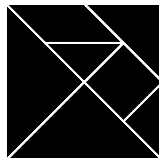
GSDLAB TECHNICAL REPORT

A Taxonomic Space for Increasingly Symmetric Model Synchronization

Zinovy Diskin, Arif Wider, Hamid Gholizadeh, Krzysztof Czarnecki

GSDLAB–TR 2014-02-1 February 2014



Generative Software Development Laboratory
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

WWW page: http://gsd.uwaterloo.ca/

The GSDLAB technical reports are published as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, notwithstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

A Space of Model Synchronization Types: Symmetrization of Model Transformations and its Challenges

Zinovy Diskin[1,2], Arif Wider[3], Hamid Gholizadeh[2], Krzysztof Czarnecki[1]

[1] University of Waterloo, Canada
[2] McMaster University, Canada
{diskinz|mohammh}@mcmaster.ca
[3] Department of Computer Science
Humboldt-Universität zu Berlin, Germany
wider@informatik.hu-berlin.de

Abstract. A pipeline of unidirectional model transformations is a well-understood architecture for model driven engineering tasks such as model compilation or view extraction. However, modern applications seem to require a shift towards *networks* of models related in various ways, whose synchronization often needs to be incremental and bidirectional. This new situation demands new features from transformation tools and a solid semantic foundation. We address the latter by presenting a taxonomy of model synchronization types, organized into a 3D-space. Each point in the space refers to its set of synchronization requirements and a corresponding algebraic structure modeling the intended semantics. The space aims to help with identifying and communicating the right tool and theory for the synchronization problem at hand. It also intends to guide future theoretical and tool research.

1 Introduction

Early model-driven engineering (MDE) was based on a simple generic scenario promoted by the *Model Driven Architecture* (MDA) vision [10]: platform-independent models describing a software system at a high-level of abstraction are transformed stepwise to platform-dependent models, from which executable source code is automatically generated. The generated code can be discarded anytime, whereas models are the primary artifacts to be maintained. Software development in the MDA perspective appears as a collection of model-transformation chains or streams "flowing through the MDE-pipe", as shown in Fig. 1.

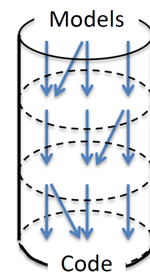
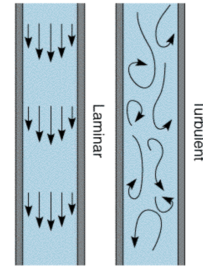


Fig. 1: MDE-pipe in MDA

However, this nice pipeline architecture fails to capture two important aspects of practical MDE. First, it turns out that some changes are easier to make in lower-level models (including code) rather than high-level models. This requirement leads to round-trip engineering, in which MT-streams in the MDE-pipe flow

back and forth. Second, models (on the same or different abstraction levels) are typically overlapping rather than disjoint, which in our pipe analogy means that MT-streams interweave rather than flow smoothly. Round-tripping and overlapping thus change the flow from "laminar" to "turbulent," as illustrated by the inset figure. Instead of separated and weakly interacting MT-streams, we have a network of intensively interacting models with bidirectional horizontal (the same abstraction level) and vertical (round-tripping) arrows as shown in Fig. 2.



"Turbulency" of modern MT brings several theoretical and practical challenges. Semantics of turbulent MT is not well understood, and clear semantics is crucial for synchronization tools because otherwise users have no trust in automatic synchronization. Moreover, tool users and tool builders need a common language to communicate required and provided features because not every synchronization problem requires the same set of features, and implementation of unnecessary features can be costly and increases chances of unwanted interaction. Having a taxonomy of synchronization behaviors, with a clear semantics for each taxonomic unit, would help to manage these problems.

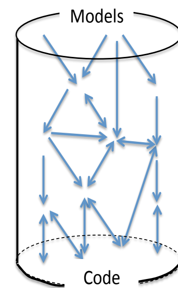


Fig. 2: Modern MDE pipe

We will analyse the basic unit of a model network – a pair of interrelated models to be kept in sync – and build a taxonomy of relationships between two models from the viewpoint of their synchronization, assuming that concurrent updates are not allowed. It is a strong simplifying assumption, yet many practically interesting cases are still covered. We identify three orthogonal dimensions in the space of such relationships, and 24 synchronization types — points in the space. The space equips the multitude of types with a clear structure: every type is characterized by a triple of its coordinates, which together determine its synchronization behavior. We term some synchronization types to be more *symmetric* than others. We observe an evolution of MDE from its early pipeline setting to its current state as a path from the asymmetric synchronization zone of the space to the symmetric zone (hence, the title of the paper).

We also propose an algebraic framework, in which synchronization types can be given a formal semantics. Algebraic laws related to a synchronization type give rise to requirements to synchronization procedures realizing the type. Hence, classifying a concrete synchronization case by its type helps to identify and communicate the right tool and the right specification for the synchronization problem at hand.

The paper is structured as follows. Section 2 introduces a series of examples that allow us to illustrate main synchronization types. In Sections 3 we organize them into a 3D-space, and Sections 4 and 5 specify the space formally. Section 7 presents related work and Section 8 concludes the paper.

2 Symmetrization via Examples

We will consider several examples of synchronization scenarios to illustrate what we mean by symmetrization.

2.1 Getting Started: "Fly with comfort!"

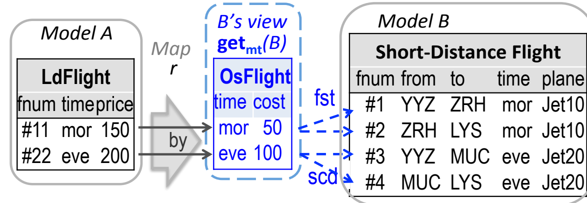


Fig. 3: Model B and mapping r implement model A

A Canadian air carrier is anticipating a significant increase in the passenger flow from Toronto to Lyon (close to Grenoble) due to the FASE'14 conference. The company decides to organize several charter one-stop flights. Two teams are created to work on the project. The first is responsible for the marketing: the number of flights to be added, their time (morning or evening), and prices. Model A in Fig. 3 (left) is a sample model that the marketing team has developed. The second team is technical: it works on implementation of marketing decisions, and deals with an optimal choice of intermediate airports and airplanes to use. Having data about airports and planes, the team can compute a crucial flight parameter: cost-per-passenger, or just cost. An example of a technical model implementing the marketing model A is also shown in Fig. 3. The model consists of a base table B of short-distance flights (sd-flights)[4], and a derived table of one-stop flights (os-flights), obtained by joining sd-Flight with themselves: an os-Flight flight self is a pair of sd-flights (self.fst, self.scd) satisfying two conditions:

$$(Q) \qquad \mathsf{self.fst.to} = \mathsf{self.snd.from} \text{ and } \mathsf{self.fst.time} \leq \mathsf{self.snd.time}.$$

The first one defines relational join and the second one assumes ordering mor< eve. We also define self.time=self.fst.time. Computing self.cost is done by some procedure using airport and airplane data. We thus have a function $\mathsf{get}_{\mathrm{mt}}$(read "get the marketing view of technical data") that computes derived table OsFlight from base table SdFlight.

Two 'by'-links relate Ld-flights to their one-stop implementations. Together they form an inter-table mapping by: LdFlight $\rightarrow$ OsFlight. This constitutes a *model-correspondence mapping* $r\colon A \rightarrow \mathsf{get}_{\mathrm{mt}}(B)$ (mapping r could contain more submappings like by, if model A would have more tables). Note that implementation is a pair (B, r), but we will often say 'implementation' B, leaving the correspondence mapping implicit. We consider implementation (B, r) to be *correct*, if for each ld-flight self we have:

$$(C) \qquad \mathsf{self.time} = \mathsf{self.by.time} \text{ and } \mathsf{self.price} \geq \mathsf{self.by.cost} + 100.$$

[4] We use standard airport codes: YYZ – for Pearson Int., Toronto, ZRH – Zurich, LYS – Lyon Saint-Exupéry, and MUC – Munich "Franz Josef Strauß".

Many different correct implementations of the same model A are possible. For example, Fig. 4 shows a variant with ZRH serving as a hub for several flights. It implies that the relational join table OsFlight has extra flights, and thus mapping r is not surjective. This is a typical situation: an implementation platform normally provides many possibilities, not all of which are used in a concrete implementation.

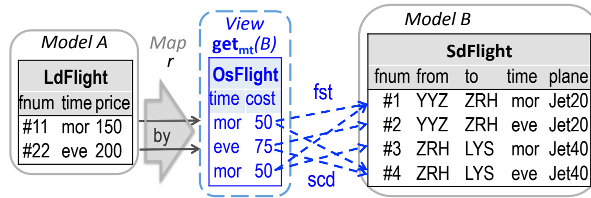


Fig. 4: Another Implementation

Also, to be precise, what is implemented is not the entire model A but its projection on the time and price columns (attribute `fnum` does not play any role in creation of model B). This projection is a technical view $\mathsf{get}_{\mathrm{tm}}(A)$ of marketing model A, which can be more complex if there are other attributes affecting the price (wholesales, promotions, etc.) but not relevant for side B. Figure 5 refines Fig. 4 by taking both views into account.

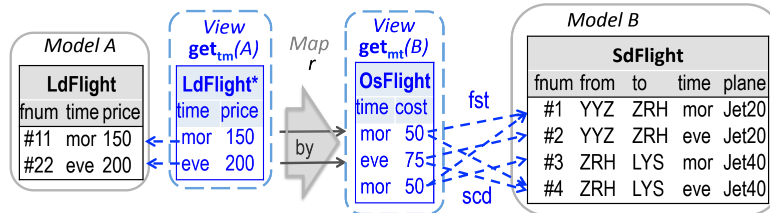


Fig. 5: Model B and mapping r implementing model A

Since model $A_t = \mathsf{get}_{\mathrm{tm}}(A)$ is fully determined by model A, we say that A *informationally dominates* A_t and write $A \geq_{\mathsf{inf}} A_t$. We will also say that model A has its *private* information invisible to A_t. Similarly, $\mathsf{get}_{\mathrm{mt}}(B) \leq_{\mathsf{inf}} B$. However, neither of models A and B fully dominates the other, as they both have their own private information. We call it *informational symmetry*, $A \asymp_{\mathsf{inf}} B$.

2.2 A Tour of Synchronization Symmetries and Asymmetries

There are several ways to organize the interaction between the marketing team (model A) and the technical team (model B). We will distinguish five basic interaction patterns, each described in one of the following subsections.

2.2.1 Any implementation is good enough, and the user does not care which one is chosen; e.g., either of the two possibilities shown in Fig. 3, 4, or yet another one, would suit the user. In other words, the marketing team pushes the policies and the technical team needs to find a way to implement whatever is decided by marketing. This is an unlikely scenario for flight implementation, but it can be often encountered in code generation, when the user does not have

access to code. We will refer to this situation by saying that model A *strongly dominates* B *organizationally*, and write $A \gg_{\mathsf{org}} B$ (or, equivalently, $B \ll_{\mathsf{org}} A$).

2.2.2 Implementation is an asset. Now suppose that the technical team works intensively with model B, tries different variants, analyzes them, and strives to find an optimal solution. Discarding results of these efforts with every change in model A would be discouraging for side B. A much better solution is to implement changes on side A incrementally as shown in Fig. 6: the change, or *delta*, on side A, Δ_A, is propagated to a delta on side B, Δ_B, which together with the original implementation B provides an updated implementation. In the figure, solid lines and shaded tables refer to given data, and dashed lines and blank tables denote data produced by the operation of delta propagation. In more detail, Δ_A makes explicit that flight #11 is preserved and flight #22 is added. Correspondingly, delta Δ_B keeps flights #1 and #2, and adds two Sd-flights implementing the required one-stop flight. The range of possible implementations is captured by placing *labeled nulls* $?_i$ into the table; nulls with the same label must be substituted with the same value (unknown so far), nulls with different labels are independent, but may be also substituted with the same value. In this way, uncertain model B' captures the implementation in Fig. 3 with $?_1$=MUC, and that one in Fig. 4 with $?_1$=ZRH, and others possibilities as well. Correspondingly, the derived OsFlight table is also uncertain: the cost value is given by applying some known procedure say, F, to unknown argument values $?_i, i = 1, 2, 3$.

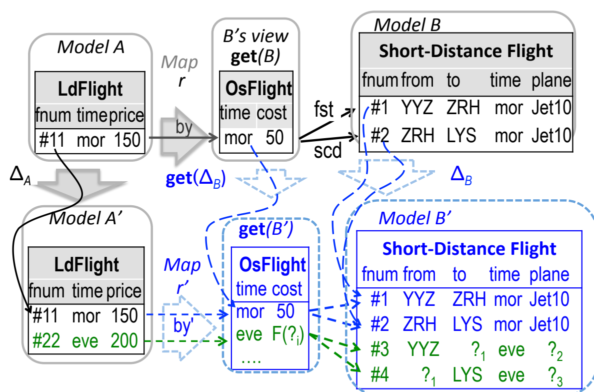


Fig. 6: Incremental implementation

Incremental propagation as described above gives model B more independence than in case 2.2.1. However, while update propagation from A to B is allowed, update propagation from B to A is prohibited as we explain in the following.

Suppose that model B is the result of customizing some previous model B^-, i.e., the update delta Δ^- leading to B was independently produced on side B rather than propagated. We assume some policy that does allow such changes on side B if they do not affect side A, that is, original model A^-=A (consistent with B^-) and updated model B are still consistent. Then Customization on side B will be saved even if A changes and the change is propagated incrementally as described by Fig. 6. However, the policy is prohibitive towards B, if updated B^- (i.e., B) and A^- are inconsistent; to restore consistency the policy would require to roll back Δ^-. We will refer to this situation where update propagation from A to B is allowed but update

propagation from B to A is prohibited by saying that model A dominates B *organizationally* (but not strongly), and write $A >_{\mathsf{org}} B$ or $B <_{\mathsf{org}} A$. We will also term the case as *organizational asymmetry*; then we may call the case in 2.2.1 *strong* organizational asymmetry.

2.2.3 Implementation is as important as specification: Round-tripping. Consider a different business context, in which the technical team gains a greater authority and administrative weight than before. Now, if while working with the B-model originated from model A, the team would find a good modification B', e.g., new profitable os-flights, but B' is inconsistent with A, then the B-team may require to modify model A to a state A' consistent with B'. In other words, updates now can be propagated in both directions. We will refer to the case as *organizational symmetry* of A and B, and write $A \asymp_{\mathsf{org}} B$.

2.2.4 In-between organizational asymmetry and symmetry. In this case only some updates can be propagated: Assume that model B is a database of SdFlights, and A is its materialized view comprising *all* OsFlights. Of course, the attribute `fnum` is to be skipped, and mapping `'by'` is bijective: $A \cong \mathsf{get}(B)$.

In our previous examples, view A was *prescriptive* in the sense that model B was thought of as an implementation of A. Now we consider view A as an ordinary *descriptive* view on the data source B, as is typical for databases. Yet we still want to allow the user to update the view, say, to state A', and propagate the update back to the source. The problem is that there are many states B' such that $\mathsf{get}(B') = A'$, but we cannot arbitrarily choose one of them: updated state A' reflects an updated state of the world, which is, in turn, reflected in the unique updated source B' (recall that B is a view of the real world data). In other words, choosing an arbitrary update policy does not work anymore.

If the view update is insertion, we can manage the uniqueness problem by filling unknown slots with labeled nulls (Section 2.2.2). Suppose, however, that the view update is a deletion, for example, ld-flight #11 in Fig. 6 is deleted. This deletion can be caused by a real world deletion of sd-flight #11.by.fst from YYZ to ZRH, or flight #11.by.scd from ZRH to LYS, or both. We cannot arbitrarily choose one of these choices, because rows of table SdFlight must represent actual flights existing in the schedule. Hence, deletions in the view A must be prohibited.

Thus, some of view updates are propagatable, and some are not; however, any view update is reachable from the source side B. The case is thus more symmetric than organizational asymmetry in 2.2.2, but less symmetric than round-tripping in 2.2.3. We will term the situation as organizational *semi-symmetry* (or *partial* round-tripping) and write $A \leq_{\mathsf{org}} B$.

A dual semi-symmetry case, $A \geq_{\mathsf{org}} B$, is also possible. It means that all view updates are propagatable, but only some of the source updates are allowed. For example, we can imagine a propagation policy when sd-flight updates changing costs, and hence possibly affecting prices, are propagatable, whereas updates that imply deletions or additions of ld-flights are prohibited.

3 From a Line of Examples to a Space of Types

The series of scenarios in Section 2 is intuitively perceived as linearly progressing from less to more symmetric synchronization behaviour. However, a more thorough analysis reveals a richer structural landscape which is a three-dimensional taxonomic space formed by three *orthogonal* synchronization features shown in Fig. 7: *Organisational symmetry, Informational symmetry, and Incrementality.* Symmetry of each feature is "measured" along its axis, and black and grey circles present *synchronization types*, each one characterized by its triple of coordinates. Below we will explain why six bottom types are split. Our sample scenarios are naturally placed on a trajectory as shown by the solid blue arrow in Fig. 7. We classify scenario from section 2.2.1 as (010), 2.2.2 as (011), 2.2.4 as ($\frac{1}{2}$11), and 2.2.3 as (111).

In more detail, axis X is for indexing organizational symmetry (org-symm): asymmetry is indexed by 0, symmetry by 1, and semi-symmetry or partial round-tripping has index $\frac{1}{2}$. Axis Y is for informational symmetry (info-symm): asymmetry and symmetry are indexed by 0 and 1 resp. Note that *strong* organizational asymmetry ($A \gg_{\mathsf{org}} B$ in section 2.2.1) turns out to be a compound concept: it is a combination of organizational asymmetry (org-symm.=0) and lack of incrementality (incr.=0).

Axis Z is for "measuring" with how much incrementality (if at all) models are synchronized. Incrementality index (incr.) is set either to 0, if model transformation is always executed from scratch, or to 1, if the transformation is incremental, i.e., realized via delta propagation as shown in Fig. 6. A special case of incremental transformation is when deltas are degenerated into pairs of states (A, A'), (B, B') and etc. (see [2]). We index this case by incr. $= \frac{1}{2}$ and call it *discrete incrementality.* For example, to generate code from a changed model

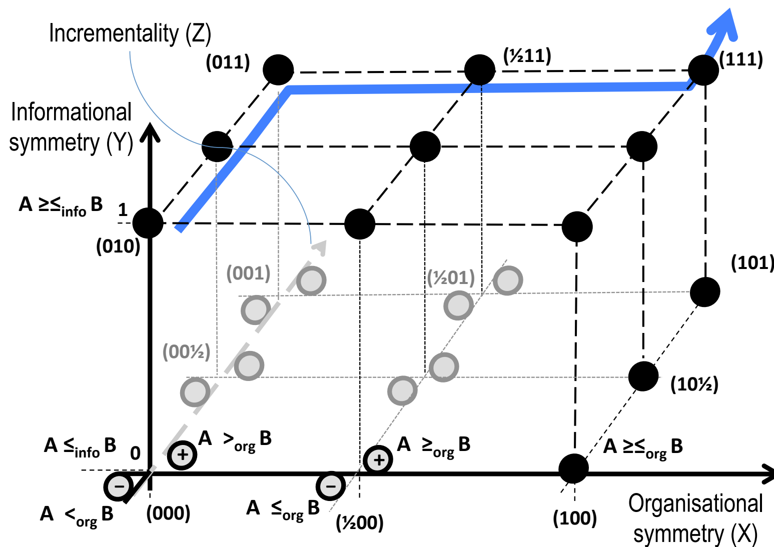


Fig. 7: Taxonomic space of synchronization types

state A', the transformation uses the original version of code B (generated from the original model A). As mentioned in the introduction, concurrent updates are not considered.

Computation vs. Organisation. In a synchronization scenario, we call forward (source to view/target) and backward (view/target to source) synchronization operations `get` and `gen`, respectively. These synchronization operations form a *computational framework* of synchronization. On the other hand, there is an *organizational framework*: It organizes different policies that prescribe one or another use of synchronization operations. These two aspects – computation and organization – are independent and should be separated in a foundational synchronization framework. This is one of the main ideas of our space.

For example, in a database context, we sometimes require that views are created and updated independently rather than receiving updates from the source, and the source data cannot be changed other than via the view (i.e., source data cannot be updated directly). With this policy it seems that we only require operation gen for our purpose, but we also need operation get to check the correctness of source generation, i.e., to ensure equality $\mathsf{get}(\mathsf{gen}(A)) = A$ or inclusion $\mathsf{get}(\mathsf{gen}(A)) \sqsupseteq A$ for a given view A (see [4] for invertibility requirements etc.). In this scenario, the view is *active* – it issues updates that have to be accepted by the source, while the source is a *passive* receiver of the updates and cannot change the view. We also say that the view *organizationally dominates* the source. However in another scenario, the organizational dominance can be reversed so that the source is active, i.e., is created and updated independently, while the view is a passive receiver of updates from the source. This is typical for databases, but it is known that for some simple views, view updates are allowed to be propagated back to the source, and then operation gen is needed.

In both scenarios, we need both forward and backward operations, but we use them for different purposes. Thus, the presence of two mutually inverse operations (get, gen) (the computational framework) does not say anything about which side is active or passive, and how updates are propagated (the organizational framework). The latter can be set independently of the computational base: computation and organization are two orthogonal dimensions of model synchronization. Different types of computational frameworks are placed in the YZ-plane of the space, which is described in Sect. 4. Different types of synchronization policies form axis X orthogonal to the YZ-plane are specified in Sect. 5.

Split types. An intricacy of the space is that some coordinates split into two subtypes because of the asymmetry of dominance relations (see XZ-plane in Fig. 7). When a type xyz comprises two asymmetric relations, it splits into two subtypes depending on whether two dominant models coincide or not. We will denote two corresponding types by xyz^- (even less symmetry, since the same model is "suppressed" in both relations), or xyz^+ (if one model dominates in one relation, and the other model in the other relation). Typical instances of type $(001)^-$ are incrementally maintained database views, or incremental reverse engineering. Typical instances of type $(001)^+$ are incremental unidirectional refinements, particularly, code generation. As a results of this splitting, there are

$3{\times}2 + 3{\times}2 + 3 = 15$ types with info-symm=0 and 9 types with info-symm=1. Hence, the total number of types is $15 + 9 = 24$.

4 Computational framework: Lenses

A well-known mathematical framework for modeling bidirectional transformations is based on algebraic structures called *lenses* [11,2]. Roughly, an *asymmetric* lens is a tuple $L^{\leq} = (\mathsf{V}, \mathsf{S}, \mathsf{get}, \mathsf{gen})$ with V and S the *source* and *view model spaces* resp., and $(\mathsf{get}, \mathsf{gen})$ a pair of operations mapping models from S to models in V (*get the view of the source*), and back (*generate the source from the view*). The operations are required to be *mutually inverse* in one or another sense. The structure of model spaces and arities of the operations depend on whether the transformations are incremental, and how incrementality is realized.

4.1 Non-incremental lenses

In the simplest case of non-incremental transformation, model spaces are just sets of models, get and gen are ordinary unary operations, $\mathsf{get}\colon \mathsf{V} \leftarrow \mathsf{S}$ and $\mathsf{gen}\colon \mathsf{V} \rightarrow \mathsf{S}$, and their invertibility amounts to equation $A = \mathsf{get}.(A.\mathsf{gen})$ holding for any model $A \in \mathsf{V}$. We have chosen the directions of get and gen as above to correspond to the example in Section 2. Also, to recall directions of operations, we write the arguments of get to the right, and of gen to the left, of the function symbol.

Importantly, lenses do not require $(\mathsf{get}.B).\mathsf{gen} = B$ for all $B \in \mathsf{S}$ as there may be multiple different sources with the same view.

The algebraic structure described above is called a *non-incremental* (asymmetric) lens. We denote them by double arrows $L_0^{\leq}\colon \mathsf{V} \leftrightharpoons \mathsf{S}$ (think of two operations) with the upper semi-arrow giving the direction of get. The superscript in the symbol $L_0^{\leq}$ also points to the view space, and the subscript refers to zero-incremental (i.e., non-incremental) transformation.

4.2 Incremental case

For modeling incremental transformations, i.e., update (delta) propagation, we assume model spaces are categories, i.e., directed graphs with sequentially composable arrows, whose objects/nodes are models, and arrows are intermodel deltas. That is, a model space M is a pair $(\mathsf{M}_\bullet, \mathsf{M}_\Delta)$ with $\mathsf{M}_\bullet$ a set of models, and M_Δ a set of composable directed intermodel deltas, which includes an identity loop delta $\mathsf{id}_A\colon A \rightarrow A$ for every model $A \in \mathsf{M}_\Delta$ (see [4] for a detailed motivation). Incremental transformations are now modeled by operations get and gen acting over models *and* deltas: $\mathsf{get}\colon \mathsf{V} \leftarrow \mathsf{S}$ is a functor[5], and $\mathsf{gen}\colon \mathsf{V}_\Delta{\times}\mathsf{S}_\bullet \rightarrow \mathsf{S}_\Delta$ an operation, which takes a view delta $a\colon A \rightarrow A'$ and an original source B such

[5] a mapping sending nodes to nodes and edges to edges so that their incidence is preserved, and arrow composition and identities are preserved as well

that $A = \mathsf{get}.B$, and produce a source delta $b\colon B \to B'$. In the lens jargon, this is phrased as "put the view update back to the source", and such binary gen is usually denoted by put. Invertibility is specified by the PutGet law: if $(a\colon A \to A', B).\mathsf{put} = b\colon B \to B'$, then $a = \mathsf{get}.b$ for any $a{\in}\mathsf{V}_\Delta$, $B{\in}\mathsf{S}_\bullet$. Thus, a delta lens is a tuple $(\mathsf{V}, \mathsf{S}, \mathsf{get}, \mathsf{put})$ denoted by $L^{\leq}_1\colon \mathsf{V} \leftrightharpoons \mathsf{S}$ with subindex 1 meaning incrementality. We thus have two notions of lenses, $L^{\leq}_{\mathsf{incr}}$ with $\mathsf{incr}= 0$ or 1 for non-incremental and incremental transformations resp.

A *symmetric lens* of incremental type $\mathsf{incr} \in \{0, 1\}$ is a pair of lenses, the *left* one, $L^{\geq}_{\mathsf{incr}}\colon \mathsf{M} \rightleftharpoons \mathsf{V}$, and the *right* one, $R^{\leq}_{\mathsf{incr}}\colon \mathsf{V} \leftrightharpoons \mathsf{N}$, working over a shared view space V. The operations of *forward*, from M to N, and *backward*, from N to M, delta propagation are defined by composing the corresponding gets and puts. For a left delta $a\colon A \to A'$ and right model $B{\in}\mathsf{N}_\bullet$ such that $A.\mathsf{get}_L = \mathsf{get}_R.B$, the corresponding right delta $b\colon B \to B'$ can be computed by setting $b = (a.\mathsf{get}_L, B).\mathsf{put}_R$. We thus have the operation $\mathsf{fPpg}\colon \mathsf{M}_\Delta{\times}\mathsf{N}_\bullet \to \mathsf{N}_\Delta$ of forward update propagation. Dually defined is the operation of backward propagation $\mathsf{bPpg}\colon \mathsf{M}_\Delta \leftarrow \mathsf{M}_\bullet{\times}\mathsf{N}_\Delta$: for given $A \in \mathsf{M}_\bullet$ and $b\colon B \to B' \in \mathsf{N}_\Delta$, $\mathsf{bPpg}(A, b) = \mathsf{put}_L(A, \mathsf{get}_R.b)$ gives a left delta $a\colon A \to A'$. We will write a symmetric lens as a tuple $S^{\geq\leq}_{\mathsf{incr}} = (\mathsf{M}, \mathsf{N}, \mathsf{fPpg}, \mathsf{bPpg})$ omitting alignment operations, and denote them by double-arrows $S^{\geq\leq}_{\mathsf{incr}}\colon \mathsf{M} \leftrightharpoons \mathsf{N}$. [6] Spaces M and N are called the *left* and N the *right sources* resp.

Note that an asymmetric lens $R^{\leq}_{\mathsf{incr}}\colon \mathsf{V} \leftrightharpoons \mathsf{S}$ can be considered as a special symmetric lens $S^{\geq\leq}_{\mathsf{incr}}\colon \mathsf{V} \leftrightharpoons \mathsf{S}$ whose left component is the identity lens $Id^{\geq}\colon \mathsf{V} \rightleftharpoons \mathsf{V}$, and the right component is $R^{\leq}_{\mathsf{incr}}$.

Thus, we have four notions of lenses, which constitute four points in the YZ-plane of our space.

4.3 Private vs. public deltas.

Let pair of lenses, $L^{\geq}_1\colon \mathsf{M} \rightleftharpoons \mathsf{V}$ and $R^{\leq}_1\colon \mathsf{V} \leftrightharpoons \mathsf{N}$, be a symmetric lens providing update/delta propagation from the left to the right space and back. Delta $a\colon A \to A'$ in the left space is called *private*, if $a.\mathsf{get}_L = \mathsf{id}_{A.\mathsf{get}_L}$. Non-private deltas are called *public*. Thus, we have a partition $\mathsf{M}_\Delta = \mathsf{M}^{\mathrm{prv}}_\Delta \uplus \mathsf{M}^{\mathrm{pub}}_\Delta$. Similarly, delta $b\colon B \to B'$ is called *private* if $\mathsf{id}_{\mathsf{get}_R(B)} = \mathsf{get}_R(b)$, and $\mathsf{N}_\Delta = \mathsf{N}^{\mathrm{prv}}_\Delta \uplus \mathsf{N}^{\mathrm{pub}}_\Delta$.

5 The Process: Synchronization Cases and Types

5.1 Models as trajectories

Organizational symmetry is about change propagation, and we will consider a changing model as a *trajectory* in the respective space. Let M be a metamodel, and $\mathsf{M} = (\mathsf{M}_\bullet, \mathsf{M}_\Delta)$ be the space of its instances and deltas as described above. We define a model over M to be a mapping $A\colon I \to \mathsf{M}_\bullet$, whose domain I is a finite

[6] Symmetric lenses can be defined in a more general way by defining fPpg and bPpg axiomatically [2,3], but we do not need this generality for the present paper.

linearly ordered set $\{i_0 < i_1 < \ldots < i_n\}$ of *version numbers* or *indexes*. Thus, A appears as the model's immutable identity whereas its state $A(i)$ changes as index i runs over I. To simplify notation, below we will write A_i for $A(i)$, and by the abuse of terminology often call model's states just models.

If $i{\in}I$ is a version number and $i{-}1$ is its parent (the previous version number), we have a (directed) delta $a_i\colon A_{i-1} \to A_i$ specifying the change. We may consider the pair $(i{-}1, i)$ as an arrow from $i{-}1$ to i, and delta a_i as the A-image of this arrow in M_Δ. This makes I a directed graph, and A a graph mapping. Moreover, we can make I a category I with nodes $\mathsf{I}_\bullet = I$ and arrows $\mathsf{I}_\Delta = \{i_1 i_2\colon\ i_1 < i_2 \in I{\times}I\}$, arrow composition $i_1 i_2; i_2 i_3 = i_1 i_3)$ and identities ii. Then a model trajectory is a functor $A\colon \mathsf{I} \to \mathsf{M}$, i.e., a graph mapping such that $A(i_1 i_3) = A(i_1 i_2); A(i_2 i_3)$ and $A(ii) = \mathsf{id}_{A_i}$. For any non-initial index i, we write a_i for delta $A(i{-}1, i)\colon A_{i-1} \to A_i$ to be read "the update that created model A_i".

To traverse set I, we will use operations $i{-}1, i{-}2, \ldots$ and $i{+}1, i{+}2, \ldots$ with the evident meaning. (Not defined are $i_0 - 1$ and $i_n + 1$.)

Class of all possible trajectories in space M is denoted by $\boldsymbol{Tr}[\mathsf{M}]$.

5.2 Synchronization cases and types

Synchronization of two models is about maintaining certain correspondences between two trajectories, say, $A\colon \mathsf{I} \to \mathsf{M}$ and $B\colon \mathsf{J} \to \mathsf{N}$, in two computationally related model spaces. For example, we may assume the spaces are related by a delta lens $\lambda\colon \mathsf{M} \leftrightharpoons \mathsf{N}$ comprising operations of forward and backward delta propagation as described in Section 4.

Partitioning of model deltas into private and public determines a similar partitioning of index deltas and, resp., indexes themselves: $I = I^{\text{prv}} \uplus I^{\text{pub}}$ with

$$I^{\text{prv}} = \{i \in I\colon\ a_i \in \mathsf{M}_\Delta^{\text{prv}}\},\ \text{and}\ I^{\text{pub}} = \left\{i \in I\colon\ a_i \in \mathsf{M}_\Delta^{\text{pub}}\right\},$$

and similarly $J = J^{\text{prv}} \uplus J^{\text{pub}}$.

Since public updates destroy consistency, as soon as a public update is committed on one side, it must be propagated to the other side to restore consistency. The propagated update is also public, but we call it *passive* while the original update is *active*. For example, suppose that we have a public update $a_i\colon A_{i-1} \to A_i$, i.e., $i \in I^{\text{pub}}$. There are two possibilities for the case. Either update a_i was initiated on the A side (we say a is *active*) and then was propagated to the B side. This means that there is a version number $i{\rhd} \in J$ and update $b_{i\rhd}\colon B_{i\rhd-1} \to B_{i\rhd}$ produced by this propagation (we then say that b is passive), such that consistency of A_i and B_j is restored: $A_i \sim B_{i\rhd}$. Alternatively, a_i is the result of propagation from the other side (now a is *passive*) of some (active) update $b_j\colon B_{j-1} \to B_j$ so that $i = {\lhd}j$ and we again have consistency $A_{\lhd j} \sim B_j$ resulted from this backward propagation.

Then for a pair of synchronized trajectories as above, we have a partitioning $I^{\text{pub}} = I^{\text{act}} \uplus I^{\text{pas}}$, a partitioning $J^{\text{pub}} = J^{\text{act}} \uplus J^{\text{pas}}$, and a pair of order-preserving

bijections,

$$_\triangleright : I^{\text{act}} \to J^{\text{pas}} \text{ and } \triangleleft_ : I^{\text{pas}} \leftarrow J^{\text{act}}.$$

This data imply an evident isomorphism $i \triangleright\!\triangleleft j$ between I^{pub} and J^{pub} such that for any pair $i \triangleright\!\triangleleft j \in I \times J$, models A_i and B_j are consistent: $A_i \sim B_J$.

The next definition gives an accurate formal specification of the case.

Definition 1 (Synchronization case). Given a lens $\lambda\colon \mathsf{M} \leftrightharpoons \mathsf{N}$, a *(consistent) synchronization case* is a pairs of trajectories, $A\colon \mathsf{I} \to \mathsf{M}$ and $B\colon \mathsf{J} \to \mathsf{N}$, with the following additional structure.

(a) Sets I^{pub} and J^{pub} are further partitioned, $I^{\text{pub}} = I^{\text{act}} \uplus I^{\text{pas}}$ and $J^{\text{pub}} = J^{\text{act}} \uplus J^{\text{pas}}$, and two isomorphisms are given: $_\triangleright : I^{\text{act}} \to J^{\text{pas}}$ and $\triangleleft_ : I^{\text{pas}} \leftarrow J^{\text{act}}$. This established an isomorphism $\triangleright\!\triangleleft : I^{\text{pub}} \to J^{\text{pub}}$, such that for any pair of corresponding indexes $i \triangleright\!\triangleleft j \in I \times J$, models A_i and B_j are consistent: $A_i \sim B_j$. Particularly, we assume that initial indexes correspond, $i_0 \triangleright\!\triangleleft j_0$, and thus $A_0 \sim B_0$.

Note also that set $I^{\text{pub}} \subset I$ is also linearly ordered and has its own traversal operations i——1 (*long minus*) and i++1 (*long plus*).

(b) Now we require the following. If $i \triangleright\!\triangleleft j$ and $i{+\!\!+}1 \in I^{\text{act}}$, then $j{+\!\!+}1 \in J^{\text{pas}}$ and

$$a_{i{+\!\!+}1}.\mathsf{fPpg}(B_{j{+\!\!+}1-1}) = b_{j{+\!\!+}1},$$

where $a_{i{+\!\!+}1} : A_{i{+\!\!+}1-1} \to A_{i{+\!\!+}1}$, $b_{j{+\!\!+}1} : B_{j{+\!\!+}1-1} \to B_{j{+\!\!+}1}$, and $A_{i{+\!\!+}1-1} \sim B_{j{+\!\!+}1-1}$ as $A_i \sim B_j$ and all updates between i and $i{+\!\!+}1$, and j and $j{+\!\!+}1$ are private by the definition of long plus.

If $i \triangleright\!\triangleleft j$ and $i{+\!\!+}1 \in I^{\text{pas}}$, then $j{+\!\!+}1 \in J^{\text{act}}$ and

$$a_{i{+\!\!+}1} = \mathsf{bPpg}(A_{i{+\!\!+}1-1}).b_{j{+\!\!+}1},$$

The necessary modification of this definition for the case of non-incremental lens $\lambda\colon \mathsf{M} \leftrightharpoons \mathsf{N}$ is evident.

Below we will often write a synchronization case over a lens $\lambda\colon \mathsf{M} \leftrightharpoons \mathsf{N}$ as a pair of trajectories (A, B) in spaces M and N resp., thus leaving all the necessary extra structure specified in Definition 1 implicit.

5.3 Synchronization Types

The class of all synchronization cases over a given lens $\lambda\colon \mathsf{M} \leftrightharpoons \mathsf{N}$ is denoted by $\boldsymbol{SC}[\lambda]$. It is a big set of pairs of synchronized trajectories that contains all possible synchronization cases. If special *organizational* relations between the models are assumed, some cases can be a priori prohibited. For example, we can make side A an entirely passive receiver of changes from side B by requiring $I^{\text{act}} = \varnothing$. Thinking extensionally (i.e., in terms of sets), an organizational relation *is* a set of synchronization cases considered legal wrt. this relation. We call such sets synchronization types.

Our first condition is quite general; it is a sort of a liveness condition for types.

Definition 2 (Completeness). A set $T \subset \boldsymbol{SC}[\lambda]$ of synchronization cases is called a *(synchronization) type* if it is *complete* in the following sense: for any deltas $d \in \mathsf{M}_\Delta$ and $e \in \mathsf{N}_\Delta$, there is a case (A, B) with $i \in I$ and $j \in J$ such that $a_i = d$ and $b_j = e$.

Below we will describe several properties of synchronization cases. Each such a property, say, Φ, determines a corresponding type, $[\Phi] \stackrel{\text{def}}{=} \{sc \in \boldsymbol{SC}[\lambda]\colon\ sc \models \Phi\}$. We will name these properties and types in terms of organizational symmetry or asymmetry.

We begin with a general pattern/schema of synchronization case properties.

Definition 3 ((P, Q)-symmetry). Let (A, B) be a sync case over a lens $\lambda\colon \mathsf{M} \leftrightarrows \mathsf{N}$, and a pair of sets $P \subset \mathsf{M}_\Delta$ and $Q \subset \mathsf{N}_\Delta$ of *propagatable* deltas is given. We say the case is (P, Q)-*compatible* if $a_i \in P$ for all $i \in I^{\text{act}}$, and $b_j \in Q$ for all $j \in J^{\text{act}}$. That is, a public update a_i, $i \in I^{\text{pub}}$ is allowed to propagate to the other side iff $a_i \in P$, that is, equivalence $i \in I^{\text{act}}$ iff $a_i \in P$ holds. Similarly, b_j, $j \in J^{\text{act}}$ is possible iff $b_j \in Q$. We will also call (P, Q)-compatible case (P, Q)-*symmetric* and write $A{>\!\!<}^{PQ}_{\mathsf{org}}B$.

Importantly, sets P and Q are determined organizationally rather than technologically in the sense that propagation operations are defined for non-propagatable deltas. For example, when code is generated from a UML model by a forward operation gen, the backward operation get is defined for all code deltas, and is theoretically important for checking correctness of code generation (the GenGet law). However, only some (or none) of code deltas are allowed to be propagated back to the model.

The general PQ-symmetry has several important specializations.

Definition 4 (Organizationally asymmetric types). We say *model A is organisationally dominated by B* or, equivalently, *B dominates A*, if $P = \varnothing$ whereas $Q = \mathsf{N}^{\cup}_\Delta$. Then we write $A <_{\mathsf{org}} B$ or $B >_{\mathsf{org}} A$.

If the lens is symmetric, then properties $A <_{\mathsf{org}} B$ and $A >_{\mathsf{org}} B$ are equivalent wrt. permutation of A and B. However, if the lens is asymmetric $\lambda^{\leq}\colon \mathsf{M} \leftrightarrows \mathsf{N}$, then we have two different situation depending on which side of the lens is dominated: we write $A <^{+}_{\mathsf{org}} B$ if $A <_{\mathsf{org}} B$, and $A <^{-}_{\mathsf{org}} B$ if $A >_{\mathsf{org}} B$.

Definition 5 (round-tripping). Synchronization case (A, B) is called *organizationally symmetric*, if $P = \mathsf{M}^{\cup}_\Delta$, $Q = \mathsf{N}^{\cup}_\Delta$, and both sets I^{act} and J^{act} are not empty. We then write $A{>\!\!<}_{\mathsf{org}}B$.

A very special subtype of this type is *interleaving*, for which, in addition, the following holds: $I^{\text{prv}} = \varnothing = J^{\text{prv}}$. In other words, the two models actually share the same set of version indexes, and all changes on either sides are at once propagated to the other side in the interleaving mode.

Definition 6 (Partial roundtripping or semi-symmetry). Synchronization case (A, B) is called *organizationally semi-symmetric*, if $P \subsetneq \mathsf{M}^{\cup}_\Delta$, $Q = \mathsf{N}_\Delta$, and I^{act} and J^{act} are not empty. We then write $A \leq_{\mathsf{org}} B$ and call the type *partial round-tripping.*

Another partial round-tripping type $A \bowtie_{\mathsf{org}}^{PQ}$ is when both sets of propagatable deltas are partial: $P \subsetneq \mathsf{M}_{\Delta}^{\cup}$ and $Q \subsetneq \mathsf{N}_{\Delta}^{\cup}$, and I^{act} and J^{act} are not empty.

6 Symmetrization and its Challenges

The examples we discussed previously point to the importance of dealing with networks of interacting models and model transformations rather than unidirectional pipelines. Synchronization tasks in such network scenarios demand transformation tools that support bidirectionality, incrementality, informational symmetry, and ultimately concurrent updates. We call this trend *symmetrization* of model transformations. It poses several challenges for transformation tools. Here, we want to attract attention to those challenges and make suggestions for future practical and theoretical research.

Flexible Tool Architectures Developing synchronization tools that meet all the requirements posed by symmetrization is hard to accomplish. We have shown formally that several of these requirements are independent of each other. Sometimes features only need to be supported to some extent as we have shown with $\frac{1}{2}$ indices on the incrementality and organisational axis.

Tool architectures need to reflect feature orthogonality and fine-grained requirements by allowing for flexible combination of required features. For example, when update procedures propagate deltas instead of states, the task of obtaining deltas from given states can be flexibly assigned. They can be provided directly by the editing tool or by comparing states and applying heuristics. The former allows for fine-grained incremental updates, while the latter is better suitable for concurrent, i.e. independent, updates.

Semantics of Synchronizations When synchronizing two models, two procedures of update propagation, from A to B and from B to A, must be consistent between themselves, and satisfy some invertibility property (see [4,2] for details). When implementing those procedures separately, proving and maintaining consistency and invertibility for complex synchronization becomes a major maintenance issue. The goal of bidirectional transformations (BX) is to specify a consistency relation and let the update propagation procedures be inferred from this specification, so that they are always consistent by construction. Because there are usually many possibilities to restore consistency, the behaviour of the inferred procedures must be predictable for a user. The situation with QVT-R shows how unclear semantics of BX hinders tool implementation and user acceptance [11]. Thus, a main challenge of symmetrization is to provide solid semantics of BX, especially in combination with incrementality and informational symmetry. The formal framework presented in the previous section is meant to serve as a foundation for providing BX semantics.

Informational Symmetry Many BX approaches rely on informational asymmetry. In practice, informational asymmetry rarely occurs. Even scenarios that seem to be info-asymmetric at first glance are often info-symmetric because most models contain some private part – e.g., layout information – that needs

to be preserved (though info-asymmetry can be a useful simplification). While combinators for info-asymmetric BX have been shown to be very useful, their symmetric counterparts are still missing.

Independent and Concurrent Updates When models that need to be synchronized can be created independently and/or updated concurrently, solving conflicts and matching models heuristically become important tasks. This differs substantially from a situation that prohibits independent changes. Theoretical and practical research is needed to deal with this situation adequately.

7 Related Work

Existing works on synchronization – practical and theoretical – usually focus only on one specific type, i.e., one point in our space. For instance, original lenses as presented by Foster et al. [5] formalize info-*a*symmetric state-based BX with incrementality incr.=1/2. Info-*s*ymmetric state-based lenses with incr.=1/2 were proposed in [11]. Delta-based lenses (incr.=1) were introduced for informational asymmetry in [4] and symmetry in [2]. A unified specification of the entire family of lens structures in Sect. 4 appears to be novel.

Triple Graph Grammars (TGG) [9,7] provide a more operational approach to BX; for example, delta-lenses can be implemented by TGG [8]. Incrementality in TGG has been also studied in [6,8].

The org-symmetry dimension has been discussed in the literature as unidirectional vs. bidirectional transformations [1,9]. We present a more fine-grained taxonomy by introducing organizational semi-symmetry. There is little related work that describes the combination of several dimensions of model synchronization and provides a formal foundation. Antkiewicz and Czarnecki's [1] is closest to ours in its intention to classify different synchronization scenarios, but deltas are not considered there, and orthogonality of the dimensions is not elaborated. We consider our work as a continuation of [1], and we are not aware of other classification work in-between.

8 Conclusions

Symmetrization of MDE, i.e., the shift from model transformation pipelines to networks of interacting models, poses several challenges for transformation tools, e.g., support of bidirectionality, incrementality, informational symmetry, and ultimately concurrent updates. Having a taxonomy of synchronization behaviors, with a clear semantics for each taxonomic unit, could help to manage these problems.

We presented a taxonomic 3D-space of model synchronization types and provided it with formal semantics. Two dimensions are computational and form a taxonomic plane classifying pairs of mutually inverse transformation operations realizing BX. The third dimension is orthogonal to the plane and classifies relationships of *organizational dominance* between the models to be kept in sync.

The space can be used to locate the type of the synchronization problem at hand; from this type, we can infer the requirements for model transformations tools and theories to be applied to the problem. We think of the space as a communication medium for tool users and tool builders, in which they can specify tool capabilities and behavior. We hope that our space could also guide future research concerning bidirectional transformations. Concurrent updates are not covered yet, although we are aware of its importance for MDE applications, and leave it for future work.

References

1. Antkiewicz, M., Czarnecki, K.: Design Space of Heterogeneous Synchronization. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE. Lecture Notes in Computer Science, vol. 5235, pp. 3–46. Springer (2007)
2. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In: Whittle et al. [12], pp. 304–318
3. Diskin, Z., Maibaum, T.S.E.: Category Theory and Model-Driven Engineering: From Formal Semantics to Design Patterns and Beyond. In: Golas, U., Soboll, T. (eds.) ACCAT. EPTCS, vol. 93, pp. 1–21 (2012)
4. Diskin, Z., Xiong, Y., Czarnecki, K.: From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. Journal of Object Technology 10, 6: 1–25 (2011)
5. Foster, J.N., Greenwald, M., Moore, J., Pierce, B., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)
6. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8, 21–43 (2009)
7. Golas, U., Lambers, L., Ehrig, H., Giese, H.: Toward bridging the gap between formal foundations and current practice for triple graph grammars: Flexible relations between source and target elements. In: ICGT'2012
8. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y.: Correctness of Model Synchronization Based on Triple Graph Grammars. In: Whittle et al. [12], pp. 668–682
9. Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars. In: ICGT. pp. 411–425 (2008)
10. Soley, R., et al.: Model Driven Architecture. OMG White Paper (2000)
11. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software and System Modeling 9(1), 7–20 (2010)
12. Whittle, J., Clark, T., Kühne, T. (eds.): Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings, Lecture Notes in Computer Science, vol. 6981. Springer (2011)