Incremental View Model Synchronization Using Partial Models

Kristóf Marussy[a,b], Oszkár Semeráth[a,b], Dániel Varró[a,b,c]
[a]Budapest University of Technology and Economics,
Department of Measurement and Information Systems, Hungary
[b]MTA-BME Lendület Cyber-Physical Systems Research Group, Hungary
[c]McGill University, School of Electrical and Computer Engineering, Canada

ABSTRACT

View models are abstractions of a set of source models derived by unidirectional model transformations. In this paper, we propose a view model transformation approach which provides a fully compositional transformation language built on an existing graph query language to declaratively compose source and target patterns into transformation rules. Moreover, we provide a reactive, incremental, validating and inconsistency-tolerant transformation engine that reacts to changes of the source model and maintains an intermediate partial model by merging the results of composable view transformations followed by incremental updates of the target view. An initial scalability evaluation of an open source prototype tool built on top of an open source model transformation tool is carried out in the context of the open Train Benchmark framework.

ACM Reference Format:
Kristóf Marussy, Oszkár Semeráth, Dániel Varró. 2018. Incremental View Model Synchronization Using Partial Models. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239412

1 INTRODUCTION

Complex industrial toolchains used for designing cyber-physical systems frequently depend on various models on different levels of abstraction. Abstract models [13] can be derived and synchronized by view model transformations upon changes of one or more underlying source models.

View synchronization challenges are addressed by using either *general purpose model transformation tools* (e.g. ATL [33, 43], ETL [37], Henshin [6], Viatra [56]), *bidirectional model synchronization* (e.g. various TGG tools [24, 27, 40, 48] and QVTr [46]), or dedicated *view transformation* techniques (e.g. View TGGs [5, 32], Active Operations [7], Viatra Views [18], QuEST [23]).

To tackle complex scenarios, view model transformations are desirably *defined in a compositional way* to reuse existing transformations without further changes. While sequential composition (chaining) is widely supported, existing tools need to impose major restrictions in case of parallel composition (merging) of target views. An ideal (forward only) *view transformation engine* is *reactive* (i.e. reacts to source model changes), target *incremental* (i.e. updates only affected target elements), *consistent* (i.e. continuously maintains a transformation relation between source and target models) and *validating* (i.e. the target model is a valid, materializable instance of the target language).

Currently, there is a significant trade-off in existing tools between the expressiveness and compositionality of the view transformation language, and the level of support for desirable features of the view transformation engine. On the one hand, fully reactive behavior is a challenge in itself supported by only few tools (e.g. [7, 43, 56]), while incrementality, consistency and validity is provided at the same time for very restrictive transformation languages. Practical model transformation engines frequently fail to restore consistency between models [53].

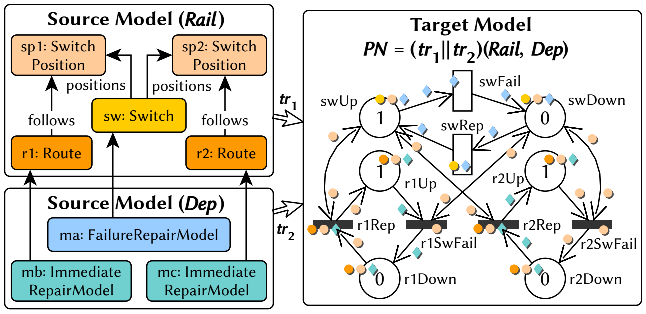
Our main contribution in the paper is a unidirectional view transformation approach with a (1) a *fully compositional view transformation language*, and (2) a *reactive, incremental, validating and inconsistency-tolerant transformation engine*. The view transformation language explicitly reuses the Viatra Query Language [55] to declaratively capture relevant source and view patterns by following the principles of ramification [39]. Moreover, *inconsistency-tolerant partial models* (a generalization of partial models of [22, 57]) provide the conceptual core of the transformation engine.

The transformation engine reacts to aggregated changes of the source model observed in the result set of graph queries (hence *reactive*), then it builds and maintains a partial model as a knowledge base with traceability links. Once the partial view model becomes a valid instance of the target metamodel (i.e. relevant aggregated changes are observed in the knowledge base, and structural constraints are respected), the target view model is *incrementally updated by providing a corresponding change* (e.g. model delta, notification or API call). Our engine is *inconsistency tolerant* in the sense that inconsistencies are semantically persisted in the internal knowledge base. This allows to keep a large fragment of the source and view models in sync in case of inconsistent source changes and provides hippocratic behavior (i.e. avoids the unnecessary deletion and recreation of elements).

The transformation engine is implemented as a prototype tool [1] and integrated into the open source Viatra transformation framework [9]. Moreover, we carry out an initial scalability evaluation by adapting an existing view model transformation from an industrial research project (aiming to carry out dependability evaluation of automotive designs) to the open Train Benchmark [54]. Artifacts related to this paper are also available from [2].

MODELS '18, October 14–19, 2018, Copenhagen, Denmark

ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239412



Trace ($tr_1: Rail \to PN$): sw ↦ {swUp, swDown, swRep, swFail}, r1 ↦ {r1Up, r1Down, r1Rep}, r2 ↦ {r2Up, r2Down, r2Rep}, sp1 ↦ {swUp, swDown, r1Up, r1Down, r1Rep, r1SwFail}, sp2 ↦ {swUp, swDown, r2Up, r2Down, r2Rep, r2SwFail}
Trace ($tr_2: Dep \to PN$): ma ↦ {swUp, swDown, swRep, swFail}, mb ↦ {r1Up, r1Down, r1Rep}, mc ↦ {r2Up, r2Down, r2Rep}

Markers denote the trace of the nodes; multiple markers mean objects merged from both view transformations.

Figure 1: Source and target models with traceability links.

2 A OVERVIEW OF COMPOSITIONAL VIEW TRANSFORMATIONS

A view transformation $Trg = tr(Src_1, \ldots, Src_k)$ aims to derive a target view model *Trg* as an abstraction of a set of source models $Src_1, \ldots, Src_k$. A *tr* is a mapping from source(s) to target models typically with loss of information.

Moreover, in a typical view synchronization scenario, each target change is causally dependent on some (aggregate) change of the source model (e.g. a model delta or notification upon model update). This causal dependence can be captured by a *match* of a *view transformation rule* in the source model which triggers the simultaneous creation of respective target elements together with some traceability links between source and target elements.

A motivating example. The running example of the paper is adapted from an industrial project where formal dependability analysis of automotive models were carried out by composing two view transformations: (1) $PN = tr_1(Aut)$ maps automotive component models *Aut* to stochastic Petri nets *PN* [3], and (2) $PN = tr_2(Dep)$ is a reusable mapping [12, 42] from a domain-independent dependability model *Dep* to stochastic Petri nets. The target Petri net model is defined as the (parallel) composition of the two transformations $PN = tr_1 \parallel tr_2(Aut, Dep)$ calculated over the two input models.

Due to IP restrictions of automotive models, we present the challenge using a public model of railway networks developed as part of the Train Benchmark [54], a cross-technology macrobenchmark of graph-based model query tools. A sample source and target model are shown along with the traceability links in Fig. 1, while some transformation rules will be illustrated later in Fig. 7.

2.1 Levels of compositional definitions

To categorize the levels of compositionality in view transformations, let us assume the existence of two view transformations, tr_1 and tr_2 and a single source model *Src* to simplify the discussion. Transformations tr_1 and tr_2 can be composed in different ways.

In many practical scenarios [5, 29], chaining of view transformations is necessitated, which is a *sequential composition* of transformations $Trg = tr_2 \circ tr_1(Src) = tr_2(tr_1(Src))$ where tr_2 takes the output of tr_1 as its source model, and the target model of this transformation chain is the subsequent result of tr_2. The definition of sequential composition is supported in several tools [29, 44].

Given two existing view transformations $Trg_1 = tr_1(Src)$ and $Trg_2 = tr_2(Src)$, another relevant aspect is *parallel composition* $Trg = tr_1 \parallel tr_2(Src) = tr_1(Src) \oplus tr_2(Src)$ where the target model is derived by merging (or gluing) the results of transformations tr_1 and tr_2 both applied on the same source model *Src*. If the two transformations are independent, the target model is the union of the individual transformations, otherwise the aggregated result can be computed e.g. by category-theoretical foundations [16, 19, 20]. Below, we briefly categorize the *major assumptions for parallel composition* $tr_1 \parallel tr_2$ used in existing transformation tools.

(1) In the *independent* case, each target object is fully defined by a single rule in one transformation, thus a union of target elements can be taken without merge, i.e. $Trg = tr_1(Src) \cup tr_2(Src)$. Otherwise, a new transformation tr_3 needs to be written manually.

(2) In the *serializable* case, the parallel composition is turned into a sequential composition where one transformation (e.g. tr_1) is taken as-is (called *primary*) while the other transformation tr_2 (called *secondary*) needs to be manually changed to tr'_2, i.e. $tr_1 \parallel tr_2(Src) = tr'_2(tr_1(Src), Src))$ or $tr'_1(tr_2(Src), Src))$.

(a) Certain transformation languages (e.g. ATL [33]) *restrict* primary rules, i.e. at most one serialization $tr'_2(tr_1(Src), Src))$ or $tr'_1(tr_2(Src), Src))$ can exist. In ATL, outgoing references of an object can only be defined in a primary rule (to ensure multiplicity constraints in the target language), thus a static check will prevent serializing the transformations in the wrong way.

(b) Other serializable view transformation approaches [9, 29] are *unrestricted* to allow both serializations $tr'_2(tr_1(Src), Src))$ and $tr'_1(tr_2(Src), Src))$, but one of the transformations still needs to be adapted to take the output of the other (instead of *Src*).

(3) *Fully compositional* view transformation approaches allow to compose tr_1 and tr_2 as $tr_1 \parallel tr_2(Src) = tr_1(Src) \cup_? tr_2(Src)$ without changing the transformations by using some model merge operator $\cup_?$ to weave the target models of individual transformations into a joint result.

(a) In *ID-based* $tr_1 \parallel tr_2(Src) = tr_1(Src) \cup_{\mathrm{ID}} tr_2(Src)$ composition, rules assign the same ID to objects that need to be merged in the final target model. The ID attribute can be selected from the metamodel intrusively [46] or added by *augmentation* [39].

(b) *Relation-based* $tr_1 \parallel_g tr_2(Src) = tr_1(Src) \cup_{g(Src)} tr_2(Src)$ composition can mark unrelated objects constructed separately by transformations tr_1 or tr_2 to be merged. The merge operation is a parameter, i.e. it can be specified as a categorial colimit with a suitable *reference* or *connection* model [19, 21, 47], by *direct mappings* [16], or by *graph bisimulation* [15].

2.2 Properties of view transformation engines

A view transformation engine $Out^{(i)} = exec(tr, In^{(i)})$ repeatedly executes a transformation *tr* at a given logical time point *i* on an input $In^{(i)}$ (which can be the source model $Src^{(i)}$ or a delta $\Delta^{(i)}_{Src}$) to derive an output $Out^{(i)}$ (the target model $Trg^{(i)}$ or a delta $\Delta^{(i)}_{Trg}$) while

Table 1: Comparison of view model transformation techniques.

	Parallel composition	Engine properties React.	Incr.	Cons.	Valid.	Comment
Our approach	relation-based	R	●	IT	●	
Reactive ATL [34, 43]	independent	R	●	C	●	Restrictions in source and trace language
TGG Virtualized View [32]	independent	R	●	C	○	Only single node or reference in rule target side, limited NAC support
TGG Materialized View [5]	independent	R	●	C	●	Only single node or reference in rule target side, limited NAC support
Viatra Views [18]	independent	R	●	C	●	Only single node or reference in rule target side
QueST [23]	independent	D	●	C	●	Only single node or reference in rule target side
Incremental QVTr [50]	restricted serializable	R	●	?	?	Cons., valid. difficult to determine due to QVTr semantic issues [26, 52]
EMF Views [14]	independent	NR	○	C	○	Infers target metamodel
Active Operations [7]	restricted serializable	R	●	C	○	Transformation also defines target metamodel
Hearnden et al. [28]	restricted serializable	D	●	IT	○	Produces deduction tree tree as target model
ATL (no imperative code) [33]	restricted serializable	NR	○	C	●	Restrictions on outgoing references in non-primary rules
ATL (+imperative code) [33]	serializable	NR	○	NC	●	No consistency checking for imperative actions
eMoflon TGG [40, 41]	restricted serializable	D	●	C/A	●	Restrictions for negative application conditions (NAC)
Viatra [29, 56]	serializable	R	●	NC	●	No consistency checking for imperative actions
QVTr [46] M2M [58]	ID-based	NR	○	?	?	Cons., valid. difficult to determine due to QVTr semantic issues [26, 52]
Epsilon ETL [37] + EML [38]	relation-based	NR	○	NC	●	Merge operators for composition in separate language
JTL [17]	serializable	D	○	C/A	●	No answer if the target cannot satisfy constraints
RAMification [39, 45]	ID-based	NR	○	C	○	Metamodel constraints are *relaxed*
GRoundTram, ATLGT [30, 31]	relation-based	D	●	C	○	Graph bisimulation based data model, non-EMF
BiGUL [36]	relation-based	NR	○	C/A	●	PutBack-based functional programming, may be adapted to EMF [4]

Legend: R reactive, D delta-based, NR non-reactive (batch); C consistent, C/A consistent or aborts, NC non-consistent, IT inconsistency tolerant; ● yes, ○ no

(a) maintaining the consistency relation $Trg = tr(Src)$ between the source and target models and (b) keeping the target model a valid instance of the target language ($Trg \models MM_T$).

(1) A *batch* engine takes the entire source model at any step: $Out^{(i)} = exec(tr, Src^{(i)})$. A *delta-based* engine takes a model change as input, but it executes on-demand: $Out^{(i)} = exec(tr, Src^{(i-1)}, \Delta^{(i)}_{Src})$. A *reactive* engine executes in response to source model changes [11, 43] by receiving deltas as model notifications: $Out^{(i)} = exec(tr, \Delta^{(i)}_{Src})$. Delta-based and reactive engines load the source model as a large delta upon initialization.

(2) An *incremental* engine updates only those target elements which are affected by a specific source model change, that is $\Delta^{(i)}_{Trg} = Out^{(i)} = exec(tr, In^{(i)})$, thus the new target model is obtained by applying this delta: $Trg^{(i)} = Trg^{(i-1)} + \Delta^{(i)}_{Trg}$. A *non-incremental* engine derives the new target model from scratch:
$Trg^{(i)} = Out^{(i)} = exec(tr, In^{(i-1)})$.

(3) A *consistent* engine continuously enforces consistency (correctness) constraints between source and target elements: if $Out^{(i)} = exec(tr, In^{(i)})$ then $Trg^{(i)} = tr(Src^{(i)})$. A *non-consistent* engine does not guarantee these constraints if the transformation rules are conflicting with each other (e.g. in case of a specific source change).

(4) A *validating* engine derives the view model as a valid instance model of the target metamodel (or viewtype) where all metamodel constraints (e.g. aggregation, multiplicity) are satisfied: if $Trg^{(i)} = exec(tr, In^{(i)})$ then $Trg^{(i)} \models MM_T$. Checking these structural constraints of the target metamodel is out of scope for a *non-validating* engine, thus $Trg^{(i)} \not\models MM_T$. A validating engine can be used for both materialized and virtualized viewtypes [13].

Fully compositional view transformations need to face the conceptual challenge that while enforcing the consistency between the source and target models, one may easily violate the structural constraints imposed by a metamodeling framework like EMF [51].

2.3 Related work

A desired view transformation approach offers a fully compositional *language* and a reactive, incremental, consistent and validating *engine* but no transformation tools currently exist which support all these properties. Our overview of (the significant amount) of related work primarily focuses on existing transformation tools by categorizing the level of support (1) for parallel compositionality in transformation languages, and (2) for desirable transformation engine properties in Table 1. For space considerations, we highlight only the typical restrictions found in the context of multiple tools.

Imperative transformation approaches reactively build the target model (like imperative ATL, Viatra or ETL) but they do not provide consistency guarantees, i.e. certain target models may not be consistent with a source model. Unfortunately, such inconsistencies can propagate to future stages of the transformation.

Bidirectional model synchronization tools (like different TGG implementations or JTL) either guarantee consistency or they abort the execution of the transformation. These tools offer a certain level of serializability, but they are not fully compositional.

Dedicated view transformation approaches (like TGG Views, Viatra Views, Reactive ATL) use a restricted transformation language (wrt. their regular transformation counterpart) to provide desirable engine behavior. However, parallel composition of different transformations is very limited.

Most existing fully compositional approaches (like QVTr, ramification, Epsilon with combined ETL and EML languages) are neither reactive nor incremental and only EML is validating. Only GRoundTram and ATLGT support target incrementality and delta-based source incrementality, but over a custom (non-EMF compliant) model representation. The closest approaches to ours are [17, 28] as they build a knowledge base based on first order logic and target models are derived by logical inference, but these approaches are not fully compositional.

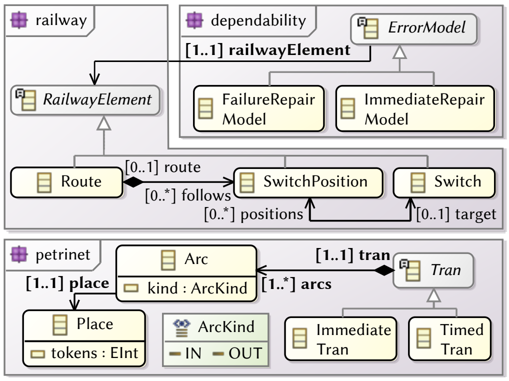
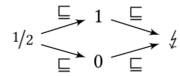


Figure 2: Two source and one target metamodels.



$$(X \sqsubseteq Y) \coloneqq (X = 1/2) \lor (X = Y) \lor (Y = ↯)$$

(a) Lattice of logical values.

X	$\neg X$
0	1
1	0
1/2	1/2
↯	↯

$\lor$	0	1	1/2	↯
0	0	1	1/2	↯
1	1	1	1	1
1/2	1/2	1	1/2	1
↯	↯	1	1	↯

$\land$	0	1	1/2	↯
0	0	0	0	0
1	0	1	1/2	↯
1/2	0	1/2	1/2	0
↯	0	↯	0	↯

$\oplus$	0	1	1/2	↯
0	0	↯	0	↯
1	↯	1	1	↯
1/2	0	1	1/2	↯
↯	↯	↯	↯	↯

(b) Logic connectives ($\neg$, $\lor$, $\land$) and information merge ($\oplus$).

Table 2: Belnap-Dunn 4-valued logic.

Our work provides a view transformation approach with (1) a *fully compositional* transformation language built on top of an existing declarative query language, and (2) a transformation engine which is *reactive*, *incremental*, *validating* and *inconsistency-tolerant* at the same time. The inconsistency-tolerant engine is a relaxed version of a consistent engine where $Trg^{(j)} \neq tr(Src^{(j)})$ may happen after some conflicting source model changes $Out^{(j)} = exec(tr, \Delta^{(j)}_{Src})$, but all other desirable properties are preserved. Most of the target model satisfying MM_T is preserved, while inconsistencies are explicitly highlighted by the framework. Lastly, by delaying notifications to engine, *reactive* behavior can be optionally replaced with *delta-based* processing.

3 INCONSISTENCY-TOLERANT PARTIAL MODELS

Our view transformation technique builds on inconsistency-tolerant partial models which store inconsistent and unknown information in models by generalizing the merging of inconsistent and incomplete views in conceptual models [47]. This section provides theoretical foundations based on Belnap-Dunn 4-valued logic [8, 35].

3.1 Preliminaries: Foundations of metamodels

A metamodel contains the main concepts and relations of a domain, and captures the basic structure of the models. Formally, a *metamodel* defines a signature $\Sigma = \{C_1, \ldots, C_t, R_1, \ldots, R_r, \sim\}$, which is a vocabulary of unary type predicate symbols $\{C_i\}_{i=1}^{t}$ defined for each class, binary relation predicate symbols $\{R_j\}_{j=1}^{r}$ defined for each reference and attribute, and additionally, an equivalence relation $\sim$. In our running example, Fig. 2 defines two source (railway and dependability) and one target metamodels (petrinet).

Metamodeling tools impose additional *structural constraints* on instance models to enforce a basic structure. In the Eclipse Modeling Framework (EMF) [51], violating such a structural constraint would prevent the materialization (saving) of a model.

Type hierarchy. A metamodel defines a type system by *supertype* relations and *abstract* classes. For each object o, there shall be a single class C, where (i) C is non-abstract, and (ii) o is an instance of C' when C' is a supertype of C. In the petrinet metamodel in Fig. 2, an abstract Tran is either an ImmediateTran or a TimedTran.

Type compliance. The metamodel restricts the classes C_1, C_2 of objects at the ends of a reference R: $\forall o_1, o_2 \colon R(o_1, o_2) \Rightarrow C_1(o_1) \land C_2(o_2)$. E.g., the target of a tran reference has to be a Tran.

Multiplicity constraints are placed on upper bounds on the number of references adjacent to an object: $\forall o, o_1, o_2 \colon R(o, o_1) \land R(o, o_2) \Rightarrow o_1 \sim o_2$. For example, an Arc can have only one tran.

Inverse relations. Some references R and R' always occur in pairs: $\forall o_1, o_2 \colon R(o_1, o_2) \leftrightarrow R'(o_2, o_1)$. See e.g., tran and arcs.

Containment hierarchy. EMF models are arranged in a strict tree hierarchy via the containment references. EMF restricts objects not to (i) have multiple containers, and (ii) form circles via containment references. E.g., an Arc cannot be contained by multiple Trans.

Equivalence relation $\sim$ is *reflexive*: $\forall o \colon o \sim o$, *symmetric*: $\forall o_1, o_2 \colon o_1 \sim o_2 \Rightarrow o_2 \sim o_1$, and *transitive*: $\forall o_1, o_2, o_3 \colon o_1 \sim o_2 \land o_2 \sim o_3 \Rightarrow o_1 \sim o_3$. In a regular instance model, objects are different from one other, but partial models may have explicit $\sim$ relations.

3.2 Inconsistency-tolerant partial models

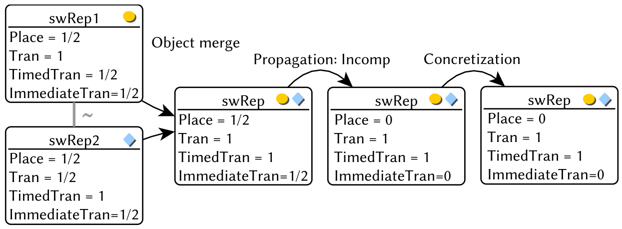
For a flexible composition of parallel view transformations, we propose *inconsistency-tolerant partial models* as a generalization of partial models [22, 57] that explicitly represents inconsistencies and uncertain parts of view models.

Belnap-Dunn logic. As a semantic basis, we use the 4-valued Belnap-Dunn logic [8] with regular *true* and *false* values (denoted by 1 and 0, respectively), an *unknown* value (1/2) to represent unspecified properties (which can be either 1 or 0), and an *inconsistent* value (↯) to represent errors where both 1 and 0 values simultaneously hold. An information ordering relation $\sqsubseteq$ is introduced (see Fig. 2a) where ↯ is larger than 1 and 0 while 1/2 is less than 1 and 0. Operation $X \oplus Y$ denotes the merge of information values by taking the maximum of two logic symbols with respect to $\sqsubseteq$. The 4-valued truth table for basic logic connectives is listed in Fig. 2b.

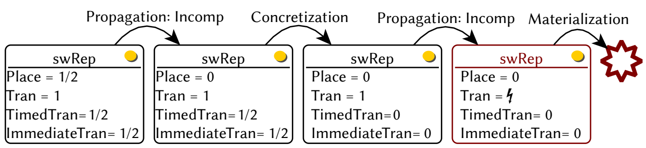
Inconsistency-tolerant partial models. A partial model $P = \langle Obj_P, \mathcal{I}_P \rangle$ is a 4-valued logic structure of Σ, where Obj_P is a *finite* set of objects, and $\mathcal{I}_P$ is a 4-valued interpretation of the relation symbols in Σ with:

- $\mathcal{I}_P(C_i) \colon Obj_P \to \{0, 1, 1/2, ↯\}$ for each C_i;
- $\mathcal{I}_P(R_i) \colon Obj_P \times Obj_P \to \{0, 1, 1/2, ↯\}$ for each R_j, and
- $\mathcal{I}_P(\sim) \colon Obj_P \times Obj_P \to \{0, 1, 1/2, ↯\}$ for equivalence relation.

A partial model P is *concrete*, if (i) there are only 0 and 1 values in $\mathcal{I}_P$, and (ii) $o_1 \sim o_2$ iff o_1 and o_2 are the same element of Obj_P. A concrete partial model can be interpreted as an instance model M



(a) Partial model refinement by merge functions



(b) Inconsistency found during partial model refinement

Figure 3: Sample chain of partial models

$$
\begin{gathered}
[\![R(o_1,\dots,o_n)]\!]^P_Z := \mathcal{I}_P(R_i)(Z(o_i),\dots,Z(o_j)) \\
[\![o_1 \sim o_2]\!]^P_Z := \mathcal{I}_P(\sim)(Z(o_1),Z(o_2)) \qquad [\![\neg\varphi]\!]^P_Z := \neg[\![\varphi]\!]^P_Z \\
[\![\varphi_1 \wedge \varphi_2]\!]^P_Z := ([\![\varphi_1]\!]^P_Z \wedge [\![\varphi_2]\!]^P_Z) \\
[\![\varphi_1 \vee \varphi_2]\!]^P_Z := ([\![\varphi_1]\!]^P_Z \vee [\![\varphi_2]\!]^P_Z) \\
[\![\forall o : \varphi(o)]\!]^P_Z := \textstyle\bigwedge_{x\in Obj_P} [\![\varphi(o)]\!]^P_{Z,o\mapsto x} \\
[\![\exists o : \varphi(o)]\!]^P_Z := \textstyle\bigvee_{x\in Obj_P} [\![\varphi(o)]\!]^P_{Z,o\mapsto x}
\end{gathered}
$$

Figure 4: Semantics of 4-valued predicates

(i.e. a labeled graph). If all structural constraints are also respected ($M \models MM$) then M can be *materialized* into a regular EMF model.

Example 3.1. A sequence of partial models corresponding to Tran swRep of Fig. 1 is listed in Fig. 3b. For example, the left-most partial model states that element swRep is a Tran (1), and it is unknown (1/2) if it is also Place, a TimedTran or ImmediateTran.

3.3 Graph predicates

A *graph predicate* $\varphi(v_1,\dots,v_n)$ is a first-order logic (FOL) predicate over an infinite set of variables $(o_1,o_2,\dots)$, the relation symbols of Σ (C_i, R_j, $\sim$), standard logic connectives ($\neg,\wedge,\vee$), and quantifiers ($\exists,\forall$). The *semantics of a graph predicate* $[\![\varphi(v_1,\dots,v_n)]\!]^P_Z$ can be evaluated on a partial model P with variable binding $Z\colon \{v_1,\dots,v_n\} \to Obj_P$ to yield a logic value 0, 1, 1/2 or ↯ as defined in Fig. 4. For concrete (2-valued) models this semantics is equivalent to standard FOL. A variable binding Z of $\varphi(v_1,\dots,v_n)$ is called a *match*, if $1 \sqsubseteq [\![\varphi(v_1,\dots,v_n)]\!]^P_Z$, i.e., there is a real match or an inconsistency.

Following [57], the structural constraints of a metamodel MM are captured by a *malformedness predicate* φ_{MM} *where a match of the predicate highlights elements that violate the constraint.* If P is an instance model M, and there is no match of predicate φ_{MM} ($1 \not\sqsubseteq [\![\varphi_{MM}]\!]^P_Z$ for all variable bindings Z, i.e. it can be 0 or 1/2), then M is a valid instance model: $M \models MM$, thus it can be materialized.

Example 3.2. A sample graph predicate derived from a structural constraint of the petrinet metamodel (see Fig. 2) captures that a Tran needs to be either a TimedTran or a ImmediateTran: $\forall o : \mathrm{Tran}(o) \Rightarrow \mathrm{TimedTran}(o) \vee \mathrm{ImmediateTran}(o)$.

Type Hierarchy:

$$\textsc{SuperUp}\colon \frac{C_2(o)}{C_1(o)\!\uparrow},\ \textsc{SuperDn}\colon \frac{\neg C_1(o)}{C_2(o)\!\downarrow} \text{ if } C_1 \text{ is a supertype of } C_2,$$

$$\textsc{Join}\colon \frac{C_1(o)\wedge\dots\wedge C_n(o)\wedge\neg C'_1(o)\wedge\dots\wedge\neg C'_m(o)}{C^*(o)\!\uparrow}$$

if among types that are not subtypes of any C'_j, C^* is the unique most generic non-abstract common subtype of all C_i ($n \ge 1$, $m \ge 0$, and C^* may be equal to one of $C_1,\dots,C_n$),

$$\textsc{Incomp}\colon \frac{C_1(o)\wedge\dots\wedge C_n(o)\wedge\neg C'_1(o)\wedge\dots\wedge\neg C'_m(o)}{C^*(o)\!\downarrow}$$

if among types that are not subtypes of any C'_i, $C_1,\dots,C_n$ and C^* have no common non-abstract subtype (not even an improper subtype, i.e. one of C_i or C^*),

Relations:

$$\textsc{RelUp}\colon \frac{R(o_1,o_2)}{C_1(o_1)\!\uparrow\ C_2(o_2)\!\uparrow},\ \textsc{RelDn}\colon \frac{\neg C_1(o_1)\vee\neg C_2(o_2)}{R(o_1,o_2)\!\downarrow} \text{ if } C_1 \text{ and } C_2 \text{ are the source and target of } R,$$

$$\textsc{Mult}\colon \frac{R(o,o_1)\wedge\neg(o_1\sim o_2)}{R(o,o_2)\!\downarrow} \text{ if } R \text{ has upper multiplicity 1,}$$

$$\textsc{ContMult}\colon \frac{R_1(o_1,o)\wedge\neg(o_1\sim o_2)}{R_2(o_2,o)\!\downarrow} \text{ if } R_1, R_2 \text{ are containment,}$$

$$\textsc{ContLoop}\colon \frac{R_1(o_1,o_2)\wedge\dots\wedge R_{n-1}(o_{n-1},o_n)}{R_n(o_n,o_1)\!\downarrow} \text{ if all } R_i\ (1\le i\le n) \text{ are containment}$$

Equivalence:

$$\sim\!\textsc{Symm}\colon \frac{o_1\sim o_2}{o_2\sim o_1\!\uparrow},\ \sim\!\textsc{Tran}\colon \frac{o_1\sim o_2\wedge o_2\sim o_3}{o_1\sim o_3\!\uparrow},\ \sim\!\textsc{Refl}\colon \frac{1}{o_1\sim o_1\!\uparrow}$$

Figure 5: Propagation rules for EMF structural constraints.

3.4 Merge functions for partial models

In order to unify the semantic treatment of partial model concretization, view model merge and rule application, we define a *merge function* $m\colon Obj_P \to Obj_Q$ between objects of partial models P and Q. Function m is defined to ensure a *refinement relation* $\sqsubseteq\colon P\times Q$ between partial models P and Q [57], which respects information ordering as stated by the following conditions for all $o_1,o_2 \in Obj_P$:

- $\mathcal{I}_P(C_i)(o_1) \sqsubseteq \mathcal{I}_Q(C_i)(m(o_1))$ for all $C_i \in \Sigma$,
- $\mathcal{I}_P(R_j)(o_1,o_2) \sqsubseteq \mathcal{I}_Q(R_j)(m(o_1),m(o_2))$ for all $R_j \in \Sigma$,
- $\mathcal{I}_P(\sim)(o_1,o_2) \sqsubseteq \mathcal{I}_Q(\sim)(m(o_1),m(o_2))$.

Partial model refinement is information preserving in the sense that all true (resp. false) predicates remain true (resp. false) in any refinement of a partial model (as proved in [57]).

Example 3.3. Before the formal definitions, merge functions are informally illustrated along two different sequences in Fig. 3. The first sequence (Fig. 3a) starts from a partial model where two objects are marked as equivalent (~), thus (a) an *object merge* function can be applied, which merges information from input objects: swRep becomes both a Tran (due to the top object) and an TimedTran (due to the bottom object). (b) Then an Incomp propagation rule will refine the model in accordance with the type hierarchy since a TimedTran object cannot be a Place or an ImmediateTran. Finally, (c) the concretization step has no further effect, and we obtain an instance model on the right.

The second sequence (Fig. 3b) first (a) applies an Incomp propagation rule to ensure that a Tran is no longer a Place. Then (b) concretization is executed to set 1/2 values to 0 for TimedTran and ImmediateTran. Now (c) another Incomp propagation rule finds that an abstract Tran needs to be refined into either a TimedTran or an ImmediateTran thus it changes their 0 value to the inconsistent value $\lightning$ (both 0 and 1 at the same time). (d) If a materialization step is now executed then the inconsistent object is removed.

Below, we define the different merge functions for partial models:

(1) *Propagation rules* handle type inferencing over 4-valued logic. A propagation rule (detailed in Figure 5) takes the form $prop = \frac{\varphi(v_1,\ldots,v_n)}{\alpha_i\uparrow \cdots \alpha_k\downarrow}$, where φ is a precondition, and $\alpha_i\uparrow$ (known to be true) and $\alpha_k\downarrow$ (known to be false) are atomic *actions* over the free variables of φ. For every match Z of φ (with $1 \sqsubseteq [\![\varphi(v_1,\ldots,v_k)]\!]^P_Z$), we obtain a merge function $prop_Z$ from P to a new partial model Q with $Obj_Q = Obj_P$, $prop_Z(o) = o$, and $\mathcal{I}_Q$ is obtained from $\mathcal{I}_P$:

$$[\![\alpha]\!]^Q_Z = \begin{cases} [\![\alpha]\!]^P_Z \oplus 1, & \text{if } \alpha\uparrow \text{ is an action of } prop,\\ [\![\alpha]\!]^P_Z \oplus 0, & \text{if } \alpha\downarrow \text{ is an action of } prop,\\ [\![\alpha]\!]^P_Z, & \text{otherwise.} \end{cases}$$

The function $prop_Z$ is a merge function, because both $A \oplus 1$ and $A \oplus 0$ respect the refinement $\sqsubseteq$ of logical values.

(2) *Object merge om*: $Obj_P \to Obj_Q$ merges two distinct objects $o_1, o_2 \in Obj_P$ into a joint object $o_{1,2} \in Obj_Q$ if $1 \sqsubseteq \mathcal{I}_P(\sim)(o_1,o_2)$ and leaves the object unchanged otherwise. Formally, $Obj_Q = Obj_P \setminus \{o_1,o_2\} \cup \{o_{1,2}\}$, and $\mathcal{I}_Q$ is obtained by combining the contents of the two elements of $\mathcal{I}_P$ with $\oplus$ i.e.

$$\mathcal{I}_Q(\mathsf{C}_i)(o) = \begin{cases} \mathcal{I}_P(\mathsf{C}_i)(o_1) \oplus \mathcal{I}_P(\mathsf{C}_i)(o_2), & \text{if } o = o_{1,2},\\ \mathcal{I}_P(\mathsf{C}_i)(o) & \text{otherwise.} \end{cases}$$

The function om_{o_1,o_k} is a merge function, because $\oplus$ respects the refinement $\sqsubseteq$ of logical values.

(3) *Concretization* is a merge function $conc\colon Obj_P \to Obj_Q$ that refines a partial model P to a concretized (partial) model Q by setting all 1/2 values to 0. Partial model Q can only contain 0, 1 and $\lightning$ values. If Q has no $\lightning$ values then it is a concrete instance model. Concretization preserves partial model refinement, i.e., $P \sqsubseteq Q$.

A *materialization* function $mat\colon Obj_P \to Obj_Q$ takes a *concretized* partial model P and removes all inconsistent elements by setting all $\lightning$ values to 0 to obtain an instance model Q. In general, materialization is not a merge function (as $P \not\sqsubseteq Q$), since information preservation is violated when rewriting predicates ($\lightning \mapsto 0$). However, if a concretized (partial) model is free from $\lightning$ values, then materialization is a trivial merge function due to being idempotent. Materialization is non-invasive, as it keeps all valid model elements in a concretized model, but removes inconsistent model elements to make the instance model EMF-compliant (e.g., serializable).

Correctness of merging partial models. Computations over 4-valued partial models carried out by a sequence of merge functions and finalized by concretization and materialization. Formally, if P is a partial model and $m = m_k \circ \cdots \circ m_m$ a *maximal* sequence of propagations and object merges applied to P, then $Q = (mat \circ conc \circ m)(P)$ is an instance model and $1 \not\sqsubseteq [\![\varphi_{MM}]\!]^Q$, where φ_{MM} is the disjunction of error patterns corresponding to enforced metamodel constraints from Section 3.1. Therefore the final result is always a valid instance model. Moreover, if we have $[\![\varphi_{MM}]\!]^P = 0$, then $P \sqsubseteq Q$, which means that no information is lost.

Proof sketch: As merge functions are closed over composition, m and $conc \circ m$ are merge functions. Propagation rules ensure that $[\![\varphi_{MM}]\!]^{m(P)} \in \{0, 1/2, \lightning\}$. By changing 1/2 values in $\mathcal{I}_{m(P)}$ by *conc*, we obtain $[\![\varphi_{MM}]\!]^{(conc \circ m)(P)} \in \{0, \lightning\}$, and the 1/2 values are removed from $\mathcal{I}_{(conc \circ m)(P)}$. Lastly, because enforced metamodel constraint violations φ_{MM} can be corrected by removing objects, $[\![\varphi_{MM}]\!]^Q = 0$, hence *mat* ensures that Q is an instance model.

If we initially have $[\![\varphi_{MM}]\!]^P = 0$, i.e. the partial model is surely valid, m (or any other sequence of propagations and object merges defined within this paper) does not introduce any $\lightning$ values to $\mathcal{I}_{m(P)}$. Thus *mat* is the identity function and $P \sqsubseteq Q$. Otherwise a portion of objects is removed when obtaining Q to avoid violating metamodel constraints φ_{MM}.

```
‹view› ::= ‹rule› (‹rule›)*
‹rule› ::= rule ‹pattern-dec› (=> ‹pattern-dec›)? (‹lookup›)*
‹lookup› ::= lookup ‹pattern-dec› => ‹param-list›
‹pattern-dec› ::= ‹pattern-name› ‹param-list›
‹param-list› ::= (‹variable› (, ‹variable›)*)
‹pattern-def› ::= ‹pattern-dec›; ‹pattern-body› (or ‹pattern-body›)*
‹pattern-body› ::= {‹constraint›; (‹constraint›;)*}
‹constraint› ::= C_i(‹variable›) | C_i.R_i(‹variable›, ‹variable›)
               | ‹variable› == ‹variable› | ‹variable› != ‹variable›
               | (find | neg find | count find) ‹pattern-dec›
               | (check | eval) (‹expression›)
```

Figure 6: A compositional view transformation language.

4 VIEW MODEL TRANSFORMATIONS

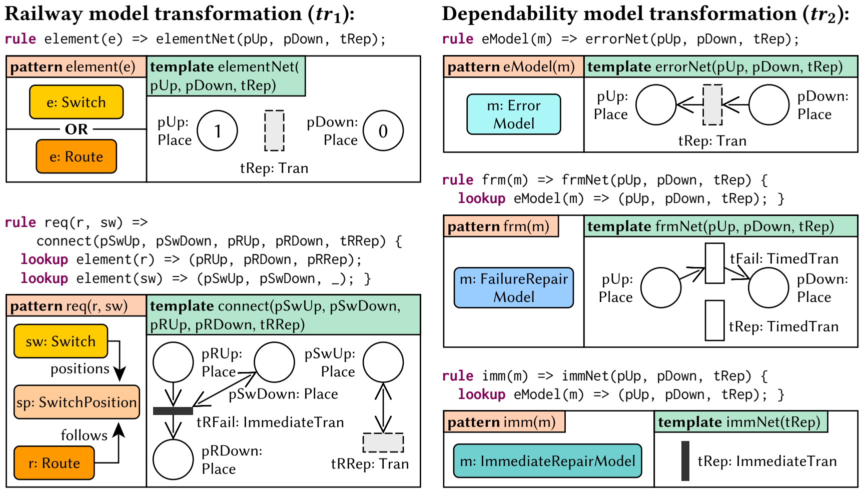
In this section we propose a view transformation language with relation-based composition along with a reactive, incremental, validating and inconsistency-tolerant execution engine. The view transformation is based on 4-valued partial models.

4.1 View definition by graph patterns

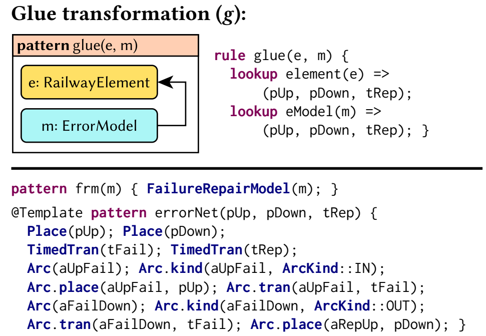
In this paper we introduce a declarative and fully compositional view transformation language based on graph queries. We reuse the Viatra Query Language [55] to form view transformation rules by using pairs of *precondition* patterns, *template* patterns and *lookups* to reference (matches of) other transformation rules.

A graph pattern captures structural constraints with a graph predicate. In the concrete syntax of Viatra (see Fig. 6), a pattern is declared (*‹pattern-dec›*) by a unique name (*‹pattern-name›*), and a list of formal pattern parameters (*‹param-list›*). The predicate of a pattern is defined by a disjunction of pattern bodies (*‹pattern-body›*) connected by the **or** keyword. A pattern body contains a conjunction of constraints that can be type and reference checks (C_i() and R_i(,)), equivalence check (==), positive, negative and aggregated pattern calls to compose complex patterns (resp. **find**, **neg find** and **count find** keywords), or external Java source code (using **check** or **eval** keywords) for attribute checks.

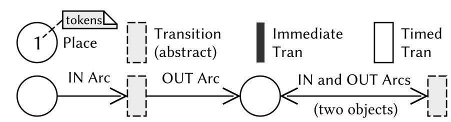
As templates of a view transformation rule, we define a restricted set of graph patterns (denoted by the underlined part of Fig. 6), which disallows multiple bodies, inequality constraints, negative and aggregated pattern calls, and **check** or **eval** expressions. In summary, a template pattern is a conjuction of atomic constraints.



(a) Rules for the railway model (tr_1), dependability model (tr_2) and glue (g) transformation definitions, which are composed to obtain the transformation $tr_1 \parallel_g tr_2$.



(b) Precondition pattern frm and template frmNet with Viatra Query textual syntax.



(c) Graphical syntax for stochastic Petri nets.

Figure 7: View transformation rules for Train Benchmark dependability example.

A view transformation definition consists of a set of view transformation rules, where each rule consists of a (i) a *precondition pattern for the source language*, (ii) a(n optional) *template pattern for the target language* built from a restricted subset of pattern language elements, and (iii) a list of *lookups* for traceability links and parameter bindings. A *lookup* refers to implicit traceability links between source and target elements created when the source pattern was matched and the corresponding target elements were created by the transformation rule referred in the lookup.

Example 4.1. View transformation rules of our running example are defined in Fig. 7. A detailed description is provided for the *frm* rule (for dependability transformation) in Fig. 7b. Its precondition pattern matches a single FailureRepairModel element m, assuming that the *eModel* rule has already been applied in the context of m as defined by the corresponding lookup. As a result of the rule, the *frmNet* template is applied on the target model, which specifies the creation of two places (pUp and pDown), two TimedTran elements (tFail an tRep), and two corresponding Arcs between them (from pUp to tFail and from tFail to pDown). However, due to the right side lookup directive, the two places pUp and pDown as well as the transition tRep need to be merged with corresponding target Petri net elements already created when rule *eModel* was applied – as defined by the unification introduced by identical variable names.

4.2 Execution of view transformations

View models are constructed in four steps as shown in Fig. 8.

(1) First, each view transformation rule creates a partial model representing the application of a template predicate in isolation. Next, (2) the partial models are merged together by linking different view fragments along equivalences $\sim$ based on the lookups in rules. After that, (3) the merged partial model is refined by various merge functions to enforce target metamodel constraints. Finally, (4) as the merged view may contain inconsistencies due to the contradicting view specifications, a materialization step operation removes $\lightning$ values from the partial model to end up with a regular target instance model.

I. Reactive (source-incremental) execution. First, the precondition φ^S of a rule R is matched against the source model by calculating the match set $ZS = \{Z \mid 1 \sqsubseteq [\![\varphi^S]\!]^P_Z\}$. We explicitly reuse existing features of the Viatra framework. Changes in the match set of source predicates are handled by using the incremental graph query engine of Viatra [55]. All subsequent processing steps in our engine are triggered and executed as a reactive transformation [56], therefore our entire engine becomes *reactive*.

II. Template instantiation and model merge. Then each rule R is applied independently. For each match $Z \in ZS$ of rule R a template partial model $T = \langle Obj_T, \mathcal{I}_T \rangle$ is created for each rule according to the target predicate φ^T. This T is constructed as:

- Each variable v of φ^T is mapped to an object of Obj_T
- Constraints of φ^T are translated to a 1 value in $\mathcal{I}_T$:
  - If there is a $\mathsf{C}_i(v)$ in the predicate φ^T, and variable v is mapped to an object o, then $\mathcal{I}_T(\mathsf{C}_i)(o) = 1$
  - If there is a $\mathsf{R}_j(v_1, v_2)$ in the predicate φ^T, and variable v_1, v_2 are mapped to objects o_1, o_2, then $\mathcal{I}_T(\mathsf{R}_j)(o_1, o_2) = 1$
  - If there is a $v_1 \sim v_2$ in the predicate φ^T, and variable v_1, v_2 are mapped to objects o_1, o_2, then $\mathcal{I}_T(\sim)(o_1, o_2) = 1$
- Every other values of $\mathcal{I}_T$ are $1/2$.

Next, each independently created template partial model $\{T_1, \ldots, T_n\}$ is copied together into a merged partial model $MP = \langle Obj_{MP}, \mathcal{I}_{MP} \rangle$ in order to represent all templates and lookups.

- Obj_{MP} consists of the union of objects of the template partial models: $Obj_{T_1} \cup \cdots \cup Obj_{T_n}$.
- $\mathcal{I}_{MP}$ is the same as the $\mathcal{I}_{T_i}$ of template partial model T_i: for each objects $o_1, \ldots, o_n$ in a template model T_i, and for each symbol $\alpha \in \Sigma$: $\mathcal{I}_{MP}(\alpha)(o_1, \ldots, o_n) = \mathcal{I}_{T_i}(\alpha)(o_1, \ldots, o_n)$
- Between the templates, **lookup** rules set additional $\mathcal{I}_{MP}(\sim)$ to 1 to add connections between templates.
- In all other cases $\mathcal{I}_{MP}(\alpha)(o_1, \ldots, o_n)$ is $1/2$.

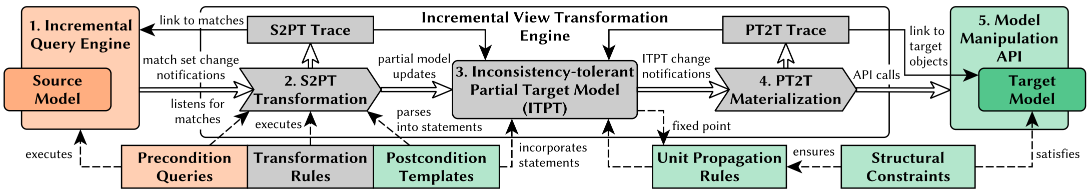


Figure 8: Overview of the view transformation

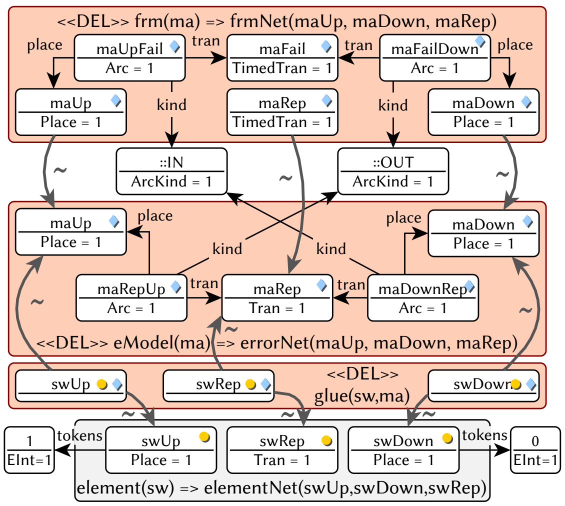


Figure 9: Initial partial model derived from predicates with the effects of a source change shown as «DEL» stereotypes.

The partial model *PM* obtained after this step is *para-complete* [8, 35], thus it may contain 1/2 and 1 values, but no 0 and ↯ values.

Example 4.2. Fig. 9 illustrates the application of the *frm* rule (from Fig. 7) for the source models of *Rail* and *Dep* from Fig. 1.

- First, the precondition of the rule *frm* checks for the existence of an FailureRepairModel element, and then the template *errorNet* is applied. As a result, the bottom part of the partial model (marked by a dashed rectangle) is created with model elements corresponding to the template.
- Since the rule contains a lookup to another rule *eModel*, partial model elements created by the two rules need to be merged. This is initiated by adding equivalence relations ∼ between the corresponding elements defined by variables such as maUp, maDown and maRep.
- Similar equivalences are declared by applying other transformation rules from Fig. 7 and Fig. 9 presents the entire partial model derived by all rules. The *glue* rule is a special view transformation rule where no target elements are created but only equivalences are declared.

III. Reactive object merge and propagation. By now, all objects of the partial model created by different templates are identified to be merged by marking them with equivalence relations. The merge functions defined for inconsistency tolerant partial models in Section 3.4 are executed in an incremental way.

Each propagation rule $prop = \varphi/\alpha_i$ has a graph predicate φ as a precondition which can be captured by a regular graph query evaluated over 4-valued logic. The execution of a propagation rule can be carried out reactively by extending the constraint rewriting technique [49] to provide 2-valued *may* and *must* graph predicates for under- and over-approximation. For the incremental execution of an object merge *om*, we rely upon incremental maintenance techniques for strongly connected components used for graph queries with transitive closure [10].

As a result of this step, all 1/2 values are removed, and all equivalent objects (marked by ∼) are merged, thus the partial model becomes *para-consistent* [8, 35] as it contains only 0, 1 and ↯ values. However, during the propagation phase, the partial model may contain both uncertain 1/2 and inconsistent ↯ values.

Example 4.3. The effects of object merge and propagation rules were illustrated in Fig. 3a. The two swRep objects of the partial model created by rules *element* (yellow dot) and *frm* (blue diamond).

The Fig. 3b case corresponds to a hypothetical source change where the match of rule *frm* no longer exists, thus the effects of the template need to be removed. The exact merge procedure was discussed in Example 3.3. Templates removed from the partial model due to this source change are shown as «DEL» stereotypes in Fig. 9.

IV. Incremental materialization. At the final step, erroneous elements of the target model are removed by a materialization step. After materialization, the partial model is equivalent to the target instance model, thus (1) all structural constraints of the target metamodel are ensured in accordance with the correctness of merge functions (see Section 3.4), hence our technique is *validating*. Moreover, (2) each change in this final partial model can be incrementally propagated to the target instance model, hence our approach is (target-)*incremental*. If a source model change does not affect a view model model, then no change is propagated to the target view model. Therefore, (3) our approach is *hippocratic*.

Concerning the (source-target) *consistency* of our approach, we need to separate the case when no ↯ symbols need to be removed during materialization. In such a case, all steps are valid refinement steps, thus it is guaranteed that the final model *P* refines all applied templates T_i ($T_i \sqsubseteq P$) which ensures consistency. If an ↯ symbol is removed during materialization, then the cause of this inconsistency can be shown by a corresponding match of a propagation rule precondition tracing the found issue back to the applied templates, the source model and the enforced structural constraint of the target metamodel.

Example 4.4. If all the propagation steps are executed for the partial model of Fig. 9 then the target Petri net instance model of Fig. 1 is obtained.

5 EVALUATION

Research Questions. Our view transformation approach is fully implemented as an open source project [1]. We carried out an experimental evaluation to address three research questions:

RQ1. What is the complexity of different execution phases in our view transformation engine?

RQ2. What is the performance overhead for the initial run of our view transformation engine compared to reactive imperative transformations with explicit traceability?

RQ3. What is the performance overhead for change-driven behavior of our view transformation engine compared to reactive imperative transformations with explicit traceability?

Case studies. We selected two substantially different view transformation challenges for our investigation. (1) *Dependability* is an extended version of the case study used in this paper which aims to compose two separate transformations in a way that the target Petri net model is significantly larger than any of the two source models. (2) *VirtualSwitch* is a filtering transformation taken from [18] where the size of the source model is significantly larger than the size of the target model. We believe that these transformations are representative for key practical applications of view transformations: the *VirtualSwitch* scenario is typical for in traditional view models with information loss [13] while the transformation challenges in the *Dependability* case are common for the formal analysis of extra-functional properties of systems [25, 42].

Compared approaches. First, we instrumented our *ViewModel* transformation approach to enable the clear separation of different transformation phases to address **RQ1**. Then we compare our approach with two different view transformation styles available in Viatra[1]. These solutions use an *explicit traceability model* (vs. implicit traceability in our approach) and *imperative actions* in transformation rules using Java/Xtend (vs. declarative query-based templates). However, differences in query performance can be mitigated to a large extent. (i) The *source-reactive* solution [18] uses exactly the same source queries as our view transformation approach, but rule priorities had to be set carefully. (ii) The *trace-reactive* solution [29] uses queries with both source and traceability elements as part of its precondition. Since both the level of compositionality and the properties of the view transformation engine are different in these approaches compared to our view transformation approach (see Section 2.3), our evaluation may reveal the performance trade-offs of the increased expressiveness of our approach.

Experiment setup. To investigate the initial transformation runs (**RQ2**), our measurement setup contains 5 source models of increasing size. For the *Dependability* case, the source models ranged 1K to 25K while the target models ranged from 3K to 72K. For the *VirtualSwitch* case, the source models were ranging from 25K to 425K elements, while the target models were ranging from 500 to 9K elements. In each case, we measured the initial time for populating the caches of queries and the execution time of the first transformation, while the load time of source models was excluded. To address **RQ1**, we measure how much time the different phases of our view transformation approach takes during this initial run.

To investigate the change-driven behavior (**RQ3**), we first created 10 different elementary changes (modifications of one element) and 5 change mixes containing 100 elementary changes each (with fix ratio between different types of change within each mix). Due to space restrictions, we only present results for 3 change mixes within the paper, while all other measurements (and plots) are available in [2]. Change mix (A) presents a balanced mix of changes, while types of changes in mixes (B) and (C) were selected from those elementary changes that caused longer synchronization times in the *Dependability* and *VirtualSwitch* cases, respectively.

Each experiment was executed 30 times after 10 warmup runs on a cloud-based virtual environment (with 4 CPU, 16 GB memory and 8 GB disk size) on Amazon AWS.

Results. Our evaluation results comparing the performance of core reactive Viatra transformations and our view model approach are presented in Fig. 10a where the two Viatra transformations (source vs. trace-reactive) have very similar behavior. The two key internal phases of our approach separating the source-to-partial model (S2PT) transformation and partial-model-to-target (PT2T) materialization stages (with propagation and concretization) are presented in Fig. 10b.

Since the *VirtualSwitch* case is dominated by the size of the source model while the *Dependability* case is dominated by size of the target model, the logarithmic horizontal (x) axis presents a combined model size as the geometric mean ($\sqrt{|src| * |trg|}$) of source and target model sizes (i.e. number of objects) which is compatible with the logarithmic scale of the plots. The logarithmic vertical (y) axis presents the execution times (in ms).

The intermediate partial model for the largest source models had (1) 222K partial model variables and 401K partial model atomic statements to represent 72K target objects (*Dependability*) and (2) 38K partial model variables and 58K partial model atomic statements which represents 8K target objects (*VirtualSwitch*).

Discussion. Based on these experimental results, we make the following observations related to the research questions:

RQ1: Both major view transformation phases seem to grow polynomially in model size, but more data points (model sizes) would be necessitated for a firm statement.

Dependability: The construction of the partial target model and its materialization are both challenging. The S2PT phase (0.4 s on smallest, but 12 s on largest) and the PT2T (0.3 s on smallest, but 14 s on largest) were within 0.5 orders of magnitude, while PT2T wass slower on large models as it has to perform type inferencing and complex object merges.

VirtualSwitch: The key challenge is to filter the source model, thus the intermediate partial model is smaller and necessitates fewer complex merges than above. Thus PT2T was 1 order of magnitude faster (S2PT 3.7 s on largest vs PT2T 0.65 s on largest).

RQ2: The initial query took exactly the same time (0.15 s for largest *Dependability*, 150 s for largest *VirtualSwitch*) for each implementation of the transformations, because the same queries and the same query engine (Viatra) was used, thus our measurements highlight the differences in the transformation phase. There was a

[1] Our repository contains an implementation of the transformations in batch ATL and a partial implementation in eMoflon, but the different performance optimizations in those tools would disallow to separate query performance from transformation performance.

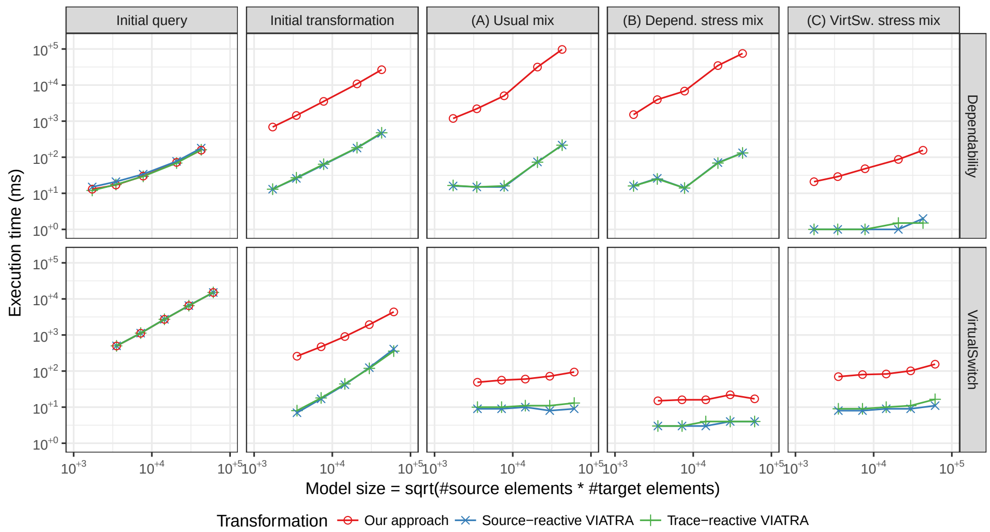
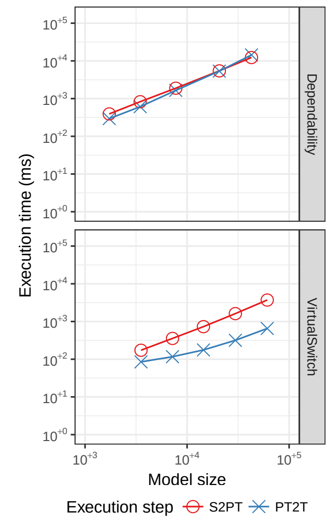


Figure 10: (a) Complexity of source query initialization, initial transformation, and the sychronization of a (A) balanced mix of modifications of 100 operations, and modification mixes of 100 operations focused on stressing the (B) Dependability, (C) VirtualSwitch transformation.



(b) Complexity of the two execution phases in our approach during initial transformation.

2 orders of magnitude difference in *Dependability* (26.7 s vs 0.48 s on largest), and 1 in VirtualSwitch (4.4 s vs 0.4 s on largest) between execution times in favor of reactive VIATRA transformations.

RQ3: In the *Dependability* case, we observed 2.5 orders of magnitude difference in mixes (A) and (B) which cause major changes in the target model (94 s vs 0.2 s on largest). In mix (C), which cause significantly fewer target changes as only attributes of places are modified, VIATRA was instantaneous, but our approach also took only 10–150 ms depending on model size to process the change.

In the *VirtualSwitch* case, VIATRA was instantaneous even in the modification mix specifically designed to cause target model changes. In (A) and (C), our approach took around 100-150 ms, which is significantly less than the initial transformation.

Conclusion. Our approach is more sensitive to target model size than source model size. The incremental behavior of our approach is also dominated by the size of the implied target change. For small target deltas, the overhead of our approach was less than 150 ms. The S2PT phase takes more time for complex model filtering and weaving challenges, while PT2T is slower when it has to materialize a large partial model. Unlike reactive VIATRA [18, 56], our approach achieves compositional and consistent view transformations (i.e. no manual adaptations to compose the original transformations). The performance penalty of this increased expressiveness is about 1-2 orders of magnitude increase in execution time compared to an industrial model transformation engine.

Threats to validity. To mitigate *internal validity*, 10 warm-up runs were included prior to the measurements to decrease the fluctuation of runtime caused by JVM. While our measurements were executed in the cloud (AWS), the same virtual machine was used for comparing the different approaches in a fair way.

To address *external validity*, we selected two transformations with substantially different characteristics (massive filtering in *VirtualSwitch* vs. complex merging in *Dependability*). Train Benchmark models serve as a common source model used in both cases, which may reduce the generalizability of our result to other domains. However, the Train Benchmark [54] has been actively used within the MDE community as a performance benchmark for different query and transformation tools, thus external validity is not compromised.

6 CONCLUSIONS AND FUTURE WORK

We proposed a fully compositional view transformation language executed by a reactive, incremental, validating and inconsistency-tolerant view transformation engine. Our approach reuses the VIATRA Graph Query Language [55] to define target fragments which are merged during transformation using the novel concepts of inconsistency tolerant partial models based on 4-valued logic foundations to gracefully handle temporal inconsistencies during transformations. The execution engine reuses existing support for incremental graph queries as available in the VIATRA framework [56] to provide reactive behavior, while graph predicates used in merge functions also enable incremental propagation of changes while ensuring structural constraints of the target language.

Our experimental evaluation also highlighted that such an increased expressiveness on the view transformation language level does not come for free as the core (imperative and reactive) VIATRA engine executes 1-2 orders of magnitude faster for the case studies – but the individual transformations had to be modified manually to achieve the necessitated merge functionality.

The detailed evaluation of the different execution phases also points to key directions for future work for a hybrid view transformation engine. A sophisticated static analyzer may automatically reveal transformation rules where compositionality falls into a more simple class, thus many optimizations available in existing view transformation tools would become amenable to improve performance. Nevertheless, our view transformation approach already provides strong support for the most challenging composition problems for a very expressive view transformation language.

REFERENCES

[1] 2018. ViewModel project repository. https://github.com/ftsrg/viewmodel
[2] 2018. ViewModel Tool and Benchmark Results for "Incremental View Model Synchronization Using Partial Models". https://doi.org/10.5281/zenodo.1318156 Approved by the MODELS '18 Artifact Evaluation Committee.
[3] Marco Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. 1994. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons.
[4] Anthony Anjorin, Zinovy Diskin, Frédéric Jouault, Hsiang-Shang Ko, Erhan Leblebici, and Bernhard Westfechtel. 2017. BenchmarX Reloaded: A Practical Benchmark Framework for Bidirectional Transformations. In *BX@ETAPS 2017 (CEUR Workshop Proceedings)*, Vol. 1827. CEUR-WS.org, 15–30.
[5] Anthony Anjorin, Sebastian Rose, Frederik Deckwerth, and Andy Schürr. 2014. Efficient Model Synchronization with View Triple Graph Grammars. In *ECMFA 2014*. Springer.
[6] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 2010. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *MODELS 2010*. Springer, 121–135.
[7] Olivier Beaudoux, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2010. Active Operations on Collections. In *MODELS 2010*. Springer.
[8] Nuel D. Belnap. 1977. A Useful Four-Valued Logic. In *Modern Uses of Multiple-Valued Logic*. Springer, 5–37.
[9] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. 2015. Viatra 3: A Reactive Model Transformation Platform. In *ICMT 2015*. Springer, 101–110.
[10] Gábor Bergmann, István Ráth, Tamás Szabó, Paolo Torrini, and Dániel Varró. 2012. Incremental Pattern Matching for the Efficient Computation of Transitive Closure. In *ICGT 2012*. Springer, 386–400.
[11] Gábor Bergmann, István Ráth, Gergely Varró, and Dániel Varró. 2012. Change-driven model transformations. *Softw. Syst. Model.* 11, 3 (2012), 431–461.
[12] Andrea Bondavalli, Ivan Mura, and István Majzik. 1999. Automatic Dependability Analysis for Supporting Design Decisions in UML. In *HASE '99*. IEEE, 64–74.
[13] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2017. A feature-based survey of model view approaches. *Softw. Syst. Model.* (2017).
[14] Hugo Brunelière, Jokin Garcia Perez, Manuel Wimmer, and Jordi Cabot. 2015. EMF Views: A View Mechanism for Integrating Heterogeneous Models. In *ER 2015*. Springer, 317–325.
[15] Peter Buneman, Mary Fernandez, and Dan Suciu. 2000. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB J.* 9, 1 (2000).
[16] Marsha Chechik, Shiva Nejati, and Mehrad Sabetzadeh. 2012. A relationship-based approach to model integration. *Innov. Syst. Softw. Eng.* 8, 1 (2012), 3–18.
[17] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2010. JTL: A Bidirectional and Change Propagating Transformation Language. In *SLE 2010*. Springer, 183–202.
[18] Csaba Debrezeni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró. 2014. Query-driven incremental synchronization of view models. In *VAO '14*. ACM, 31–38.
[19] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. 2011. Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In *MODELS 2010*. Springer, 165–179.
[20] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (Eds.). 1999. *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol. 1. World Scientific Publishing Co., Inc., River Edge, NJ, USA. 163–246 pages.
[21] Gregor Engels, Reiko Heckel, Gabriele Taentzer, and Hartmut Ehrig. 1997. A Combined Reference Model- and View-Based Approach to System Specification. *Int. J. Softw. Eng. Knowl. Eng.* 7, 4 (1997), 457–477.
[22] Michais Famelis, Rick Salay, and Marsha Chechik. 2012. Partial models: Towards modeling and reasoning with uncertainty. In *ICSE '12*. IEEE.
[23] Hamid Gholizadeh, Zinovy Diskin, and Tom Maibaum. 2014. A Query Structured Approach for Model Transformation. In *Workshop on Analysis of Model Transformations (CEUR Workshop Proceedings)*, Vol. 1277. CEUR-WS.org, 54–63.
[24] Holger Giese, Stephan Hildebrandt, and Leen Lambers. 2014. Bridging the gap between formal semantics and implementation of triple graph grammars. *Softw. Syst. Model.* 13, 1 (2014), 273–299.
[25] Stephen Gilmore, László Gönczy, Nora Koch, Philip Mayer, Mirco Tribastone, and Dániel Varró. 2010. Non-functional properties in the model-driven development of service-oriented systems. *Software & Systems Modeling* 10, 3 (2010), 287–311. https://doi.org/10.1007/s10270-010-0155-y
[26] Joel Greenyer. 2006. *A study of technologies for model transformation: Reconciling TGGs with QVT*. Diplomarbeit. Universität Paderborn.
[27] Joel Greenyer and Ekkart Kindler. 2007. Reconciling TGGs with QVT. In *MODELS 2007*. Springer, 16–30.
[28] David Hearnden, Michael Lawley, and Kerry Raymond. 2006. Incremental Model Transformation for the Evolution of Model-Driven Systems. In *MODELS 2006*. Springer, 321–335.
[29] Ábel Hegedüs, Ákos Horváth, István Ráth, Rodrigo Rizzi Starr, and Dániel Varró. 2016. Query-driven soft traceability links for models. *Softw. Syst. Model.* 15, 3 (2016), 733–756.
[30] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. 2011. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In *ASE 2011*. IEEE.
[31] Sochiro Hidaka and Massimo Tisi. 2016. *Partial Bidirectionalization of Model Transformation Languages*. Technical Report. https://hidaka.cis.k.hosei.ac.jp/research/papers/scp2016.pdf
[32] Johannes Jakob, Alexander Königs, and Andy Schürr. 2006. Non-materialized Model View Specification with Triple Graph Grammars. In *ICGT 2006*. Springer, 321–355.
[33] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A Model Transformation Tool. *Sci. Comput. Program.* 72, 1–2 (2008), 31–39.
[34] Frédéric Jouault and Massimo Tisi. 2010. Towards Incremental Execution of ATL Transformations. In *ICMT 2010*. Springer, 123–137.
[35] Norihiro Kamide and Hitoshi Omori. 2017. An Extended First-Order Belnap-Dunn Logic with Classical Negation. In *LORI 2017*. Springer, 79–93.
[36] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: a formally verified core language for putback-based bidirectional programming. In *PEPM '16*. ACM, 61–72.
[37] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. [n. d.]. The Epsilon Transformation Language. In *ICMT 2008*. Springer, 46–60.
[38] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2006. Merging Models with the Epsilon Merging Language (EML). Springer, 215–229.
[39] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. 2010. Explicit Transformation Modeling. In *MODELS 2009*. Springer, 240–255.
[40] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. 2012. Bidirectional Model Transformation with Precedence Triple Graph Grammars. In *ECMFA 2012*. Springer, 287–303.
[41] Erhan Leblebici, Anthony Anjorin, Lars Fritsche, Gergely Varró, and Andy Schürr. 2017. Leveraging Incremental Pattern Matching Techniques for Model Synchronisation. In *ICGT 2017*. Springer, 179–195.
[42] István Majzik, András Pataricza, and Andrea Bondavalli. 2002. Stochastic Dependability Analysis of System Architecture Based on UML Models. In *Architecting Dependable Systems*. Springer, 219–244.
[43] Salvador Martínez, Massimo Tisi, and Rémi Douence. 2017. Reactive model transformation with ATL. *Sci. Comp. Prog.* 136 (2017), 1–16.
[44] Sergey Melnik, Philip A. Bernstein, Alon Halevy, and Erhard Rahm. 2005. Supporting executable mappings in model management. In *SIGMOD '05*. ACM, 167–178.
[45] Bart Meyers. 2016. A Multi-Paradigm Modelling Approach to Design and Evolution of Domain-Specific Modelling Languages.
[46] Object Management Group. 2016. MOF Query/View/Transformation Specification. http://www.omg.org/spec/QVT/1.3/ Version 1.3.
[47] Mehrdad Sabetzadeh and Steve Easterbrook. 2006. View merging in the presence of incompleteness and inconsistency. *Requir. Eng.* 11, 3 (2006), 174–193.
[48] Andy Schürr. 1995. Specification of Graph Translators with Triple Graph Grammars. In *WG 1994*. Springer, 151–163.
[49] Oszkár Semeráth and Dániel Varró. 2017. Graph Constraint Evaluation over Partial Models by Constraint Rewriting. In *ICMT 2017*. 138–154.
[50] Hui Song, Gang Huang, Franck Chauvel, Wei Zhang, Yanchun Sun, Weizhong Shao, and Hong Mei. 2011. Instant and Incremental QVT Transformation for Runtime Models. In *MODELS 2011*. Springer, 273–288.
[51] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.
[52] Perdita Stevens. 2010. Bidirectional model transformations in QVT: semantic issues and open questions. *Soft. Syst. Model.* 9, 7 (2010).
[53] Perdita Stevens. 2014. Bidirectionally tolerating inconsistency: partial transformations. Springer, 32–46.
[54] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. 2017. The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Softw. Syst. Model.* (2017).
[55] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* 98, 1 (2015), 80–99.
[56] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi. 2016. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and Systems Modeling* 15, 3 (2016), 609–629.
[57] Dániel Varró, Oszkár Semeráth, Gábor Szárnyas, and Ákos Horváth. 2018. Towards the Automated Generation of Consistent, Diverse, Scalable and Realistic Graph Models. In *Graph Transformation, Specifications, and Nets*. Springer, 285–312.
[58] Edward D. Willink. 2017. The Micromapping Model of Computation; The Foundation for Optimized Execution of Eclipse QVTc/QVTr/UMLX. In *ICMT 2017*. Springer, 51–65.