Using component frameworks for model transformations by an internal DSL

Georg Hinkel and Lucia Happe

Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
{georg.hinkel,lucia.kapova}@kit.edu

Abstract. To increase the development productivity, possibilities for reuse, maintainability and quality of complex model transformations, modularization techniques are indispensable. Component-Based Software Engineering targets the challenge of modularity and is well-established in languages like Java or C# with component models like .NET, EJB or OSGi. There are still many challenging barriers to overcome in current model transformation languages to provide comparable support for component-based development of model transformations. Therefore, this paper provides a pragmatic solution based on NMF Transformations, a model transformation language realized as an internal DSL embedded in C#. An internal DSL can take advantage of the whole expressiveness and tooling build for the well established and known host language. In this work, we use the component model of the .NET platform to represent reusable components of model transformations to support internal and external model transformation composition. The transformation components are hidden behind transformation rule interfaces that can be exchanged dynamically through configuration. Using this approach we illustrate the possibilities to tackle typical issues of integrity and versioning, such as detecting versioning conflicts for model transformations.

1 Introduction

In Model-Driven Engineering (MDE), systems are designed in models that conform to a metamodel. This formal definition of the model makes it possible to transform models to other artifacts like code or other models by means of model transformations. As MDE is getting applied in more complex scenarios, these model transformations also get very complex. As a consequence, it gets more important to divide these transformations into parts, i.e. components, with the goal to reuse these components in new scenarios as much as possible.

However, Wimmer, Kusel et al. indicate that reuse functionality of current model transformations is hardly established in practice [1, 2], especially reuse among different metamodels.

On the other hand, the situation is entirely different in most object-oriented general purpose languages. Here, classes can hide implementation details even to

inheriting classes by use of private methods or fields. Functionality can be reused through these classes, even independent of domains through the usage of generic types. These classes are assembled to components that have well-defined public interfaces. Furthermore, these components are explicitly versioned, so that a system can automatically detect versioning conflicts.

To pick an example, in the .NET component model, components (assemblies) consist of a provided interface (the publicly visible types), references to required assemblies, an explicit version information and possibly a digital signature. References to other assemblies also include the referenced version and the digital signature where applicable. Assemblies can be reused, deployment is simplified as dependencies are explicitly specified, integrity can be ensured through the usage of digital signatures and the runtime can detect version conflicts. A version conflict is detected when a required component with the specified major and minor version number (but possibly higher build or revision number) cannot be found. A similar component model is also required for model transformations. Unlike other component models like OSGi, the .NET component model does not allow components to be exchanged, e.g. through configuration. This is typically solved by using dependency injection frameworks.

Solving this problem by inventing a new component model for model transformations yields all risks of duplicate concepts, as e.g. a high maintenance effort. Thus, we think that it is a better approach to adopt existing component models where possible to use the interface mechanism for model transformation. As interfaces of .NET components are the publicly visible types, i.e. classes, we only need a meaningful mapping from model transformation concepts like transformation rules to classes in order to be able to reuse such a component model for internal model transformation composition, i.e. composing a single model transformation of multiple reusable parts.

The .NET component model is a particularly interesting candidate, as it has been used for a wide range of languages, including originally imperative languages like C# or VB.NET as well as more recently functional languages like F#. This variety of language paradigms yields the question of whether it can also be reused for more tailored languages like model transformation languages.

One approach for such a mapping is the definition of an internal DSL where concepts of model transformations are represented in the host language. Such an internal DSL is provided e.g. by NMF Transformations, which has been applied to several cases at the Transformation Tool Contest (TTC) 2013 [3, 4]. Furthermore, it is used within the .NET Modeling Framework (NMF)[1] to generate code for the model representation, quite similar to EMF[2]. Especially the latter transformation is rather large, so there is a need to make it more modular.

This embedding gives us a range of benefits regarding both internal and external model transformation composition. We can have model transformation components that specify interfaces, we can ensure their integrity through digital

[1] http://nmf.codeplex.com/
[2] http://www.eclipse.org/modeling/emf/

signatures and detect versioning conflicts. Here, integrity means that a model transformation component cannot be replaced by a forged component with the same public interface. These benefits come at no maintenance cost for the required infrastructure, as the infrastructure e.g. to ensure integrity through digital signatures is still maintained by the original owner, i.e. Microsoft. Furthermore, our embedding allows us to specify how model transformation rules can be dynamically exchanged through configuration by means of dependency injection.

Because model transformation concepts like transformation rules and a trace are represented directly by classes, NMF Transformations can act as a baseline for mappings of other transformation languages to classes as well, e.g. by translating model transformations of other languages to NMF Transformations.

The rest of this paper is structured as follows: Section 2 shows related work. Section 3 explains the running example of finite state machines and Petri Nets. Section 4 gives a brief overview of NTL, before Section 5 explains how NTL can be used to specify model transformation components mapped to assemblies. Finally, Section 6 concludes this paper.

2 Related Work

The idea of using internal DSLs [5] for model transformation has already been applied several times, but for different reasons. These reasons include a low implementation effort [6], type safety [7] or extensibility [8, 7, 9, 10]. However, their implications on reusing underlying component models have not been analyzed so far.

For external languages, experiences of reusing existing component models exist, as e.g. Xtend[3] is reusing the whole technology stack of Java, including the organization in Jar archives as the technical component model. However, unlike NTL, Xtend is a general-purpose language with support for model-to-text transformations through Xpand templates. Tailored model-to-model transformation languages implemented as external DSLs, like most commonly known QVT-O [11] or ATL [12], usually specify module reuse concepts (as surveyed by Wimmer, Kusel et al. [1, 2], plus the more recent approach from Rentschler et al. for modular QVT-O [13]), but cannot reuse component infrastructure that provides support regarding versioning conflicts or checking the integrity of model transformation components.

Rather, these languages are mainly organized in files that do not specify version information. References to other files are specified as import links. These links also do not specify a version information and possible version conflicts are not automatically detected. This is different for our approach where components of model transformations are represented by assemblies that specify references enriched with both version information and digital signatures in assemblies. This rather technical information has to be specified separately from the transformation specification and does not pollute the latter. Essentially, our work reuses

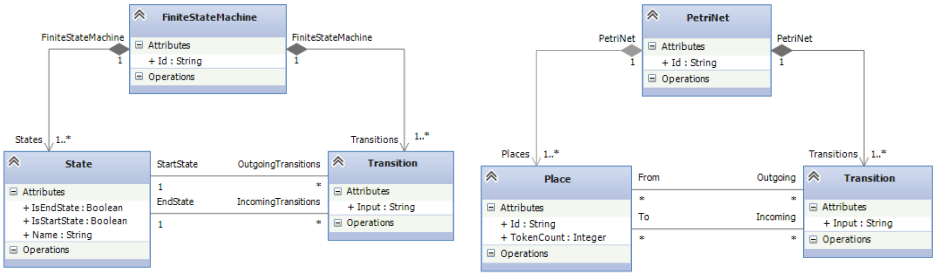
[3] http://www.eclipse.org/xtend/

the specification of component dependencies from existing component models as these dependency specifications are not specific to the domain of model transformation.

Model transformations can also be composed of multiple transformations through external composition such as chaining model transformations of multiple languages (see e.g. [1, 2] for a survey). Our embedding approach is also applicable for external composition, but yields the limitation that all model transformations must be embedded in the same platform. This limitation is typically avoided in specialized component models for model transformation [14], but these component models suffer from duplicated concepts and interoperability issues with other components.

3 Finite State Machines and Petri Nets

This section will introduce the running example of the transformation of finite state machines to Petri Nets. Both finite state machines and Petri Nets are popular formalization techniques to model processes, for example business processes. One is sometimes interested to transform finite state machines into Petri Nets because Petri Nets are more expressive.



(a) Finite state machines (b) Petri Nets

Fig. 1. Metamodels of finite state machines and Petri Nets

The metamodels for finite state machines and Petri Nets are depicted in Figure 1. The transformation maps each state of the state machine to a corresponding place. Each transition is mapped to a transition with an according input and output. Places corresponding to start states have an initial token and those corresponding to end places have a transition without a target.

4 NMF Transformations

NMF Transformations consists of two parts: a model transformation framework and a DSL that provides an easy syntax for this framework. This language

is called NTL (NMF Transformations Language) and is an internal DSL embedded in C#. Its abstract syntax is depicted in Figure 2. In NMF Transformations, model transformations consist of transformations and patterns (which we omit for brevity here). These rules can have dependencies to each other, letting computations of a transformation rule depend on one or multiple other computations.

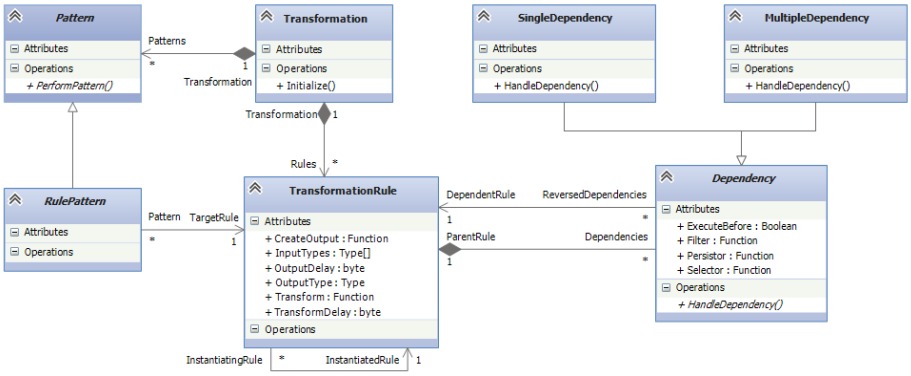


Fig. 2. The abstract syntax of NMF Transformations

In NTL, model transformations are represented as classes with their transformation rules as public nested classes. The transformation rules then specify how model elements should be transformed. Listing 1 shows an example of a transformation transforming finite state machines to Petri Nets, accompanied with the start rule (implicitly the first rule that matches the transformation request). The `Transform` method is used to specify the actions to be done when a finite state machine is to be transformed, i.e. initializes the transformation rule output where it may access the transformation context for tracing purposes.

```
1  using NMF.Transformations;
2  using NMF.Transformations.Core;
3
4  public class FSM2PN : ReflectiveTransformation {
5   public class Automata2Net : TransformationRule<FSM.FiniteStateMachine, PN.
        PetriNet>
6   {
7    public override void Transform(...)
8    {
9     output.ID = input.ID;
10   }
11  }
12 }
```

Listing 1. The transformation rule FiniteStateMachine2PetriNet

To specify dependencies, transformation rule classes may override the method `RegisterDependencies` and call several methods that create such dependencies using lambda expressions. An example is shown in Listing 2. The first dependency is a special one, as it specifies when the rule is going to be called (instead

of what other rules should be called). The last argument specifies how dependent computations should be saved. For example, the place corresponding to the start state of a finite state machine transition should be added to the `From` collection of the corresponding Petri Net transition.

```
1  public class Transition2Transition : TransformationRule<FSM.Transition, PN.
       Transition>
2  {
3   public override void RegisterDependencies()
4   {
5    CallForEach(Rule<Automata2Net>(),
6      selector: fsm => fsm.Transitions,
7      persistor: (net, transitions) => net.Transitions.AddRange(transitions));
8
9    Require(Rule<State2Place>(),
10     selector: t => t.StartState,
11      persistor: (t, place) => t.From.Add(place));
12
13   Require(Rule<State2Place>(),
14     selector: t => t.EndState,
15      persistor: (t, place) => t.To.Add(place));
16  }
17 }
```

Listing 2. The rule Transition2Transition with multiple dependencies

The language also supports inheritance mechanisms that can be facilitated for model transformation components. This includes both inheritance of transformations (like superimposition in ATL) as well as two applicable concepts for transformation rules, inheritance and instantiation. Transformation rule inheritance really is inheritance of the transformation rule class (similar to masked rules in ATL), whereas instantiation is what most other transformation languages (including ATL) call inheritance, unfortunately. An inherited rule may really override the body of that rule, as well as its dependencies. An instantiating rule must not do so, but may instead take control over the creation of outputs. Whereas rule inheritance aims for extensibility, instantiation is rather to support inheritance hierarchies and thus omitted in this paper for brevity.

Let us, for example, extend the above transformation to use colored Petri Nets. Listing 3 shows the implementation with rule inheritance. The behavior of the *Transition2Transition* rule is simply overridden in that it now creates a *ColoredTransition* with the default color. The transformation engine will simply instantiate a *ColoredTransition2Transition* rule instead of a *Transition2Transition* rule, because a *ColoredTransition2Transition* rule **is a** *Transition2Transition* rule and marked as overriding.

```
1  public class FSM2ColoredPN : FSM2PN {
2   [OverrideRule]
3   public class ColoredTransition2Transition : Transition2Transition
4   {
5    public override PN.Transition CreateOutput(...) {
6      return new PN.ColoredTransition() { Color = DefaultColor };
7    }
8   }
9  }
```

Listing 3. Introducing colored Petri Nets through rule inheritance

Next to transformation rule inheritance, Listing 3 demonstrates the inheritance of transformations. The transformation *FSM2ColoredPN* inherits *FSM2PN* and thus inherits its transformation rules (except for *Transition2Transition* which is overridden).

5 Components in model transformations with NTL

One of the advantages of using an internal DSL for model transformation is the easy integration of arbitrary code of the host language. For NTL, this is C# (or in theory any other .NET language). The `Transform` method is just executed normally and can contain arbitrary C# code, e.g. also using third-party components referenced by the assembly that contains the model transformation. This embedding allows target model elements to depend on source model elements in more sophisticated forms than simple transformations such as adding a prefix but more complex analysis. Being an ordinary method call, such a call is of course possible through dependency injection, so targets of external calls can be replaced by means of configuration. Likewise, model transformations can be called from arbitrary .NET assemblies as a transformation in NTL can be triggered by a method call. This yields way to any means of external model transformation composition. This composition is however restricted only to model transformations written for a common platform, which in our case is .NET.

Fig. 3. Dependent metamodels as components

An example of this is the generated model representation code for the involved metamodels, as demonstrated in Figure 3. One usually wants to separate this model representation code in own assemblies. Because both metamodel code and transformation code lie in assemblies that carry a version information, the transformation has an explicit knowledge of which metamodel version it is using.

Being .NET classes, transformation rules in NTL are perfectly allowed to also split the implementation of their interface (which by default most importantly consists of the two methods `Transform` and `RegisterDependencies`) in as many private methods as they wish, hiding their implementation. More interestingly, they can also have virtual, abstract or final methods. Thus, transformation rules can decide how their behavior can (or must) be overridden. They can e.g. mark the overridden `RegisterDependencies` method final to prevent child classes to modify dependencies. Alternatively, transformation rules can specify new virtual methods that are called e.g. from their `Transform` implementation, enabling extension rules to override only parts of its behavior. Making the `Transform` method final, the extension is also limited to these parts.

With these virtual methods, such transformation rules do provide an interface for other components that wish to extend them. If they are additionally marked as abstract, the only remaining differences are that true .NET interfaces do not interfere the inheritance hierarchy and type parameters can be covariant or contravariant. The interference with the inheritance hierarchy is not relevant to transformation rules, as transformation rules in NTL must eventually inherit from `TransformationRule<,>` anyhow. The restriction of covariancy/contravariancy means that implementations of such a transformation rule interface must stick to the same signature, which we argue is an acceptable restriction. Implementations of such an interface are transformation rules that inherit from the interface transformation rule.

Since `TransformationRule<,>` itself is an abstract class, we can also use it as an interface. This is possible because in the .NET platform, the runtime is still aware of generic type arguments (unlike e.g. Java). This is even directly supported by NTL. So instead of specifying a concrete transformation rule in listing 2, one can also just specify the transformation rule signature (i.e. the input and output type of the transformation rule) having the transformation engine apply the dependency for all registered transformation rules that correspond to this signature (without having to know the exact transformation or its implementation).

Using generic transformation rule classes as transformation rule interfaces has another advantage, as this decouples a transformation rule from the metamodels used by this rule. Instead, the transformation rule interfaces can require the input and/or output type to adhere to some type constraints or require implementations to inject domain knowledge through abstract methods and generic type arguments.

Now consider the transformation to create a language-independent model representation code model. One of the challenges of this transformation is to cope with the fact that the metamodels may use multiple inheritance whereas .NET only supports single inheritance. This challenge is independent from the exact mapping how model elements are transformed to type members and is thus a good candidate for a reusable component.

Using external composition techniques, one would first transform the metamodel to a code model with multiple inheritance and chain a separate transformation that removes the multiple inheritance by introducing interfaces and merging the implementing classes. Since our embedding allows to treat the involved model transformations just like any other method call, we can put them into separate components and hide them behind interfaces as we wish (using platform standard interfaces).

On the other hand, we can also create a system of transformation rules in a separate component where these rules fix their output type but leave the input type open (using generic type parameters). This way, we compose the transformation by refining specialized rules for this task, i.e. in a manner of internal model transformation composition. These specialized rules would then contain domain-specific extension points that allow to alter the merging step in

many places. With this approach, the second merging step becomes more like a transformation framework in its own, based on the transformation language. The advantage of this is that the merge process can be controlled in much more detail as derived transformation rules may choose an extension point to override.

Finally, we can also load single transformation rules from other components, adding e.g. the merging step as a separate transformation rule that runs delayed. This way, the merging step runs in the very same transformation run and has access to all the trace. If we had multiple components that realize this functionality in different ways, we could swap the implementation by means of a dependency injector dynamically as NTL also allows to load transformation rules in a method. The only restriction here is that a transformation always requires fresh transformation rule instances (transformation rule instances cannot be shared across multiple transformations), but most dependency injectors can be configured to fit this requirement, i.e. creating a new instance per request.

Thus, the mapping of model transformation concepts to classes yields a clear and precise notion of interfaces for model transformation rules as well as model transformation components, where the components are the .NET assemblies each containing a subset of transformation rules. But as they are assemblies, they also inherit the version information of assemblies.

As a consequence, developers can (and have to) specify the version of their model transformation components explicitly as version of the assembly that contains the model transformation component and likewise the version of referenced components. These version numbers consist of four parts, major, minor, build and revision. Changes of build and revision number by default are interpreted as bug fixes, i.e. the runtime will load assemblies with higher build or revision numbers instead of the specified referenced assembly. Higher major or minor version numbers are interpreted as breaking changes regarding the assemblies public interface, which for NTL are publicly accessible model transformation rules (or other public e.g. helper types). If only a higher version of a referenced component is found (components are allowed to be present in multiple versions concurrently), the runtime raises an exception, detecting possible versioning issues.

6 Conclusion

In this paper, we have shown how a mapping from model transformation concepts to object-oriented general-purpose constructs can be used to reuse component models. We have achieved this goal through NMF Transformations, a framework and internal DSL embedded in C#. This embedding allows us to:

- Integrate existing .NET components (assemblies) into model transformations and vice versa
- Detect versioning conflicts of model transformation components
- Compose model transformations as extension of model transformations specified in other components
- Specify transformation rule interfaces

- Compose model transformations of instancies previously defined transformation rule interfaces loaded from referenced components
- Ensure integrity of model transformation components by means of digital signatures.

Reusing existing dependency injectors further yields the chance to reconfigure model transformations based on configuration files without new compilation. Thus, we showed that a mapping of model transformation concepts to general-purpose programming, as e.g. with an internal DSL such as NTL, can be used to adopt existing component models for model transformation and reuse a lot of concepts and tools.

References

1. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Fact or fiction–reuse in rule-based model-to-model transformation languages," in *Theory and Practice of Model Transformations*. Springer, 2012, pp. 280–295.
2. A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reuse in model-to-model transformation languages: are we there yet?" *Software & Systems Modeling*, pp. 1–36, 2013.
3. 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.
4. ——, "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.
5. M. Fowler, *Domain-specific languages*. Addison-Wesley Professional, 2010.
6. H. Barringer and K. Havelund, *TraceContract: A Scala DSL for trace analysis*. Springer, 2011.
7. 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.
8. 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.
9. T. Horn, "Model Querying with FunnyQT," in *Theory and Practice of Model Transformations*. Springer, 2013, pp. 56–57.
10. G. Hinkel, "An approach to maintainable model transformations using internal DSLs," Master thesis, 2013.
11. Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification," http://www.omg.org/spec/QVT/1.1/PDF/, 2011.
12. F. Jouault and I. Kurtev, "Transforming models with ATL," in *Satellite Events at the MoDELS 2005 Conference*. Springer, 2006, pp. 128–138.
13. A. Rentschler, D. Werle, Q. Noorshams, L. Happe, and R. Reussner, "Designing information hiding modularity for model transformation languages," in *Proceedings of the of the 13th international conference on Modularity*. ACM, 2014, pp. 217–228.
14. J. S. Cuadrado, E. Guerra, and J. de Lara, "A component model for model transformations," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.