Tool Support for Model Transformations: On Solutions using Internal Languages

Georg Hinkel[1] and Thomas Goldschmidt[2]

Abstract: Model-driven engineering (MDE) has proven to be a useful approach to cope with todays ever growing complexity in the development of software systems, yet it is not widely applied in industry. As suggested by multiple studies, tool support is a major factor for this lack of adoption. Existing tools for MDE, in particular model transformation approaches, are often developed by small teams and cannot keep up with advanced tool support for mainstream languages such as provided by IntelliJ or Visual Studio. In this paper, we propose an approach to leverage existing tool support for model transformation using internal model transformation languages and investigate design decisions and their consequences for inherited tool support. The findings are used for the design of an internal model transformation language on the .NET platform.

Keywords: Model-driven Engineering, Model Transformation, Internal DSL, C#

1 Introduction

While in the past, increasing complexity of software systems has been tackled by an increasing abstraction of the programming language, it seems like the abstraction level of modern programming languages can hardly be raised without losing general purpose applicability. Therefore, in recent years, many domain-specific languages (DSLs) [Fow10] have been proposed that offer a raised abstraction level at the price of limited expressiveness.

The usage of such domain-specific language requires a specification how these languages are executed. A popular approach for this is to map a domain-specific language to a target platform by a transformation. Since such a transformation determines the execution semantics of the source languages instances, model transformations are sometimes called the ‘heart and soul’ of model-driven approaches that should be supported by dedicated languages [SK03].

This has lead to a variety of model transformation approaches [CH06] incorporating high-level abstractions like the composition of model transformations into rules describing the transformation for a particular model element. While these languages produce more concise, more understandable and sometimes even more performant (cf. eg. [GR13]) model transformations than general purpose languages, the model-driven approach is still not

[1] FZI Forschungszentrum Informatik, Software Engineering Division, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, hinkel@fzi.de
[2] ABB Corporate Research, Software Systems, Wallstadter Straße 59, 68526 Ladenburg, thomas.goldschmidt@de.abb.com

widely adopted in industry and the question is why. Multiple studies [Sta06; Moh+09; Whi+13] suggest that a major factor in this decision is the tool support, particularly also of model transformations given their importance in model-driven approaches. Recent studies [Moh+13] suggest that the tool support is still not satisfactory.

But as the model-driven approach is not widely adopted, relatively few resources are spent to improve the tools, at least in comparison to IDEs for mainstream languages like IntelliJ or Visual Studio. These tools have a massive user base and thus much more resources are spent for the improvement of the tools. Furthermore, many model transformation tools are maintained by researchers with few incentives to implement in principle long-known tool concepts in their model transformation tools, simply because of the lack of insights generated by these mostly laborious tasks.

Furthermore, as Meyerovich suggests [MR13], many developers do not appreciate to switch their primary programming languages and do so only if there is a significant amount of code they can reuse or if management requires them to do so. This is reasonable since such a change often makes valuable knowledge of particular technologies superfluous. Furthermore, similar concepts are sometimes implemented in slightly different ways, causing subtle bugs. As an example in the world of model transformations, the difference between *is-kind-of* and *is-type-of* in OCL often causes confusions for developers not confronted with it on an everyday-basis.

A promising approach to tackle this problem of 1. general purpose languages with the lack of model transformation concepts on the one side and 2. dedicated model transformation languages with lacking tool support on the other is to combine both worlds using an internal DSL. To gain best tool support, the model transformation language should be hosted in a mainstream general-purpose language such as Java or C#. This allows to combine high-level abstractions for model transformations with advanced tool support.

However, so far internal transformation languages do not exist for all often used general-purpose languages. This raises the problem how to design an internal transformation for a language so far not covered and how to implement this transformation DSL with respect to tool support.

In this paper, we tackle this problem as we extract our experience with the design of the NMF Transformation Language (NTL)[Hin13] regarding design for tool support reuse. We discuss the design alternatives how to map model transformation concepts (in particular transformation rules) to code artifacts in an internal DSL and explain limitations and consequences. We then describe briefly how these design decisions are implemented in NTL.

In the remainder of this paper, we first present related work in Section 2. Section 3 discusses how transformation rules can be embedded in general-purpose object-oriented programming concepts aiming for optimal tool support reuse. Section 4 explains our implementation in the internal model transformation language NTL. Finally, we conclude the paper in Section 5.

2 Related Work

There exists a variety of model transformation approaches, surveyed and summarized for example by Czarnecki et al. [CH06]. In the remainder of this section, we concentrate on those implemented as internal DSLs.

A language that has been used as host language for model transformation several times is Scala [GWS12; KCF14] . Scala has not as many users as Java or C# but still advanced tool support is available. However, the language adoption problem remains, i.e. fewer developers know Scala than Java or C#.

Surprisingly, the most often used mainstream languages to the best of our knowledge have hardly been used as a host language for model transformation yet. Next to NTL and its close relative NMF Synchronizations [Hin15], we are only aware of two approaches using Java [TL12] which is rather focussed on pattern matching and SDMLib [Zün+13], an internal DSL for the Fujaba tool using a method chaining syntax.

In this paper, we discuss how an internal model transformation language can be constructed specifically for the inheritance of tool support, but DSLs have been used in a number of approaches for a multitude of different rationales, including the ease of development [BH11], type safety [GWS12] or language adoption [Hin+15].

3 Implementing Transformation Rules with respect to Tool Support

This section discusses the implications of different ways to represent transformation rules in the object-oriented design aiming specifically to gain optimal tool support for model transformations. Our discussion is based on strongly-typed multi-paradigm programming languages such as Java or C# as host languages where there is rich tool support available.

3.1 Editing Support

According to a study on the usage of the Eclipse IDE with 41 Java developers, the basic editing operations like delete, copy, cut and paste are upon the most commonly used editing commands [MKF06]. These commands are supported by any editor. However, the study showed that also more sophisticated tools were often used, indicating that they raise productivity. The most often used tool support beyond the basic commands was code completion that was used by every developer, making up in the average 6.7% of all executed commands. This study was made in 2006 and code completion has been improved by learning from examples, frequency or mined associations [RL08; BMM09]. While many model transformation tools support a basic code completion listing all available members in alphabetic order, more advanced code completion is usually limited to large mainstream IDEs.

Code completion requires strongly typed environments since in this case the tool knows what methods are available for a given object. Thus, the signature of a transformation rule must be known to the compiler. This can be achieved either when the transformation rule is represented by a method or by turning the transformation rule signature into generic type parameter. The problem with the representation as a method is that it is very hard to decide when to create a trace entry. Transformation rules like ATL, ETL and QVT-O solve this problem by dividing the transformation rule execution into phases. In SIGMA which represents transformation rules as methods, the problem is solved by allowing only a single transformation phase specified by the user.

The approach of turning a transformation rule signature into generic type parameters is more flexible, but there are multiple possible implementations. The difference between these implementations is the artifact that represents a given transformation rule. Generic type parameters can be created for methods or classes. In case of generic parameters of methods, each transformation rule would be a call of a generic transformation rule method that creates the transformation rule, taking in additional configuration such as different phases of the transformation.

```
var state2place = TransformationRule<State, Place>(
  createOutput: (state, context) => ...,
  transform: (state, place, context) => ...
);
```

List. 1: Representing transformation rules as method calls

An example how the transformation of states of a finite state machine to places of a Petri net looks like when transformation rules are implemented as method calls is shown in Listing 1. In this listing, we assume an optional `context` parameter that can be used for tracing purposes. The example uses named parameters which are not available in all languages (and usually optional where they are available). Other options include method chaining syntaxes. An example of these languages is SCALAMTL, as method chains have a suitable syntax in Scala.

The representation as inheritance means that there is a generic transformation rule class that is inherited from for each transformation rule, passing the type signature again as generic type parameters. Here, different phases of a transformation rule can easily be represented by different methods of the class which the transformation rule has to override.

```
class State2Place : TransformationRule<State, Place> {
  Place CreateOutput(State state, Context context) { ... }
  void Transform(State state, Place place, Context context) {...}
}
```

List. 2: Representing transformation rules as classes

The representation of transformation rules as classes with a certain inheritance relation is sketched in Listing 2. Here, the concept of a transformation rule is implemented in a generic class which is inherited from. The transformation phases are represented as overridden methods.

The design decision whether to implement transformation rules as method calls or as classes has several important consequences. While the syntax of the method calls contains less syntactic boilerplate and is thus more concise in terms of lines of code, the latter version using inheritance has the important advantage that being types, transformation rules are reflected in metadata. This has advantages for visualization as we will discuss in the next section. Furthermore, an object as the result of a method call can only be referenced once it has been created while a class can be referenced regardless of the order in the code (in most languages). This is problematic when the abstract syntax (cf. Fig. **??**) contains a cross-reference to transformation rules, when traces are explicit. Thus, for example in SCALAMTL, the traces are implicit and cannot be made explicit.

3.2 Navigation Support

A large proportion of development activities is devoted on the analysis of existing code and navigation through it [MKF06]. However, the navigation support of the mostly used search commands (searching for references to a selected element or its definition) can be derived independently of the transformation rule representation, if there is a representation as a code element at all (as opposed e.g. to pure naming conventions).

However, Rentschler et al. have shown that a visualization of a model transformations structure aids the navigation and is thus helpful for the maintainability of model transformations [Ren+13]. A similar visualization is getting common for general-purpose object-oriented code visualizing the usage of members within a class or the usage of classes within a package, as for example with Code Maps in Visual Studio. This analysis is based on metadata, i.e. classes, methods and their interrelations based on the methods' bodies. Objects as results of method calls are not part of this metadata since they are runtime artifacts and cannot be predicted at compile-time in general. As a consequence, code visualizations based on this metadata is not available when transformation rules were represented in method calls.

For internal languages, inherited visualization is of particular importance. Unlike external languages, they are merely guidelines how to use a framework but these guidelines are not enforced by the host language compiler[3]. As a consequence, static analysis like visualization specifically created for the internal language is of limited applicability. For analyses that look at the big picture like visualizations of the entire transformation, this means that it is hard to create something above the inherited visualization support. Alternatively, dynamic visualization as supported by SDMLIB are a viable approach, but are hard to integrate in the development environment and laborious to develop.

For the implementation of transformation rules as types as outlined above, the usage of a transformation rule is the same as the usage of this type. Therefore, visualization techniques to show dependencies in object-oriented programming can be used to visualize the structure of model transformations. An implementation of transformation rules as method cancels the possibility of inherited visualizations.

[3] Technologies like the modular compiler Roslyn give internal languages the chance to enrich the host language compiler

4 The NMF Transformations Language (NTL)

This section describes the NMF Transformations Language (NTL), the concrete syntax of the model transformation framework of NMF. The language uses C# as host language and is able to describe model transformations from and to arbitrary runtime objects. The implementation is part of NMF and thus available as open-source[4].

As we are aiming for a comprehensive transformation language, we represent transformation rules as classes that inherit from a common generic transformation rule class and pass in their signature as generic type parameters (cf. Section 3). The different phases of the transformation rule (cf. Section **??**) are specified by overriding methods from the base class. Thus, transformation rules in NTL exactly look like sketched in Listing 2 except for some syntactic boilerplate: In C#, overriding methods must repeat the entire signature of the base method accompanied by the `override` keyword.

```
public class FSM2PN : ReflectiveTransformation {
  public class State2Place : TransformationRule<State, Place> { ... }
}
```

List. 3: A transformation in NTL

The assembly of a model transformation of transformation rules is done by adding the transformation rule as public nested types of the transformation class which in turns inherit from `ReflectiveTransformation`. An example is presented in Listing 3. As a consequence, the declaration of the transformation rule and its registration with the transformation coincide, decreasing maintenance efforts.

```
1 public class State2Place : TransformationRule<State, Place> {
2   protected override RegisterDependencies() {
3     CallMany(Rule<Transition2Transition>(),
4      selector: s => s.Outgoing,
5      persistor: (p,transitions) => p.From.AddRange(transitions));
6   }
7 }
```

List. 4: Specifying dependencies in NTL

The specification of dependencies for a transformation rule is sketched in Listing 4. Dependencies are created in a dedicated method `RegisterDependencies` (see line 2) which is run by the transformation engine at initialization of the transformation.

Unlike transformation rules, dependencies themselves are merely uninteresting in code visualization. Furthermore, dependencies need not to be cross-referenced. Therefore, we represent them as method calls in a dedicated function of transformation rules. The attributes of dependency elements of the abstract syntax are simply passed as method call arguments. In line 3, such a call is made creating a dependency of the *State2Place* rule to *Transition2Transition* for each outgoing transition of a state in the state machine (line 4). The resulting transitions are then added to the *From* reference of the Petri net place which is the transformation result of the current state (line 5).

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

The entire finite state machine to Petri nets transformation can be found in [Hin13]. We abbreviate it here for space limitations.

5 Conclusion

Despite the improvements in terms of productivity, Model-driven engineering still lacks an industry adoption. In this paper, we have proposed an approach how this tool support problem can be solved by internal model transformation languages by exploring the design alternatives how model transformation rules can be represented in object-oriented design. We have shown how this discussion has lead to the development of NTL.

There is no unique way of implementing transformation rules in an object-oriented language. The implementation choices are trade-off decisions. An implementations of transformation rules as methods or method calls lead to a more concise syntax with less syntactic boilerplate but yield restrictions. Implementations as methods restrict the transformation language to a single operational phase in transformation rules and method calls make explicit tracing hard and cancel inherited visualizations based on metadata. An implementation alternative without these shortcomings at the price of a less concise language is the implementation as classes inheriting from a common transformation rule class.

References

[BH11] H. Barringer and K. Havelund. *TraceContract: A Scala DSL for trace analysis*. Springer, 2011.

[BMM09] M. Bruch, M. Monperrus, and M. Mezini. “Learning from examples to improve code completion systems”. In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2009, pp. 213–222.

[CH06] K. Czarnecki and S. Helsen. “Feature-based survey of model transformation approaches”. In: *IBM Systems Journal* 45.3 (2006), pp. 621–645.

[Fow10] M. Fowler. *Domain-specific languages*. Addison-Wesley Professional, 2010.

[GR13] P. V. Gorp and L. Rose. “The Petri-Nets to Statecharts Transformation Case”. In: *Sixth Transformation Tool Contest (TTC 2013)*. EPTCS. 2013.

[GWS12] 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.

[Hin+15] G. Hinkel et al. “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.

[Hin13] G. Hinkel. *An approach to maintainable model transformations using internal DSLs*. Master thesis, Karlsruhe Institute of Technology. 2013.

[Hin15] G. Hinkel. “Change Propagation in an Internal Model Transformation Language”. In: *Theory and Practice of Model Transformations*. Springer, 2015, pp. 3–17.

[KCF14] F. Křikava, P. Collet, and R. B. France. “SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations”. In: *Model-Driven Engineering Languages and Systems*. Springer, 2014, pp. 569–585.

[MKF06] G. C. Murphy, M. Kersten, and L. Findlater. “How are Java software developers using the Elipse IDE?” In: *Software, IEEE* 23.4 (2006), pp. 76–83.

[Moh+09] P. Mohagheghi et al. “MDE adoption in industry: challenges and success criteria”. In: *Models in Software Engineering*. Springer, 2009, pp. 54–59.

[Moh+13] P. Mohagheghi et al. “An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases”. In: *Empirical Software Engineering* 18.1 (2013), pp. 89–116.

[MR13] 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.

[Ren+13] A. Rentschler et al. “Interactive visual analytics for efficient maintenance of model transformations”. In: *Theory and Practice of Model Transformations*. Springer, 2013, pp. 141–157.

[RL08] R. Robbes and M. Lanza. “How program history can improve code completion”. In: *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE. 2008, pp. 317–326.

[SK03] S. Sendall and W. Kozaczynski. “Model transformation: The heart and soul of model-driven software development”. In: *Software, IEEE* 20.5 (2003), pp. 42–45.

[Sta06] 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.

[TL12] B. Trancón y Widemann and M. Lepper. “Paisley: pattern matching à la carte”. In: *Theory and Practice of Model Transformations*. Springer, 2012, pp. 240–247.

[Whi+13] J. Whittle et al. “Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?” In: *Model-Driven Engineering Languages and Systems*. Vol. 8107. LNCS. Springer Berlin Heidelberg, 2013, pp. 1–17.

[Zün+13] A. Zündorf et al. “Story Driven Modeling Libary (SDMLib): an Inline DSL for modeling and model transformations, the Petrinet-Statechart case”. In: *Sixth Transformation Tool Contest (TTC 2013), ser. EPTCS* (2013).