following subsections as they are needed.

3.2 Dependency Injection

Dependencies are defined by linking feature calls, that is, when an attribute or when a reference is used in a MT rule, to MT steps. 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. Dependencies are represented as references to the class `ComputationStep` in the class `FeatureCall` in the object model of Fig. 2. A dependency thereby links either an object ς or a feature call (ς, f) to the computation steps in which it is used. Such dependencies are detected when a MT rule is processed, 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 `when` conditions. When a match is found to be valid, the collection of dependencies is injected into the dependency registry for the MT step that uses that

Rule	Source Match	Target Match	Dependencies from M_s
C->T	C $\mapsto$ 1	T $\mapsto$ 1, PK_COL $\mapsto$ 4	(1,name), (1,att), (5, type)
C->T	C $\mapsto$ 4	T $\mapsto$ 6, PK_COL $\mapsto$ 7	(4, name), (4, attr)
A->C	A $\mapsto$ 2	COL $\mapsto$ 2	(2, name)
A->C	A $\mapsto$ 3	COL $\mapsto$ 3	(3, name)
R->FK	OTHER $\mapsto$ 5	COL $\mapsto$ 5	(5,name), (5,type), (1, name), (4,name)

Table 1 Analysis of dependencies for the initial MT $t : M_s \xrightarrow{*} M_t$ of Fig. 3.

match. Otherwise, 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 links between matched objects ς and the MT step in which they are matched. Such links are represented as ReverseMatches in the object model of Fig. 2.

Dependencies may also be found when executing a MT step, e.g., while executing initialization expressions associated with object features, either attributes or references, in model patterns in RHS and in where clauses. An initialization expression, sometimes referred to as binding expression, compute a value that is then bound to an object feature. Such initialization expressions are evaluated within the context of a MT rules and can refer features of objects that have been matched by LHS variables. In rule A->C of Fig.1, the initialization expression name = AN binds the value of the attribute name of the matched object of type Attribute to the attribute name of the newly created object. When an attribute or reference is used, either set or queried, there is a feature call that is injected as a dependency. In such cases, the transformation engine implicitly injects a dependency for the MT step every time a feature call in the source model is detected. As a result, note that several MT 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 Fig. 1 in initialization mode from model M_s. Each row in the table represents a MT 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 MT step. These dependencies are stored as FeatureCalls according to the object execution model of Fig. 2. Reverse matches are extracted from LHS matches, by reading them in the opposite direction. It is worth noting that the second match of the rule C->T has less dependencies than the first one because the rule R->FK, implicitly called in the where clause, does not find any attribute.

Dependency injection is specified with an aspect whose pointcut matches feature calls under a user-defined namespace. By default all feature calls under the specified namespace will be injected but this can be narrowed down to specific packages or classes depending on domain-specific needs. For example, when run-time performance is important and/or memory resources are limited. 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.

3.3 Representable Model Changes

The EMF Change Model [46] defines the language of changes that are applicable to an instance of any other EMF model. It is built-in in EMF and, therefore, available for any EMF-compliant tool. Moreover, it can be regarded as a technology-independent language for defining model changes because it is defined as a metamodel, which happens to be implemented in EMF. An EMF model is represented using a set of root objects. Each object is tagged with feature values, either attribute values or reference values, or simply references. There are two types of references, cross-references, simply called references, denoting the graph structure in the model, and containment references, denoting structural (*part-of*) relationships among objects. Root objects in a model are effectively roots of hierarchies of objects via containment references.

In this section, we describe how a documented change is represented using the EMF Change Model and how it can be automatically defined given any atomic model change that is received at run time, i.e., a *live* atomic model change.

In the EMF Change Model, a model change is represented as a ChangeDescription which contains a map of objectChanges, which refer to those objects that are updated in a model and, for each such object, it contains a list of FeatureChanges. That is, a change description describes whether root objects are added to the model or deleted from it, and changes to feature values, within a given object in the model. 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. 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 changes to feature values for an object but EMF also permits adding and removing root objects to a model, 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 [46].

Table 2 shows a classification of atomic model changes that are representable with the EMF Change Model as explained above. In the table, the column *cases* refers to how many cases are considered in the row, the column *granularity* indicates whether model changes are atomic or composite, the column *level* indicates whether the change is applied in a root object or within an object, the column *feature* denotes the type of feature involved, the column *change action* denotes the type of change, the column *change representation* indicates how the change is represented using the EMF Change Model, the column *DO* flags whether the change affects objects, the column *DFC* indicates whether the change affects feature values. Note that moving an object structurally−case 12 *move (inter.)*−is represented in a composite model change 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.

A model change, 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 changes* as a `ChangeDescription` for either a root object, a collection of root objects, a resource or a resource set. The recorded `ChangeDescription` is the representation of a *history model change* [6], from the changed 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. The recorded `ChangeDescription` can be reversed representing a *documented model change*, capturing the original model change performed by the user. This a documented model change can then be propagated along a model transformation.

The EMF change recorder enables the possibility of storing EMF change models to the point in which they are propagated, enabling durability, saving memory resources, and interoperability. Furthermore, recorded (history) changes can be regarded as a rollback mechanism for implementing transactional model changes, which may be performed live.

Given the model transformation sequence $t : M_s \xrightarrow{*} M_t$ of Fig. 1, Fig. 3 shows examples of documented model changes δ, defined over the source model M_s of the running example, which lead to a model transformation sequence $t' : \delta(M_s) \xrightarrow{*} M'_t$. Such model changes 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 changes, including moves, are highlighted in grey colour. Objects that are added, and thus created, have a new identifier. Objects that are changed and/or moved preserve their identifier. Removals are highlighted by using dashed lines for the contour lines of the corresponding shapes. The given model changes are instantiations of:

- model change `a` (case 4), changing the name of the class `Order` to `Invoice`;
- model change `b` (case 1), adding a root class `Product`;
- model change `c` (case 9), adding a single-valued attribute `amount` to class `Item`;
- model change `d` (case 10), removing the attribute `date` from class `Item`; and
- model change `e` (case 11), 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 model changes is discussed and the aforementioned examples will be used for illustrating them.

3.4 Impact Analysis

In this subsection, we discuss how source documented model changes are analyzed in order to determine which MT steps are affected. This analysis is comprised of

Cases	Granularity	Level	Feature	Change action	Change 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 change types with their representation in EMF.

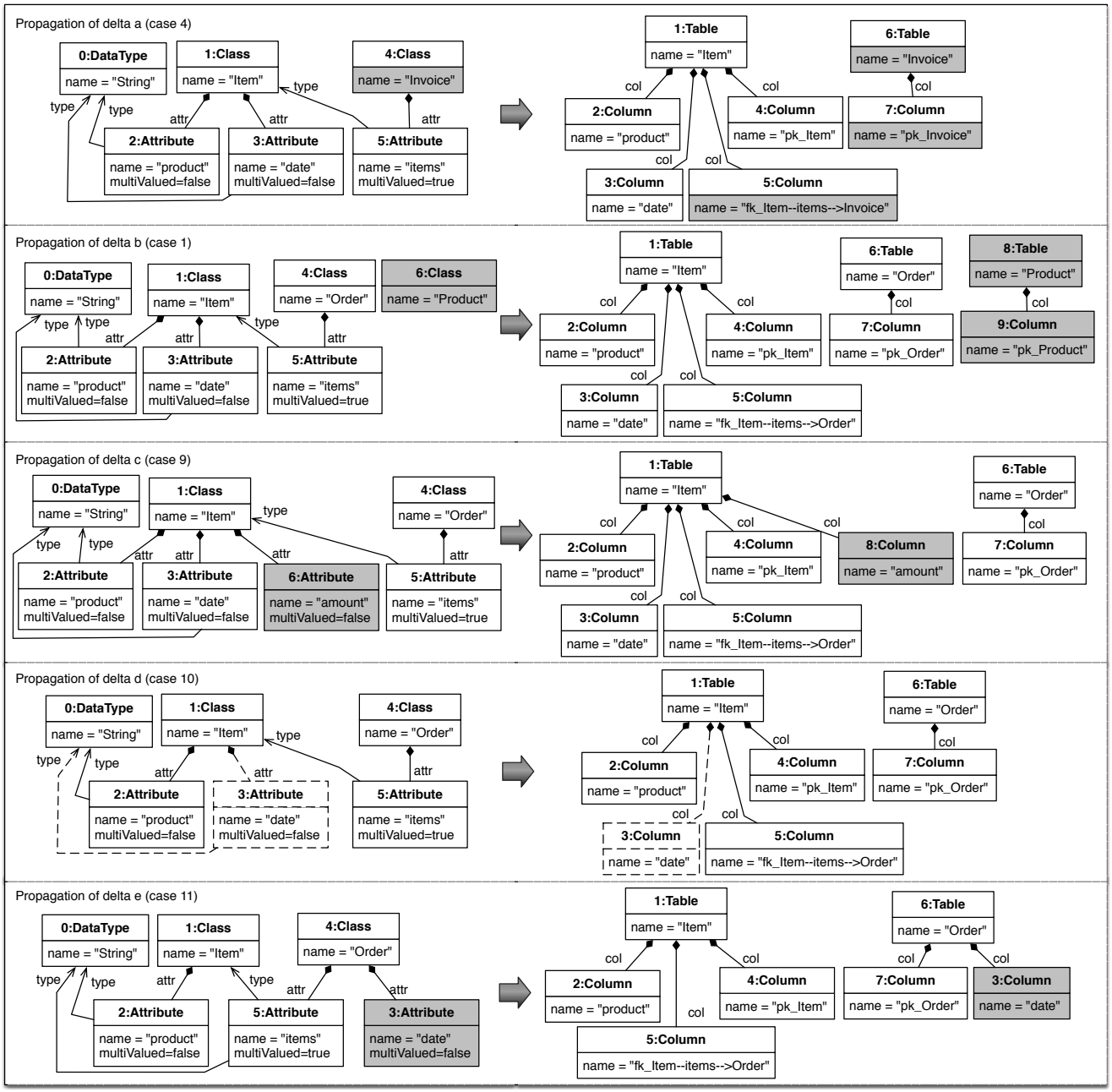

Fig. 3 Source/target metamodels, initial synchronized models and forward change propagation (a-e).

	Case	DO	DFC	Rule	Source Match	Target Match	changeMatchPool	dirty?
a	4	–	(4, name)	C->T	c ↦ 4	t ↦ 6, pk_col ↦ 7	✓	✓
b	1	(6, *ADD*)	–	C->T	c ↦ 6	t ↦ 8, pk_col ↦ 9	✓	
c	9	–	(1, attr)	C->T	c ↦ 1	t ↦ 1, pk_col ↦ 4	✓	✓
		(6, *ADD*)	–	A->C	att ↦ 6	c ↦ 8	✓	
d	10	–	(1, attr)	C->T	c ↦ 1	t ↦ 1, pk_col ↦ 4	✓	✓
		(3, *DEL*)	–	A->C	att ↦ 3	col ↦ 3		✓
e	11	–	(1, attr),	C->T	c ↦ 1	t ↦ 1, pk_col ↦ 4	✓	✓
		–	(4, attr)	C->T	c ↦ 4	t ↦ 6, pk_col ↦ 7	✓	✓

Table 3 Impact analysis of source model changes a-e.

three main steps: identification of atomic model changes from a documented model change, update of the set of objects potentially affected newly enabled rules, and marking of MT steps impacted by changes.

Identification of atomic model changes. In the first step, given a model change represented as a `ChangeDescription` in the EMF Change Model, 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. If an object is moved, either within the same collection or structurally, it is not considered as an addition or as a removal.

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 model change, `FeatureChange`s and `ResourceChange`s are traversed considering two cases. The first case happens 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 model change, which could occur by removing it either from a containment reference or from the root contents of the resource. The second case happens when an object is deleted and it is not added elsewhere in the model change. `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 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 change types are represented using the EMF Change Model (column *model change 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 model changes of Fig. 3, so that each model change is internally represented by two sets `DO` and `DFC`. Note that the sets `DO` and `DFC` decouple the transformation engine from the EMF Change Model and provide another entry point for defining model changes programmatically, which can be used for capturing atomic *live changes* received via observers of change events[2].

Update of the set of objects potentially affected by newly enabled rules. For each dirty object $(\varsigma, \texttt{ADD})$, the object ς is added to the set of objects associated with the type of an object ς, denoted $\texttt{type}(\varsigma)$. This set of objects is referred to as the *extent of an object type* $\texttt{type}(\varsigma)$ and it is used to compute matches for MT rules. The augmentation of the extent of an object type with new objects potentially enables new matches when rules are matched during the change propagation phase. Π is used to denote the union of all type extents involved in a model transformation sequence.

Marking of impacted MT steps. In this step, MT steps that are affected by the atomic changes in the source model change are marked as dirty. For each dirty object $(\varsigma, \texttt{ADD}) \in \texttt{DO}$, 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 $(\varsigma, \texttt{DEL}) \in \texttt{DO}$, we obtain the list of MT steps that are affected from the map of reverse matches. Such MT steps will then remain transient and the objects in their target match will not be referenced from other objects in the target models. A transient MT step is not realized and its effects are forgotten when the model transformation ends its execution. In particular, note that when processing root objects or a containment reference, an object that is removed in the model change is not present in the changed source model and, therefore, it does not trigger the MT step that had been executed in the initial transformation.

For each dirty feature call $(\varsigma, f) \in \texttt{DFC}$ we obtain the list of MT steps that are affected from the registry of dependencies. For each such MT step, the satisfaction of its source match is checked. If such source match is still valid, then it is inserted into `changeMatchPool`, the

[2] Implemented as EMF adapters in EMF

pool of matches that are used to apply rule applications during the change propagation phase.

For each atomic change in Fig. 3, Table 3 shows the marking of MT steps that are (re-)applied according to the dependencies of Table 1. In Table 3, the column *case* refers to the type of model change from Table 2; the column `DO` indicates the set of dirty objects involved in the change, if any; the column `DFC` indicates the set of dirty feature calls involved in the model change, if any; the column *rule* indicates the rule of a MT step affected by the model change; the column *source match* indicate the objects involved in the source match of the MT step; the column *target match* indicate the objects involved in the target match of the MT step; the column ***changeMatchPool*** indicates whether the MT step is selected to be re-applied; and the column *dirty?* indicates whether the MT step is marked as dirty or not. Whenever a model change affects more than one MT step, each MT step is added in a separate line within the row for the model change.

In summary, if a MT step is re-applied, its current source and target matches are included, it is marked as dirty and included in `changeMatchPool`. If a MT step is not to be re-executed, it is simply marked as dirty. New MT steps, with fresh matches due to new objects, are found in the matching phase of the model change propagation phase, explained in the next subsection.

3.5 Model Change Propagation

After the impact analysis phase, model change propagation proceeds by executing a model transformation using the matching and execution phases, as outlined in §2. Fig. 3 illustrates the propagation of source model changes according to the model transformation of Fig. 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 type extent used for model change propagation, initialized during the change impact analysis phase. Therefore, new matches may be found for objects that have been created by the source model change. Those matches are inserted both into `matchPool` and `changeMatchPool`, creating new MT steps. Table 3 shows that two new MT steps are applied, one for rule `C->T` in model change `b`, and one for rule `A->C` in model change `c`.

Execution Phase. During the execution phase, MT steps determined by the matches in `changeMatchPool` are executed. Such matches originate from the impact analysis phase, corresponding to MT steps that are *dirty* and need to be re-executed, and from the matching phase above, corresponding to new MT steps.

The re-execution of a MT step is performed as in the batch/initialization mode but for the creation of MT steps. Whereas a newly applied MT step needs to get its output objects initialized (instantiated for output elements), a dirty MT 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 MT 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 MT 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 MT step that produced it [9]. Therefore, when a dirty MT 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 MT step will be transformed from scratch. References to already initialized objects will be simply fetched. These can be updated in the calling MT step if needed. Hence, the granularity of the target model change is as fine grained (at binding level) as the source model change for the underlying graph structure of the model.

3.6 Indirect Dependencies

In this subsection, dependencies (introduced in §3.2) injected for MT steps are augmented by considering all of the subsequent MT steps that can be affected by a source model change. This involves considering dependencies between computation steps and the type of model changes that can be propagated along them, characterizing when a model transformation can be executed incrementally.

There are two main types of dependencies: *direct* dependencies, linking a feature call to a computation step where it is used−as presented in §3.2,− and *indirect* dependencies, linking a feature call to all computation steps that may be affected if the object feature

Dependency	Data	Control Flow
`helper` → `trafoStep`	`allInstances` (implicit)	`where` (implicit)
`trafoStep` → `trafoStep`	`insertDependency` (explicit)	`where` (implicit)/`insertDependency` (explicit)

Table 4 Types of dependencies and how they are tracked.

is changed. That is, we are considering dependencies between computation steps in order to infer whether a change in an object feature value affects a MT step that does not call that object feature directly. Hence the name, indirect dependencies. In the following, indirect dependencies, common in rule-based model transformation, are summarized indicating how they are captured. Then the previous impact analysis and change propagation phases are extended to consider them.

The algorithm, therefore, considers new features of the model transformation language that had not been discussed in [13]. On the one hand, implicit dependencies between helper queries and MT steps enable the incremental execution of the model transformation when a helper needs to be re-evaluated. On the other hand, explicit dependencies, with the statement `insertDependency`, between MT steps, enable the incremental execution of MT steps with side effects that are encoded using programming facilities that are external to the model transformation language. Furthermore, deleting model changes can also be propagated by considering `undo` actions for MT rules.

3.6.1 Injection of Indirect Dependencies

Indirect dependencies are recorded by injecting dependencies between computation steps, as `successor` references in the execution model of Fig. 2, where two different subtypes can be identified: *data dependencies*, when the data changed by one computation step affects a successor computation step (either enabling it, disabling it, or affecting its results), and *control flow* dependencies, when the successor computation step can only be executed after the predecessor computation step and there is no necessary data dependency between them.

Given that helpers are evaluated at the start of the transformation as query steps, a computation step cannot be succeeded by a query step. The main types of indirect dependencies are listed in Table 4, where the column *dependency* indicates the predecessor of a dependency, the column *data* indicates when there is a data dependency between two computation steps, and the column *control flow* indicates when there is a control dependency between two computation steps.

Data dependencies are injected when the operation `allInstances`, used to fetch the extent of a type, is used in helpers and when helpers are invoked in a MT step. In addition, dependencies between MT steps are declared explicitly with the statement `insertDependency`. The scenarios where these dependencies arise are explained next.

Control dependencies are implicitly injected when a `where` clause is executed and explicitly injected using the statement `insertDependency`. For example, when the user wants to impose some control flow dependencies between rules with different priorities.

3.6.2 Outline of Extended Algorithm

Fig. 4 lists the extended algorithm for propagating model changes, where the parameters are a model change δ, represented as a `ChangeDescription`, a `matchPool` with a collection of MT steps that witnesses the execution of a model transformation, the union of object type extents Π, and the collection `dependencies` linking MT steps, in `matchPool`, to feature calls made during the execution of the MT step. In Fig. 4, lines $(3)-(5)$ correspond to the impact analysis phase and lines $(7)-(12)$ correspond to the change propagation phase, which will be explained in the following subsections. The steps in the algorithm highlight the logic required for processing indirect dependencies and data dependencies between algorithmic steps. The resulting values of the algorithm are: the updated `matchPool`, where MT steps may have been added, modified or removed in order to propagate the model change δ; the updated union object type extents Π, where objects may have been added or removed; and the updated collection `dependencies`, where dependencies may have been added or removed depending on the effect of the model change δ on MT steps in `matchPool`. These resulting values enable the application of the operation *propagateModelChange* on another model change δ for processing sequences of model changes iteratively. Consistency maintainers between different models can be implemented as model transformations that may enforce the consistency of those models by propagating model changes along the corresponding model transformations.

Throughout the rest of this section, excerpts of the object model of Fig. 2 are referred to as parameters in the algorithm, assuming that a directional association is implemented as a map data structure, mapping a source element to a target element, when the upper bound of the association is one, or to a list of elements, when the upper bound of the association is many. In

```
propagateModelChange(δ, matchPool, Π, dependencies) =                                          (1)
    // impact analysis phase                                                                   (2)
    ⟨DO, DFC⟩ := identifyAtomicChanges(δ)                                                      (3)
    type := updateChangeTypeExtent(DO, DFC, Π)                                                 (4)
    ⟨changeMatchPool, U, H⟩ := markTrafoSteps(DO, DFC, Π)                                      (5)
    // model change propagation phase                                                          (6)
    do {                                                                                       (7)
        ⟨successors, U⟩ := evaluateHelper(H, U)                                                (8)
        ⟨changeMatchPool, dependencies⟩ := match(type, changeMatchPool, dependencies, successors, U, H)   (9)
        ⟨matchPool, dependencies, successors⟩ := execute(changeMatchPool, dependencies, U)     (10)
        matchPool := undo(matchPool, U)                                                        (11)
    } until (successors = ∅)                                                                   (12)
    return⟨matchPool, Π, dependencies⟩                                                         (13)
```

Fig. 4 Algorithm for propagating model changes augmented with indirect dependencies.

particular, `matchPool` refers to the store of MT steps, each of which includes both a source match and a target match, Π refers to the type extent that maps a type to its population of objects, `dependencies` refers to dependencies indicating the MT steps in which a feature call is used, `successors` refers to indirect dependencies, either between a query step and the MT steps where they are used or between two MT steps. Two auxiliary data structures are used in the algorithm, a list `H` of affected helpers, those whose list `Helper::classifiers` are affected by changes, and a list `U` of MT steps to be undone, should the model change require so.

3.6.3 Impact Analysis

A model change affects an indirect data dependency, with a predecessor computation step and a successor MT step, when the model change affects the predecessor step as explained in §3.4. Indirect dependencies are implicitly injected, in the clause `where` of a rule, when the resulting value of the helper is used−using a query step as predecessor. Other types of dependencies are captured explicitly with the operation `insertDependency`(R, ς), with a rule name R and an input matched object ς, to be used in an out element action. This operator fetches all MT steps of rule R involving the object ς in their source match and creates indirect dependencies between the current MT step (as predecessor) and them (as successors), where the operation `insertDependency` is invoked. User-defined dependencies enable the incremental evaluation of MT rules with side effects, defining which MT steps work with the same shared variables. Thereby, the model transformation language does not prescribe the type of data structures that can be used to store side effects, facilitating interoperability with external libraries.

Three use case scenarios need to be considered when dealing with indirect dependencies: *a)* objects are created or removed, affecting the corresponding type extent, which is used by the `allInstances` operator in a query step; *b)* a data structure is used to store side effects by the predecessor computation step, for example a collection data structure that is shared among several MT steps; and *c)* the predecessor step uses a helper. In what follows, the impact analysis in §3.4 is extended to consider indirect dependencies.

In case *a)*, when a dirty object−either to be added or to be removed−is identified in line (3) of the algorithm, the helpers using expressions involving `allInstances` that traverse the extent of the type of the dirty object are singled out. The field `Helper::classifiers`, from Fig. 2, is used to associate a helper with the contextual type used in an expression involving `allInstances`. In the marking of impacted MT steps, in line (5) of the algorithm, helpers whose classifiers are affected by changes are added to the list `H`.

In case *b)*, when a MT step that changes a collection, which is shared between different MT steps, is marked as dirty in line (3) and it is not re-applied, it is not re-executed and the data structure would remain populated with stale data. To avoid this situation, the right-hand side of a MT rule is extended with `undo` actions (a formula over the right-hand side pattern), indicating how to undo side effects. MT steps to be undone are identified during the impact analysis phase, in line (5), by checking which dirty objects that are removed enable source matches of MT steps. Such MT steps are added to the list `U`.

In case *c)*, indirect dependencies are injected during the propagation phase either when a helper is invoked, in line (8), or when a MT step with a statement `insertDependency` is executed, in line (10). Line (8) of

the algorithm identifies which MT steps need to be re-executed by providing the list of `successors`. Line (10) injects dependencies, via statements `insertDependency` in rules, while MT steps are re-executed.

3.6.4 Change Propagation

When propagating source model changes with indirect dependencies in line (10), the re-execution of the predecessor computation step (either a query step with a helper or a MT step) may invalidate the successor computation step, whose validity needs to be checked and re-executed if needed. The main algorithm in §3.5 is then extended to consider affected query steps (involving helpers) and MT steps that need to be undone, as identified in the previous phase.

Affected helpers are re-evaluated and successor MT steps are marked as dirty in line (8). Matches for all new objects are found in line (9) and, if MT steps are to be executed fully (including actions in the right-hand side), they are re-executed in line (10), as explained in §3.5. This step produces a new map `successors` of indirect dependencies if there are statements `insertDependencies` in the rules of the MT steps being executed. MT steps that remain dirty are then executed using their `undo` actions in line (11), cleaning up any stale data that may be left in shared variables.

Change propagation occurs in an iterative fashion, starting from computation steps affected by the source model change, and iterating over the steps above while there are indirect dependencies between MT steps (due to the evaluation of an operation `insertDependency`).

3.7 Domain-Specific Optimizations

A model transformation can be executed incrementally without user specification overhead in declarative transformations that do not use auxiliary data structures. However, as model transformations are optimized using libraries that are external to the model transformation language, e.g., performing side effects on auxiliary shared data structures, some domain-specific logic needs to be programmed manually as already explained in previous subsections. That is, when actions affecting object patterns in the right-hand side of a rule need to account for the fact that a MT step may be re-executed. The following cases need to be considered:

- when auxiliary data structures are used and an object that is inserted in a shared collection in a MT step, the re-execution of the MT step may insert it again, and duplicates may need to be avoided;
- in a similar situation, when the MT step is invalidated, its effects result in stale data, which needs to be refreshed using `undo` actions;
- dependencies between MT steps, due to the use of shared data structures, need to be explicitly declared using `insertDependency` statements.

Checks of this type are domain-specific and are defined manually.

4 VIATRA CPS Benchmark

The VIATRA CPS benchmark [50] provides the specification of a problem [52] solvable as a model transformation and a framework for comparing the performance of different model transformation tools, with incremental execution of model transformations.

In this section, we borrow material from [50] to present the main idea behind the problem of the benchmark and explain how it has been solved with YAMTL.

4.1 Problem Specification

The main problem consists in deploying the specification of a cyberphysical system (CPS). A CPS specifies application types, whose behaviour is specified as a state machine, and host types, specifying computational capacity (default RAM, CPU, HDD). In a state machine, transitions may specify an action for sending or waiting for a signal. A CPS also specifies a topology of host instances to be used for deployment. In a CPS model, application instances, which have an active state, can be allocated to host instances.

A deployment model contains a topology of host instances with deployed applications running on them, where each application's behaviour is given in terms of states and transitions. Each application's behaviour has a current state and transitions may trigger other transitions. A transition is triggered when it is waiting for a given signal and such triggers are recorded explicitly, each sending transition refers to each target transition that it triggers.

From a more technical perspective, given a CPS model, a solution must generate a deployment model, mapping host instances, application instances and state machines to their corresponding counterparts in the target model. The most computational expensive parts consist in instantiating the state machine for each deployed application, and in computing trigger references in the deployment model. Moreover, such mappings must be recorded by generating a traceability model instance, which relates elements of a CPS model with those of a deployment model.

Rule	Rule Type	Advanced Features
`CyberPhysicalSystem_To_Deployment (C2D)`	matched	
`HostInstance_To_DeploymentHost (HI2DH`	matched	
`ApplicationInstance_To_DeploymentApplication (AI2DA)`	matched	undo
`StateMachine_To_DeploymentBehavior (SM2DB)`	lazy	
`State_To_BehaviorState (S2BS)`	unique lazy	
`Transition_To_BehaviorTransition (T2BT)`	unique lazy	explicit dependencies
`Transition_To_BehaviorTransition_Trigger (T2BTT)`	matched, transient	

Table 5 Outline of YAMTL Transformation Rules.

Solutions are implementable with a model transformation. The performance of solutions is measured by taking the execution time of the transformation in different cases [51], which define different communication models by modifying the topology of host instances:

- *ClientServer*: this case creates a forest of stars, where each tree is formed by a server and several clients. Transition triggers point from clients to servers.
- *LowSynch*: this case creates a CPS model where state machines have only a few transitions with actions. Therefore only a few transitions are triggered.
- *PublishSubscribe*: this case is similar to the *ClientServer* case, but both the communication and triggering directions are inverted to have a single publisher send messages to subscribers.
- *SimpleScaling*: in this case application instances are allocated uniformly to host instances, without imposing constraints on the communication topology.
- *StatisticBased*: this case uses model statistics from industrial UML models mapped to the CPS domain.

In the benchmark, a solution transformation is executed in a two-phased approach: first, it is applied to a given CPS model and, second, after updating the CPS model, the transformation is applied again. The benchmark also considers two scenarios for assessing the performance of solutions: *batch scenario*, where the transformation is executed in batch mode in both transformation applications, and *incremental scenario*, where the transformation is executed incrementally, propagating only those model changes that are applied in the source model in the second phase.

4.2 Solution to the VIATRA CPS Benchmark

In this section, the YAMTL solution for the VIATRA CPS benchmark is briefly described, illustrating some of the advanced features mentioned in previous sections. The solution is available on GitHub [11].

Table 5 enumerates the rules that form the model transformation definition. Each rule maps one object from a cyber physical system to a deployment model according to the benchmark transformation specification [52]. In the table, the name of each rule indicates which classes are mapped. In addition, the table also highlights the type of rule and whether it uses advanced features for managing incremental propagation.

The model transformation contains two helpers: `waitingTransitions` and `sendingTransitions`, which group transitions with a `wait` action and with a `send` action, respectively, by their signal identifier and by their application type identifier, in that order.

Rule `C2D` creates the root object of the deployment model and rule `HI2DH` creates deployment hosts in it. Rule `AI2DA` creates deployment applications with a unique behaviour by invoking the rule `SM2DB`. This rule is declared as *lazy* and every time it is invoked it transforms a state machine into a fresh deployment behaviour, and maps the state machine to the deployment behaviour in an auxiliary map for analysing reachability of signals through the corresponding host instance topology. Rule `SM2DB` instantiates the behaviour type, represented as a state machine, by mapping states and transitions using the rules `S2BS` and `T2BT`, respectively. These last two rules are declared as *unique lazy*. The final rule, `T2BTT`, is a matched rule with lower priority, that updates triggers in transitions with `send` actions by performing reachability analysis though the host instance topology. Transitions with `send` actions are matched with a `when` clause, represented as a filter condition in the corresponding YAMTL MT rule. Trigger references denote which `wait` transitions are triggered from transitions with a `send` action. As it has lower priority, it traverses all of the transitions once they have been created in the deployment model by the rule `T2BT`.

Rule `T2BT` injects explicit dependencies, using the statement `insertDependency`, from MT steps of the rule `T2BT`, processing a transition with a `send` action, to MT steps of the rule `T2BTT`, updating the trigger reference of the corresponding behaviour transitions. These two rules are executed with different priority and they use a map to share how state machine transitions are mapped to behaviour transitions. `T2BT` creates an entry when a behaviour transition is created and `T2BTT` uses this information for computing trigger references. The use of this shared data structure results in a data dependency between MT steps of these two rules: modifica-

tions in `send` actions of a transition may modify trigger references in behaviour transitions. As explained in §3.6, such type of indirect dependencies cannot be inferred automatically because the transformation uses programming constructs of the host language, Xtend/Java, which are not managed by the YAMTL transformation engine.

Rule `AI2DA` declares an action `undo`, defining how to incrementally undo the side effect of a MT step that creates a deployment application in case the input application instance is deleted. This `undo` action is needed because application instances are stored in an auxiliary map, used for computing reachability of signals when computing trigger references. Such map is not managed by the transformation engine and is used to implement domain-specific logic.

5 Performance Analysis

In the present work, we aimed at answering the following research questions: *(RQ1)* How does the overhead imposed by the dependency injection mechanism, both during the initial phase and during the incremental propagation phase, hinder performance? *(RQ2)* How does our solution scale with respect to the size of the given data set during the initialization phase? *(RQ3)* How does our solution scale with respect to the size of the given data set during the change propagation phase? The methodology used below enables analysing the scalability of the solution developed in YAMTL both in absolute terms, by looking at performance results, and in relative terms, by comparing those performance results with those of the other solutions, developed with state-of-the-art model transformation tools.

5.1 Methodology

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 [52]. The model transformation definition implemented for our model transformation engine, named *YAMTL-incr* and presented in §4.2, 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 [10] and YAMTL is available at `https://yamtl.github.io/`. The experiments were run on a MacBookPro11,5 Core i7 2.5 GHz, with four cores and 16 GB of RAM. For the experiments the following software was used: ATL/EMFTVM (4.0.0); ATL SDK (4.0.0); CPS metamodels (0.1.0); Eclipse (4.7.3); EMF SDK (2.13.0); JRE (build 1.8.0_72-b15); VIATRA SDK (1.7.2); and Xtend SDK (2.13.0).

This evaluation is an extension of the one performed for the batch component of the VIATRA CPS benchmark in [9]. From the original VIATRA CPS benchmark, two incremental variants of the transformation implemented with *EMF-IncQuery* have been selected: *ExplicitTraceability* (EXPL) [48] and *QueryResultTraceability* (QRT) [49], out of which the first one is the best performing solution up to now. In addition, a recent solution for the CPS benchmark using ATL [16,14], with the Active Object Framework (AOF), has also been included in the evaluation.

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 [25] 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.

All of the scenarios provided in the original benchmark were considered. The CPS model generator [53] 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, very large models.

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, model change 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 change propagation phase includes the application of the model change to the source model and its propagation. Only initial transformation and model change 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.

5.2 Results

In the solutions *ATL*, *EXPL* and *QRT*, the model change is applied to the source model by directly modifying the resource containing the model. In the solution with YAMTL such model change was recorded and persisted using the EMF Change Model as described in §3.3. To analyze whether this feature could become a threat to validity, a separate experiment was run by excluding the query part of the model change (searching for the objects to be changed) 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. 5 shows the performance results obtained, both for the initial model transformation and for forward model change propagation, for the models generated for the different cases. 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.

5.2.1 How does the overhead imposed by the dependency injection mechanism, both during the initial phase and during the incremental propagation phase, hinder performance? (RQ1)

The measurements of the performance of the execution of YAMTL in batch mode (*YAMTL-batch*) over the source model has been included in Fig. 5. It can be seen that injection of dependencies incurs a small penalty during initialization time, as expected.

5.2.2 How does our solution scale with respect to the size of the given data set during the initialization phase? (RQ2)

In the initialization phase, the VIATRA solutions (*EXPL* and *QRT*) operate several orders of magnitude slower. The ATL solution shows better initialization performance than VIATRA although it was more memory-demanding in some cases, e.g., *publish-subscribe*. The solution in YAMTL showed similar scalability to other solutions but with a more efficient run-time performance, in some cases (e.g., statistic-based), with an improvement of several orders of magnitude.

5.2.3 How does our solution scale with respect to the size of the given data set during the change propagation phase? (RQ3)

In the propagation phase, for the cases *client-server*, *lowSynch* and *statistic based*, it can be observed that while *YAMTL-incr* exhibits a constant propagation time (in μs.) for the source model change, the cost of the other solutions depends on the size of the input model. However, it was found out that this behaviour is not generalized.

In the cases *publish-subscribe* and *simple scaling*, whenever an application instance is created with a behaviour that contains a transition with an action `wait` for a signal, then all behaviour transitions with an action `send` for both the same signal and the same application type identifier must be changed in order to trigger appropriate transitions. In the YAMTL solution, the behaviour transitions that need to be changed are designated in the explicit dependencies inserted in rule `T2BT`. These explicit dependencies refer to the MT steps of the rule `T2BTT` that change the triggers of these behaviour transitions. As the re-execution of each such execution step involves a search of those behaviour transitions with `wait` actions that are reachable from behaviour transitions with `send` actions, the computational cost of the model change propagation increases with the size of the target model change (including indirect dependencies). One of the VIATRA solutions (EXPL) outperforms YAMTL in these cases, possibly because the trigger feature of a behaviour transition with a `send` action is changed incrementally, whereas YAMTL recomputes all of the triggers for such a behaviour transition within the MT step.

The worst case scenario is embodied by a source model change that shares no context with the already transformed source model. For such type of model changes, a batch model transformation is more efficient.

5.3 Threats to Validity

To ensure construct validity, we have reused the VIATRA CPS benchmark, which proposes a problem solvable with model transformations, including a random model instance generator for different scenarios, test cases to ensure the correctness of solutions and a standardized way of measuring performance across solutions. However, the VIATRA CPS benchmark is designed to compare the performance of solutions, as run on top of a model transformation engine. Therefore, there is a layer of indirection, and all claims regarding scalability of transformation engines need to be regarded through the solution that is being executed.

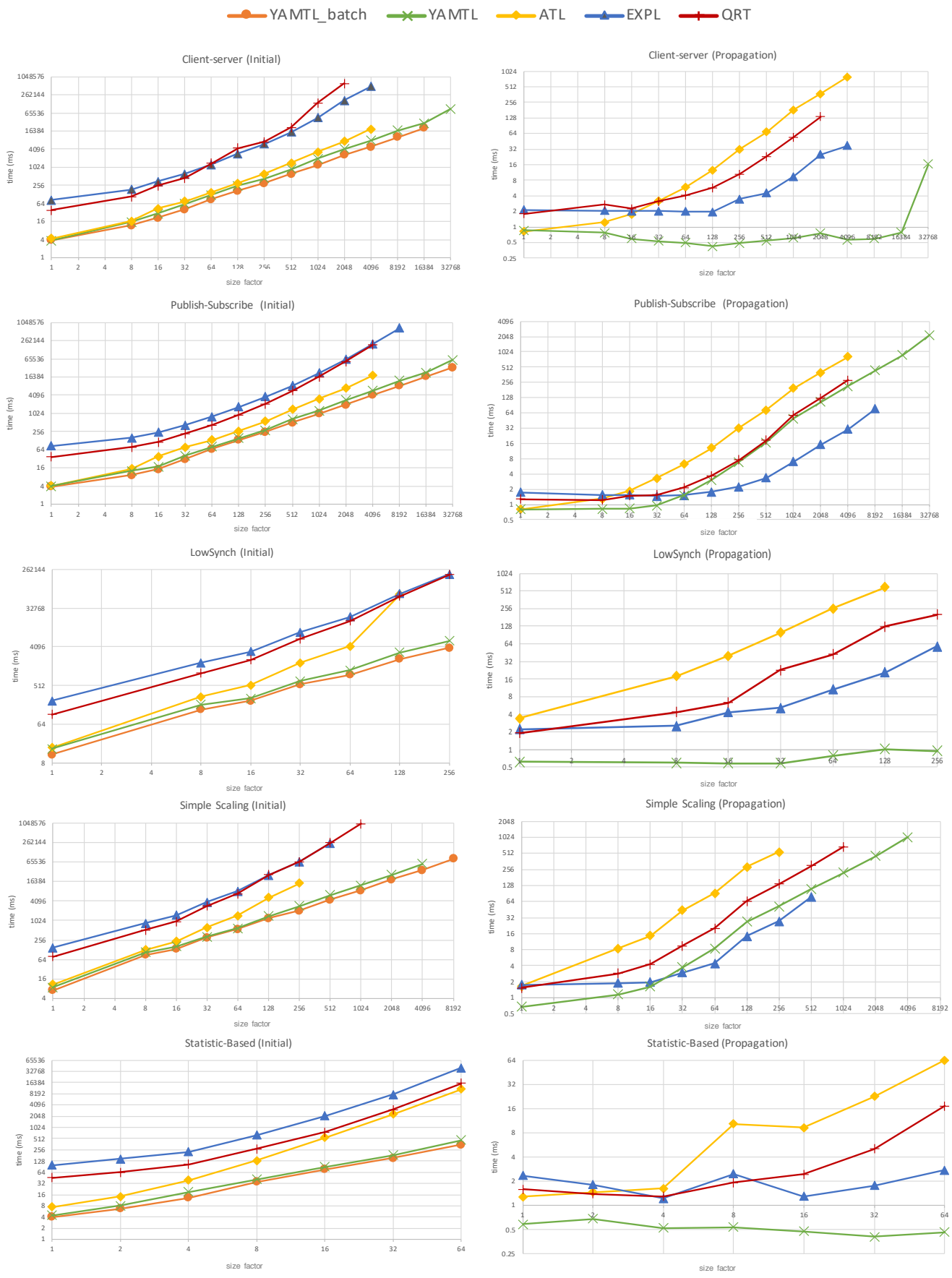

Fig. 5 Performance of initialization (left) and model change propagation (right).

Regarding internal validity, the experiments were performed with model instances generated by the CPS random model generator and, once generated, the same instance was used for all of the experiments. While this ensures reliability across experiments involving different tools, all of the experiments used the same model instance. Intentional bias has been avoided by using a benchmark proposed by the research community, where solutions to the CPS benchmark have been proposed by experts in the corresponding transformation tools. YAMTL does not excel in all of the scenarios, although the overall performance, combining running times in the initial phase and in the incremental phase, are indeed highly performant.

Regarding external validty, while representative model transformation tools with support from incremental evaluation of queries and transformations have been chosen from the research literature, no attempt has been made at completely covering all model transformation tools. Furthermore, the VIATRA CPS benchmark restricts solutions to those model transformations using EMF as their underlying metamodeling framework. While EMF is a de facto metamodeling framework, there are model transformation tools that do not provide in-built support for it. Any generalization of results must consider these restrictions into account.

In the following section, the mechanisms used to implement incremental execution of model transformations in VIATRA and ATL are discussed in more detail.

6 Related Work

In this section, approaches for incremental execution of model transformation leading to the tools used in the evaluation section, ATL and VIATRA, are discussed first. Then approaches for bidirectional model transformation which also provide support for incremental evaluation are discussed at the end.

6.1 Reactive Model Transformation

Reactive model transformation [41,5] enables the propagation of model changes from source models to target models on demand. State-of-the-art tool support relies on notification mechanisms, enabling live detection of source model changes either for immediate processing, as in VIATRA [5] and in ATOL [16], or for deferred processing, as in ReactiveATL [41]. At present, approaches providing support for immediate processing are based on incremental evaluation of model queries whereas those using deferred processing are not. Our proposal also focuses on a coarse incremental execution of MT rules, where queries are not evaluated incrementally, requiring a small memory footprint while keeping an acceptable performance when compared to fine-grained evaluation approaches, as shown in §5.

In reactive model transformation, source model change 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 changes, as in the proposed model change propagation procedure, completely decouples detection of model changes from the transformation engine, freeing model change 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 a change notification, information about the intent of the overall model change, i.e., the contextual information relating different atomic changes, is lost. This problem is avoided using documented model changes, which may be serialized for implementing distributed systems, enabling their processing, e.g., aggregating composite changes like the *move operation*, and optimization, e.g., reduction of atomic operations that are cancelled when composed. We refer the reader to [19] for an additional discussion of change-based approaches against state-based approaches.

In the following, we discuss the body of research underneath the tools used in the experimentation.

6.1.1 ATL

ATL's incremental execution mode [35] enables the incremental forward propagation of model changes to target models. However, some features of ATL, notably helpers, rules with multiple source elements and lazy rules are not supported. In our approach, helpers and the MT steps, including those corresponding both to lazy rules and to unique lazy rules, that depend on their evaluation are re-executed incrementally.

In [35], model changes are not explicitly represented, as in our approach, and they are applied directly in the model and notified to EMF adapters (observers of change events). Moreover, only atomic model changes were considered (element creation, element deletion, property change), whereas we also consider composite model changes and optimizations via the EMF Change Model.

Furthermore, forward change propagation is *live* in [35], regarding both *models* and *model changes*. In their

approach, *live* model changes are notified when the change is applied in a source model in an immediate, *synchronous* way. This requires the source models to be linked to the ATL engine, restricting the physical location of the source models. In our approach, model changes are given either as EMF Change Model instances, which may be serialized, and are applied *asynchronously*, or using a change notification API, which allows changes to be applied *synchronously*.

The initial evaluation of an ATL transformation in incremental mode collects dependency information, which properties (either attributes or references) of which objects, for each OCL expression that is evaluated. Once a model change is notified, either for a filter condition or for a binding expression, dependency information is used to re-evaluate the affected OCL expression, as a whole. Their approach is not optimal−as described by the authors−requiring the re-evaluation of OCL expressions even when changes in property values do not affect the resulting value of the OCL expression. With lazy transformations [47], model changes are propagated on demand when objects are accessed in the target model.

Reactive ATL [41] builds on the expression injection mechanism of [35] to detect which parts of an ATL transformation need to be executed and on lazy evaluation of [47] to defer computation. This approach suffers from the same limitations of [35] as explained above. In ReactiveATL, model changes are applied in the source model and injected in the transformation engine but only applied when requested from the target model, which is read-only (i.e., only the reactive engine can update target models−source model changes can be propagated to target models but target models cannot be updated themselves directly). In our approach, feature calls are tracked and model changes are analyzed to detect which matched object features have been updated. Thus, changes are detected at the binding level and MT steps are re-executed while preserving the target context that is not affected by the channge.

A different line of research on ATL addresses the limitation of degree of granularity in model change propagation, when evaluating queries, by means of active operations [3]. These rely on the notion of box to designate mutable values (either collections or singleton values, which can be nullable) that can be updated. A box acts as a notifier that alerts observers when the value is modified. An active operation relates input boxes to output boxes modelling how model changes are propagated along the evaluation of the operation. Such operations facilitate fine-grained change propagation and preservation of collection ordering when an OCL query involving collection operations is re-evaluated incrementally in the presence of some model changes [33]. These properties have been recently brought to ATL [16], allowing the incremental execution of model transformations by means of a compiler from the declarative excerpt of ATL to Java using active operations.

6.1.2 VIATRA

In live model transformations [42], model changes are applied to a model and detected by the incremental pattern matcher that keeps the execution context of trigger rules in memory (match set for the pattern and shared variables). Although the different types of model changes are classified, the notification of model changes from the moment they are received to the point when they are used in the pattern matcher is not specified.

Change-driven MT rules [6], manually defined by a user, model the valid changes that can be performed to a source model, explicitly representing how those changes are propagated to the target model. MT rules are augmented with guards that, when evaluated in the context of the changes that are applied to a source model, trigger the corresponding rule if it is an appropriate reaction for a given change. In our approach, however, model changes are given at the instance level and there is no need for a user to define the synchronization of model changes by using change MT rules. All possible changes that are valid within the EMF framework, i.e., those model changes that are instances of the EMF Change Model, can be propagated using the proposed generic synchronization mechanism. That is, a model transformation definition need not be aware of the type of changes that can be performed.

Complex-event processing was integrated with change-driven model transformation in [18], combining serialization and exchange of model changes with live model transformations in VIATRA. Our approach is also based on live model transformations but is not reactive. Our offline model changes describe self-contained model changes, which may be atomic or composite, and can be serialized and exchanged. As it has been empirically shown, with the VIATRA CPS benchmark in §5, the implementation of the proposed mechanism for propagating model changes scales well and it is a feasible candidate for processing model changes that occur at a high rate, in real time.

6.1.3 Other approaches

Tefkat [26] is a logic-based transformation language with data-driven evaluation, where models are encoded in a fact database. In Tefkat, the evaluation of declarative

rule-based transformations is driven by a search for solutions to a Prolog-like goal (query) by relying on SLD resolution (Selective Linear Definite clause resolution). After the first transformation, the execution environment is stored as a SLD tree, where the initial goal is associated with the root node. Every time a goal in the tree unifies with a rule, a child is created with the target goal created by SLD resolution, partially solving the previous goal. When a goal is resolved, Tefkat stores the generated SLD tree augmented with dependencies in order to support incrementality. In the SLD tree, each branch represents a rule application and all possible resolution paths are represented. Tefkat augments each fact with tags linking facts to the parts of the tree where they are used, corresponding to our notion of *dependency*. Source changes are represented as changes in facts, and propagated incrementally: additions are used to create new branches in the tree whereas deletions are used to prune branches.

A strong point of Tefkat is that SLD resolution is complete (all solutions can be deduced) and sound (no wrong solutions are produced).

Partial evaluation techniques are applied in order to execute QVT-OM (Operational Mappings), which are used to specify unidirectional model transformation, in an incremental way in QVTMix [43]. This approach requires human intervention to anticipate where changes can occur on source models using a tagging mechanism for declaring variable parts. QVTMix then precomputes the intermediate results of the transformation that do not depend on variable parts of the source model, which will be computed for each given model. From a technical perspective, QVTMix manipulates the abstract syntax tree of a transformation pre-computing those parts that are static. Apart from change-driven transformations, the other incremental approaches, including ours, are more flexible as they do not rely on meta-information on the expected changes.

6.2 Incrementality in bidirectional MT

Among bidirectional model transformation approaches, Triple Graph Grammars (TGG), introduced in [44], 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 [23,24]. Efficient approaches for TGG synchronization [37,40,38] avoid analyzing the whole model by relying on dependencies which hint at the impact of a model change directly. Precedence-based approaches [37,40] 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 [37] overestimates the actual dependencies by defining them at the type level, others underestimate them relying on user feedback [40] or on special correspondences [24]. [38] decouples impact analysis of model changes 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 model change propagation mechanism proposed in the present work, both the creation of new MT steps and the revocation of existing ones cannot trigger further applications because rule matches are computed against the source model and they are unique, unless explicit dependencies have been introduced with the statement `insertDependency`. A new MT step may be found when new elements are inserted in the source model. On the other hand, when a MT step is revoked, no other rule can be applied or a conflict would have been detected when the rule was applied the first time. When explicit dependencies exist, successor MT steps are re-evaluated nonetheless.

Incremental evaluation of model analyses, involving model queries, has been studied in [31] with the .NET Modeling Framework (NMF) [27]. NMF Expressions are presented as the implementation of an incremental computation system where the incrementalization of a model analysis function can be regarded as a functor. In NFM, generic incrementality is made possible thanks to the use of NMF expressions, where a model analysis is syntactically represented as an expression tree, whose nodes correspond to instructions. A dynamic dependency graph (DDG) representing each executed instruction is built at run time. When a value in the DDG changes, dependent nodes in the expression tree are notified of the change, which is then propagated up to the root of the expression tree until the value of a subexpression does not change. NMF expressions are extensible with specific incrementalized algorithms, which take preference over the generic change propagation mechanism, in order to enable user-defined optimizations. NMF has been used for bidirectional model transformations in [30,29,28]. The scalability of NMF has been studied from a model analysis perspective but

not in a model transformation setting. A comparison with our approach for model analysis was presented in [12], where YAMTL showed better scalability.

Active operations, as supported by AOF [34] also provide support for bidirectional change propagation but this feature has not been yet surfaced through ATL.

To the best of our knowledge, among the tools discussed above, only NMF supports offline representation of model model changes, and none of them use a standardized notation for them, such as the EMF Change Model, which can be regarded as the de-facto standard for representing model changes in the EMF modeling tool ecosystem.

7 Concluding Remarks

The main contribution of this work is the design of a model change propagation mechanism for executing change-driven model transformations, which has been implemented in YAMTL. Such a model change propagation mechanism forms the foundation for developing efficient consistency maintainers between very large models. The design of this mechanism has been isolated from implementation details documenting the requirements that must be satisfied by a model transformation engine in order to adopt it. Advanced model transformation constructs have been studied for analyzing the impact of a model change in a source model, characterizing the type of model transformations that can be incrementalized. The novelty of the approach consists in the use of a standardized representation of model changes, which facilitates interoperability with EMF-compliant tools, and in the use of dependency injection mechanism, which allows the transformation engine to be aware of model changes without having to rely on an observer pattern implementation.

The VIATRA CPS benchmark has been used to justify that *(1)* the initial transformation in YAMTL incurs a small penalty with respect to its batch counterpart and that *(2)* propagation of model changes can be performed in real time for very large models. Moreover, the presented mechanism and its implementation in YAMTL is several orders of magnitude faster than the up-to-now fastest incremental solutions in the benchmark. Our experiments have been performed using computational resources found in a standard computer, differing from previous analyses. Hence, YAMTL shows satisfactory scalability in incremental execution of model transformations on very large models in a wide variety of execution environments. While the VIATRA CPS benchmark is a first step towards comparing incremental execution of model transformations, it proposes a fairly coarse-grained problem and additional cases, for example in the Tool Transformation Contest, should help in exploring the advantages and disadvantages of the incremental execution mechanisms used in each transformation engine.

Lines of future research consist in providing formal guarantees on synchronization properties and the extension of the technique presented in this work for designing and implementing low-latency systems where propagation of changes can be performed concurrently, both from source to target and from target to source.

Acknowledgements The author would like to thank Théo Le Calvar for assisting in the reuse of the ATL solution, Ábel Hegedüs and Zoltán Ujhelyi for helping with the reuse of the VIATRA CPS benchmark, and the anonymous reviewers for their constructive, useful feedback.

References

1. AUTOSAR: AUTOSAR development partnership (2019). `http://www.autosar.org`
2. Baker, P., Loh, S., Weil, F.: Model-driven engineering in a large industrial context — motorola case study. In: MoDELS, vol. 3713, pp. 476–491. LNCS (2005)
3. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.: Active operations on collections. In: MODELS, *LNCS*, vol. 6394, pp. 91–105. Springer (2010)
4. Benelallam, A., Gómez, A., Tisi, M., Cabot, J.: Distributing relational model transformation on mapreduce. Journal of Systems and Software **142**, 1–20 (2018)
5. Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., Varró, D.: Viatra 3: A reactive model transformation platform. In: ICMT, vol. 9152, pp. 101–110. LNCS (2015)
6. Bergmann, G., Ráth, I., Varró, G., Varró, D.: Change-driven model transformations - change (in) the rule to rule the change. Software and System Modeling **11**(3), 431–461 (2012)
7. Biermann, E., Ermel, C., Taentzer, G.: Precise Semantics of EMF Model Transformations by Graph Transformation. In: MODELS, pp. 53–67. LNCS 5301 (2008)
8. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. Software and System Modeling **11**(2), 227–250 (2012)
9. Boronat, A.: Expressive and efficient model transformation with an internal dsl of xtend. In: Proceedings of the 21th ACM/IEEE International Conference on MoDELS, pp. 78–88. ACM (2018)
10. Boronat, A.: YAMTL evaluation repository with the incremental component of the VIATRA CPS benchmark (2018). `https://github.com/YAMTL/viatra-cps-incr-benchmark`
11. Boronat, A.: YAMTL incremental M2M transformation for the VIATRA CPS benchmark (2018). `https://github.com/YAMTL/viatra-cps-incr-benchmark/tree/master/m2m.incr.cps2dep.yamtl`
12. Boronat, A.: YAMTL solution to the TTC 2018 social media case. In: A. García-Domínguez, G. Hinkel, F. Krikava (eds.) Proceedings of the 11th Transformation Tool Contest, co-located with the 2018 Software Technologies: Applications and Foundations, TTC@STAF

2018, Toulouse, France, June 29, 2018., vol. 2310, pp. 65–78. CEUR-WS.org (2018)
13. Boronat, A.: Offline delta-driven model transformation with dependency injection. In: FASE 2019, *LNCS*, vol. 11424, pp. 134–150. Springer (2019)
14. Calvar, T.L.: Atl solution for the viatra cps benchmark using atol/aof (2019). `https://github.com/TheoLeCalvar/viatra-cps-benchmark`
15. Calvar, T.L.: Viatra cps benchmark results (including atol) (2019). `https://theolecalvar.github.io/viatra-benchmark-results/`
16. Calvar, T.L., Jouault, F., Chhel, F., Clavreul, M.: Efficient ATL incremental transformations. Journal of Object Technology **18**(3), 2:1–17 (2019)
17. Daniel, G., Jouault, F., Sunyé, G., Cabot, J.: Gremlin-ATL: A scalable model transformation framework. In: ASE, pp. 462–472. IEEE Computer Society (2017)
18. Dávid, I., Ráth, I., Varró, D.: Foundations for streaming model transformations by complex event processing. Software and System Modeling **17**(1), 135–162 (2018)
19. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: The symmetric case. In: MODELS, vol. 6981, pp. 304–318. LNCS (2011)
20. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
21. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, 1st edn. Addison-Wesley Professional (2012)
22. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA (1995)
23. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: MoDELS, vol. 4199, pp. 543–557. LNCS (2006)
24. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and System Modeling **8**(1), 21–43 (2009)
25. Harmath, D., Ráth, I.: Viatra/query/faq: Performance optimization guidelines (2016). `https://wiki.eclipse.org/VIATRA/Query/FAQ#Performance_optimization_guidelines`
26. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: MoDELS, vol. 4199, pp. 321–335. LNCS (2006)
27. Hinkel, G.: Change propagation in an internal model transformation language. In: ICMT, vol. 9152, pp. 3–17. LNCS (2015)
28. Hinkel, G.: An NMF solution to the families to persons case at the TTC 2017. In: Proceedings of the 10th Transformation Tool Contest (TTC@STAF), *CEUR Workshop Proceedings*, vol. 2026, pp. 35–39 (2017)
29. Hinkel, G.: An NMF solution to the smart grid case at the TTC 2017. In: Proceedings of the 10th Transformation Tool Contest (TTC@STAF), *CEUR Workshop Proceedings*, vol. 2026, pp. 13–17. CEUR-WS.org (2017)
30. Hinkel, G., Burger, E.: Change propagation and bidirectionality in internal transformation dsls. Softw Syst Model (2017)
31. Hinkel, G., Heinrich, R., Reussner, R.: An extensible approach to implicit incremental model analyses. Software & Systems Modeling **18**(5), 3151–3187 (2019)
32. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of mde in industry. In: ICSE, pp. 471–480. ACM (2011)
33. Jouault, F., Beaudoux, O.: On the use of active operations for incremental bidirectional evaluation of OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling, *CEUR Workshop Proceedings*, vol. 1512, pp. 35–45. CEUR-WS.org (2015)
34. Jouault, F., Beaudoux, O.: Efficient ocl-based incremental transformations. In: Proceedings of the 16th International Workshop on OCL and Textual Modelling, *CEUR Workshop Proceedings*, vol. 1756, pp. 121–136. CEUR-WS.org (2016)
35. Jouault, F., Tisi, M.: Towards incremental execution of atl transformations. In: L. Tratt, M. Gogolla (eds.) ICMT, *LNCS*, vol. 6142, pp. 123–137. Springer (2010)
36. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The grand challenge of scalability for model driven engineering. In: M.R.V. Chaudron (ed.) Models in Software Engineering (MiSE). Collocated with MODELS., vol. 5421, pp. 48–53. LNCS (2008)
37. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient model synchronization with precedence triple graph grammars. In: ICGT, vol. 7562, pp. 401–415. LNCS (2012)
38. Leblebici, E., Anjorin, A., Fritsche, L., Varró, G., Schürr, A.: Leveraging incremental pattern matching techniques for model synchronisation. In: ICGT, vol. 10373, pp. 179–195. LNCS (2017)
39. OMG: Meta Object Facility (MOF) 2.5.1 Core Specification (2016). URL `http://www.omg.org/spec/MOF/`
40. Orejas, F., Pino, E.: Correctness of incremental model synchronization with triple graph grammars. In: ICMT, vol. 8568, pp. 74–90. LNCS (2014)
41. Pérez, S.M., Tisi, M., Douence, R.: Reactive model transformation with ATL. Sci. Comput. Program. **136**, 1–16 (2017)
42. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live model transformations driven by incremental pattern matching. In: ICMT, pp. 107–121 (2008)
43. Razavi, A., Kontogiannis, K.: Partial evaluation of model transformations. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 562–572 (2012)
44. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: WG, pp. 151–163. LNCS 903 (1994)
45. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Software **20**(5), 42–45 (2003)
46. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)
47. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Lazy execution of model-to-model transformations. In: J. Whittle, T. Clark, T. Kühne (eds.) MODELS, pp. 32–46. LNCS (2011)
48. VIATRA Team: Explicit traceability M2M transformation (2016). `https://github.com/viatra/viatra-docs/blob/master/cps/Explicit-traceability-M2M-transformation.adoc`
49. VIATRA Team: Query result traceability M2M transformation (2016). `https://github.com/viatra/viatra-docs/blob/master/cps/Query-result-traceability-M2M-transformation.adoc`
50. VIATRA Team: VIATRA CPS benchmark (2016). `https://github.com/viatra/viatra-cps-benchmark/wiki/Benchmark-specification`

51. VIATRA Team: VIATRA CPS benchmark (cases) (2016). https://github.com/viatra/viatra-cps-benchmark/wiki/Benchmark-specification#cases
52. VIATRA Team: VIATRA CPS benchmark (cps to deployment transformation) (2016). https://github.com/viatra/viatra-docs/blob/master/cps/CPS-to-Deployment-Transformation.adoc
53. VIATRA Team: VIATRA CPS benchmark (model generator) (2016). https://github.com/viatra/viatra-docs/blob/master/cps/Model-Generator.adoc