Journal of Software & Systems Modelling manuscript No.
(will be inserted by the editor)

Change Propagation and Bidirectionality in Internal Transformation DSLs

Georg Hinkel · Erik Burger

Received: June 17, 2016 / Revision: July 3, 2017/ Accepted: not yet

Abstract Despite good results in several industrial projects, Model-Driven Engineering (MDE) has not been widely adopted in industry. Although MDE has existed for more than a decade now, the lack of tool support is still one of the major problems, according to studies by Staron and Mohaghegi [52, 58]. *Internal languages* offer a solution to this problem for model transformations, which are a key part of MDE. Developers can use existing tools of host languages to create model transformations in a familiar environment. These internal languages, however, typically lack key features such as change propagation or bidirectional transformations. In our opinion, one reason is that existing formalisms for these properties are not well suited for textual languages. In this paper, we present a new formalism describing incremental, bidirectional model synchronizations using synchronization blocks. We prove the ability of this formalism to detect and repair inconsistencies and show its hippocraticness. We use this formalism to create a single internal model transformation language for unidirectional and bidirectional model transformations with optional change propagation. In total, we currently provide 18 operation modes based on a single specification. At the same time, the language may reuse tool support for C#. We validate the applicability of our language using a synthetic example with a transformation from finite state machines to Petri nets where we achieved speedups of up to multiple orders of magnitude compared to classical batch transformations.

Keywords Model-driven Engineering · Model Synchronization · Domain-Specific Language · Change Propagation · Bidirectional · Incremental

Georg Hinkel
FZI Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
E-mail: hinkel@fzi.de

Erik Burger
Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany
E-mail: burger@kit.edu

1 Introduction

Model-driven engineering (MDE) is an approach to raise the level of abstraction of systems in order to cope with increasing system complexity. Although MDE is widely adopted in academia, it is not as popular in industry, primarily because of the lack of stable tool support [52, 58]. In addition, Meyerovich et al. [51] have shown that most developers only change their primary language when either there is a hard technical project limitation or there is a significant amount of code that can be reused.

Model transformations are the "heart and soul" of MDE [57]. Since general-purpose languages are not suitable for this task [57], there is a plethora of specialized model transformation languages. This may hamper the adoption of MDE in industry, since developers may not want to use model transformation languages for the reasons found by Meyerovich.

To solve both of these issues, *internal languages* offer a promising approach: Model transformation languages can be abstracted from, and integrated into general-purpose languages. This way, tool support for the host language is inherited, and developers may stick to the languages that they are used to.[1]

[1] This does not mean that internal DSLs superior per se. One has to be very careful when designing an internal DSL to gain the benefits of the host language integration while still preserving the advantages of a model transformation language such as a

Several languages follow this approach [4, 20, 29, 41, 47, 63, 64]. We have observed that most of these languages only operate in a rather imperative way, which means that they contain less control flow abstractions than declarative model transformation languages, such as, for example, QVT-R [54]. In particular, only few approaches support bidirectional transformations, most notably the Scala DSL created by Wider [63]. To the best of our knowledge, none of these languages supports change propagation, a feature that is mostly provided by declarative languages such as Triple Graph Grammars (TGGs)[2]. For these, implementations exist that support change propagation [21, 22, 24].

A key problem when creating an internal language for model transformation that shall support bidirectionality and change propagation is the semantic difference between a typical programming language and formalisms that support bidirectionality and change propagation such as TGGs. TGGs are based on graph patterns, divided into left hand side, right hand side and correspondence part. The only implementation of graph patterns in an internal language that we are aware of, VIATRA Query, uses method calls to represent graph patterns in an object-oriented language. This entirely loses the expressions language of object-oriented languages, which we think is a pity.

Therefore, we propose a new formalism for bidirectional and incremental model synchronization that is created to take the expression language of object-oriented programming language better into account. The formalism is based on synchronization blocks that describe expressions that, based on an isomorphism, should be synchronized. The synchronization rests on the lenses approach originally introduced by Foster et al. [16]. We describe this formalism and prove its ability to repair inconsistencies between heterogeneous models in a hippocratic manner.

Further, we present *NMF Synchronizations*, an internal language in C#. Therefore, we show that the current lack of change propagation, especially combined with bidirectionality, is not a general restriction of internal languages. The language supports multi-directional model transformation as well as multiple change propagation patterns in combination. The implementation, however, has a few limitations, which we discuss in Section 8, but we believe they are only technical restrictions.

(presumably) better understandability. The latter can be more easily accounted for in a dedicated, i.e., external, language.

[2] VIATRA Query (formerly named EMF-IncQuery) does support change propagation and is implemented as an internal DSL in Xtend, but does not support bidirectional transformations.

We have validated our approach using an example transformation of Finite State Machines to Petri Nets. With our prototype language, we only have a single specification and are able to obtain 18 different model transformations.

This paper is an extension of a previous version submitted to the ICMT 2015 [31]. It extends this prior work with a formal theory how change propagation and bidirectionality are achieved and a more thorough explanation and related work section.

The rest of this paper is structured as follows: Section 2 explains our running example, the synchronization of finite state machines and Petri Nets. Section 3 introduces the foundations that our work is based on. Section 4 introduces the concept of synchronization blocks and proves some of their basic properties. Section 5 explains how synchronization blocks can be implemented in an internal DSL. Section 6 describes our prototype language through applying it to the motivational example of synchronizing finite state machines and Petri Nets. Section 7 validates the performance of our language by benchmarking our example synchronization. Section 8 shows the limitations of our approach. Finally, Section 9 lists related work before Section 10 concludes the paper.

2 Finite State Machines to Petri Nets

Throughout the paper, both to explain our approach and for validation, we use the running example of the transformation of Finite State Machines to Petri Nets, two well-known formalisms in theoretical computer science. Both of them are well suited to describe behaviors but each of them has its advantages. Therefore, both of them are widely used. Finite state machines can be easily transformed to Petri nets.

However, for model synchronization, the example of Finite State Machines and Petri Nets is a rather synthetic one as usually only one of these formalisms is used. We use it as our running example though as the involved metamodels are rather simple and structurally similar but yet different. Real application scenarios would rather center on the synchronization of artifacts like the source code, architecture information in UML diagrams and potentially performance engineering models such as the Palladio Component Model (PCM) [5].

The metamodel that we use for finite state machines is depicted in Figure 1. Finite state machines consist of states and transitions where transitions hold a reference to the incoming and outgoing states and states hold a reference to the incoming and outgoing transitions. States can be start or end states.

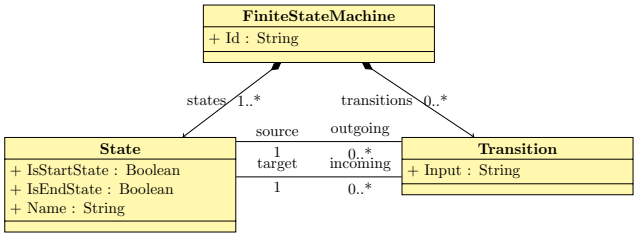


Figure 1 The metamodel for finite state machines

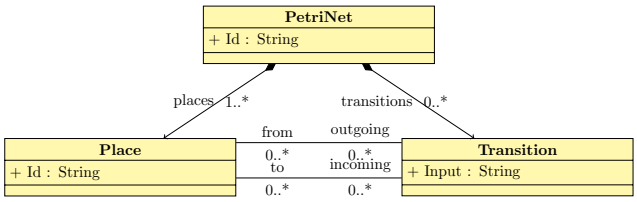


Figure 2 The metamodel for Petri Nets

The metamodel of Petri Nets is depicted in Figure 2. Petri Nets consist of places and transitions. Unlike state machines where states are modeled explicitly, the state of a Petri Net is the allocation of tokens in the network.

The transformation from finite state machines to Petri Nets transforms each state to a place. Transitions in the finite state machine are transformed to Petri Net transitions with the source and target places set accordingly. Final states are transformed to a place with an outgoing transition that has no target place and therefore 'swallows' tokens.

An example of this transformation is illustrated in Figure 3, where the state machine to manage the lifecycle of a simulation is depicted. The advantage of a Petri net is here that using tokens, Petri nets allow to represent the state of multiple simulations in the same diagram.

The backward transformation from Petri Nets to finite state machines is not always well defined since Petri Net transitions may have multiple source or target places. However, if the Petri Net is an image of a finite state machine under the above transformation, then the backward transformation is useful to have.

3 Foundations

Our approach is a bridge between technologies that already exist. We combine and adapt a model transformation framework with a framework for incremental computation and enrich this framework with lenses 3 to make bidirectionality possible. Thus, we briefly introduce both the model transformation framework *NMF Transformations*, the framework for incremental computation *NMF Expressions*, its foundation in category theory and the theory of lenses that we require for bidirectionality.

3.1 NMF Transformations

NMF stands for .NET Modeling Framework [32] and is an open-source project to support MDE on the .NET platform. NMF Transformations [29] is a sub-project of NMF that supports model transformation. It consists of a model transformation framework and internal DSL for C# on top of it (NMF Transformations Language, NTL). Both framework and DSL are inspired by the transformation languages QVT [54] and ATL [43] but work with arbitrary .NET objects. The language has been applied internally in NMF and at the Transformation Tool Contest (TTC) in 2013 [35, 36].

Figure 4 shows an excerpt of NMF Transformations' abstract syntax. Model transformations consist of transformation rules that are represented as public nested classes of a model transformation class in NTL. A transformation rule represents how a particular model element should be transformed. NTL allows multiple transformation rules to operate on the same model element type(s). Transformation rules can have dependencies specifying which other transformation rules should be called if the transformation is executed for given model elements. These dependencies may contain selectors, filters and persistors which are called to register the dependent model elements on the target. NTL distinguishes between single dependencies that only demand calling the dependent transformation rule for a single other tuple of model elements or multiple dependencies that may execute the dependent transformation rule several times for different inputs. In NTL, these dependencies are specified using special method calls where function typed attributes of the dependencies like selectors, filters or persistors are specified as lambda expressions.

Because NMF Transformations operates independently of containment hierarchies, the structure of the model transformation is entirely encoded in the transformation dependencies. The idea is that the transformation rules specify locally what other elements should be transformed and whether they should be transformed before the current transformation rule. The transformation engine then resolves these dependencies and executes all computations when their dependencies are met. The rules themselves are imperative with an access to the trace, i.e. to all correspondences that have been found so far. In NTL, the rule body is specified as an overridden method that takes the input and output model element of the transformation rule as well as a

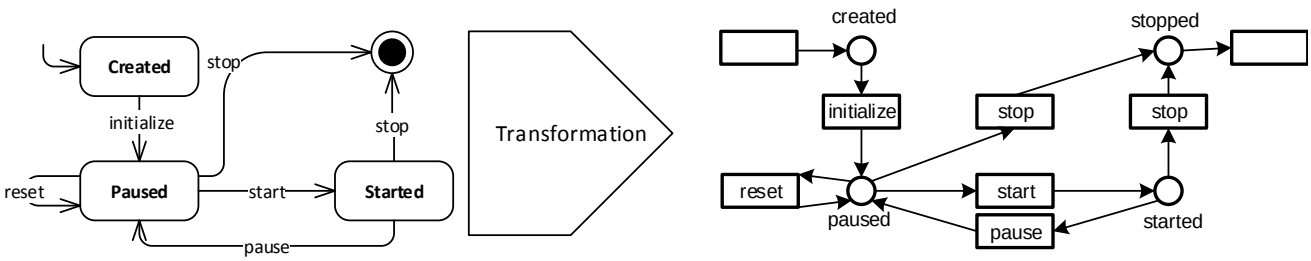


Figure 3 Illustration of the considered example transformation from finite state machines to Petri nets exemplified for a simulation lifecycle.

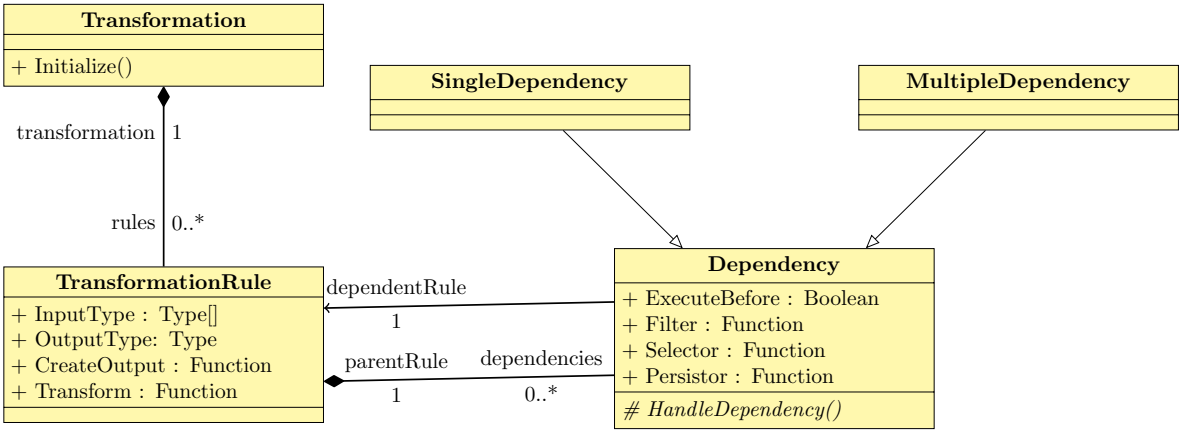


Figure 4 Abstract syntax of NMF Transformations

transformation context which can be used to query the trace.

3.2 A very brief introduction to Category Theory

The goal of this section is to briefly introduce most of the category theory that is used in this paper for reference purposes. The concepts are not explained in depth as suitable explanations can be obtained from many textbooks on category theory. For the interested reader, we would recommend the books by Lawvere [49] or Crole [11]. The book by Lawvere is a good general introduction, whereas Crole's book is more focused on applications to algebraic type theories. The latter is also the original source for the definitions in this section.

Definition 1 (Category) A category $\mathcal{C}$ consists of a collection $ob\mathcal{C}$ of objects and collections of morphisms between objects of $\mathcal{C}$ equipped with an associative operator $\circ$. Furthermore, for each object A, the identity id_A must exist and for each $f \in Mor(A, B)$ it must hold that $f \circ id_A = f = id_B \circ f$.

For given objects $A, B \in \mathcal{C}$, the set of morphisms is denoted as $Mor_{\mathcal{C}}(A, B)$ or simply $Mor(A, B)$ if $\mathcal{C}$ is clear from the context. The collection of all morphisms in $\mathcal{C}$ is denoted as Mor with mappings $source, target : Mor \to ob\mathcal{C}$ determining the source and target object of a morphism.

Remark 1 The associativity means that for any $f \in Mor(A, B), g \in Mor(B, C), h \in Mor(C, D)$ where A, B, C, $D \in \mathcal{C}$ that $(h \circ g) \circ f = h \circ (g \circ f)$.

Example 1 (Sets) One of the most important categories is the category $\mathcal{S}$ of sets. Here, the morphisms are the mappings between sets and the identity for a given set A is the identity mapping on A.

Remark 2 In category theory, equations are often visualized as graphs. Here, objects of a category form the vertices of the graph whereas the directed edges are the morphisms between the objects. The terminology that such a diagram commutes equals to saying that following either path through the diagram yields the same result.

Definition 2 (Functor) A (covariant) functor $\mathcal{F} : \mathcal{C} \to \mathcal{D}$ between categories $\mathcal{C}$ and $\mathcal{D}$ is a mapping between the objects of $\mathcal{C}$ and $\mathcal{D}$ and the morphisms such that for each objects A and B and $f \in Mor(A, B)$ in $\mathcal{C}$, we have that $\mathcal{F}(f) \in Mor(\mathcal{F}(A), \mathcal{F}(B))$. Further, a functor has to respect composition, i.e. if $f : A \to B$ and $g : B \to C$, then it must hold that $\mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f)$ and $\mathcal{F}(id_A) = id_{\mathcal{F}(A)}$ in $\mathcal{D}$ and for each object A in $\mathcal{C}$.

Example 2 (Identity functor) An important functor is the identity functor $id_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ for a category $\mathcal{C}$ that maps each object $A \in \mathcal{C}$ to itself and likewise each mapping $\phi \in Mor(A, B)$ to itself.

Example 3 There are three prominent collection functors on $\mathcal{S}$:

1. The powerset functor $\mathbb{P} : \mathcal{S} \to \mathcal{S}$ sends each set to its powerset and for each morphism $f : A \to B$ we have that

   $$\mathbb{P}(f) : \mathbb{P}(A) \to \mathbb{P}(B), S \mapsto f(S) := \{f(s) | s \in S\}.$$

2. The multiset functor $\mathbb{M} : \mathcal{S} \to \mathcal{S}$ sends each set S to the set of multisets with elements of S, i.e. to a function $S \to \mathbb{N}_0$ that assigns each element a multiplicity in the multiset. A morphism $f : A \to B$ is mapped to

   $$\mathbb{M}(f) : \mathbb{M}(A) \to \mathbb{M}(B), m \mapsto (b \mapsto \sum_{a \in f^{-1}(\{b\})} m(a)).$$

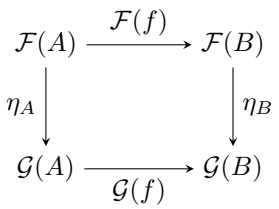
3. The Kleene closure $* : \mathcal{S} \to \mathcal{S}$ maps each set A to its Kleene closure $A*$, which is the monoid of finite sequences of elements of A. A morphism $f : A \to B$ is mapped to

   $$*(f) : A* \to B*, (a_1; \ldots; a_n) \mapsto (f(a_1); \ldots; f(a_n)).$$

These three functors can be seen as a formalization of collections.

Remark 3 Functors are the 'natural' mapping constructs between categories. This is because indeed, the collection of categories forms the category $\mathcal{C}at$ where the morphisms between categories $\mathcal{C}$ and $\mathcal{D}$ (which are themselves objects of $\mathcal{C}at$) are the functors $\mathcal{F} : \mathcal{C} \to \mathcal{D}$.

Definition 3 (Natural transformation) A natural transformation $\eta : \mathcal{F} \to \mathcal{G}$ between two functors $\mathcal{F}, \mathcal{G} : \mathcal{C} \to \mathcal{D}$ is a set of mappings $\eta_A \in Mor(\mathcal{F}(A), \mathcal{G}(A))$ for each $A \in \mathcal{C}$ (called components of η) such that for each $A, B \in \mathcal{C}$ and $f \in Mor(A, B)$ it holds that $\eta_B \circ \mathcal{F}(f) = \mathcal{G}(f) \circ \eta_A$. That is, the following diagram commutes:



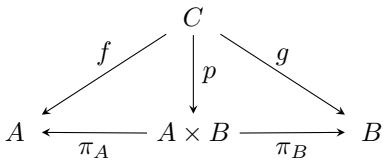
If all η_A are isomorphisms, η is called a natural isomorphism between $\mathcal{F}$ and $\mathcal{G}$.

Example 4 An important example of a natural transformation between functors is the identity transformation on a given functor $\mathcal{F}$. For each object A in $\mathcal{C}$, the transformation component for A is simply the identity, i.e. $(id_{\mathcal{F}})_A = id_{\mathcal{F}(A)}$.

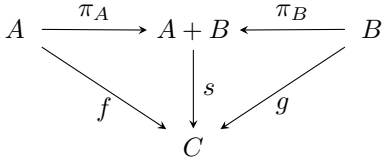
Definition 4 (Monad) A monad $\mathcal{T} : \mathcal{C} \to \mathcal{C}$ is a functor equipped with two natural transformations $\eta : id_{\mathcal{C}} \to \mathcal{T}$ and $\mu : \mathcal{T}^2 \to \mathcal{T}$ such that $\mu \circ \mathcal{T}\mu = \mu \circ \mu\mathcal{T}$ and $\mu \circ \mathcal{T}\eta = \mu \circ \eta\mathcal{T} = id_{\mathcal{T}}$. Here, the natural transformation η 'lifts' objects into the monad and is thus sometimes called the unit operation, while μ simplifies a nested monad.

Example 5 A well known example of a monad are collections. Here, the functor maps each type A to a generic collection of type A. The functor application of a function corresponds to a mapping. The natural transformation η treats an item of type A as a collection of type A that just contains this element, while μ flattens a collection of collections of type A into a collection of type A.

Definition 5 (Product, Sum) Let A and B be objects of a category $\mathcal{C}$. The product of A and B in $\mathcal{C}$ is an object $A \times B$ of $\mathcal{C}$ together with two projection morphisms $\pi_A : A \times B \to A$ and $\pi_B : A \times B \to B$ such that for every object C and every pair of morphisms $f : C \to A$ and $g : C \to B$, there is a unique morphism $p : C \to A \times B$ such that $f = \pi_A \circ p$ and $g = \pi_B \circ A$. That is, the following diagram commutes:

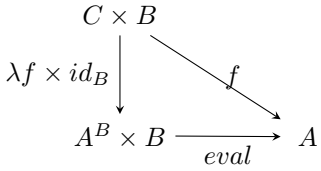


A sum of objects A and B in $\mathcal{C}$ simply is the product of A and B in $\mathcal{C}^{op}$. That is, it is an object $A + B$ together with two morphisms $\imath_A : A \to A + B$ and $\imath_B : B \to A + B$ such that for every object C and every pair of morphisms $f : A \to C$ and $g : B \to C$, there is a unique morphism $s : A + B \to C$ such that $f = s \circ \imath_A$ and $g = s \circ \imath_B$. That is, the following diagram commutes:



Definition 6 (Exponential) Let $\mathcal{C}$ be a category such that for each objects A and B their product exists. Then the exponential of A and B is an object A^B together

with a morphism $eval : A^B \times B \to A$ such that for any morphism $f : C \times B \to A$, there is a unique morphism $\lambda f : C \to A^B$ such that for every $c \in C$ and $b \in B$, $f(c, b) = eval(\lambda f(c), b)$. That is, the following diagram commutes:



Definition 7 (Initial object, terminal object) An initial object $\perp$ of a semicategory $\mathcal{C}$ is an object such that for every object A in $\mathcal{C}$, there exists exactly one morphism from $\perp$ to A.

Conversely, a terminal object $\top$ of a semicategory $\mathcal{C}$ is an object such that for every object A in $\mathcal{C}$, there exists exactly one morphism from A to $\top$.

An initial object of $\mathcal{C}$ is a terminal object of $\mathcal{C}^{op}$ and vice versa. Initial and terminal objects are unique up to isomorphism, i.e. if A and B are initial objects of the same (semi-)category, then there is an isomorphism from A to B.

Example 6 In the category $\mathcal{S}$ of sets, the initial object is the empty set. The terminal objects are the sets that contain exactly one element.

Definition 8 A category $\mathcal{C}$ is called cartesian-closed, if it satisfies the following properties:

- It contains a terminal object (unique up to isomorphism)
- For any objects A and B, the product $A \times B$ exists and is an object of $\mathcal{C}$.
- For any objects A and B, the exponential A^B exists and is an object of $\mathcal{C}$.

3.3 Incremental computation

Self-adjusting or incremental computation refers to the idea to adjust a computation using dependency tracking rather than recomputing the whole computation when the input data changes. This is done by modifiable references and a system that creates a dynamic dependency graph based on these [1]. Further research has shown that such self-adjusting programs can be implicitly inferred from a batch specification [10]. That means, from an expression $x + y$ where x and y are modifiable references, a dynamic dependency graph is built where x and y are nodes. Each node holds its current value. In this situation, the system builds a new node for $x+y$ holding a reference to both x and y so that the sum changes as soon as either x or y change. Creating a self-adjusting program from a traditional (batch) specification is possible for purely functional programs [10] since they do not contain side effects. However, approaches for imperative languages exist as well [2, 25] but are not working implicitly.

As explored already by Carlsson [9], the modifiable references that make up incremental computation can be described with a monad in Haskell. Monads in Haskell originate from monads in category theory which may be used to describe algebraic type systems [11]. However, the unit function of incrementalization is not natural: It does matter whether a computation is made statically or incrementalized as only the latter will produce incremental updates. Therefore, incrementalization only is a functor in the terminology of category theory.

Our interpretation is based on mutable references, so an instance $a \in A$ of some type A refers only to its identity while its attributes and references may change over time. We do not model the state per class but only keep a global model state Ω, inspired from stochastics.

One of the merits of category theory is that it often does not require an in-depth understanding of the inner structure of objects but rather reasons on their behavior, i.e. the value or the uniqueness of certain morphisms. This is useful for us, because it enables a formalization at a very high level of abstraction that yields a good flexibility in a later implementation. In particular, we do not make any assumptions on the structure of the state space Ω or a given type $A \in \boldsymbol{T}$ except that there is a relation that checks whether a given object has a certain type. We use the element notation $a \in A$ to depict that an object a is an instance of A.

An attribute or reference of A is modeled as a function $f : A \times \Omega \to B$. The reason for a global state is that changes to attributes or references usually may have side-effects to other attributes or references, for example if there is an opposite reference. If one is to set a reference that has an opposite reference, implicitly also the opposite reference is set. We call this system a mutable type category:

Definition 9 (Mutable Type Category) A mutable type category $\mathcal{C}$ for a set of types $\boldsymbol{T}$ and a state space Ω is a cartesian-closed category that consists of tuples $ob\ \mathcal{C} := \{A \times \Omega \mid A \in \boldsymbol{T}\}$ as objects and morphisms $Mor(A \times \Omega, B \times \Omega)$ between two types A and B as functions $A \times \Omega \to B \times \Omega$.

Definition 10 (Notation) In the remainder of the paper, we use a slightly simplified notation where we write $f : A \to B$ for $f \in Mor(A \times \Omega, B \times \Omega)$. We also say that $A \in \mathcal{C}$ to denote that $A \times \Omega \in ob\ \mathcal{C}$ if it is

clear that A refers to a type. Further, a functor $\mathcal{I}$ applied to a given object $A \times \Omega$ in $\mathcal{C}$ must be an object $\mathcal{I}(A \times \Omega) = A' \times \Omega$. We notate this type $A' \in \boldsymbol{T}$ as $A' = \mathcal{I}(A)$ such that $\mathcal{I}(A \times \Omega) = \mathcal{I}(A) \times \Omega$.

We refer to changes of the global state as set-theoretic functions $\Delta\omega \in \Delta\Omega := \Omega \to \Omega$. We are interested in the incrementalization of side-effect free morphisms as per the following definition:

Definition 11 (Side-effect free morphisms) A morphism $f : A \to B$ in a mutable type category $\mathcal{C}$ is said to be side-effect free if and only if for all $a \in A$ and $\omega \in \Omega$, we have that

$$\pi_\Omega(f(a, \omega)) = \omega$$

where we denote π_Ω as the set-theoretic projection to the state.

Remark 4 It is clear that compositions of side-effect free morphisms are side-effect free and therefore the category $\mathcal{C}_\Omega$ with objects $obj\ \mathcal{C}_\Omega = obj\ \mathcal{C}$ and morphisms $Mor_{\mathcal{C}_\Omega}(A, B) = \{f \in Mor_{\mathcal{C}}(A, B) | f \text{ is side-effect free}\}$ is a subcategory of $\mathcal{C}$.

To define an incrementalization system formally, we need a last proposition as follows:

Proposition 1 *Let $\mathcal{C}$ be a mutable type category and $\mathcal{I}$ an endofunctor on $\mathcal{C}_\Omega$. Then, the point-wise tuple $\mathcal{I} \times \Delta\Omega$ that consists of objects $\mathcal{I}(A) \times \Delta\Omega$ and morphisms $(f, \Delta\omega)$ where A is a type in $\mathcal{C}_\Omega$ and f a morphism in $\mathcal{C}_\Omega$, is a category. The concatenation operator $\circ$ is defined element-wise. Further, the mappings $\mathcal{F}(A) : \mathcal{I}(A) \mapsto \mathcal{I}(A) \times \Delta\Omega$ and $\mathcal{F}(f) : \mathcal{I}(f) \mapsto (\mathcal{I}(f), Id_\Omega)$ form a functor. We identify this functor with the assigned category if this is clear from the context.*

Definition 12 (Incrementalization System) An incrementalization system over a mutable type category $\mathcal{C}$ is an endofunctor $\mathcal{I} : \mathcal{C}_\Omega \to \mathcal{C}_\Omega$ equipped with four transformations $\eta : Id \to \mathcal{I}$, $\mu : \mathcal{I}^2 \to \mathcal{I}$, $value : \mathcal{I} \to Id$ and $apply : \mathcal{I} \times \Delta\Omega \to \mathcal{I}$ from which $\mu, value$ and $apply$ are natural and the following conditions hold:

$$\begin{aligned} \mu \circ \mathcal{I}\mu &= \mu \circ \mu\mathcal{I}, \\ \mu \circ \mathcal{I}\eta &= \mu \circ \eta\mathcal{I} = id_\mathcal{I}, \\ value \circ \eta &= Id_\mathcal{C} \\ apply \circ (Id_\mathcal{I}, Id_\Omega) &= Id_\mathcal{I}. \end{aligned}$$

Remark 5 For a given type A, the *value*-transformation shall return the current value of a modifiable reference $a \in \mathcal{I}(A)$. The *apply*-transformation applies a given state change to an incremental value. Whether this works by recomputing the value from scratch or performing an inexpensive propagation is left as an implementation detail to the incrementalization system $\mathcal{I}$. The transformation η plays the role of an elevation of a given instance of a type A to a constant of that type, i.e. an incremental value whose value as per *value* transformation never changes after a state change has been applied. In the running example, it could be used to fix a certain state machine instance as object of consideration. The transformation μ is required to simplify nested modifiable references, for example a name reference of a modifiable state reference.

The first two validity constraints mean that an incrementalization system actually is a monad with the slight exception that η does not have to be natural as this would be too restrictive: If a result of a computation is elevated to a constant, this must not yield the same result as performing the computation incrementally on constant arguments - the latter may change due to state changes as well.

The *value*-transformation can (and should) be natural so that there is no difference in the result whether the current value of a modifiable reference is processed incrementally or not, besides that an incremental processing also means that the result are refreshed upon a state change.

The last two constraints mean that the value of a constant should always be the original instance the constant was created from. The last constraint implies that if no changes are made to the global state, modifiable references must not change.

Proposition 2 *The value of an updated modifiable reference matches the case where the value would have been recomputed from scratch, i.e. for each $\Delta\omega \in \Delta\Omega$, $f : A \to B$, $\omega \in \Omega$ and $a \in A$, we have that*

$$f(a, \omega) = value(\mathcal{I}(f)(\eta(a), \omega)) \quad \text{and} \tag{1}$$
$$f(a, \Delta\omega(\omega)) = value(apply(\mathcal{I}(f)(\eta(a), \omega), \Delta\omega)). \tag{2}$$

Proof This proposition mainly is a consequence from the naturality of *value* and *apply*.

The key benefit of the incremental approach is that the right side of Equation 2 knows the function f before a model change $\Delta\omega \in \Delta\Omega$ is applied such that the incrementalization system may insert and manage buffer memories to accelerate the recomputation of f, for example through a dynamic dependency graph. For the correctness, only the naturality of *value* and *apply* have to be checked, which can be done for each atomic morphism individually. The naturality for composite morphisms follows by stacking together the commutative diagrams.

Therefore, the key application of the incrementalization system $\mathcal{I}$ is to map any function $f : A \to B$ to the function, i.e. a function $\mathcal{I}(f) : \mathcal{I}(A) \to \mathcal{I}(B)$.

In this paper, we use an implementation of these ideas within the NMF project, NMF Expressions[3], applied to the Train Benchmark case at the TTC 2015 [39]. This approach is suitable for our needs as it contains dedicated collection support and is likewise implemented as an internal DSL for C# and therefore suitable to combine it with NMF Transformations. Furthermore, unlike [10] it does not operate on the source code and therefore can be used in a compiling environment. NMF Expressions operates on CLR classes that implement the .NET platform default notification interfaces, similar to the EMF Notification API. For a metamodel, such classes can be generated.

3.4 Lenses

Lenses are a an algebraic construct originally introduced by Foster et al. [16] to solve the view-update-problem. They operate on a set of tree structures $\mathcal{V}$ and are able to compute updates to the original tree when views have changed. In the remainder of this section, we briefly introduce this concept and the most relevant definitions from [16].

Definition 13 (Lens) A *lens* l is a pair of two partial functions $l \nearrow: \mathcal{V} \rightharpoonup \mathcal{V}$ called the GET-function of l and $l \searrow: \mathcal{V} \times \mathcal{V} \rightharpoonup \mathcal{V}$ called the PUT-function of l. The intuition is that $l \nearrow$ computes a view on an element while $l \searrow$ applies changes to the view back to the original element.

Definition 14 (Well-behavedness) Let C and A be subsets of $\mathcal{V}$. A lens is called well-behaved and total from C to A, if it maps arguments of C to results of A ($l \nearrow (C) \subset A$ and $l \searrow (A \times C) \subset C$) and complies with the following laws:

$$l \searrow (l \nearrow (c), c) = c \quad \text{for all } c \in C \qquad \text{(GetPut)}$$
$$l \nearrow (l \searrow (a, c)) = a \quad \text{for all } (a, c) \in A \times C. \qquad \text{(PutGet)}$$

Intuitively, these laws state that when no modification is performed in the view, the PUT function should not modify the original element; otherwise it should store this information such that recomputing the view would not change the results.

Definition 15 The composition operator ; puts two lenses l and k in sequence:

$$(l; k) \nearrow: c \mapsto k \nearrow (l \nearrow (c))$$
$$(l; k) \searrow: (a, c) \mapsto l \searrow (k \searrow (a, l \nearrow (c)), c).$$

[3] http://github.com/NMFCode/NMF/tree/master/Expressions/

Proposition 3 *The composition $l; k$ of a well-behaved total lens l from A to B and a well-behaved, total lense k from B to C is a well-behaved, total lens from A to C.*

Proof The proof can be found in [16].

4 Model Synchronization with Synchronization Blocks

In this section, we present our model synchronization approach. First, we introduce the underlying synchronization theory our implementation is built upon. We then explain synchronization blocks as the synchronization primitives of our approach and how these primitives are composed to model synchronizations.

4.1 Combining Bidirectionality and Change Propagation

To combine incrementality and bidirectionality, one must find a suitable formalization able to describe both of them. On the one hand, we have incrementality which can be described by monads as manifested by Carlsson [9], on the other hand, we have bidirectionality, where Foster et al. have proposed the lenses approach [16]. In this section, we detail on our approach for such a common formalization through incremental lenses.

To do that, we use that both approaches can be described in terms of category theory. However, there is an important difference of what entities are modeled in the categories: While the objects in Croles categories reconstructed from algebraic type theory are types, lenses often consider entire models as objects [13].

There have been many different versions of lenses with close correlations [42]. Diskin, Xiong and Czarnecki argue that the original lenses have problems as they do not know the change sequence and propose delta-lenses as a solution [14]. However, this problem only arises when the differences between two states is not distinct as shown by Johnson and Rosebrugh [42]. As Diskin uses categories where objects represent entire models[4] [13], deltas require model differencing, which in general has no unique solution.

In our case, the objects of the category consist of identities of model elements or simple values. Thus, the differencing is easy and unique: A change sequence is uniquely described by the previous model element identity and the new identity. Therefore, state-based lenses suffice.

[4] also referred to as model-at-a-time, as opposed to object-at-a-time

Therefore, we have to adjust the notion of lenses to our situation a bit and apply it to the categorical type system introduced in Definition 9.

Definition 16 (In-model Lenses) In this interpretation, a (well-behaved) in-model lens (m-lens) $l : A \hookrightarrow B$ between types A and B of a mutable type category $\mathcal{C}$ consists of a side-effect free GET morphism $l \nearrow\colon A \to B$ and a morphism $l \searrow\colon A \times B \to A$ called the PUT function that satisfy the following conditions for all $a \in A$ and $\omega \in \Omega$:

$$l \searrow (a, l \nearrow (a)) = (a, \omega)$$
$$l \nearrow (l \searrow (a, c, \omega)) = (a, \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega.$$

Here, the object associated with the tuple type $A \times B$ in $\mathcal{C}_\Omega$ is the space $A \times B \times \Omega$, because morphisms in $\mathcal{C}_\Omega$ are stateless. Because the objects of $\mathcal{C}$ and $\mathcal{C}_\Omega$ are the same, this type also exists in $\mathcal{C}$. After all, it does not make sense to consider tuples where the elements live in different global states.

The first condition is a direct translation of the original PutGet law. Meanwhile, the second line is slightly weaker than the original GetPut because the global state may have changed. In particular, we allow the PUT function to change the global state.

Example 7 The most important lenses that we will be working with are property accesses. Here, the GET morphism is the evaluation of the property for a given model element in a given state. The PUT morphism changes the global state such that the property then points to the new value for the given model element. In most modeling environments, this may also have side-effects such as setting the opposite reference or resetting stale references in case a model element has been deleted (for example by setting its parent reference to `null`).

Definition 17 (Stateless morphism) A side-effect free morphism $f : A \to B$ for some types A and B in a mutable type category $\mathcal{C}$ is stateless, if for all $a \in A, \omega, \tilde{\omega} \in \Omega$, we have that $\pi_B(f(a, \omega)) = \pi_B(f(a, \tilde{\omega}))$, i.e., f ignores the state.

Proposition 4 *There is a composition operator $\circ$ that maps an in-model lens $f : A \hookrightarrow B$ and an in-model lens $g : B \hookrightarrow C$ to a combined in-model lens $(g \circ f) : A \hookrightarrow C$ if $g \nearrow$ is stateless by the following definition:*

$$(g \circ f) \nearrow : (a, \omega) \mapsto g \nearrow (f \nearrow (a, \omega))$$
$$(g \circ f) \searrow : (a, c, \omega) \mapsto$$
$$f \searrow (a, g \searrow (\pi_B(f \nearrow (a, \omega)), c, \omega)).$$

Note that in the last line, the result of $g \searrow$ expands to two arguments of the morphism $f \searrow$.

Proof $(g \circ f) \nearrow$ is side-effect free as concatenation of side-effect free morphisms. Let $a \in A, c \in C$ and $\omega \in \Omega$. We first proof GetPut:

$$\begin{aligned}
&(g \circ f) \searrow (a, (g \circ f) \nearrow (a, \omega)) \\
&\quad = f \searrow (a, g \searrow (\pi_B(f \nearrow (a, \omega)), (g \circ f) \nearrow (a, \omega))) \\
&\quad = f \searrow (a, g \searrow (\pi_B(f \nearrow (a, \omega)), g \nearrow (f \nearrow (a, \omega)))) \\
&\quad = f \searrow (a, f \nearrow (a, \omega)) = (a, \omega).
\end{aligned}$$

Here, $(g \circ f) \nearrow$ is side-effect free as a concatenation of side-effect free morphisms and therefore does not change the state. Then, we first applied GetPut for g and then for f.

To see PutGet, we note that

$$\begin{aligned}
&(g \circ f) \nearrow ((g \circ f) \searrow (a, c, \omega)) \\
&\quad = g \nearrow (f \nearrow ((g \circ f) \searrow (a, c, \omega))) \\
&\quad = g \nearrow (f \nearrow (f \searrow (a, g \searrow (\pi_B(f \nearrow (a, \omega)), c, \omega)))) \\
&\quad = g \nearrow (f \nearrow (f \searrow (a, g \searrow (\pi_B(f \nearrow (a, \omega)), c, \omega)))) \\
&\quad = g \nearrow (\pi_C(g \searrow (\pi_B(f \nearrow (a, \omega)), \omega)), \tilde{\omega}) \\
&\quad = (c, \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega.
\end{aligned}$$

Here, we first applied PutGet for f, but the problem is that $f \searrow$ may change the state from whatever $g \searrow$ returned to some state $\tilde{\omega}$. Because we do not know anything about $\tilde{\omega}$, we have to demand that $g \nearrow$ returns the same result regardless of the global state. As we have that, we know that $g \nearrow$ returns the same result as in the case of the PutGet of g and since we know that $g \nearrow$ is side-effect free, we even know that the final state is $\tilde{\omega}$.

Remark 6 The composition operator $\circ$ is closely related to ;, with the exception of the parameter order. In category, it is common to read $g \circ f$ as 'g after f', whereas the original lens concatenation $(f; g)$ means 'f, then g', which intuitively is the same.

Example 8 An example of lenses where the GET morphism is stateless are arithmetic operations, because the information what has changed is already encoded in the reference to the number. Consider for example the lens $+c : \mathbb{Z} \to \mathbb{Z}$ given by $+c \nearrow\colon (i, \omega) \mapsto (i + c, \omega)$ and $+c \searrow\colon (i, j, \omega) \mapsto (j - c, \omega)$ for some $c \in \mathbb{Z}$. Informally, the lens simply adds a constant number.

Example 9 An example of an operation beyond arithmetics is `FirstOrDefault` that returns the first item of a collection or the default value of a type (`null` for a reference type and zero for numeric types) if the collection is empty. If we were to assign $x.FirstOrDefault() = y$, i.e., evaluate $FirstOrDefault \searrow (x, y, \omega)$, we can distinguish the following cases:

1. The collection x contains y and y is the first element. In this case, we do not have to change x since the assignment is already satisfied.
2. The collection x contains y but not as the first element (if the collection is ordered). In this case, we have multiple options. We could either move y to be the first element (matching the semantics of getting the literally first element) or leave the collection unchanged (with the semantics of getting any element e.g. in an unordered collection[5]). This is because a single functional implementation can implement multiple semantics that need different reversability behaviors.
3. The collection x does not contain y. In this case, we add y to the collection x. We can either add it as first element if x is an ordered collection or add it to x at all, if x is unordered.
4. The element y is the element type default value. In this case we again have multiple options. In our implementation we clear the collection x.

The main learning point from this example is that the same operational implementation of an operator can match multiple lens semantics. In the example of `FirstOrDefault`, we have two versions (with different names) realizing the two options in case 2. On the other hand, this limits the possibility for implicitly inferring a reversibility semantics from existing code since there we do not know how a particular operator has been used. Thus, we decorate each operator with its reversability behavior explicitly.

Example 10 An example that breaks PutGet for composed lenses is the following: Consider a very simple metamodel of old-fashioned relationships depicted in Figure 5. It consists only of two classes `Man` and `Woman` that have a bidirectional reference to each other. A man may or may not have a wife and a woman may or may not have a husband.

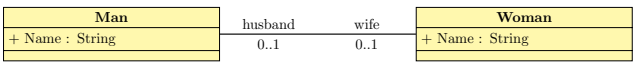


Figure 5 A simple metamodel of men and women

In most metamodeling languages such as Ecore or NMeta, such a bidirectional reference is modeled as two separate references with a set opposite. This means, as soon as the developer sets this reference, implicitly also the opposite reference is set. We refer to these references as lenses *wife* and *husband*. We will consider their concatenation $(wife \circ husband)$.

Now consider the example instance depicted in Figure 6.

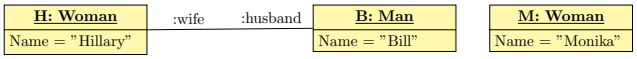


Figure 6 An example instance of men and woman at state ω_0

The example instance consists of three model elements H, B and M. In state ω_0, H is the wife of B and conversely, B is the husband of H. M has no husband. We want to see whether PutGet holds for the tuple (H, M, ω_0).

For this, we first have to evaluate $(wife \circ husband) \searrow (H, M, \omega_0)$. Because $wife \searrow$ will keep the identity of the model element it is based on, it will return B, but change to a new state:

$$
\begin{aligned}
&(wife \circ husband) \searrow (H, M, \omega_0) \\
&\quad = husband \searrow (H, wife \searrow (\underbrace{husband \nearrow (H, \omega_0)}_{B}, M, \omega_0)) \\
&\quad = husband \searrow (H, B, \omega_1).
\end{aligned}
$$

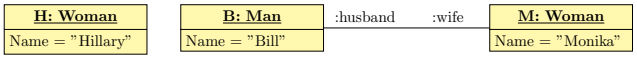


Figure 7 An example of men and women at state ω_1

This new global state is depicted in Figure 7. As a side-effect of $wife \searrow$, also the reference *husband* has changed both for H and for M: Because *wife* has a maximum cardinality of 1, H is no longer a *wife* of B which in turn resets the *husband* reference. On the other hand, M now is a wife of B and therefore the *husband* reference is set appropriately.

If we go on and evaluate $husband \searrow (H, B, \omega_1)$, this again sets the *wife* reference of B because it is an opposite of the *husband* reference. Because *wife* still has a maximum cardinality of 1, M is no longer a *wife* of B and we finally arrive back in state ω_1.

The problem is now that in state ω_0, we have that $(wife \circ husband) \nearrow (H, \omega_0) = (H, \omega_0)$ whereas PutGet would demand this to be M. Even worse, because *wife* and *husband* are opposite references, there must not be a state $\omega \in \Omega$ such that $(wife \circ husband) \nearrow (H, \omega) = (M, \omega)$. In particular, evaluating $(wife \circ husband) \nearrow$ for H in state ω_1 even throws an exception because $husband(H, \omega_1)$ returns a null-reference.

[5] Due to the slight difference in the semantics for ordered and unordered collections, one could also separate the two of them in different methods. Our counter-arguments are that this increases the size of the API. Furthermore, it is often not clear statically if a collection is ordered.

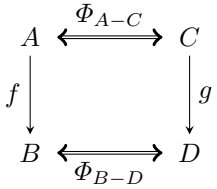


Figure 8 Schematic overview of synchronization blocks

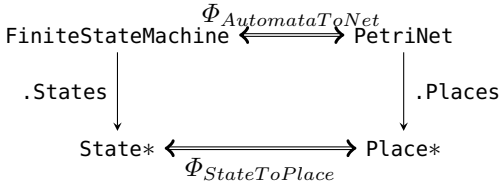


Figure 9 Synchronization of the states of a finite state machine with the places of a Petri net

Remark 7 We do not yet have a clear criterion to automatically decide whether a given GET morphism is stateless or not. Furthermore, the fact that a given GET morphism is not stateless does not immediately imply that the resulting pair of morphisms breaks PutGet. Therefore, our implementation currently assumes that the developer is aware.

4.2 Synchronization Blocks

Before we describe synchronization blocks, we need a small further definition.

Definition 18 A lens $l : A \hookrightarrow B$ is called persistent if for all $a \in A, b \in B$ and $\omega \in \Omega$, we have that $\pi_A(l \searrow (a, b, \omega)) = a$. This means that the PUT operation may only change the state, but not the identity.

Example 11 A property access is a persistent lens as we have shown in previous examples.

Proposition 5 *Let $f : A \hookrightarrow B$ a persistent lens and $g : B \hookrightarrow C$ a lens such that $(g \circ f)$ fulfills the PutGet law and therefore is a lens. Then, $(g \circ f)$ is persistent.*

Proof The proof follows straight from the definition of $(g \circ f) \searrow$.

The very basic idea behind our approach is to describe the correspondence between elements of heterogeneous models through *isomorphisms* that are incrementally build up during a synchronization through *synchronization blocks* as in the following definition:

Definition 19 (Synchronization Block) A (single-valued) synchronization block $\boldsymbol{S}$ is an octuple (A,B,C, D,Φ_{A-C},Φ_{B-D},f,g) of four types with two isomorphisms and two persistent lenses. Here, A, B, C and D are types for which a correspondence isomorphism Φ_{A-C} is defined between A and C and likewise Φ_{B-D} between B and D. The types A and B originate from a mutable type system $\mathcal{C}_L$, meanwhile C and D originate from a type system $\mathcal{C}_R$. We further have persistent lenses $f : A \hookrightarrow B$ and $g : C \hookrightarrow D$ in their respective type systems $\mathcal{C}_L$ and $\mathcal{C}_R$ to navigate through the models.

A schematic overview of a synchronization block is depicted in Figure 8. We call the isomorphism Φ_{A-C} the base isomorphism of $\boldsymbol{S}$, denoted as $\boldsymbol{S}_s$ and say that Φ_{A-C} depends on Φ_{B-D} through $\boldsymbol{S}$. Likewise, the morphism Φ_{B-D} is called the target morphism and is denoted as $\boldsymbol{S}_t$.

A multi-valued synchronization block is a synchronization block where the lenses f and g are typed with collections of B and D, i.e., $f : A \hookrightarrow B*$ and $g : C \hookrightarrow D*$. Here, the stars denote Kleene closures.

Remark 8 The semantics of such a synchronization block is to declaratively specify validation constraints that must hold for any elements $a \in A$ and $c \in C$ when they have a correspondence $(a, c) \in \Phi_{A-C}$. The lenses allow us to enforce these constraints in one direction or the other.

Remark 9 Kleene closures in this formalization are immutable, ordered collections. In an implementation, one would also like to allow mutable collections that may not be ordered. Conceptually, mutable collections or unordered collections are very similar to Kleene closures, but their formalization is a lot more complex, yet does not offer much insight. Therefore, in the remainder of the paper, the full theory of single-valued synchronization blocks and multi-valued synchronization blocks using ordered, immutable collections is presented, but the equivalent definitions and propositions for other types of collections is omitted.

Example 12 As a first example, we want to synchronize the states of a finite state machine with the places of a Petri net. This can be realized through the synchronization block depicted in Figure 9.

The synchronization block in Figure 9 states that for each state of a state machine, there should be a place in the Petri net (and vice versa).

Example 13 An important special case is when $B = D$ and we can simply use the identity as Φ_{B-D}. This case is particularly relevant for the synchronization of attributes as their data types are typically used in many independent models. However, this can also be interesting when models have an overlap in model classes.

In the following definitions and propositions, we show how a synchronization block is used.

Definition 20 (Consistency with respect to single-valued synchronization blocks) Let $\boldsymbol{S} = (A,B,C,D,\Phi_{A-C},\Phi_{B-D},f,g)$ be a single-valued synchronization block. We denote the state spaces of the type systems $\mathcal{C}_L$ and $\mathcal{C}_R$ with Ω_L and Ω_R, respectively. Let further $\omega_L \in \Omega_L$ and $\omega_R \in \Omega_R$. We say that the state pair (ω_L, ω_R) is consistent for a tuple $(a,c) \in \Phi_{A-C}$ regarding $\boldsymbol{S}$, if

$$(f \nearrow (a, \omega_L), g \nearrow (c, \omega_R)) \in \Phi_{B-D}.$$

We say that the state tuple (ω_L, ω_R) is consistent regarding $\boldsymbol{S}$, if it is consistent for all tuples $(a,c) \in \Phi_{A-C}$.

Definition 21 (Consistency with respect to multi-valued synchronization blocks) In case $\boldsymbol{S}$ is a multi-valued synchronization block in the last definition, we say that the states (ω_L, ω_R) are consistent for the tuple (a,c) with respect to $\boldsymbol{S}$, if the following conditions hold:

- $f \nearrow (a, \omega_L)$ and $g \nearrow (c, \omega_R)$ have the same length and
- For each index i for $f \nearrow (a, \omega_L)$, we have that $(f \nearrow (a, \omega_L)_i, g \nearrow (c, \omega_R)_i) \in \Phi_{B-D}$.

Example 14 With respect to the synchronization block from Figure 9, two states ω_L and ω_R are consistent, if for each pair (f,p) of a state machine and a petri net, there is an isomorphism between the states of f and the places in p, namely the isomorphism $\Phi_{State2Place}$ restricted to the states of f.

Definition 20 clearly can be used to check whether two models that should be treated equally (meaning that they are treated as isomorphic), but the more interesting use case of synchronization blocks is to repair inconsistencies. This is captured in the following propositions.

Definition 22 Let $\boldsymbol{S} = (A,B,C,D,\Phi_{A-C},\Phi_{B-D},f,g)$ be a single-valued synchronization block. The right repair operator $\mathcal{R}_R : A \times C \times \Omega_L \times \Omega_R \to \Omega_R$ for $\boldsymbol{S}$ is defined as

$$\mathcal{R}_R(a,c,\omega_L,\omega_R) := \pi_{\Omega_R}(g \searrow (c, \Phi_{B-D}(\pi_B(f \nearrow (a,\omega_L))), \omega_R)).$$

Here, the repair operator is a function from the space $A \times C \times \Omega_L \times \Omega_R$ to Ω_R, not a morphism in either $\mathcal{C}_L$ or $\mathcal{C}_R$.

In case $\boldsymbol{S}$ is a multi-valued synchronization block, we exchange $\Phi_{B-D}(\pi_B(f \nearrow (a,\omega_L)))$ with $\bar{\Phi}_{B-D}(\pi_B * (f \nearrow (a,\omega_L)))$ where

$$\bar{\Phi}_{B-D} : B* \to D*$$
$$(b_1; \ldots; b_n) \mapsto (\Phi_{B-D}(b_1); \ldots; \Phi_{B-D}(b_n)).$$

This means, we convert the items in the collection separately through the isomorphism.

Definition 23 Let $\boldsymbol{S} = (A,B,C,D,\Phi_{A-C},\Phi_{B-D},f,g)$ be a single-valued synchronization block. The left repair operator $\mathcal{R}_L : A \times C \times \Omega_L \times \Omega_R \to \Omega_L$ is defined as

$$\mathcal{R}_L(a,c,\omega_L,\omega_R) := \pi_{\Omega_L}(f \searrow (a, \Phi^{-1}_{B-D}(\pi_D(g \nearrow (c,\omega_R))), \omega_L)).$$

In case $\boldsymbol{S}$ is a multi-valued synchronization block, we exchange

$$\Phi^{-}_{B-D}1(\pi_D(g \nearrow (c,\omega_R)))$$

with

$$\bar{\Phi^{-}_{B-D}}1(\pi_D * (g \nearrow (c,\omega_R)))$$

where the closure of the isomorphism defined as above.

Proposition 6 *Let $\boldsymbol{S} = (A,B,C,D,\Phi_{A-C},\Phi_{B-D},f,g)$ be a synchronization block and $\omega_L \in \Omega_L, \omega_R \in \Omega_R$ be states such that the tuple (ω_L,ω_R) is not consistent for a tuple $(a,c) \in \Phi_{A-C}$ with respect to $\boldsymbol{S}$. Then, the operator $\mathcal{R}_R$ can repair this inconsistency. This means, the tuple $(\omega_L, \mathcal{R}_R(a,c,\omega_L,\omega_R))$ is consistent for (a,c) with respect to $\boldsymbol{S}$.*

Proof Assume that $\boldsymbol{S}$ is single-valued. We need to check that

$$\pi_D(g \nearrow (c, \mathcal{R}_R(a,c,\omega_L,\omega_R))) = \Phi_{B-D}(\pi_B(f \nearrow (a,\omega_L))).$$

To see this, we have that

$$\begin{aligned} & g \nearrow (c, \mathcal{R}_R(a,c,\omega_L,\omega_R)) \\ &= g \nearrow (c, \pi_{\Omega_R}(g \searrow (c, \Phi_{B-D}(\pi_B(f \nearrow (a,\omega_L))), \omega_R))) \\ &= g \nearrow (g \searrow (c, \Phi_{B-D}(\pi_B(f \nearrow (a,\omega_L))), \omega_R)) \\ &= (\Phi_{B-D}(\pi_B(f \nearrow (a,\omega_L))), \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega_R. \end{aligned}$$

The $\tilde{\omega}$ is precisely the result from the repair operator $\mathcal{R}_R$ and hence the result of the Put operation of g.

Here, we used that g is persistent and we therefore know that $\pi_C(g \searrow (c, \ldots)) = c$. The rest follows from the PutGet for g. The projection of the resulting tuple is exactly what is requested. The proof for the multi-valued case is exactly equivalent.

Remark 10 It may be possible that resolving Φ_{B-D}, there is not yet a corresponding element for $\pi_B(f \nearrow (a,\omega_L))$. In that case, the engine may decide whether or not to extend the isomorphism Φ_{B-D} dynamically by creating an entry for the tuple $(\pi_B(f \nearrow (a,\omega_L)), \pi_D(g \nearrow$

$(c, \omega_R)))$. Whether or not this is intended is usually application specific. In our implementation, we deny creating such a trace entry if either of the elements is a `null`-reference and ask the developer otherwise, using a virtual method. In case of collections, the reconstruction of Φ_{B-D} is done element-wise.

Example 15 To give an example to the last remark, consider again the synchronization block from Figure 9 that synchronizes the states of a state machine with the places of a Petri net. Consider that we start to synchronize two consistent models, but the isomorphism $\Phi_{State2Place}$ is not yet populated, for example because the synchronization is run in an offline scenario. In that case, the engine has two options: It could either create entirely new places and discard the existing ones, or it could try to reuse the existing places. Because creating new model elements may lead to information loss, our implementation always tries to reuse existing model elements. To match the states to the existing places, we request the developer to specify when a new trace entry should be created. Therefore, we may specify that creating such a new correspondence tuple is permitted if the names of the state and place match (cf. Section 6).

Proposition 7 *Let $\boldsymbol{S} = (A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ be a synchronization block and $\omega_L \in \Omega_L, \omega_R \in \Omega_R$ be states such that the tuple (ω_L, ω_R) is not consistent for a tuple $(a, c) \in \Phi_{A-C}$ with respect to $\boldsymbol{S}$. Then, the operator $\mathcal{R}_L$ can repair this inconsistency. This means, the tuple $(\mathcal{R}_L(a, c, \omega_L, \omega_R), \omega_R)$ is consistent for (a, c) with respect to $\boldsymbol{S}$.*

Proof The proof is exactly symmetric to the proof of Proposition 6 as synchronization blocks are entirely symmetric.

Proposition 8 *The right repair operator $\mathcal{R}_R$ is hippocratic in the sense that if the states (ω_L, ω_R) are consistent for the tuple $(a, c) \in \Phi_{A-C}$ (with respect to $\boldsymbol{S}$), then $\mathcal{R}_R(a, c, \omega_L, \omega_R) = \omega_R$.*

Proof Again, we proof this proposition only for single-valued synchronization blocks as the proof for multi-valued synchronization blocks is equivalent. We have that $\Phi_{B-D}(\pi_B(f \nearrow (a, \omega_L))) = \pi_D(g \nearrow (c, \omega_R))$. Therefore,

$$\begin{aligned} & g \searrow (c, \Phi_{B-D}(\pi_B(f \nearrow (a, \omega_L))), \omega_R) \\ & \quad = g \searrow (c, \pi_D(g \nearrow (c, \omega_R)), \omega_R) \\ & \quad = g \searrow (c, g \nearrow (c, \omega_R)) = (c, \omega_R). \end{aligned}$$

This is because $g \nearrow$ is side-effect free and therefore always returns the same state it was executed with – in this case ω_R. The last line is a consequence of GetPut. The projection of the resulting tuple is ω_R as requested.

Proposition 9 *The left repair operator $\mathcal{R}_L$ is also hippocratic.*

Proof The proof is once again exactly symmetric to the proof for $\mathcal{R}_R$.

Remark 11 If one of the input model changes, reflected by a state change, then all synchronization blocks must be revisited to check whether the states of the input models are still consistent with respect to this synchronization block. Depending on the size of the base isomorphism but also the complexity of the involved lenses, this can be very time-consuming (which is not reflected in the formalization). Therefore, we use the incrementalization to accelerate this process.

In particular, the synchronization engine may keep an incrementalization $i = \mathcal{I}(f \nearrow)(\eta_A(a))$. In this case, calling $f \nearrow (a, \omega)$ is replaced by $value(i)$ which yields the same result due to (1). In case the model state $\omega \in \Omega_L$ is updated, the system may use the *apply* transformation to apply the model change sequence $\Delta\omega \in \Delta\Omega_L$ that lead to the new state directly to the incremental value of the lens.

We observed that it sometimes comes in very practical to be able to also have synchronization blocks that only allow to repair inconsistencies in one direction. This may be because one of the models contains not invertible analysis results from the other model, the transformation should be only uni-directional or such a one-way synchronization block accounts for a flaw in some other synchronization block where the lens does not respect the PutGet law in some cases[6]. After all, the design-aim of one-way synchronization blocks is to give the developer a choice what information he would like to have synchronized and which information should not be synchronized. In the latter case, the synchronization engine may still be used to detect any inconsistencies. This is the subject of the next two definitions:

Definition 24 (One-way synchronization block) A one-way synchronization block $\boldsymbol{S}$ is an octuple (A, B, C, D,Φ_{A-C},Φ_{B-D},f,g) like a regular synchronization block with either of the following exceptions:

- f is not a lens, but a regular morphism $f : A \to B$ (single-valued) or $f : A \to B*$ (multi-valued). In this case, we call the one-way synchronization block a Left-to-Right synchronization block.
- g is not a lens, but a regular morphism $g : C \to D$ (single-valued) or $g : C \to D*$ (multi-valued). In this case, we call the one-way synchronization block a Right-to-Left synchronization block.

[6] This may be acceptable because the – let us call such a thing for the moment a semi-lens – can be specified much more generic in this way. An example of such a construct is given in Section 6

The consistency for one-way synchronization blocks is the same as for regular synchronization blocks except that the missing lens GET has to be replaced by the respective regular morphism.

Remark 12 The advantage of a one-way synchronization block is that the choice of the function is more liberal and the transformation developer may also chose non-invertible functions.

Proposition 10 *The consistency repair operator $\mathcal{R}_R$ is also applicable for Left-to-Right synchronization blocks and in that case is also hippocratic.*

Likewise, the consistency repair operator $\mathcal{R}_L$ is also applicable for Right-to-Left synchronization blocks and in this case, is also hippocratic.

Proof The proof is equivalent to the case of regular synchronization blocks where we again exchange the GET operation of the missing lens by the regular morphism.

4.3 Composition of Synchronization Blocks

A synchronization block is made to define how an isomorphism should be populated based on the knowledge of another one. In the implementation, these isomorphisms are usually synchronization rules, so that the synchronization block in Figure 9 specifies how to build up the isomorphism *StateToPlace* from the isomorphism *AutomataToNet*. This stacking process is the subject of the next definition:

Definition 25 (Model Synchronization) A model synchronization is a set of synchronization blocks S with an entry isomorphism s such that for each $\boldsymbol{S} \in S$, we have that either $\boldsymbol{S}_s = s$ or there is another synchronization block $\tilde{\boldsymbol{S}} \in S$ such that $\boldsymbol{S}_s = \tilde{\boldsymbol{S}}_t$.

Remark 13 The synchronization blocks of a model synchronization can be regarded as a graph where the nodes are the isomorphisms and the edges are the synchronization blocks. A synchronization block $\boldsymbol{S}$ then points from $\boldsymbol{S}_s$ to $\boldsymbol{S}_t$. This graph may be an arbitrary directed graph, it is not required to be free of circles. Rather, circles are required to handle composite structures such as expressions.

The start isomorphism s of a model synchronization determines the signature of the model synchronization, i.e., what model elements it can synchronize. Usually, this isomorphism is defined between the root model class of the LHS and the root model class of the RHS.

Definition 26 Because the synchronization blocks having a given isomorphism Φ as their base isomorphism describe validity constraints for tuples in this isomorphism, we associate them with Φ and refer to them as the synchronization blocks of Φ.

In particular, within a model synchronization (S, s), we have that

$$blocks_S(\Phi) := \{\boldsymbol{S} | \boldsymbol{S}_s = \Phi\}.$$

Remark 14 In practice, the isomorphisms are created and dedicated for a given model synchronization scenario. Therefore, we omit the subscript in the last definition.

Definition 27 Let (S, s) be a model synchronization, $\boldsymbol{S} \in S$ a single-valued synchronization block with source isomorphism $\boldsymbol{S}_s : A \to C$ and target isomorphism $\boldsymbol{S}_t : B \to D$. We call a tuple of states $(\omega_L, \omega_R) \in \Omega_L \times \Omega_R$ fully consistent for a tuple $(a, c) \in \boldsymbol{S}_s$ with respect to $\boldsymbol{S}$ in S, if

- The states are consistent for the tuple (a, c) with respect to $\boldsymbol{S}$ and
- The states are fully consistent for the tuple $(f \nearrow (a, \omega_L), g \nearrow (c, \omega_R))$ with respect to all synchronization blocks of $\boldsymbol{S}_t$ in S.

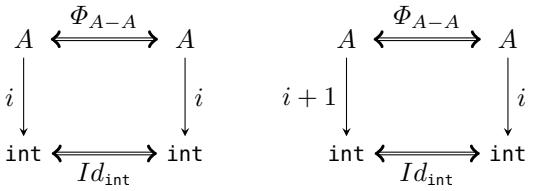
If $\boldsymbol{S}$ is a multi-valued synchronization block, then the states have to be fully consistent for all tuples spanned by $f \nearrow (a, \omega_L)$ and $g \nearrow (c, \omega_R)$ with respect to all synchronization blocks of $\boldsymbol{S}$ in S.

Definition 28 Let (S, s) be a model synchronization. We call a tuple of states $(\omega_L, \omega_R) \in \Omega_L \times \Omega_R$ consistent for a tuple $(a, c) \in s$ with respect to (S, s), if the states are fully consistent for the input tuple with respect to all synchronization blocks in s within S.

Remark 15 Like for synchronization blocks, one would like to obtain generic consistency repair operators $\tilde{\mathcal{R}}_R$ and $\tilde{\mathcal{R}}_L$ that are able to repair any possible inconsistency. Such a repair operator could be obtained by repeatedly executing $\mathcal{R}_R$ or respectively $\mathcal{R}_L$ for all inconsistencies that arise. However, we have no guarantee that repairing one consistency does not open a new one.

Remark 16 To repair an inconsistency, the synchronization blocks only change the respective model through PUT operations. This means that in case of heterogeneous models, any information not contained in the other model simply is ignored, meaning that it is not propagated to the other model but left intact.

Example 16 Consider a metamodel with just a single class A that has an integer-valued property called i, which we extend to a lens. We look at synchronizing two copies of this metamodel with the following two synchronization blocks:



It is clear that no state tuple can be consistent for any pair of instances of A, because the i property of both copies must be the same and different at the same time, which is not possible. If we go ahead and start repairing inconsistencies, then each repair will create a new inconsistency, which is why the synchronization does not terminate.

Proposition 11 *If the model synchronization terminates to apply $\mathcal{R}_R$ and $\mathcal{R}_L$, it returns a new consistent state.*

Proof This proposition is an immediate consequence from the fact that the non-existence of further inconsistencies is used as the termination criterion.

Proposition 12 *The model synchronization is hippocratic, i.e., if applied to consistent changes, the model synchronization does not change the states.*

Proof This proposition is an immediate consequence from the fact that the repair operator(s) for the individual synchronization blocks is hippocratic.

Remark 17 As the last example shows, repairing the inconsistencies between two states may not be terminating. To find a suitable theory to proof termination is subject of future work. To us, it is unclear whether such a theory may even exist, in particular, whether or not the construct of synchronizations as presented in this paper is Turing-complete.

Nevertheless, we have a formal tool that is able to repair inconsistencies one by one. If only finitely many new inconsistencies arise from fixing existing ones and the synchronization only consists of regular (two-way) synchronization blocks, then any inconsistency can be repaired automatically.

Problems as in the above example can be avoided if the properties used in synchronization blocks are mutually exclusive. In that case, it is unlikely that the repairing an inconsistency just produces another inconsistencies.

Remark 18 Propositions 11 and 12 are independent of the order in which inconsistencies are resolved. In practice, this order may be important. This is because very often, isomorphisms are defined based on other isomorphisms, such as for example Listing 6 in Section 6. Our implementation uses the literal order in the transformation specification. The fact that the correctness of the synchronization process is independent of this order also means that the transformation developer can play with it to make sure that all elements are available when the synchronization is executed.

5 Implementation in an Internal DSL

In this section, we present our approach for implementing model synchronization through synchronization blocks in an internal DSL. For this, we first describe how the primitives that are used in synchronization blocks, lenses, morphisms and isomorphisms are represented in the language before we describe the synchronization modes and how the model synchronization is executed.

5.1 Lenses and Morphisms

To support multiple transformation modes, we need to operate on incrementalizable lenses. However, most general-purpose programming languages only provide predefined operators or simple method calls, combined in a compiled code – an artifact which is often hard to analyze. To solve this problem, we need to obtain a model of the code. This could be created by a fluent syntax [18].

In our implementation, we use a feature of our host language C# to retrieve this model of the code even simpler. C# allows us to obtain the abstract syntax tree (called expression tree) from an expression instead of compiled code. This means that the transformation developer writes regular C# code but the compiler does not compile this code down to intermediate language. Instead, the compiler generates code to create a model of the abstract syntax tree. Fowler calls this construction principle of an internal DSL *Parse Tree Manipulation* [19, p. 455].

The rationale behind this decision is that 1. the language adoption problem is mitigated because the transformation developer writes regular C# code, 2. the understandability is improved because the code does not contain syntactic boilerplate and 3. the tool support is maintained: The compiler still checks the correct types and the editor offers support such as code completion or navigation.

This ability to step into the compilation process is the one and only syntax feature that we use from C# that makes our language in this form impossible to implement in many other languages (apart from Visual Basic). However, we believe that other languages such as Java or in particular Xtend will soon adapt this feature as well, making our approach applicable to other languages.

For operators built into the host language, we implement default Put operators, if they can be considered as lenses.

For example, consider the expression $x + c$ for some modifiable references x and c. Through the modifiable monad, we know that whenever x changes its value, also the value of the sum may change. For the lens, the expression resembles the Get function. The lens allows us to assign a value, say 42 to the sum given that the reference c is constant. This is applied by setting $x = 42 - c$, the Put function of the respective lens. The lens is represented by its Get function which we expect to be decorated with a Put function reference. In our implementation, this reference is realized through an annotation.

Other operators such as value equality cannot be reverted in general. It is unclear how to set an expression $x == c$ to `false`, in particular, what value to assign to x. This can be solved by additional parameters that are only taken into account when reversing the operation, such as a method `EqualsOrDefault` providing the missing information with a third parameter.

To reconstruct the lens from a method call, we decorate each allowed method with an information how this method can be inverted – the developer may annotate the respective Put operation. Since this decoration is publicly accessible, this even allows developers of our approach to extend the API that can be used to specify a model synchronization.

For an example, consider the *FirstOrDefault* function that has been considered previously in Section 4.1. Ideally, one would like to define this function in a generic way, such that for each type A, $\mathit{FirstOrDefault} : A* \to A$. To make this a lens, the developer has to specify a Put method $\mathit{FirstOrDefault} \searrow: A* \times A \to A*$. Alternatively, we also allow the developer to specify an operation $\mathit{FirstOrDefault} \searrow': A* \times A \to \bot$ to specify that the lens is persistent.

This method is specified using a type (or type template) and a method name. The problem here is that the method may be generic, but method annotations must not be generic. However, if the Get method is generic, the Put method must also be generic with the same type arguments. Therefore, the implementation collects generic type arguments in the Get method and applies them automatically to the selected method. The system also checks the selected method that shall be used as Put operation for type conformance.

In the .NET platform, for a given type A, the Kleene closure is represented by the array type $A[]$, as long as indices of the array are not changed directly. The Put annotation is called `LensPut` in NMF. This annotation has to be put on a given method to specify its corresponding Put operation. In the example, the Put operation of `FirstOrDefault` is `PutFirst`. Therefore, an implementation for a *FirstOrDefault* lens for arrays is the one depicted in Listing 1.

```
1  static class Helpers
2  {
3    [LensPut(typeof(Helpers), "PutFirst")]
4    T FirstOrDefault<T>(this T[] array)
5    {
6      return array != null && array.Length > 0 ? array[0] : default(T);
7    }
8
9    T[] PutFirst<T>(this T[] array, T element)
10   {
11     ...
12   }
13 }
```

Listing 1 Implementation of a FirstOrDefault lens for arrays

When the framework is asked to build a lens of a given expression, it uses the abstract syntax tree of that expression and tries to apply Proposition 4 repeatedly to it. Our implementation currently does not enforce yet that g from this proposition is indeed stateless, but this is currently left to the transformation developer. However, it is usually not a problem because most synchronization blocks that we have come across so far either are very simple, most of them only consist of a single property access.

The ability to extend the language with custom lenses gives the transformation developer the possibility to extend the capabilities of the synchronization language whenever necessary.

If the lens is to be used also in an incremental setting, we also require to specify a function $\mathcal{I}(\mathit{FirstOrDefault}) : \mathcal{I}(A*) \to \mathcal{I}(A)$, though the implementation is actually able to lift a function $\mathit{FirstOrDefault}' : A* \to \mathcal{I}(A)$ by reevaluating the latter when the input changes. Similar to the Put operation, we annotate this manual incrementalization using an annotation `ObservableProxy`. If this function is not provided, then the system automatically assumes that the function only changes when the input reference changes.

In the case of the *FirstOrDefault* function, such an annotation is not required because the length of an array is fixed. If the collection type was mutable, then such a proxy method must be available. As an example, Listing 2 contains the same lens for generic lists. For the incremental evaluation, we simply reuse the incrementalization system of NMF through the class `ObservingFunc`.

```
static class Helpers
{
  private static ObservingFunc<IList<T>, T> firstOrDefaultFunc =
    ObservingFunc<IList<T>,T>.FromExpression(list =>
       list != null && list.Count > 0 ? list[0] : default(T));

  [LensPut(typeof(Helpers), "PutFirst")]
  [ObservableProxy(typeof(Helpers), "FirstOrDefault")]
  T FirstOrDefault<T>(this IList<T> list)
  {
    return firstOrDefaultFunc.Evaluate(list);
  }

   INotifyValue<T> FirstOrDefault<T>(this INotifyValue<IList<T>> list)
  {
    return firstOrDefaultFunc.Observe(list);
  }

  T[] PutFirst<T>(this IList<T> list, T element)
  {
    ...
  }
}
```

Listing 2 Implementation of a FirstOrDefault lens for lists

In NMF, in fact the collections we are working with are mutable. The reason for this is mostly that mutable collections allow a more fine-grained information how the collection contents have changed. We require all collections to implement an interface to record when any changes have been made to the collection contents and how these changes look like. If a model element has been added to a collection, we do not need to iterate through the entire collection to find out what has changed but already know the element that has been added. Therefore, the *FirstOrDefault* operator in our implementation does need a custom implementation of its incrementalization.

5.2 Isomorphisms

To represent isomorphisms, we distinguish between two cases: Identities and isomorphisms between model classes. The case of an identity isomorphism is easy since the model synchronization engine does not have to do anything as the identity of an object is easy to compute. In the latter case, we realize the isomorphism using two unidirectional transformation rules $\Phi^{\leftarrow}$ and $\Phi^{\rightarrow}$, one for transforming the models in each direction. Thus, the relationship $(a, c) \in \Phi$ is manifested in two trace entries $(a, c) \in \Phi^{\leftarrow}$ and $(c, a) \in \Phi^{\rightarrow}$. This of course implies a 1:1 relationship between synchronized elements, but the transformation developer is free to define arbitrarily many other isomorphisms for the same element(s)[7].

The implementation of synchronization blocks as NTL transformation rules also has another advantage: As we expose the underlying transformation rules, dependencies may be added to them which may result in executing a model transformation each time a new correspondence is set. This behavior is used for example in the TTC 2015 Java Refactoring Case [30] where a method group is created for each unique method name, meanwhile later changes to a methods name are reflected by renaming the method group.

In case custom lenses do not suffice for a given task, we allow opaque synchronization blocks where we give the transformation developer full control. In this case, the transformation developer may hook in arbitrary C# code, but has to manage all the operation modes and change propagation modes by himself.

Regarding the syntax, based on the experience we collected with our non-incremental, unidirectional transformation language NTL [33, 34], we decided to apply a similar strategy and represent synchronization rules by generic classes. However, the body of such a synchronization rule consists of synchronization blocks rather than compiled code. We therefore use a dedicated method to create these synchronization blocks through method calls. However, synchronization rules may also serve as containers for helper methods, should they be necessary for the definition of a synchronization block. The concrete syntax is presented for our motivational example in Section 6.

As a further consequence, we also inherit the modularization techniques as discussed in earlier work [34, 38]. We can therefore offer the transformation developer advanced modularization techniques, such as version conflict detection, integrity checking, creating model synchronization libraries and model synchronization rule templates.

5.3 Synchronization Modes

As an advantage of the declarative specification of synchronization blocks, they are not tied to specific operation modes and therefore can be reused in many scenarios. We have seen that we always have a choice whether to fix inconsistencies at the left or right side of the synchronization blocks. Furthermore, in some scenarios, it may be desirable to allow certain inconsistencies. We refer to the strategies of selecting the appropriate repair operator as synchronization modes and discuss them in the following.

We support six different synchronization modes that can be combined with three different change propagation modes where we adapted the terminology from Triple Graph Grammars and refer to type systems $\mathcal{C}_L$ as the Left Hand Side (LHS) and similarly $\mathcal{C}_R$ as the Right Hand Side (RHS). The synchronization modes are as follows:

[7] We do not support dynamic creation of isomorphisms, the isomorphisms must be given at compile time.

LeftToRight: the transformation ensures that all model elements on the LHS have some corresponding model elements on the RHS. However, the RHS may contain model elements that have no correspondence on the LHS. This means, we apply $\mathcal{R}_R$ as long as we find inconsistencies, but do not propagate `null`-values.

LeftToRightForced: the transformation ensures that all model elements on the LHS have some corresponding model elements in the RHS. All elements in the RHS that have no corresponding elements in the LHS are deleted. In this mode, we apply $\mathcal{R}_R$ as long as possible and also propagate `null`-values.

LeftWins: the transformation ensures that all model elements on the LHS have some corresponding model elements in the RHS and vice versa. Synchronization conflicts are resolved by taking the version at the LHS. This means, we apply $\mathcal{R}_R$ as long as possible but do not propagate `null`-values. Instead, these inconsistencies are resolved using $\mathcal{R}_L$.

RightToLeft, RightToLeftForced, RightWins: same as the above but with interchanged roles of RHS and LHS

The change propagation modes are the following:

None: no change propagation is performed. In this case, also no dynamic dependency graphs for any expressions are created as they are not necessary.

OneWay: change propagation is only performed in the main synchronization direction, i.e. LHS to RHS for the first three synchronization modes and RHS to LHS otherwise.

TwoWay: change propagation is performed in both directions, i.e. any changes on either side will result in appropriate changes in the other side.

Usage examples in the TTC 2015 [30] have shown that there are some cases where also the remaining operation mode to propagate updates from the target side of the synchronization back to the origin model make sense, but we decided not to support this operation mode in our implementation as we believe that these are rare corner-cases. However, conceptually, there is no limitation. It is also not clear whether all of the combinations between change propagation and synchronization direction are necessary but the diversity shows the flexibility how the transformation specification can be used.

Furthermore, there may be even more operation modes such as a check-only mode that only tests whether the selected constraints hold, but would not enforce these constraints.

The applicable synchronization direction and change propagation mode is specific to a synchronization run and is provided together with the input arguments, i.e., LHS and RHS initial models. At this initialization, we generate code to minimize the performance impact when no change propagation should be performed, i.e. the synchronization should run with a performance comparable to a transformation without change propagation as, e.g., pure NMF Transformations.

5.4 Execution

In this section, we explain the bidirectional execution modes that a synchronization engine implementation can support based on the concept of synchronization blocks. Throughout this section, we will use the synchronization block from Figure 9 as example.

For example, assume that the synchronization block from Figure 9 was the only synchronization block in our model synchronization and the synchronization engine was asked to execute this synchronization for a given finite state machine and Petri net model. The engine finds the synchronization rule to start with based on the synchronization rule types, in this case $\Phi_{AutomataToNet}$ and executes this rule (and therefore all of its synchronization blocks) for the given direction.

When executing the synchronization block from Figure 9, the synchronization engine uses the Get operation to obtain the states of the finite state machines and the places of the Petri net. Then, it tries to find a corresponding place for each state of the finite state machine, thereby executing the synchronization rule $\Phi_{StateToPlace}$. Synchronization rules are required to determine whether elements of LHS and RHS should correspond. For the synchronization rule $\Phi_{StateToPlace}$, a reasonable definition is that a state and a place should correspond when their names match. Once the correspondence has been established, it is saved in the trace. Because a subsequent query for the corresponding element of a given LHS element (or RHS element, respectively) results in a trace access, this guarantees that the correspondence relation stays bijective.

The result of this matching are three sets: A set $M_{\leftrightarrow}$ of tuples of states and places that correspond according to the $\Phi_{StateToPlace}$ isomorphism, a set $M_{\rightarrow\emptyset}$ of states with no corresponding place and similarly a set $M_{\emptyset\leftarrow}$ of places with no corresponding state. In all direction modes, the set $M_{\leftrightarrow}$ is traversed and the synchronization engine makes sure that each synchronization block for the dependent synchronization rule is executed for each pair.

The fact that trace entries in NTL are keyed not only by their source elements but also by their transformation rules means that a model element on the LHS

may easily mapped to multiple elements of the RHS (or vice versa) by just specifying multiple synchronization rules for that element.

For the contents of the two sets $M_{\rightarrow\emptyset}$ and $M_{\emptyset\leftarrow}$, the next step depends on the synchronization direction:

LeftToRight: In this direction, the set $M_{\emptyset\leftarrow}$ is ignored. The engine only traverses the set $M_{\rightarrow\emptyset}$ and creates a new element in the RHS, in the example a place, and establishes a correspondence, including to execute the synchronization blocks of the dependent synchronization rule. The newly created element is then added to the Petri net using the PUT operation of the RHS lens *.Places*. Since this is a collection-valued lens, this results in adding the element to the collection.

LeftToRightForced: In this direction, the RHS must look exactly like the LHS, up to isomorphism. Therefore, additionally to the processing of $M_{\rightarrow\emptyset}$ as in the *LeftToRight* direction, the engine removes elements in $M_{\emptyset\leftarrow}$, again using the collection interface of *.Places*.

LeftWins: In this direction, the set $M_{\emptyset\leftarrow}$ is similarly to the set $M_{\rightarrow\emptyset}$ as the synchronization engine creates new states, establishes a correspondence and adds the newly created states to the state machine using the PUT operation of the *.States* lens, which again means to add the state to the collection. The direction *LeftWins* is therefore almost symmetric as the sets $M_{\emptyset\leftarrow}$ and $M_{\rightarrow\emptyset}$ are treated equally. The only difference is in case of conflicts, e.g. when the lenses are single-valued. This is very often the case for attributes such as an elements name. Here, setting a new name also means to delete the old one and therefore, the synchronization engine must only apply at most one of the lenses. Here, the direction *LeftWins* specifies that in such a scenario, the engine should always apply the LHS to the RHS.

The directions *RightToLeft*, *RightToLeftForced* and *RightWins* are equivalent except for exchanged roles of LHS and RHS.

The easiest case for change propagation is of course disabled change propagation (*None*) since in this case, the synchronization does not have to perform any lifting. This means, the transformation can simply use the lens f to obtain the affected source elements and uses g to store them in the target model. Afterwards, it may forget about the synchronization block, as it has been processed.

If the change propagation is set to *OneWay*, the synchronization engine must react on changes that affect the selecting lens f. Therefore, it applies the (original) incrementalization monad $\mathcal{I}$, which for a given source element $a \in A$ yields a modifiable reference for the result $\mathcal{I}(f)(\eta(a))$. The engine can then use the value of this modifiable reference and proceed as if there was no change propagation switched on. However, the modifiable reference is stored and event handlers are registered for the event that the modifiable reference changes its value. In that case, these changes are transmitted to the RHS with the synchronization mode *LeftToRightForced* (important as otherwise the change may get lost). Both initially and in case of change notifications, the target value is stored using the PUT method of g.

Formally, if we have again the situation of Figure 8 and $a \in A, c \in C$ such that $(a, c) \in \Phi_{A-C}$ and the system is in state $\omega \in \Omega$. If change propagation is enabled for the direction LHS to RHS in the current configuration, we actually store a reference to the incrementalized getter f. We refer to this variable as $\hat{f} := \mathcal{I}(f\ \nearrow)(\eta(a), \omega)$.

Now, consider that some global state change $\Delta\omega \in \Delta\Omega$ occurs. The first step to perform is to check whether this state change actually has any effect on the synchronization block, i.e. whether

$$value(\hat{f}) \neq value(apply(\hat{f}, \Delta\omega)).$$

In the implementation, this check is implemented by an event raised by $\hat{f}$ if a state change happened that changes the current value. Should that be the case, $\hat{f}$ is assigned the new value $apply(\hat{f}, \Delta\omega)$ which in the implementation can be neglected since the objects realizing $\mathcal{I}(f\ \nearrow)$ are mutable and adjust their state automatically.

Afterwards, the change is propagated using the direction mode **LeftToRightForced**, since the change propagation demands that this change will be propagated to the RHS. Thus, the change is propagated by $\mathcal{R}_R$ as in the non-incremental case.

Compared to a repeated execution of the same model synchronization without change propagation or alternatively, a repeated execution of an unidirectional regular model transformation, the activated change propagation may lead to speedups. The speedup, however, depends on the used lenses, their performance characteristics and also the change sequences in which the models are changed. As we are striving for an extensible approach where developers can simply implement a lens, if they feel that the language misses one, a speedup can, however, not be guaranteed. Rather, the incrementalization is best effort.

The change propagation mode *TwoWay* works very similar, except that for both sides, the refined incrementalization monad $\tilde{\mathcal{I}}$ is applied and the *store* transformation is used as a replacement of the PUT operation. The

synchronization engine registers for change notifications on both monad instances and enforces the changes with direction *LeftToRightForced* or *RightToLeftForced*, respectively.

5.5 Inheritance and Superimposition

In practice, many metamodels use inheritance to avoid duplication of concepts in the metamodel. This poses a challenge for model synchronization because isomorphisms between more abstract concepts often are required to be broken down to isomorphisms that are more specific, i.e., have a smaller domain. Based on the more specific isomorphisms, more synchronization blocks may apply.

In our running example, consider the case that we added a composite structure both to finite state machines and Petri Nets. This means, states may either be simple states or composite states and likewise, places may either be simple places or composite places[8]. Clearly, we would like to keep synchronizing states of a state machine with the places of a Petri net, but in case the state is a composite state, a composite place shall be created and the inner state machine should be synchronized with the inner Petri net of that composite place. However, one would still like to continue using the abstract isomorphism between states and places, but be able to refine this isomorphism in case of more specific elements.

In the implementation, we use a concept we call *synchronization rule instantiation*, very similar to the transformation rule instantiation concept used in NTL [29, 38]. Thus, whenever the synchronization engine is asked to create a new element for a given synchronization rule, it looks out for instantiations of that synchronization rule to create the element, because the target concept may also be different according to the true type of the source element. In case of the example above, a composite place should be created if the source model element was a composite state. Such a rule instantiation may be mandatory in case the targeted class is abstract.

To implement synchronization rule instantiation, we simply reuse the rule instantiation concept of the underlying NTL transformation rules. That is, if a synchronization rule $\Phi_{concrete}$ is instantiating a synchronization rule $\Phi_{abstract}$, then simply $\Phi^{\leftarrow}_{concrete}$ instantiates $\Phi^{\leftarrow}_{abstract}$ and $\Phi^{\rightarrow}_{concrete}$ instantiates $\Phi^{\rightarrow}_{abstract}$. Note that rule instantiation can be stacked, i.e., there may be another rule that instantiates $\Phi_{concrete}$ for more concrete classes. With this feature, also more complex inheritance hierarchies can be supported. In TGGs, rule refinements are an equivalent concept.

Furthermore, we also adopted the superimposition implementation of NTL, which in turn is adopted from ATL [62]. That is, synchronization rules may be easily overridden in refined model synchronizations, one may create a library of synchronization rules or create synchronization rule templates[9]. In particular, the considerations on inherited modularization concepts as described in previous work can be reused, but it is yet unclear to which extend this is useful also for incremental and bidirectional model synchronizations. However, a detailed analysis is out of the scope for this paper and subject to future work.

6 Synchronization of Finite State Machines and Petri Nets

In this section, we apply our concepts to the motivational example of synchronizing finite state machines with Petri nets. In parallel, we also describe how the proposed model synchronization would be specified in TGGs in order to draw some comparison. Because there is a textual language available that is easier to compare with and our impression is that it is the most actively developed TGG tool, we have chosen eMoflon [3] as concrete language. However, the results should be similar when using a different TGG tool.

Like a model transformation in NMF Transformations that consists of multiple transformation rules represented by public nested classes inheriting from a `TransformationRule` base class, model synchronizations of NMF Synchronizations consist of synchronization rules. These synchronization rules implicitly define two transformation rules for NMF Transformations, one for each direction. A minimal example for a model synchronization is therefore depicted in Listing 3.

```
1 public class FSM2PN : ReflectiveSynchronization
2 {
3     public class AutomataToNet : SynchronizationRule<FiniteStateMachine
          , PetriNet> {}
4 }
```

Listing 3 A model synchronization in NMF Synchronizations

This defines the isomorphism $\Phi_{AutomataToNet}$ between finite state machines and Petri nets, but without any synchronization block. Synchronization rules in

[8] If this concept was added in only one of the metamodels, the information simply would not be propagated but no additional synchronization would be required.

[9] We have not yet used synchronization rule templates in practical use cases, yet. We suspect that they are harder to create than transformation rule templates because there is fewer support for abstract incrementalizable lenses than for regular abstract methods.

NMF Synchronizations define the LHS and RHS model elements they operate on through the generic type arguments of the `SynchronizationRule` base class they need to inherit from and have multiple methods they can override.

In particular, unlike eMoflon, we do not require a correspondence declaration. To declare the used metamodels, it suffices entirely that the compiler of the host language knows the metaclass implementations. The latter may be either in the same assembly or in a referenced assembly.

The most important method to override in the definition of a synchronization rule is the method to determine when an element of the LHS should match an element of the RHS. For the `AutomataToNet`-rule, we simply return true since both RHS and LHS model elements are the root elements of their respective models and should be unique.

```
1  public override bool ShouldCorrespond(FSM.State left, PN.Place right,
       ISynchronizationContext context)
2  {
3     return left.Name == right.Id;
4  }
```

Listing 4 Definition that states and places should correspond based on their names

Other synchronization rules may have other strategies. For example, the correspondence of the `StateToPlace`-rule is based on comparing the names as depicted in Listing 4.

The second most important method to override is the `DeclareSynchronization` method. Here, we define what synchronization blocks the synchronization rule consists of. The `DeclareSynchronization` method of *AutomataToNet* looks as depicted in Listing 5.

```
1  public override void DeclareSynchronization()
2  {
3     SynchronizeMany(SyncRule<StateToPlace>(),
4        fsm => fsm.States, pn => pn.Places);
5     SynchronizeMany(SyncRule<TransitionToTransition>(),
6        fsm => fsm.Transitions, pn => pn.Transitions.Where(t => t.To.
           Count > 0));
7     SynchronizeMany(SyncRule<EndStateToTransition>(),
8        fsm => fsm.States.Where(state => state.IsEndState),
9        pn => pn.Transitions.Where(t => t.To.Count == 0));
10    Synchronize(fsm => fsm.Id, pn => pn.Id);
11 }
```

Listing 5 The DeclareSynchronization method of AutomataToNet

The statements in Listing 5 create synchronization blocks: Lines 3 and 4 create the synchronization block we depicted earlier in Figure 9. When handling the synchronization of a finite state machine with a Petri Net, the synchronization engine should establish correspondences between the states and the places using the `StateToPlace` rule, synchronizing the states of the finite state machine with the places of a Petri Net. This synchronization rule is straight forward, matches states and places based on their names (cf. Listing 4) and synchronizes them afterwards. For a given state of a state machine, the synchronization engine only looks for corresponding places in the `Places` reference of the corresponding Petri Net.

The advantage of synchronization blocks over eMoflon is here that the specification can be much more concise. Consider for example the synchronization block in Lines 3 and 4, or depicted in Figure 9. This synchronization block is able to repair inconsistencies arising from adding or removing states from a state machine. TGGs usually require an entire rule for this, which to the best of our knowledge cannot be expressed in a single line[10]. In eMoflon, this rule consists of 36 lines. These savings are possible because a lot of declarations have to be done explicit in a TGG rule, while they can be inferred in NMF Synchronizations because of the synchronization rule a synchronization block belongs to.

Please note that the lenses at the Petri Net side violate the PutGet law in case a new transition has to be added, because the count of the `To` collection is not reversible. The `Where`-operator is currently implemented to silently ignore this inconsistency and we oblige this to the transformation developer to take care about such cases. This is fine for all the cases that we have come across, but if this should not match the transformation developers needs, he can simply override this behavior by extending the respective lens. Because it is usually only the Put operation that is problematic, the language also has some overloads to provide a custom Put method.

Similarly, the transitions of the finite state machine should be matched with the transitions of the Petri Net, but only with those that have at least one target place. Therefore, lines 5 and 6 of Listing 5 create the synchronization block depicted in Figure 10. This means that if a new transition is added to the Petri Net transitions or an existing transition is assigned a first target place, then the synchronization engine will try to match this transition to an existing finite state machine transition. If conversely, a transition is added to the finite state machine, the synchronization engine will add the corresponding transition to the Petri Net, hoping that it satisfies the condition that the count is greater than zero. To find the corresponding transition on the respective other side, the `ShouldCorrespond` method depicted in Listing 6 is used.

[10] We ignored the linebreak in the Listing, because editors usually allow 85 characters in a single line

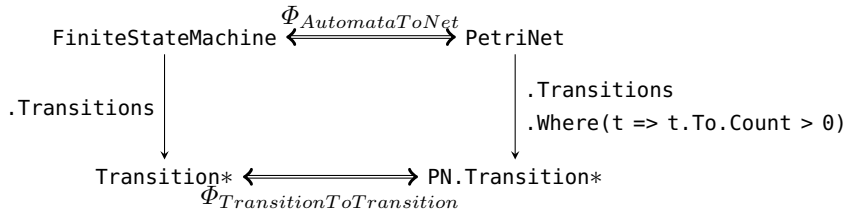


Figure 10 Synchronization of the transitions of a finite state machine with the transitions of a Petri net

```
1  public class TransitionToTransition : SynchronizationRule<FSM.
       ITransition, PN.ITransition>
2  {
3    public override bool ShouldCorrespond(FSM.Transition left, PN.
         Transition right, ISynchronizationContext context)
4    {
5      var stateToPlace = SyncRule<StateToPlace>().LeftToRight;
6      return left.Input == right.Input
7        && right.From.Contains(context.Trace.ResolveIn(stateToPlace,
             left.StartState))
8        && right.To.Contains(context.Trace.ResolveIn(stateToPlace,
             left.EndState));
9    }
10   public override void DeclareSynchronization()
11   {
12     Synchronize(t => t.Input, t => t.Input);
13     Synchronize(SyncRule<StateToPlace>(),
14                 t => t.StartState,
15                 t => t.From.SingleOrDefault());
16     Synchronize(SyncRule<StateToPlace>(),
17                 t => t.EndState,
18                 t => t.To.SingleOrDefault());
19   }
20 }
```

Listing 6 Matching transitions

This method uses the trace abilities of NMF Transformations that is still accessible in NMF Synchronizations, i.e. it accesses the corresponding place for a given state in the transformation rule $\Phi^{\rightarrow}_{StateToPlace}$ from LHS to RHS and uses it to decide whether the transitions should match. This trace entry exists regardless of the synchronization direction, as the synchronization engine always creates two trace entries.

In eMoflon, the synchronization blocks to synchronize the transitions of a finite state machine as well as the synchronization blocks to synchronize the start and end states of such a transition all can be expressed in a single TGG rule, plus a second rule if self-transitions should be supported. In terms of lines of code, however, these rule still have 57+51 lines, whereas the implementation in NMF Synchronizations requires 24 lines for the synchronization rule plus one for the synchronization block in the $\Phi_{AutomataToNet}$ synchronization rule. However, the latter also propagates partial changes, while the respective eMoflon rule would propagate everything at once or not at all[11]. Which of these strategies is better depends on the application.

[11] We believe that a partial propagation is also possible in TGGs but requires many more rules

Lines 7-9 of Listing 5 create the synchronization block depicted in Figure 11 and indicate that the remaining transitions should be synchronized with the end states of the state machine. The symmetric correspondence check fails in this case because the synchronization engine will look for a suitable state in the end states of the machine. If the state is not yet marked as an end state, the synchronization engine will not find it. Thus, we have to override this behavior and particularly look for the state which is corresponding to the transitions origin.

```
1 public override void DeclareSynchronization()
2 {
3   SynchronizeLeftToRightOnly(SyncRule<StateToPlace>(),
4     state => state.IsEndState ? state : null,
5     transition => transition.From.FirstOrDefault());
6 }
```

Listing 7 One way synchronizations

Next, it is necessary to connect or disconnect the Petri Net transition to the correct place. This only has to be done in the LHS to RHS direction since this information is already encoded in the `IsEndState` attribute in the finite state machine state. We have to limit the scope of this synchronization job because the synchronization initialization otherwise raises an exception since the conditional expression of the LHS is not a lens (we do not detect that there is a partition of values for true and false branch). This is depicted in Listing 7.

Line 10 in Listing 5 tells that the identifiers of both finite state machine and Petri Net should be synchronized. This creates a simple synchronization block as shown in Figure 12 as the dependent synchronization rule is simply the identity on strings. In the syntax, this is expressed by omitting the synchronization rule. This means that both passed λ-expressions must have the same result type such that the synchronization engine may use the identity as isomorphism.

In eMoflon, the synchronization of end states and respective transitions also requires a single TGG rule which consists of 48 lines where the solution in NMF Synchronizations requires a synchronization block in $\Phi_{AutomataToNet}$ and a synchronization rule $\Phi_{EndStateToTransition}$ consisting of 21 lines.

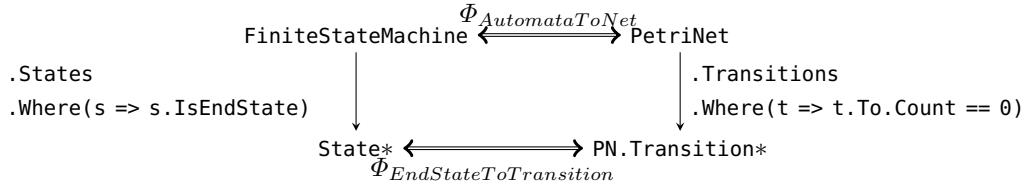


Figure 11 Synchronization of the end states of a finite state machine with swallowing transitions of a Petri net

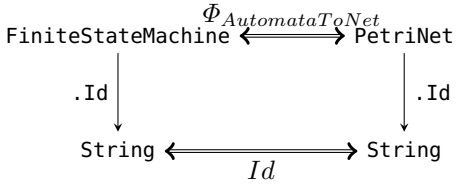


Figure 12 Synchronization of the names of a finite state machine and a Petri net

The presented synchronization is a bijection, i.e., there is no information loss. In case there was, for example if we introduced multiple types of places (for example queueing places) with no equivalent information in the finite state machines, the synchronization would still be exactly the same, similar to the synchronization mode of eMoflon. The only difference would be that the information about the type of a place would not be transmitted to the state machine. In case the user adds a state to the state machine, the default place type would be used[12]. The information of the type of existing places is retained, because NMF Synchronizations first tries to reuse an existing place to create a correspondence. This is decided based on the `ShouldCorrespond` method, so in this case based on the place name. More complex heuristics are possible, but must be implemented by the transformation developer.

The entire synchronization between finite state machines and Petri nets in NMF Synchronizations consists of 4 synchronization rules in a single file with 92 lines of code in the usual C# coding style[13]. A functionally equivalent[14] implementation in eMoflon required 5 TGG rules 217 lines in total[15], plus a correspondence definition with 32 lines. The coding style in both languages is similar to some degree such that these numbers are roughly comparable. They show that in this scenario, the specification of a synchronization in NMF Synchronizations is more concise than the eMoflon solution in terms of pure lines of code.

However, having a solution with less lines of code does not always indicate a better (for example more understandable) solution and indeed, the NMF Synchronizations solution has many more syntactic boilerplate as for example method signatures have to be repeated. Therefore, lines of code are not a very accurate instrument to measure the conciseness of a solution. Nevertheless, the comparison shows that the specification in synchronization blocks can be done in an internal language in a very compact manner about as declarative as in a TGG implementation such as eMoflon. Whether or to what degree this finding applies more generally is not subject of this paper.

Both solutions are also available online[16].

Our synchronization language is extensible since transformation developers may easily implement new lenses as discussed in Section 5.1 and store them anywhere. eMoflon also allows the developer to extend the transformation framework, but this requires the developer to switch the language and implement the extension in pure Java. Here, internal languages such as NMF Synchronizations have the benefit that developers do not have to switch the language as the integration of host language code usually can be done much more easily. In particular, as discussed in Section 5.1, extensions are simply methods with an annotation. These extensions can be made directly at the synchronization rule, in case it is only used in one place, or can be extracted into a library.

7 Validation and Evaluation

We tested the correctness and evaluated the performance of NMF Synchronizations by applying it to the finite state machines to Petri Nets example that we already used to explain the approach. In typical applications of a model synchronization, the LHS side is edited in subsequent edit operations either performed

[12] Perhaps, the type for places is abstract, in that case, the synchronization would indeed have to be changed to specify a default place type for this case.

[13] This includes 14 blank lines and 28 lines with only braces.

[14] up to propagation of partial changes as discussed above

[15] This includes 41 blank lines and 47 lines with only braces.

[16] eMoflon rules: `https://github.com/NMFCode/SynchronizationsBenchmark/tree/master/eMoflon/FiniteStatesToPetriNets/src/org/moflon/tgg/mosl/rules`, NMF Synchronizations implementation: `https://github.com/NMFCode/SynchronizationsBenchmark/blob/master/Transformations/SynchronizationsImplementation.cs`

by a user through an editor or programmatically. Then, the appropriate RHS model is required for analysis purposes or as an alternate view on the modeled reality. For such subsequent model changes, it is important to minimize the response time from changing the LHS model to having the RHS model updated accordingly (or vice versa). Often it is also important to get a change notification to be able to understand what changes in the RHS model were caused by the changes to the LHS model but although such change notifications can be supplied by NMF Synchronizations with change propagation enabled we do not take this feature into account for the evaluation. Furthermore, the evaluation only considers changes from the LHS to the RHS.

To analyze the response time from elementary changes in the finite state machine to the updated Petri Net, we designed a benchmark where we generate a sequence of 20 elementary model changes to the finite state machine. After each model change, we ensure that the Petri Net is changed accordingly, either by performing change propagation or by regenerating the net fresh from scratch. To take the different sizes of finite state machines into account, we performed our experiment for different sizes ranging from 10 to 50,000 states. The generated workload on these finite state machines shall reflect edit operations as done by a user. In particular, we generate the following elementary changes (percentage on the overall change workload in brackets):

- Add a state to the finite state machine (30%)
- Add a transition to the finite state machine with random start and end state (30%)
- Remove a random state and all of its incoming and outgoing transitions (10%)
- Remove a random transition from the finite state machine (10%)
- Toggle end state of a random state (5%)
- Change the target state of a randomly selected transition to a random other state (5%)
- Rename a state (9%)
- Rename the finite state machine (1%)

The evaluation works as follows: For every run of our benchmark, we generate a finite state machine of a given size n representing the number of states with $2n$ transitions. We then generate a sequence of 20 elementary model changes acting on randomly selected model elements of the finite state machine. For each of these actions, the action itself must be performed and the Petri Net must be updated or newly created appropriately.

We compare four implementations of this task. The first solution is using NMF Synchronizations in batch mode, i.e. the synchronization is run as a transformation from its left side to its right side with change propagation switched off. Next, we use the same synchronization code without any modification and use it in incremental mode, i.e. from left to right with change propagation mode switched on to OneWay. Third, we use an implementation for this transformation task in NTL, basically taken from previous work [29]. This solution works similar to the batch mode version but lacks some of the overhead implied by the NMF Synchronizations implementation. NMF Transformations used with NTL showed good performance results compared with other (batch mode) model transformation languages at the TTC 2013 [35, 36] so we think it is a fair comparison. Lastly, we compare the NMF solutions with a solution in eMoflon, which has been discussed in Section 6.

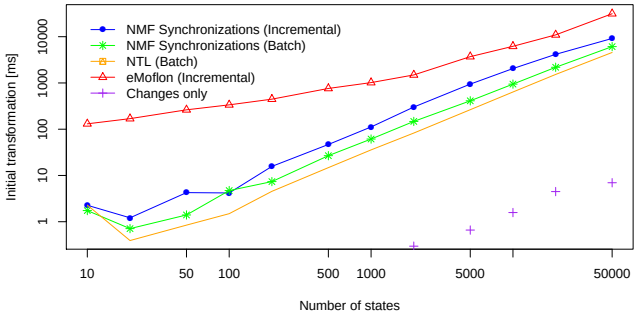


Figure 13 Performance results for the initial transformation

We did two runs of the experiment. In the first run, we check the generated Petri Net after each workload item in order to test the correctness of NMF Synchronizations. Here, we basically assume the implementation in NMF Transformations correct. In a second run of the experiment, we evaluated the execution time to apply all the elementary model changes in sequence and updating the Petri Net accordingly after each change (either by rerunning the transformation or by propagating changes). The application of 20 elementary model changes and updating the Petri Net still is a matter of milliseconds, at least for small models, but this way the precision gets in a reasonable scale.

To compare with eMoflon, we transformed the generated changes into delta specifications that can be understood by eMoflon. We then start a Java process running the eMoflon solution for the initial transformation. Afterwards, we subsequently load the delta specifications one after another and integrate them to the target model. Here, the time for transforming the changes into the delta specifications, serializing them in the benchmark driver and deserializing them in the eMoflon process is not taken into account. Rather,

we only measure the time for the integration. However, this also means that we sum up 20 time measurements, which means that these results are not as accurate.

All time measurements include performing the actual changes to the models. However, for very large models, changes such as removing an element become costly operations. Therefore, we also recorded the time to perform the generated changes without any propagation to the Petri net.

Figures 13 and 14 show the performance results achieved on an Intel Core i7-4710MQ processor clocked at 2.49Ghz in a system with 16GB RAM. The code for our used benchmark is available as open source on Github[17] so that the interested reader can obtain results for any other machines as well. The implementation uses hardware performance counters to maximize the accuracy of results and we repeated every measurement 20 times. In particular, Figure 13 shows the results for the initial transformation, Figure 14 shows the time to run 20 generated changes. The R scripts to generate these figures are also publicly available in the same GitHub repository.

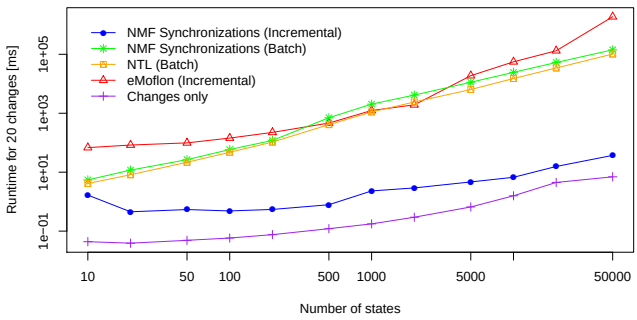


Figure 14 Performance results running 20 randomly generated changes

For the initial run of the transformation, we can see that NMF Transformations and NMF Synchronizations are much faster than the eMoflon solution. The NMF Synchronizations implementation running in batch mode is about as fast as the unidirectional implementation using NMF Transformations, while the incremental mode adds a slight overhead. This is because the engine has to create dependency graphs for all lenses, but as these lenses are rather simple, this overhead is not very large. All NMF solutions take longer to initialize for the smallest model size but this is only due to the Just-in-Time-Compiler.

For incremental change sequences, the results indicate that even for very small models such as a finite state machine with just 20 states, it is already beneficial to use the change propagation built into NMF Synchronizations: Recreating the Petri net after every change takes more than 18 times as long as propagating the change. Compared to NMF Synchronizations, the factor is even at 25. For larger models, the speedup gets higher and for the largest models with 50,000 states, the change propagation is factor 2,750 faster than recreating the Petri net using NTL or factor 3,800 faster than NMF Synchronizations in batch mode. This is because the time to propagate a change merely depends on the size of the change, rather than the overall model size: The curve for incremental change propagation in blue is almost flat in a logarithmic plot.

However, there are some operations such as the model change operations themselves, that have a linear complexity[18], which is why the speedup does not grow linearly with the size of the model. As a consequence, the distance between the blue and the purple curve in Figure 14 becomes smaller the change propagation overhead becomes smaller in relation to actual model changes: While for 20 states, the actual model changes only take about 8% of the time for incremental change propagation, this share is at 28% for 20,000 states. Given that the incremental change propagation includes a roughly similar model manipulation in the RHS, this means that the overhead for change propagation becomes very small.

Comparing the results with eMoflon, we can see that eMoflon is slower than the incremental NMF Synchronizations implementation and for most model sizes also slower than the implementations in NTL or in batch mode. The eMoflon solution also does not scale well with an increasing model size: While the curve for the incremental NMF Synchronizations solution is almost flat for up to 5,000 states, indicating a fully incremental solution, the curve for eMoflon is steeper already for smaller model sizes, indicating a worse scalability.

However, the performance of the synchronization depends on many factors. Indeed, even in the very small example of the synchronization between finite state machines and Petri nets, some types of changes such as name changes are much faster to propagate than others such as adding or removing states. As a reason, the propagation for the latter includes the execution of a transformation rule meanwhile the propagation of a name change simply means to copy the new name over to the target model. Therefore, the speedups reported in the paper are specific to the given mix of model changes and cannot be generalized to an arbitrary change sequence in the case of the example syn-

[17] `http://github.com/NMFCode/SynchronizationsBenchmark/`

[18] For example, deleting an element from an ordered list is a $O(n)$ operation.

chronization or any other model synchronization with potentially more complex patterns.

Nevertheless, the evaluation shows that the overhead for change propagation stays approximately at a constant level, indicating that NMF Synchronizations is fully incremental in the terminology of Giese and Wagner [23]. Especially for large models, the overhead is small. We see this as a consequence of the fact that changes to the RHS can be directly constructed from the model changes: The definition of the repair operators in Definitions 22 and 23 are straight and explicit.

The correctness tests are turned to automated unit tests and are therefore guaranteed to pass for every new release of NMF Synchronizations.

8 Limitations of the language

Currently, we assume in our implementation that a correspondence between model elements once established will not change during the lifecycle of both objects. This has an impact mainly on synchronization rule instantiation in the presence of filter conditions.

Consider for instance two metamodels of family relations where the gender is realized as `IsFemale` attribute (the *Persons* metamodel on the left hand of Figure 15) and using an inheritance relation (the *FamilyRelations* metamodel on the right hand of Figure 15).

An instance of the `Person` class of the *Persons* metamodel with gender *male* clearly corresponds to an instance of the `Male` class on the *FamilyRelations* metamodel. Likewise, instances with gender *female* should correspond to `Female` instances. In many cases, one would like to generalize these two correspondences into a generalized correspondence between `Person` instances in the *Persons* metamodel and `Person` instances of the *FamilyRelations* metamodel. However, because it is only an attribute, the gender of a `Person` instance of the *Persons* metamodel may change. In that case, the corresponding model element has to change (since the type of an object must not change) and all references have to be updated accordingly. Thus, the identity of one of the model elements of a correspondence relation changes.

The implementation actually does allow filters to be set on synchronization rule instantiations, but the current engine implementation throws an exception as soon as any change would cause the correspondence relation to break. This issue can of course be worked around by simply creating separate synchronization rules but the lack of a generalized isomorphism may hamper the conciseness in other parts of the model synchronization.

Though the implementation currently does not support these cases, this is only a technical problem. The formalization is fine with this: The synchronization blocks of the broken correspondence have to stop pushing incremental updates and the new correspondence link has to be created.

9 Related Work

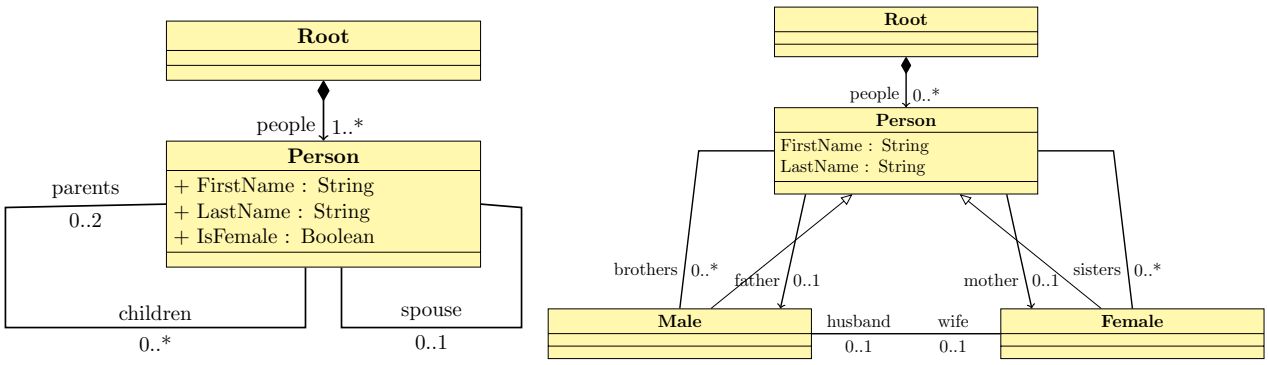
Our approach is not the first model transformation language implemented as an internal DSL and neither is the first to provide a language for bidirectional languages or languages with change propagation, yet we think we are the first to combine these concepts. The remainder of this section details on each of these aspects of model transformation languages.

Model transformation languages as internal languages
Some experiences exist with creating model transformation languages as internal languages like RubyTL [12], ScalaMTL [20], FunnyQT [41] or SDMLib[19]. The goals to use an existing language as host language are diverse and range from an easier implementation [4], reuse of the static type system [20], inherited tool support [29], reusing the expression evaluation, easier integration into the host language up to mitigated language adoption [37]. A detailed analysis on how an internal model transformation language should be designed to inherit tool support is discussed in previous work [34]. The degree in which these goals can be met depends very much on the selected host language, as e.g. tool support can only be inherited if some tool support exists but a concise syntax can usually only be achieved with host languages having a rather flexible syntax.

An example of the latter is SDMLib [64], which provides an internal DSL embedded in Java and uses the Fujaba [53] tool internally. Transformations are specified in a fluent method chaining syntax operating on generated code for each metamodel. As the language essentially builds up a model for Fujaba, it could inherit change propagation support from Fujaba. Here, we think that our solution is more easy to use since we are working directly on the abstract syntax tree, which is transparent for the developer.

An example of an internal language that is capable of change propagation is VIATRA Query [61], which is implemented as an internal DSL in Xtend. More precisely, VIATRA Query is a tool for incremental graph pattern matching. However, transformations written in this way cannot be inversed. Furthermore, Xtend is not as popular as for example C# such that the language adoption problem remains.

[19] `http://sdmlib.org/`



(a) Persons (b) FamilyRelations

Figure 15 Metamodels of the changing correspondences

Model transformation languages with change propagation Some external model transformations languages support incremental change propagation. Triple Graph Grammars, for example, have been implemented in an incremental manner [21, 22, 24] and with support for concurrent model changes and semi-automatic conflict resolution [26]. Lauder et al. provided an incremental synchronization algorithm that statically analyzes rules to determine the influence range while retaining formal properties [48]. The runtime complexity of this algorithm depends on the change rather than the model. An overview of incremental TGG tools was provided by Leblebici et al. [50].

Jouault and Tisi [44] introduced Reactive ATL, an approach for incremental model synchronization based on the ATL language that works online (the model elements must be kept in memory when changes are made to them). To accomplish this, they only support a subset of the ATL language and make changes to the ATL compiler. In contrast to our approach, ATL only supports unidirectional model transformations.

Bidirectional Model transformation languages A good overview on bidirectional model transformation languages, including a classification scheme, was created by Steven [59] or Hidaka et al. [27]. In this classification of the latter, our approach operates on the technical space of models (MDE) and consists of both forward- and backward-functional correspondences with a Turing-complete (through extensions), bidirectional specification. It reacts on live delta-based changes[20] and all operations are supported through change translation, though the enforcement of these translated changes are only checked dynamically. The explicit trace is only available in memory and we leave it to the user to persist it, if necessary.

[20] Although our formalization uses state-based lenses for simplicity, the implementation actually works delta-based, which is important for multi-valued synchronization blocks.

Among the existing bidirectional model transformation languages is the standardized QVT Relations language [55], though Stevens has identified some semantic issues with it [60]. To the best of our knowledge, there is also no approach that executes QVT-R transformations incrementally.

The most prominent incremental and bidirectional transformation languages may be Triple Graph Grammars, originally introduced by Schürr [56]. Multiple tools implemented this paradigm [28], some of them incrementally [3, 22]. Triple Graph Grammars are usually specified graphically or through external languages. As we have shown, we can express some TGG rules in a single line of code, which we believe is not easily possible for TGGs. Furthermore, our approach is extensible with user-supplied lenses.

Kramer has presented a series of languages for consistency preservation [46] in the context of the Vitruvius framework. From these, the mapping language is closest to our approach as it also defines correspondences between elements that can be enforced automatically in both directions. Similar to our approach, it specifies how the properties of model elements should be synchronized based on isomorphisms. These rules may relate to base isomorphisms. However, the synchronization properties of this approach are not formally proven and it is not clear how this formalism can be implemented in an internal language.

Lastly, Wider has created a bidirectional model transformation language as an internal DSL in Scala [63] that is based on lenses, similar to our approach. However, the approach uses pure lenses that are conceptually limited to tree-based structures.

Other approaches to bidirectional transformation include putback-based systems such as BiGUL [45]. Here, the forward transformation is automatically reconstructed from the backward transformation with the rationale

that the latter may require more attention. In comparison to these approaches, our approach is completely symmetric and therefore also supports the synchronization of heterogeneous models. BiGUL is also implemented as an internal language in Agda and Haskell. However, these languages are no mainstream languages and therefore, the language adoption problem cannot be tackled by these languages.

Lenses Based on the original approach by Foster et al. [17], a multitude of lens variants have been proposed such as delta-lenses [14], symmetric lenses [15] or edit-lenses [40]. An overview and a great comparison can be found by Johnson and Rosebrugh [42].

The main difference from these approaches to our notion of lenses is the different scope of application. While to the best of our knowledge, all other applications of lenses so far have applied them on a model space where an object is an entire model, we apply lenses at a mutable object space where an object is a set of object identities that share a common global state. Essentially, we apply lenses not between models but within models. They are used as a form of model navigation that remembers where it came from such that changes can be persisted.

Because of the different application area, the problems we face are different. The objects we are working with are much simpler (model element identities instead of models), but the shared global state causes us some problems which is why we have to suitably restrict the formalism and weaken the PutGet law. As a consequence, the compositionality of lenses breaks. To solve this problem, we do not strive to see an entire model synchronization as a big lens but rather use several small lenses and combine them to obtain a model synchronization.

We believe that the idea we employed in this paper, to use lenses as a generalization of model references, may also be beneficial for other bidirectional approaches such as TGGs. In essence, synchronization blocks are simply very simple TGG rules and the complexity comes in with more sophisticated lenses. The same lenses could also be made available to generalize the definition of TGG rules in that direction. However, we leave a detailed analysis up to future work.

From an implementation point of view, the `lens` framework by Edward Kmett[21] provides a comparable lens implementation as we use integrated into the NMF framework, but in Haskell. However, to the best of our knowledge, there is no model synchronization language built on top of it, particularly since Haskell only supports immutable objects.

[21] `https://github.com/ekmett/lens`, accessed 19/06/2017

Incremental computation Incremental computation refers to the idea that systems use a dynamic dependency graph to track how to change their outputs when the input changes rather than recomputing the whole program output. This is usually achieved either by adding explicit new language primitives [1, 8]. However, Chen et al. [10] presented an approach to infer these newly added primitives from type annotations so that effectively self-adjusting programs may be written in StandardML, which is close to NMF Expressions, a foundation of our approach. However, the approach of Chen is based on a general-purpose language that is not suitable for the specification of model transformations or synchronizations. Since the language primitives in NMF Synchronizations are fitted to the concepts of model transformation, we have more insights on how to execute the transformations incrementally.

Rather specialized on a particular class of computation, VIATRA Query [61] supports incremental pattern matching. Through a translation of OCL to graph patterns [6], this partially can be applied to textual model transformations as well. Incremental pattern matches can be used for change-driven transformations [7]. However, this approach is different to our approach because synchronization blocks specify a consistency relation, while change-driven transformations usually rather specify a concrete reaction to a given change.

10 Conclusion

In this paper, we have presented NMF Synchronizations, an internal DSL for bidirectional model transformation and synchronization with optional change propagation. Although it is only a proof-of-concept and therefore has some limitations, the approach encourages the development of model transformation languages as internal DSLs: It shows that one of the key challenges, supporting declarative model transformations, can be overcome. In particular, NMF Synchronizations supports in total 18 different operation modes from a single specification. For a synthetic example, the optional change propagation has shown speedups of up to multiple orders of magnitude, whereas the classic batch mode execution is still available with low overhead.

Acknowledgements

We would like to thank the anonymous reviewers that helped us to shape the paper as it is of today. Furthermore, we would like to thank Anthony Anjorin, who has helped us with the eMoflon-implementation of

the example finite state machines to Petri nets synchronization.

References

1. Acar, U.A.: Self-adjusting computation. Ph.D. thesis, Citeseer (2005)
2. Acar, U.A., Ahmed, A., Blume, M.: Imperative self-adjusting computation. In: ACM SIGPLAN Notices, vol. 43, pp. 309–322. ACM (2008)
3. Anjorin, A., Lauder, M., Patzina, S., Schürr, A.: emoflon: Leveraging emf and professional case tools. In: 3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe2011), Lecture Notes in Informatics (2011)
4. Barringer, H., Havelund, K.: TraceContract: A Scala DSL for trace analysis. Springer (2011)
5. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software **82**, 3–22 (2009). DOI 10.1016/j.jss.2008.03.066. URL http://dx.doi.org/10.1016/j.jss.2008.03.066
6. Bergmann, G.: Translating ocl to graph patterns. In: Model-Driven Engineering Languages and Systems, pp. 670–686. Springer (2014)
7. Bergmann, G., Ráth, I., Varró, G., Varró, D.: Change-driven model transformations. Software & Systems Modeling **11**(3), 431–461 (2012). DOI 10.1007/s10270-011-0197-9. URL http://dx.doi.org/10.1007/s10270-011-0197-9
8. Burckhardt, S., Leijen, D., Sadowski, C., Yi, J., Ball, T.: Two for the price of one: a model for parallel and incremental computation. In: ACM SIGPLAN Notices, vol. 46, pp. 427–444. ACM (2011)
9. Carlsson, M.: Monads for incremental computing. ACM SIGPLAN Notices **37**(9), 26–35 (2002)
10. Chen, Y., Dunfield, J., Hammer, M.A., Acar, U.A.: Implicit self-adjusting computation for purely functional programs. Journal of Functional Programming **24**(01), 56–112 (2014)
11. Crole, R.L.: Categories for types. Cambridge University Press (1993)
12. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: Rubytl: A practical, extensible transformation language. In: Model Driven Architecture–Foundations and Applications, pp. 158–172. Springer (2006)
13. Diskin, Z.: Model synchronization: Mappings, tiles, and categories. In: Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers, pp. 92–165 (2009). DOI 10.1007/978-3-642-18023-1_3. URL http://dx.doi.org/10.1007/978-3-642-18023-1_3
14. 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). DOI 10.5381/jot.2011.10.1.a6. URL http://www.jot.fm/contents/issue_2011_01/article6.html
15. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: The symmetric case. In: J. Whittle, T. Clark, T. Kühne (eds.) Model Driven Engineering Languages and Systems, *Lecture Notes in Computer Science*, vol. 6981, pp. 304–318. Springer Berlin Heidelberg (2011). DOI 10.1007/978-3-642-24485-8_22. URL http://dx.doi.org/10.1007/978-3-642-24485-8_22
16. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. SIGPLAN Not. **40**(1), 233–246 (2005). DOI 10.1145/1047659.1040325
17. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (2007). DOI 10.1145/1232420.1232424. URL http://doi.acm.org/10.1145/1232420.1232424
18. Fowler, M.: Domain-specific languages. Addison-Wesley Professional (2010)
19. Fowler, M., Parsons, R.: Domain Specific Languages, 1st edn. Addison-Wesley, Reading, MA, USA (2010)
20. George, L., Wider, A., Scheidgen, M.: Type-Safe model transformation languages as internal DSLs in Scala. In: Theory and Practice of Model Transformations, pp. 160–175. Springer (2012)
21. Giese, H., Hildebrandt, S.: Efficient model synchronization of large-scale models. 28. Universitätsverlag Potsdam (2009)
22. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: Model Driven Engineering Languages and Systems, pp. 543–557. Springer (2006)
23. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: O. Nierstrasz, J. Whittle, D. Harel, G. Reggio (eds.) Model Driven Engineering Languages and Systems, *Lecture Notes in Computer Science*, vol. 4199, pp. 543–557. Springer Berlin / Heidelberg (2006)
24. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software & Systems Modeling **8**(1), 21–43 (2009)

25. Hammer, M.A., Acar, U.A., Chen, Y.: Ceal: a c-based language for self-adjusting computation. In: ACM Sigplan Notices, vol. 44, pp. 25–37. ACM (2009)
26. Hermann, F., Ehrig, H., Ermel, C., Orejas, F.: Concurrent model synchronization with conflict resolution based on triple graph grammars. In: J. de Lara, A. Zisman (eds.) Fundamental Approaches to Software Engineering, *Lecture Notes in Computer Science*, vol. 7212, pp. 178–193. Springer Berlin / Heidelberg (2012)
27. Hidaka, S., Tisi, M., Cabot, J., Hu, Z.: Feature-based classification of bidirectional transformation approaches. Software & Systems Modeling pp. 1–22 (2015). DOI 10.1007/s10270-014-0450-0. URL http://dx.doi.org/10.1007/s10270-014-0450-0
28. Hildebrandt, S., Lambers, L., Giese, H., Rieke, J., Greenyer, J., Schäfer, W., Lauder, M., Anjorin, A., Schürr, A.: A survey of triple graph grammar tools. Electronic Communications of the EASST **57** (2013)
29. Hinkel, G.: An approach to maintainable model transformations using an internal DSL. Master's thesis, Karlsruhe Institute of Technology (2013)
30. Hinkel, G.: An NMF Solution to the Java Refactoring Case. In: L. Rose, T. Horn, F. Krikava (eds.) Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, *CEUR Workshop Proceedings*, vol. 1524, pp. 95–99. CEUR-WS.org (2015)
31. Hinkel, G.: Change Propagation in an Internal Model Transformation Language. In: D. Kolovos, M. Wimmer (eds.) Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings, pp. 3–17. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-21155-8_1. URL http://dx.doi.org/10.1007/978-3-319-21155-8_1
32. Hinkel, G.: NMF: A Modeling Framework for the .NET Platform. Tech. rep., Karlsruhe Institute of Technology, Karlsruhe (2016). URL http://nbn-resolving.org/urn:nbn:de:swb:90-537082
33. Hinkel, G., Goldschmidt, T.: Tool Support for Model Transformations: On Solutions using Internal Languages. In: Modellierung 2016 (2016)
34. Hinkel, G., Goldschmidt, T., Burger, E., Reussner, R.: Using Internal Domain-Specific Languages to inherit Tool Support and Modularity for Model Transformations. Software & Systems Modeling pp. 1–27 (2017). DOI 10.1007/s10270-017-0578-9. URL http://rdcu.be/oTED
35. Hinkel, G., Goldschmidt, T., Happe, L.: An NMF solution for the flowgraphs case at the TTC 2013. In: Proceedings Sixth Transformation Tool Contest, TTC 2013, Budapest, Hungary, 19-20 June, 2013., pp. 37–42 (2013). DOI 10.4204/EPTCS.135.5. URL https://doi.org/10.4204/EPTCS.135.5
36. Hinkel, G., Goldschmidt, T., Happe, L.: An NMF solution for the petri nets to state charts case study at the TTC 2013. In: Proceedings Sixth Transformation Tool Contest, TTC 2013, Budapest, Hungary, 19-20 June, 2013., pp. 95–100 (2013). DOI 10.4204/EPTCS.135.12. URL https://doi.org/10.4204/EPTCS.135.12
37. Hinkel, G., Groenda, H., Vannucci, L., Denninger, O., Cauli, N., Ulbrich, S.: A Domain-Specific Language (DSL) for Integrating Neuronal Networks in Robot Control. In: 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering (2015)
38. Hinkel, G., Happe, L.: Using component frameworks for model transformations by an internal DSL. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for Component-Based Software Systems co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), *CEUR Workshop Proceedings*, vol. 1281, pp. 6–15. CEUR-WS.org (2014)
39. Hinkel, G., Happe, L.: An NMF Solution to the TTC Train Benchmark Case. In: L. Rose, T. Horn, F. Krikava (eds.) Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, *CEUR Workshop Proceedings*, vol. 1524, pp. 142–146. CEUR-WS.org (2015)
40. Hofmann, M., Pierce, B., Wagner, D.: Edit lenses. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, pp. 495–508. ACM, New York, NY, USA (2012). DOI 10.1145/2103656.2103715. URL http://doi.acm.org/10.1145/2103656.2103715
41. Horn, T.: Model querying with funnyqt. In: Theory and Practice of Model Transformations, pp. 56–57. Springer (2013)
42. Johnson, M., Rosebrugh, R.D.: Unifying set-based, delta-based and edit-based lenses. In: Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016., pp. 1–13 (2016). URL

http://ceur-ws.org/Vol-1571/paper_13.pdf
43. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Satellite Events at the MoDELS 2005 Conference, pp. 128–138. Springer (2006)
44. Jouault, F., Tisi, M.: Towards Incremental Execution of ATL Transformations. In: Theory and Practice of Model Transformations: ICMT 2010, pp. 123–137. Springer International Publishing (2010)
45. Ko, H.S., Zan, T., Hu, Z.: Bigul: A formally verified core language for putback-based bidirectional programming. In: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '16, pp. 61–72. ACM, New York, NY, USA (2016). DOI 10.1145/2847538.2847544. URL http://doi.acm.org/10.1145/2847538.2847544
46. Kramer, M.E.: Specification languages for preserving consistency between models of different languages. Ph.D. thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany (2017). DOI 10.5445/IR/1000069284. URL http://nbn-resolving.org/urn:nbn:de:swb:90-692845
47. Křikava, F., Collet, P., France, R.B.: Sigma: Scala internal domain-specific languages for model manipulations. In: Model-Driven Engineering Languages and Systems, pp. 569–585. Springer (2014)
48. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient model synchronization with precedence triple graph grammars. In: H. Ehrig, G. Engels, H.J. Kreowski, G. Rozenberg (eds.) Graph Transformations, *Lecture Notes in Computer Science*, vol. 7562, pp. 401–415. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-33654-6_27. URL http://dx.doi.org/10.1007/978-3-642-33654-6_27
49. Lawvere, F.W., Rosebrugh, R.: Sets for mathematics. Cambridge University Press (2003)
50. Leblebici, E., Anjorin, A., Schürr, A., Hildebrandt, S., Rieke, J., Greenyer, J.: A comparison of incremental triple graph grammar tools. Electronic Communications of the EASST **67** (2014)
51. Meyerovich, L.A., Rabkin, A.S.: Empirical analysis of programming language adoption. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, pp. 1–18. ACM (2013)
52. Mohagheghi, P., Gilani, W., Stefanescu, A., Fernandez, M.A.: An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. Empirical Software Engineering **18**(1), 89–116 (2013)
53. Nickel, U., Niere, J., Zündorf, A.: The FUJABA Environment. In: Proceedings of the 22Nd International Conference on Software Engineering, ICSE '00, pp. 742–745. ACM, New York, NY, USA (2000). DOI 10.1145/337180.337620. URL http://doi.acm.org/10.1145/337180.337620
54. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification. http://www.omg.org/spec/QVT/1.1/PDF/ (2011)
55. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (ptc/07-07-07) (2007). URL http://www.omg.org/docs/ptc/07-07-07.pdf
56. Schürr, A.: Specification of graph translators with triple graph grammars. In: Graph-Theoretic Concepts in Computer Science, pp. 151–163. Springer (1994)
57. Sendall, S., Kozaczynski, W.: Model transformation the heart and soul of model-driven software development. Tech. rep. (2003)
58. Staron, M.: Adopting model driven software development in industry–a case study at two companies. In: Model Driven Engineering Languages and Systems, pp. 57–72. Springer (2006)
59. Stevens, P.: A landscape of bidirectional model transformations. In: R. Lämmel, J. Visser, J.a. Saraiva (eds.) Generative and Transformational Techniques in Software Engineering II, *Lecture Notes in Computer Science*, vol. 5235, pp. 408–424. Springer Berlin Heidelberg (2008). DOI 10.1007/978-3-540-88643-3_10. URL http://dx.doi.org/10.1007/978-3-540-88643-3_10
60. Stevens, P.: Bidirectional model transformations in qvt: semantic issues and open questions. Software & Systems Modeling **9**(1), 7–20 (2010). DOI 10.1007/s10270-008-0109-9. URL http://dx.doi.org/10.1007/s10270-008-0109-9
61. Ujhelyi, Z., Bergmann, G., Ábel Hegedús, Ákos Horváth, Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: Emf-incquery: An integrated development environment for live model queries. Science of Computer Programming **98, Part 1**, 80 – 99 (2015). DOI http://dx.doi.org/10.1016/j.scico.2014.01.004. URL http://www.sciencedirect.com/science/article/pii/S0167642314000082
62. Wagelaar, D., Van Der Straeten, R., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. Software & Systems Modeling **9**(3), 285–309 (2010)
63. Wider, A.: Implementing a bidirectional model transformation language as an internal DSL in scala. In: Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference, *CEUR Workshop Proceedings*, vol. 1133, pp. 63–70.

CEUR (2014). URL http://ceur-ws.org/Vol-1133/paper-10.pdf

64. Zündorf, A., George, T., Lindel, S., Norbisrath, U.: Story Driven Modeling Libary (SDMLib): an Inline DSL for modeling and model transformations, the Petrinet-Statechart case. EPTCS (2013)

Georg Hinkel received his B. Sc. and M. Sc. degrees in computer science from the Karlsruhe Institute of Technology (KIT), in 2011 and 2014, respectively, and the B. Sc. degree in math in 2012. Currently, he is a researcher in the research division Software Engineering of the FZI research center for information technologies. His research interest covers model-driven engineering, domain-specific languages and incrementality.

Erik Burger studied Computer Science at the University of Karlsruhe. After completing his diploma thesis at SAP, Walldorf, he joined the Software Design and Quality (SDQ) group at Karlsruhe Institute of Technology in 2009, where he completed his PhD thesis in 2014. His main research interests cover model-driven software development, metamodel evolution, and view-based modeling. He is currently involved in the development of the view-centric Vitruvius approach, which is based on a virtual single underlying model and the automatic generation of view types and views.