Change Propagation in an Internal Model Transformation Language

Georg Hinkel

Forschungszentrum Informatik (FZI)
Haid-und-Neu-Straße 10-14
Karlsruhe, Germany
hinkel@fzi.de

Abstract Despite good results, Model-Driven Engineering (MDE) has not been widely adopted in industry. According to studies by Staron and Mohaghegi [1], [2], the lack of tool support is one of the major reasons for this. Although MDE has existed for more than a decade now, tool support is still insufficient. An approach to overcome this limitation for model transformations, which are a key part of MDE, is the usage of internal languages that reuse tool support for existing host languages. On the other hand, these internal languages typically do not provide key features like change propagation or bidirectional transformation. In this paper, we present an approach to use a single internal model transformation language to create 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 48 compared to classical batch transformations.

1 Introduction

Model-driven engineering (MDE) is an approach to raise the level of abstraction of systems in order to be able to cope with increasing system complexity. However, while MDE is widely adopted in academia, it is not as popular in industry, primarily because of the lack of stable tool support [1], [2]. In addition, Meyerovich et al. [3] 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. In MDE, the 'heart and soul' are model transformations [4], but as general-purpose languages are not suitable for this task [4], there is a plethora of specialized model transformation languages. This may hamper the adoption of MDE in industry as well as developers may not want to use model transformation languages for the reasons found by Meyerovich.

To solve both of these issues, a promising approach is to integrate the abstractions from model transformation languages into general-purpose languages in the form of internal languages. This way, tool support for the host language

can be inherited and developers may stick to the languages that they are used to.

Therefore, several languages exist that follow this approach. However, we observed that they only operate in a rather imperative way. In this context, rather imperative means that these languages contain less control flow abstractions than declarative model transformation languages such as QVT-R [5]. In particular, only few approaches support bidirectional transformation and to the best of our knowledge none of these languages supports change propagation, a feature that is mostly provided by declarative languages like Triple Graph Grammars (TGGs) that have an implementation supporting change propagation [6]–[8].

In this paper, we show that this is not a general restriction of internal languages. For this, we implement an internal language in C# supporting multi-directional model transformation as well as multiple change propagation patterns. This language has a few limitations that we discuss in Section 7, which we believe are only technical restrictions.

We have validated our approach on 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.

The rest of this paper is structured as follows: Section 2 explains our running example with the synchronization of Finite State Machines and Petri Nets. Section 3 introduces some foundations. In particular, Section 3.1 explains the internal model transformation language (MTL) that the approach is based on for the model transformation part while Section 3.2 explains self-adjusting computations that the change propagation mechanism is based on. Section 4 explains in short how we extended this approach for reversable expressions. Section 5 describes our prototype language and the various operation modes. Section 6 validates our language on a synthetic example. Section 7 then shows the limitations of our approach. Finally, Section 8 lists related work and Section 9 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 which is why 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) [9].

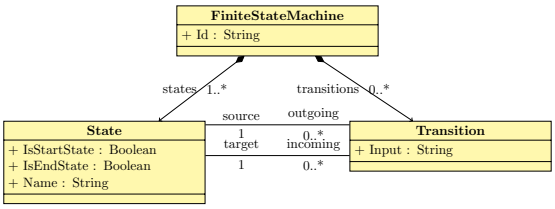


Figure 1. The metamodel for finite state machines

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.

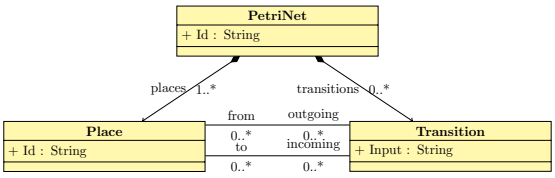


Figure 2. The metamodel for Petri Nets

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 now 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. End states are transformed to a place with an outgoing transition that has no target place and therefore 'swallows' tokens.

The backward transformation from Petri Nets to Finite state charts 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 self-adjusting computation. Thus, we briefly introduce both of them in this section.

3.1 NMF Transformations

NMF stands for .NET Modeling Framework[1] and is an open-source project to support MDE on the .NET platform. NMF Transformations [10] 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 [5] and ATL [11] but work with arbitrary .NET objects. The language has been applied internally in NMF and at the Transformation Tool Contest (TTC) in 2013 [12], [13].

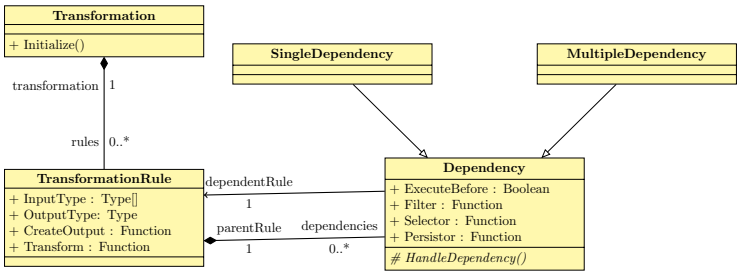


Figure 3. Abstract syntax of NMF Transformations

Figure 3 shows an excerpt of NMF Transformations' abstract syntax. Model transformations consist of transformation rules that are in NTL represented as public nested classes of a model transformation class. The transformation rules create computations that represent a transformation of a particular model element. Transformation rules can have dependencies specifying what other transformation rules should be called if a computation is executed. These dependencies may contain selectors, filters and persistors which are called to register the dependent model elements on the target. 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 NTL 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 transformation context which can be used to query the trace.

[1] `http://nmf.codeplex.com/`

3.2 Self-adjusting 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 represented by a monad [14] and a system that creates a dynamic dependency graph based on these [15]. Further research has shown that such self-adjusting programs can be implicitly inferred from a batch specification [16]. 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 [16] since they do not contain side effects. However, approaches for imperative languages exist as well [17], [18] but are not working implicitly.

In this paper, we use an implementation of these ideas within the NMF project, NMF Expressions[2]. 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 [16] it does not operate on the source code and therefore can be used in a compiling environment. NMF Expressions operates on CLR objects that implement the .NET platform default notification interfaces, similar to the EMF Notification API. A model representation code that implements these interfaces can be generated from a metamodel using NMF code generators.

4 Reversability of expressions

The essence of modifiable references from self-adjusting computation is that they inform clients whenever their value has changed. For change propagation, it is also necessary to be able to change it if possible. Therefore, we have refined the monads used in NMF Expressions (`INotifyValue` and `INotifyEnumerable`) to account for a categorial interpretation of lenses [19]. In this interpretation, a lense l between types A and B consists of a partial function $l \nearrow: A \to B$ called the get function and $l \searrow: A \times B \to A$ called the put function. In category theory, A and B are objects of the category of types.

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 lense, the expression resembles the get function. The lense now 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 lense. The lense is represented by its get function which we expect to be decorated with a put function reference.

[2] `http://nmfexpressions.codeplex.com/`

For memory efficiency reasons, the analysis whether a given expression is constant is only performed at runtime. Thus, we use a twofold mechanism. We let the classes implementing the dynamic dependency graph nodes optionally implement the refined lense monad interface and added a property to this interface to question whether an expression really is invertible, much like the `IsReadonly` property used in .NET collections.

Thus, at initialization time we know that the expression $x + c$ might be a lense, depending on whether at least one of either x or c is constant. On the other hand, other operators like the 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.

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 $x.FirstOrDefault() = y$, 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. 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). 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 lense semantics. In the example of `FirstOrDefault`, we have two versions 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 don't know how a particular operator has been used. Thus, we decorate each operator with its reversability behavior explicitly.

5 Multimode model transformations with an internal DSL

This section will first demonstrate NMF applied to the running example of Petri nets and finite state machines and afterwards explain how multimode model synchronization is achieved using this syntax.

5.1 Synchronization of finite state machines and Petri Nets

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 1.

```
1 public class PSM2PN : ReflectiveSynchronization
2 {
3   public class AutomataToNet : SynchronizationRule<FiniteStateMachine, PetriNet> {...}
4 }
```

Listing 1. A model synchronization in NMF Synchronizations

Similar to TGGs, we distinguish the sources and targets of a model transformation as Left Hand Side (LHS) and Right Hand Side (RHS) although these sides are not represented as graphs. Synchronization rules in 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.

The most important method to override 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.

The second most important method to override is the `DeclareSynchronization` method. Here, we define what actions should be taken if the synchronization rule is executed for two corresponding model elements. The `DeclareSynchronization` method of *AutomataToNet* looks as depicted in Listing 2.

```
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 2. The DeclareSynchronization method of AutomataToNet

The meaning of the statements in Listing 2 is as follows: When handling the synchronization of a finite state machine with a Petri Net, the synchronization engine should establish correspondencies 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 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.

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. 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 greather than zero. To find the corresponding transition on the respective other side, the `ShouldCorrespond` method depicted in Listing 3 is used.

```
1  public override bool ShouldCorrespond(FSM.Transition left, PN.Transition right, ISynchronizationContext
       context)
2  {
3    var stateToPlace = SyncRule<StateToPlace>().LeftToRight;
4    return left.Input == right.Input
5      && right.From.Contains(context.Trace.ResolveIn(stateToPlace, left.StartState))
6      && right.To.Contains(context.Trace.ResolveIn(stateToPlace, left.EndState));
7  }
```

Listing 3. 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 from LHS to RHS and uses it to decide whether the transitions should match. This trace entry exists regardless of the synchronization direction, i.e. the synchronization always creates two trace entries.

Lines 7-9 of Listing 2 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 4. 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 reversible. This is depicted in Listing 4.

Line 10 in Listing 2 tells that the Identifiers of both finite state machine and Petri Net should be synchronized. In this case, it is not necessary to provide a

synchronization rule since both identifiers are strings and the string will just be copied.

5.2 Multimode synchronization

To support multiple modes of transformations, especially to support optional change propagation, it is crucial to step into the compilation process of the language. If some model element is used in a change propagation, it is necessary to create dynamic dependency graphs for these expressions in order to receive updates when these expressions change their value.

Gladly, C# has an option to retrieve lambda expressions as an abstract syntax tree (called expression tree) instead of compiled code. This is the one and only syntax feature that we use from C# that makes our language impossible to implement in other languages (apart from Visual Basic). However, we believe that other languages like Java or in particular Xtend will soon adapt this feature as well, making our approach applicable to other languages.

We support six different synchronization modes that can be combined with three different change propagation modes. The synchronization modes are as follows:

- **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.
- **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.
- **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.
- **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.

We support all synchronization modes and all change propagation modes for all synchronizations. In particular, the synchronization is initialized for all possible modes and the applicable 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. However, we provide overloads of the `Synchronize` and `SynchronizeMany` methods that only act on a particular synchronization direction. This is required as some synchronizations need to assign some expressions that are not reversible and would thus otherwise raise an exception at synchronization initialization.

6 Validation

We tested the correctness and evaluated the performance of NMF Synchronizations by applying it to the Finite States 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 by a user through an editor or programatically. 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.

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 100 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 (10, 20, 50, 100, 200, 500 and 1000 states). The genereated 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 validation 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. We then generate a sequence of 100 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 three implementations of this task. The first option is the solution using NMF Synchronizations running 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. Finally, we use an implementation for this transformation task in NTL, basically taken from previous work [10]. This solution works pretty 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 [12], [13] so we think it is a fair comparison.

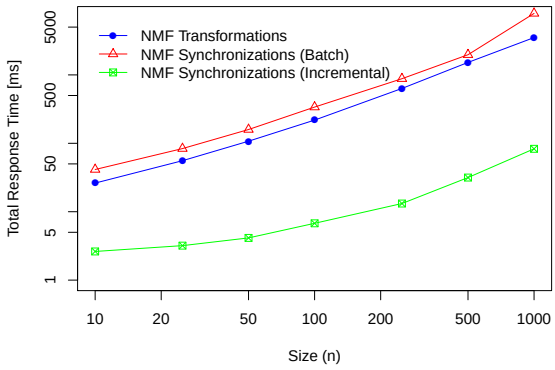


Figure 4. Performance results

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 100 elementary model changes and updating the Petri Net is still a matter of milliseconds, but this way the precision gets in a reasonable scale.

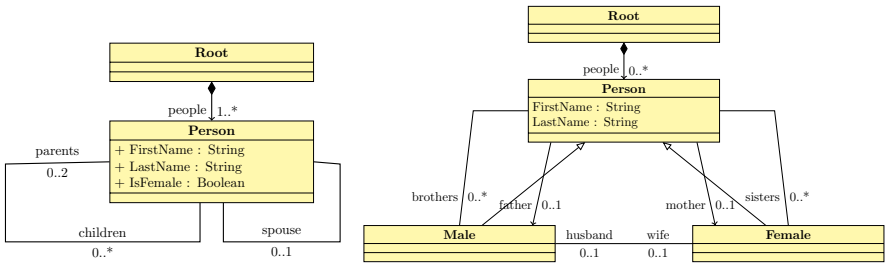
Figure 4 shows the performance results achieved on an AMD Athlon X4 630 processor clocked at 2.81Ghz in a system with 4GB RAM. However, the code

for our used benchmark is available as open source on Codeplex[3] so that the interested reader can obtain results for any other machines as well.

The results indicate that even for very small models such as a finite state machine with just 10 states, it is already beneficial to use the change propagation built into NMF Synchronizations. For the larger models, the speedup gets larger until it stabilizes at about 48 so that the curves appear parallel. Without change propagation, NMF Synchronizations is only slower than NMF Transformations by a constant factor, indicating that the transformation runs efficiently when change propagation is disabled. This may be useful in environments with limited memory or when no change propagation is needed.

7 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 is a strong assumption and there are simple counter-examples. 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 5) and using an inheritance relation (the *FamilyRelations* metamodel on the right hand of Figure 5).



(a) Persons (b) FamilyRelations

Figure 5. Metamodels of the counter-example

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. However, if the gender is changed to female for some reason, then the corresponding model element should then be a Female instance and all references should be updated accordingly. Thus, the identity of one of the model elements of a correspondence relation changes. This is currently not supported by our language although there is no technical limitation.

[3] `http://nmfsynchronizationsbenchmark.codeplex.com/`

8 Related Work

Model transformation languages as internal languages Some experiences exist with creating model transformation languages as internal languages like RubyTL [20], ScalaMTL [21], FunnyQT [22] or SDMLib[4]. The goals to use an existing language as host language are diverse and range from an easier implementation [23], reuse of the static type system [21], inherited tool support [10], reusing the expression evaluation, easier integration into the host language up to less learning points for developers. 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. To the best of our knowledge, current internal transformation languages cannot cope with change propagation. We do also believe that this implementation is only possible if the internal language can see the abstract syntax tree of the host language expressions, which is far away from being common in typical host languages. The only alternative is to use a fluent style internal language that limits the reuse of expressions and tool support.

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 [6]–[8] and with support for concurrent model changes and semi-automatic conflict resolution [24]. Lauder *et al.* [25] provided an incremental synchronization algorithm that statically analyzes rules to determine the influence range while retaining formal properties. The runtime complexity of this algorithm depends on the change not on the model. An overview of incremental TGG tools was provided by Leblebici *et al.* [26].

Self-adjusting computation Self-adjusting or 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 for self-adjusting computation [15], [27]. However, Chen et al. [16] 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 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.

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

9 Conclusion

In this paper, we have presented NMF Synchronizations, an internal DSL for bidirectional model transformation and synchronization with optional change propagation. Despite it is only a proof of concept and therefore has some limitations, the approach encourages the development of model transformation languages as internal DSLs as it shows that one of the key challenges, supporting declarative model transformations, can be overcome. In particular, NMF Synchronizations support in total 18 different operation modes from a single specification. For a synthetic example, the optional change propagation has shown speedups of up to 48, whereas the classic batch mode execution is still available with low overhead.

References

[1] M. Staron, "Adopting model driven software development in industry–a case study at two companies," in *Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 57–72.

[2] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez, "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases," *Empirical Software Engineering*, vol. 18, no. 1, pp. 89–116, 2013.

[3] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, ACM, 2013, pp. 1–18.

[4] S. Sendall and W. Kozaczynski, "Model transformation the heart and soul of model-driven software development," Tech. Rep., 2003.

[5] Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, `http://www.omg.org/spec/QVT/1.1/PDF/`, 2011.

[6] H. Giese and R. Wagner, "Incremental model synchronization with triple graph grammars," in *Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 543–557.

[7] H. Giese and S. Hildebrandt, *Efficient model synchronization of large-scale models*, 28. Universitätsverlag Potsdam, 2009.

[8] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization," *Software & Systems Modeling*, vol. 8, no. 1, pp. 21–43, 2009.

[9] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3–22, 2009.

[10] G. Hinkel, *An approach to maintainable model transformations using internal DSLs*, Master thesis, 2013.

[11] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Satellite Events at the MoDELS 2005 Conference*, Springer, 2006, pp. 128–138.

[12] G. Hinkel, T. Goldschmidt, and L. Happe, "An NMF Solution for the Flowgraphs case study at the TTC 2013," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, 2013.

[13] ——, "A NMF solution for the Petri Nets to State Charts case study at the TTC 2013," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, 2013.

[14] M. Carlsson, "Monads for incremental computing," *ACM SIGPLAN Notices*, vol. 37, no. 9, pp. 26–35, 2002.

[15] U. A. Acar, "Self-adjusting computation," PhD thesis, Citeseer, 2005.

[16] Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar, "Implicit self-adjusting computation for purely functional programs," *Journal of Functional Programming*, vol. 24, no. 01, pp. 56–112, 2014.

[17] U. A. Acar, A. Ahmed, and M. Blume, "Imperative self-adjusting computation," in *ACM SIGPLAN Notices*, ACM, vol. 43, 2008, pp. 309–322.

[18] M. A. Hammer, U. A. Acar, and Y. Chen, "Ceal: a c-based language for self-adjusting computation," in *ACM Sigplan Notices*, ACM, vol. 44, 2009, pp. 25–37.

[19] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem," *SIGPLAN Not.*, vol. 40, no. 1, pp. 233–246, 2005.

[20] J. S. Cuadrado, J. G. Molina, and M. M. Tortosa, "Rubytl: a practical, extensible transformation language," in *Model Driven Architecture–Foundations and Applications*, Springer, 2006, pp. 158–172.

[21] L. George, A. Wider, and M. Scheidgen, "Type-Safe model transformation languages as internal DSLs in Scala," in *Theory and Practice of Model Transformations*, Springer, 2012, pp. 160–175.

[22] T. Horn, "Model querying with funnyqt," in *Theory and Practice of Model Transformations*, Springer, 2013, pp. 56–57.

[23] H. Barringer and K. Havelund, *TraceContract: A Scala DSL for trace analysis*. Springer, 2011.

[24] F. Hermann, H. Ehrig, C. Ermel, and F. Orejas, "Concurrent model synchronization with conflict resolution based on triple graph grammars," in *Fundamental Approaches to Software Engineering*, ser. LNCS, vol. 7212, Springer Berlin / Heidelberg, 2012, pp. 178–193.

[25] M. Lauder, A. Anjorin, G. Varró, and A. Schürr, "Efficient model synchronization with precedence triple graph grammars," in *Graph Transformations*, ser. LNCS, vol. 7562, Springer Berlin Heidelberg, 2012, pp. 401–415.

[26] E. Leblebici, A. Anjorin, A. Schürr, S. Hildebrandt, J. Rieke, and J. Greenyer, "A comparison of incremental triple graph grammar tools," *Electronic Communications of the EASST*, vol. 67, 2014.

[27] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball, "Two for the price of one: a model for parallel and incremental computation," in *ACM SIGPLAN Notices*, ACM, vol. 46, 2011, pp. 427–444.