Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Surveying Rule Inheritance in Model-to-Model Transformation Languages

M. Wimmer[a] G. Kappel[b] A. Kusel[c] W. Retschitzegger[c]
J. Schönböck[b] W. Schwinger[c] D. Kolovos[d] R. Paige[d]
M. Lauder[e] A. Schürr[e] D. Wagelaar[f]

a. Universidad de Málaga, Spain
b. Vienna University of Technology, Austria
c. Johannes Kepler University Linz, Austria
d. University of York, United Kingdom
e. Technische Universität Darmstadt, Germany
f. INRIA & École des Mines de Nantes, France

Abstract Model transformations play a significant role in Model-Driven Engineering. However, their reuse mechanisms have yet to receive much attention. In this paper, we propose a *comparison framework* for rule inheritance in model-to-model transformation languages, and provide an *in-depth evaluation* of prominent representatives of *imperative*, *declarative* and *hybrid* transformation languages. The framework provides criteria for comparison along orthogonal dimensions, covering *static aspects*, which indicate whether a set of inheriting transformation rules is well-formed at compile-time, and *dynamic aspects*, which describe how inheriting rules behave at run-time. The application of this framework to dedicated transformation languages shows that, while providing similar syntactical inheritance concepts, they exhibit different dynamic inheritance semantics and offer basic support for checking static inheritance semantics, only.

Keywords Rule Inheritance, Model Transformation, Comparison

1 Introduction

Model-Driven Engineering (MDE) defines models as first-class artifacts throughout the software lifecycle, which leads to a shift from the "everything is an object"

M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, D. Kolovos, R. Paige, M. Lauder, A. Schürr, D. Wagelaar. *Surveying Rule Inheritance in Model-to-Model Transformation Languages*. In Journal of Object Technology, vol. 11, no. 2, 2012, pages 3:1–46.
doi:10.5381/jot.2012.11.2.a3

paradigm to the "everything is a model" paradigm [Béz05]. In this context, model transformations are crucial for the success of MDE [SK03], being comparable in role and importance to compilers for high-level programming languages. Support for large transformation scenarios is still in its infancy, since reuse mechanisms such as inheritance have received little attention so far [KKS07], although the concept of inheritance plays a major role in metamodels, as revealed, e.g., by the evolution of the UML standard [MSZJ04]. As inheritance is employed in metamodels to reuse feature definitions from previously defined classes, inheritance between transformation rules is useful in order to avoid code duplication and consequently maintenance problems. Although this need has been recognized by developers of transformation languages, the design rationales underlying individual transformation languages are not comparable at first sight. This makes it difficult to understand how these constructs are to be used.

Therefore, we propose a comparison framework for rule inheritance in model-to-model transformation languages[1] that makes explicit the hidden design rationales. The proposed framework categorizes the comparison criteria along three orthogonal dimensions – analogous to the three primary building blocks of programming languages [ASU86]. The first two dimensions comprise *static criteria*: (i) the *syntax*, a transformation language defines with respect to inheritance and (ii) the *static semantics*, which indicates whether a set of inheriting transformation rules is well-formed at compile-time. The third dimension of the comparison framework describes how inheriting rules interact at run-time, i.e., *dynamic semantics*. On the basis of this framework, inheritance mechanisms in dedicated transformation languages are compared. In order to provide an extensive survey, representatives of the three common paradigms of imperative, declarative and hybrid transformation languages have been chosen [CH06]. In this context, we examined the imperative transformation languages Kermeta [MFJ05] and QVT-Operational [OMG09], the declarative transformation languages Triple Graph Grammars (TGGs) [KKS07] and Transformation Nets (TNs) [Sch11], and the hybrid transformation languages Atlas Transformation Language (ATL) [JABK08] and Epsilon Transformation Language (ETL) [KPP08]. The results show that the inheritance semantics of these languages differ in various aspects, which has profound consequences for the design of transformation rules.

Outline. While Section 2 explains the rationale of this work, Section 3 presents the comparison framework with its three dimensions. In Section 4 we compare the inheritance mechanisms of the selected languages and present lessons learned in Section 5. Section 6 gives an overview on related work, and finally, Section 7 concludes the paper.

Please note that this paper is an extended version of [WKK^{+}11], whereby three major parts have been added. First, for the static semantics, OCL constraints have been introduced based on a generic transformation language metamodel, providing the basis to implement the static semantics in specific transformation languages. Second, in the original version, the focus was on *declarative* transformation languages as well as on *hybrid* transformations languages covering the *declarative* parts, only. In contrast, in this extended version we also investigate *imperative* languages as well as *imperative* parts of hybrid approaches. Finally, an extensive survey on reuse mechanisms in transformation languages has been added to the related work section.

[1] With the term model-to-model transformations, we refer to exogenous transformations according to [MG06].

2 Motivation

When developing a framework for comparing rule inheritance in model-to-model transformation languages, one starting point is to look at the well-known model transformation pattern (cf. Fig. 1) and to examine where the introduction of inheritance would play a role. As may be seen in Fig. 1, a transformation specification consists of a set of rules, which are responsible to describe how source models should be transformed into target models. In this context, the transformation rules may inherit from each other in order to avoid code duplication. Obviously, in order to enable rule inheritance, a transformation language must define syntactic concepts (cf. question ① in Fig. 1), which leads to the first dimension of our comparison framework, namely *syntax*. In this context, the following questions are of interest:

- *Which types of inheritance are supported?* Does the transformation language support only single or multiple inheritance?
- *Are abstract rules supported?* Is it possible to specify transformation behavior that is purely inherited?

In addition to syntax, further well-formedness constraints on the transformation rules must hold (cf. question ② in Fig. 1), which represents the second dimension, namely *static semantics*. Thereby, the following questions may arise:

- *In which way may a subrule modify a superrule?* For instance, how may the types of input and output elements be changed in subrules such that they may be interpreted in a meaningful way?

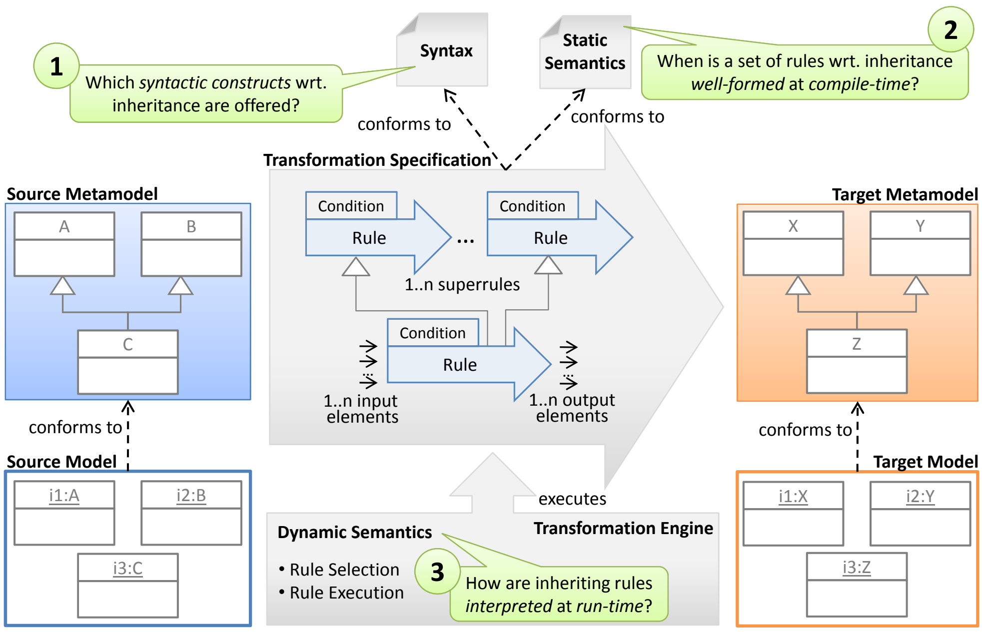

Figure 1 – Model-to-Model Transformation Pattern

- *When is a set of inheriting rules (i.e., rules that inherit from others) defined unambiguously?* Are there sets of rule definitions that do not allow selecting a single rule for specific instances?

A transformation specification is usually compiled into executable code, which is interpreted by a transformation engine that takes a source model and produces a target model by executing the transformation. The main challenge in executing the transformation is the dispatching of rules for source model instances, i.e., selecting and applying rules for specific instances. In declarative transformation languages, rule selection is performed automatically, whereas in imperative transformation languages rule selection must be performed by the transformation designer. Finally, hybrid approaches combine these two paradigms. Again several questions concerning the interpretation of inheritance at run-time arise (cf. question ③ in Fig. 1), which leads to the third dimension, namely *dynamic semantics*:

- *Which instances are matched by which rule?* If a rule is defined for a supertype, are the instances of the subtypes, i.e., the indirect instances of the supertype, also affected by this rule?
- *How are inheriting rules executed?* Either top down or bottom up the rule inheritance hierarchy?

Please note that although the last two questions seem applicable to transformations languages, which perform rule selection automatically, only, these questions are also crucial for imperative transformation languages, since dynamic dispatching of rules may be used, i.e., a general rule is called statically, but nevertheless, the most specific subrule should be executed for a given instance at run-time.

3 Comparison Framework

This section presents our framework for comparing inheritance support in model-to-model transformation languages, which are used to describe transformations between object-oriented metamodels, conforming to, e.g., Ecore[2] or MOF2[3]. Although metamodeling languages such as MOF2 support refinements between associations, e.g., `subsets` or `redefines`, these are out of scope of this paper. As shown in Fig. 2, the comparison criteria may be divided into the three dimensions of (i) *syntax*, (ii) *static semantics*, and (iii) *dynamic semantics*. These dimensions and the corresponding sub-criteria are described in the following subsections.

3.1 Syntax

This subsection provides criteria for comparing the supported syntactic concepts of model-to-model transformation languages. We consider both, general criteria (e.g., the numbers of input and output elements of a rule) and inheritance-related criteria (e.g., whether single or multiple inheritance is supported). The general criteria are included in the comparison, since they play a major role when investigating the static semantics of inheriting transformation rules (cf. Section 3.2).

[2] `http://www.eclipse.org/modeling/emf`
[3] `http://www.omg.org/mof`

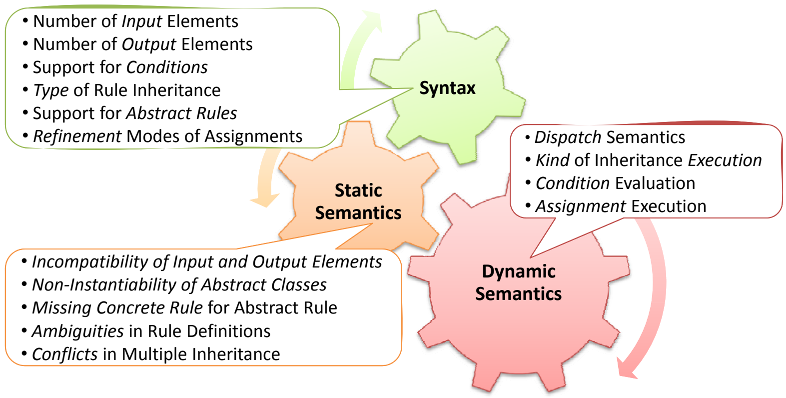

Figure 2 – Overview on the Comparison Framework

General Criteria. To identify the criteria for comparison, we analyzed (i) the features of transformation languages, and (ii) the classification of model transformation approaches presented in [CH06]. The identified features are expressed in a metamodel, shown in Fig. 3 (cf. area *Transformation Metamodel*) illustrating the core concepts of rule-based model-to-model transformation languages. A `Transformation` typically consists of several `TransformationRules`. A `TransformationRule` comprises an `InPattern`, referring to `InputElements` of the source metamodel, and an `OutPattern`, referring to `OutputElements` of the target metamodel. Please note that programmed graph transformations and TGGs distinguish between (i) rule parameters and (ii) input/output elements, whereby we consider only the latter. A general distinguishing criterion is the *allowed number of input and output elements*. Furthermore, transformation languages typically support the definition of a `Condition`, which may be interpreted in different ways (cf. Section 3.3). Furthermore, they provide the possibility of setting values for target features by means of `Assignments`. Please note that the relationships between the transformation language and the metamodeling language (cf. area *Extract of MOF*) are explicitly illustrated. In particular, `InputElements` and `OutputElements` refer to `Classes` and `Assignments` compute values for `Features`, which are contained by the classes referenced by the `OutputElements`. Finally, as already introduced, `TransformationRules` may be either applied automatically by the transformation engine or explicitly called by the transformation designer. To represent these two kinds, a rule may be marked as being *lazy*, whereby lazy means that the rule has to be explicitly called.

Inheritance-Related Criteria. In the context of inheritance-related aspects, three criteria are relevant. First, a `TransformationRule` may inherit from either one or multiple other transformation rules, depending on whether *single or multiple inheritance* is supported. Second, the concept of `abstract` rules may be supported in order to specify that a certain rule is not executable per se but provides core behavior that may be reused in subrules. Finally, one may distinguish between different *refinement modes*, which determine how inherited assignments are incorporated into inheriting rules (cf. enumeration `RefinementMode` in Fig. 3). First, *override* implies that when a subrule refines an assignment of a superrule, the assignment of the subrule is executed together with those assignments in the superrule which are not overridden. In the refinement mode *inherit* first the assignments of the superrule are executed, and then the assignments of the subrule may alter the resulting intermediate result

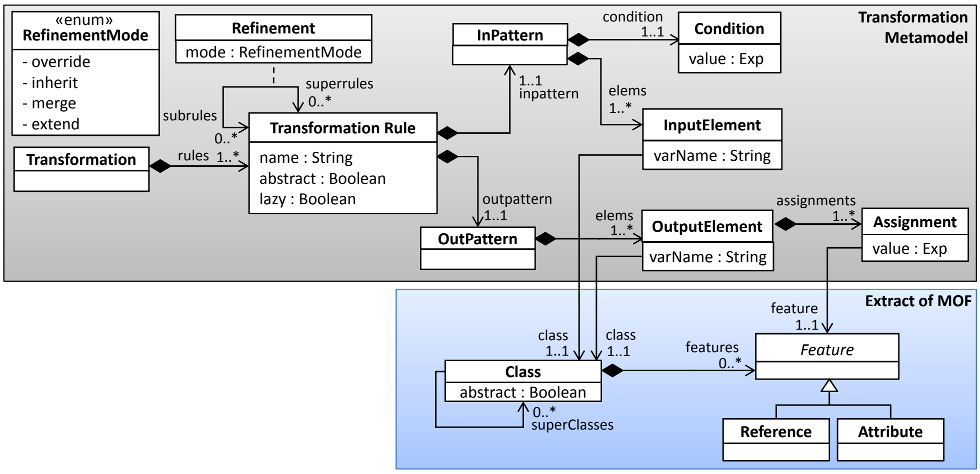

Figure 3 – Inheritance-Related Concepts of Transformation Languages

(such as by initializing some state by a supercall and then altering this intermediate result). Third, *merge* means that again both assignments are executed, but first the assignments of the subrule and then the assignments of the superrule are executed. Finally, the refinement mode *extend* induces that inherited assignments may not be changed at all. For consistency reasons, all assignments inherited from a certain rule should follow the same refinement mode. Therefore, the class `Refinement` is modeled as an association class on the association modeling the inheritance relationship between transformation rules in Fig. 3.

3.2 Static Semantics

In the previous subsection, we identified criteria targeting the comparison of syntactic concepts. Now we elaborate on criteria relevant for checking the static semantics of rule inheritance. These criteria reflect the following semantic constraints: (i) incompatibility of input and output elements of subrules and superrules in terms of type and number, (ii) non-instantiability of abstract classes, (iii) missing concrete rule for an abstract rule, (iv) ambiguities in rule definitions, and (v) conflicts in multiple inheritance. In order to clarify the semantics of each static constraint and to provide the basis to implement the static semantics in specific transformation languages, a specification on basis of OCL[4] is provided.

3.2.1 Incompatibility of Input and Output Elements

In the context of transformation rules, both feature assignments and conditions should be inheritable to subrules. Thus, it must be ensured that the *types* of the input and output elements of subrules provide at least the features of the types of the elements of the superrule. Consequently, types of the input and output elements of a subrule might become more specific than those of the overridden rule. The inheritance hierarchy of the transformation rules must thus, exhibit the same structure as the inheritance hierarchy of the metamodels. This means that co-variance for input and output elements is demanded, conforming to the principle of *specialization*

[4] `http://www.omg.org/spec/OCL`

inheritance in object-oriented programming. This is in contrast to popular design rules for object-oriented programming languages, where a contra-variant refinement of input parameters and a co-variant refinement of output parameters of methods is required to yield type substitutability, also known as *specification inheritance* [LW93].

Specification in OCL. For ensuring co-variance of input and output elements, the OCL constraint (invariant) shown in Listing 1 must hold for each transformation rule. The constraint is shown for input elements only, since it is analogous for output elements.

```
1  context TransformationRule inv CoVarianceOfInputElements:
2   -- select InputElements of context rule
3   self.inpattern.elems -> forall(ie : InputElement |
4     -- query and iterate all effectively inherited input elements
5     self.allEffInhIE() -> collect(varName)
6     -- if an effectively inherited input element is overridden
7     -> includes(ie.varName) implies
8     -- then check co-variance condition
9     ie.class.allSuperClasses() -> union(ie.class) -> includesAll(
10      self.allEffInhIE() -> select(iie : InputElement | iie.varName = ie.varName)
11      -> collect(class) -> flatten()
12    )
13  )
14
15 -- OCL operation to compute all effective (most specific) inherited input elements
16 -- this operation should be equivalent to the operation used for executing transformations
17 context TransformationRule : def allEffInhIE() : Set (InputElement)= ...
18
19 --OCL operation to compute all super classes
20 context Class: def allSuperClasses: Set(Class)=
21   self.superClasses->asSet()->union(self.superClasses->
22     collect(c| c.allSuperClasses)->asSet())
```

Listing 1 – Invariant to Check Co-Variance of Input Elements

Since co-variance must be ensured for all directly contained `InputElements` of the context rule, first an iteration over all `InputElements` has been specified (cf. line 3). Second, it is checked, if the currently processed `InputElement` overrides an `InputElement` of a superrule (cf. lines 4-7). For this, the set of all effectively

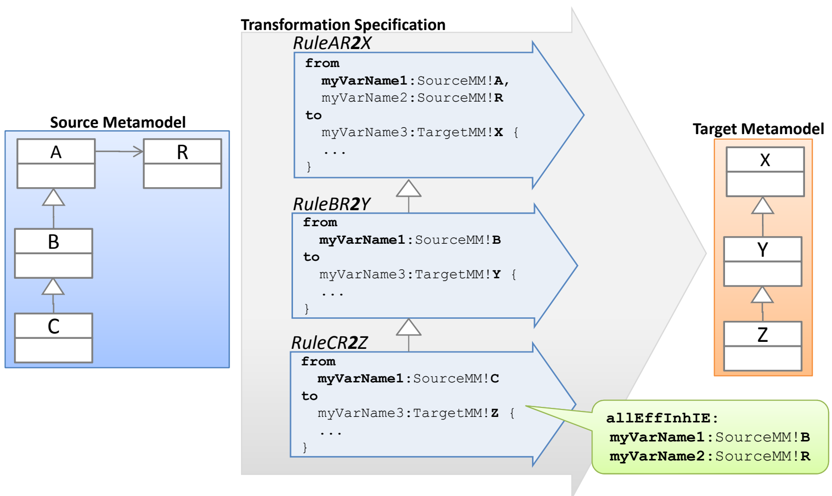

Figure 4 – Example for Co-Variance Check for Input Elements

inherited `InputElements` (cf. helper function `allEffInhIE()`) is calculated. This set contains all `InputElements`, which are directly inherited by the context rule. To exemplify this, Fig. 4 shows a simple example with three inheriting rules. The set of all effectively inherited `InputElements` for rule `CR2Z` includes `myVarName1` pointing to type `B` and `myVarName2` pointing to type `R`. `myVarName1` pointing to type `A` is not included, since it is overridden in rule `BR2Y`. Finally, if a certain `InputElement` of the current rule overrides an `InputElement` of a superrule, i.e., if the `varName` of the current `InputElement` is contained in the set of `varNames` of the effectively inherited `InputElements` (cf. lines 4-7), then the actual co-variance check is performed. For this, the set of all superclasses of the type of the current `InputElement` including the type of the current `InputElement` is calculated (cf. line 9 by calling the helper function defined on line 20). This set must then contain the type of the overridden `InputElement` (cf. lines 10-11). In case of the example depicted in Fig. 4, the set of all supertypes including the current type of `myVarName1` of rule `CR2Z` is {`C`, `B`, `A`}. Since `B`, i.e., the type of the effectively inherited `InputElement`, is included in this set, the co-variance condition for rule `CR2Z` is fulfilled.

3.2.2 Non-Instantiability of Abstract Classes

Since abstract classes cannot be instantiated, it must be ensured statically that no concrete rule tries to create instances of an abstract target class as output. Only abstract rules are allowed in this case, since they are not themselves executed, but must be refined by a subrule. The situation is different for abstract source classes: although an abstract source class cannot have any direct instances, indirect instances may be affected by the transformation rule.

Specification in OCL. For ensuring this constraint, again an invariant in OCL has been specified as shown in Listing 2. This invariant specifies that if a rule is concrete (cf. line 3) then all classes referenced by the `OutputElements` of the rule must be concrete (cf. lines 6-10).

```
1  context TransformationRule inv OnlyConcreteTargetClassesForConcreteRule:
2   -- if rule is concrete
3   (not self.abstract) implies
4   -- then all referenced target classes of output elements must be concrete
5   -- this condition has to be fulfilled also by non-overridden effectively inherited output elements
6   self.allEffInhOE() -> reject(ioe:OutputElement | self.outpattern.elems->collect(varName)
7   -> includes(ioe.varName))->union(self.outpattern.elems)
8   -> collect(class)-> flatten()
9   --check if the referenced target class is concrete
10  -> forAll(c:Class | not c.abstract)
11
12 -- OCL operation to compute all effective (most specific) inherited output elements
13 -- this operation should be equivalent to the operation used for executing transformations
14 context TransformationRule : def allEffInhOE() : Set (OutputElement)= ...
```

Listing 2 – Invariant to Check Non-Instantiability of Abstract Classes

In this context, `OutputElements` may also be inherited from superrules. Consequently, not only the directly contained `OutputElements` of the context rule must be checked, but also inherited `OutputElements`. To achieve the set of inherited `OutputElements`, the set of all effectively inherited `OutputElements` (which is analogously defined to the set of effectively inherited `InputElements` - cf. Listing 1) must be calculated with a corresponding helper function (cf. `allEffInhOE()`). In case of the rule `B2YU` of the example shown in Fig. 5, this set comprises the `OutputElements` `myVarName2` pointing to type `X` and `myVarName3` pointing to type `U`. This set must then

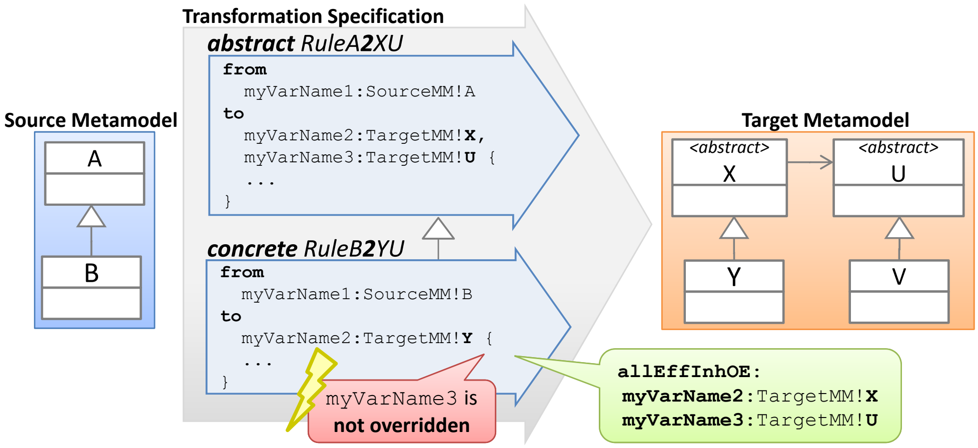

Figure 5 – Example for Non-Instantiability of Abstract Classes Check

be reduced by those `OutputElements`, which are overridden in the current rule (cf. `reject()`-operation in line 6) – in case of the example the variable `myVarName2` of the rule `A2X` is overridden and thus, removed from the set. In order to consider also the `OutputElements` of the context rule, the resulting set is unified with `OutputElements` of the context rule (cf. `union()`-operation in line 7). Consequently, the set of classes, which must not be abstract, includes type `U` referenced by `myVarName3` and type `Y` referenced by `myVarName2`. Since type `U` is abstract, the example shown in Fig. 5 does not fulfill the given constraint.

3.2.3 Missing Concrete Rule for an Abstract Rule

In order to execute the transformation code specified within an abstract rule, at least one concrete rule needs to be specified, which inherits from an abstract rule. However, since this does not lead to an error during execution, only a warning should be given to the user.

Specification in OCL. For ensuring this constraint, it is first checked if the context `TransformationRule` is abstract (cf. line 3 in Listing 3). If this is the case, then all subrules are collected by an according helper function (cf. line 7-9). In the resulting set of rules, at least one concrete rule has to exist (cf. line 4).

```
1 context TransformationRule inv ConcreteRuleToAbstractRule:
2  -- if rule is abstract
3  self.abstract implies
4      self.allSubrules()-> exists(r | not r.abstract)
5
6 -- OCL operation to compute all subrules of a Transformation Rule
7 context TransformationRule: def allSubrules: Set(TransformationRule)=
8    self.subrules->asSet()->union(self.subrules->
9      collect(r| r.allSubrules)->asSet())
```

Listing 3 – Invariant to Check if a Concrete Rule Exists For an Abstract Rule

3.2.4 Ambiguities in Rule Definitions

Provided that rules inherit from each other, it has to be ensured that a *single* rule may be determined for a specific instance or a specific set of instances, respectively. Consequently, the rules in an inheritance hierarchy must match for disjoint sets of objects, since otherwise redundant target model instances may result, if the multiple processing of input elements is not prohibited.

Although the dispatching of rules is tightly coupled to the dynamic semantics, the fact that more than one rule potentially matches for a single instance or a set of instances, respectively may be anticipated statically to a certain extent. Basically, disjoint sets may be either achieved (i) by subtyping, or (ii) by corresponding conditions, which divide the instances into the required sets, assuming that the most specific rule is applied to a certain instance and that the multiple processing of input elements is prohibited. This means also that if an object is matched and transformed by a specific rule, more general rules should not match and transform this element again. Given the fact that input elements may change in type or number in subrules, four valid cases ensuring rule compatibility exist as shown in Fig. 6 and detailed in the following.

- **Same number, different types:** In the first case, transformation rules with the same number of input elements, but with different types have been defined. In this case, the required disjoint subsets are achieved by *subtyping*, i.e., the subrule `C2Z` refines the types of the input elements from `A` and `B`, respectively to `C`. Consequently, it is clearly defined, that the subrule `C2Z` matches for `C` instances, whereas the superrules `A2X` and `B2Y` match for instances of types `A` and `B`, respectively.

- **Same number, equal types:** In this case, transformation rules with input elements of equal types and same number exist. Thus, the needed disjoint subsets may be achieved by *conditions*, only, since the input elements exhibit no other distinguishing factor.

- **Different number, different types:** The third case incorporates a different number of input elements as well as different types. Consequently, the needed subsets are built by subtyping – as in case (a).

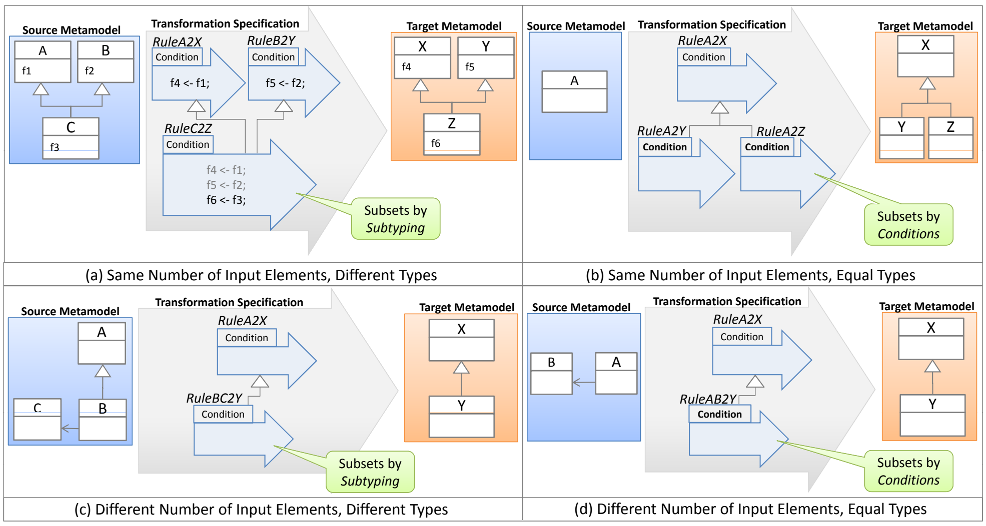

Figure 6 – Rule Compatibility

- **Different number, equal types:** Finally, the fourth case includes a different number of input elements, but the types of the input elements inherited from the superrules have not changed in the subrule (cf. input element `A`). Due to the equal typing of the inherited input elements, the required disjoint subsets may again only be achieved by `conditions`. However, this time the condition is given by the fact, that the subrule demands for more elements, i.e., the subrule `AB2Y` matches all `A` instances that exhibit a link to an instance of `B`, whereas the remaining instances may be matched by the superrule `A2X`.

In summary, it must be ensured statically, that the input and output elements are changed in a co-variant manner and the number might be extended, only. If the types of input elements are refined, subsets by subtyping are automatically built. In case that the types of input elements are not refined, it might only be checked that at least all the subrules specify conditions. The decision, whether these conditions really select disjoint subsets would be a task for program analysis. One interesting question that remains open in the context of cases (b) and (d) is whether the instances that do not fulfill any of the conditions of the subrules are matched by the superrule (provided that the superrule is concrete). Since this question is closely related to dynamic semantics, it is further discussed in Section 3.3.

Multiple Dispatching Problem. A special case of rule ambiguities may arise in the context of scenario 1 (cf. same number of input elements with different types). Provided that a rule requires multiple input elements, the situation may arise that there is no single rule for which the match in run-time types is closer than all the other rules. This is analogous to the problem that arises in multiple dispatching as needed for multi-methods (cf. [ADL91, Cha92]), since choosing a method requires the run-time type not of a *single* input element, only but of a *set* of input elements. Thus, the method whose run-time types most closely match the statically specified types should be dispatched at run-time.

A simple example of a rule ambiguity problem is depicted in Fig. 7. In this context, three transformation rules are specified taking two input elements of different metamodel types, respectively. Now, suppose that a pair of instances `(b,y)` of type `B` and `Y` is transformed, and let us assume that the rules might also match indirect instances. The transformation engine should now look for a rule, whose arguments *most closely match* the pair `(b,y)`. In this case, no single rule may be determined, since `RuleBX2...` and `RuleAY2...` are equally good matches. Thus, the set of defined transformation rules is ambiguous.

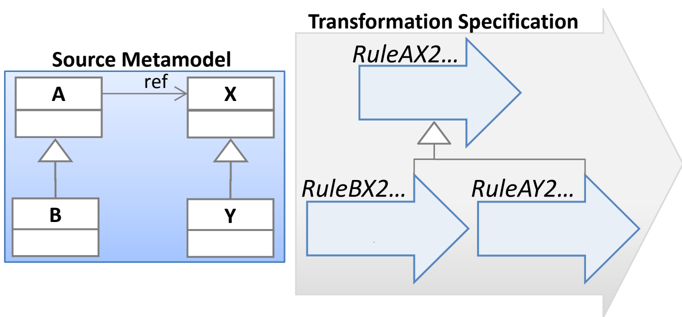

Figure 7 – Example for Rule Ambiguity

The actual check for rule ambiguities may be implemented in an imperative language such as Java, since for this check, a complex data structure has to be computed and processed. Thus, we refrain to present a concrete OCL constraint for this check. To actually implement this derived property, algorithms for explicit disambiguation in the area of multi-methods may be applied. For instance, in [AD96] an algorithm is proposed, which calculates a minimal set of method redefinitions necessary for disambiguation. This algorithm consists of two steps. In the first step, so-called *pole signatures* are calculated, whereby these pole signatures represent a minimal set of combinations of types, for which the algorithm must check for ambiguities. In the second step, the algorithm computes the *most specific applicable* (MSA) method for each pole signature, whereby the basis for method specificity is a *precedence relationship*: a method m_i is more specific than a method m_j, if *all* arguments of m_i are subtypes of the arguments of m_j. An ambiguity arises, if for a certain pole signature more than one MSA method exists.

To exemplify this, Fig. 8 depicts an example. One may see that three different rules with different arguments have been specified. To achieve the set of pole signatures, first the pole types for each argument position have to be calculated. This is achieved by collecting the types appearing at a certain position (e.g., {A,B,C}) and adding those types from the class hierarchy, which inherit from multiple classes – in the inheritance hierarchy shown in Fig. 8, the type D must be added, since this is the only type, which inherits from multiple classes (A and B) and is not yet in the set of {A,B,C}. The set of pole signatures is then achieved by building the cartesian product of the set of pole types on the different argument positions. This set of pole signatures may be reduced by the signatures appearing in the given rules, since for these signatures, no ambiguity may arise (in the example {{A,G},{B,F},{C,G}}). Consequently, in the second step of the algorithm, ambiguities have to be checked for

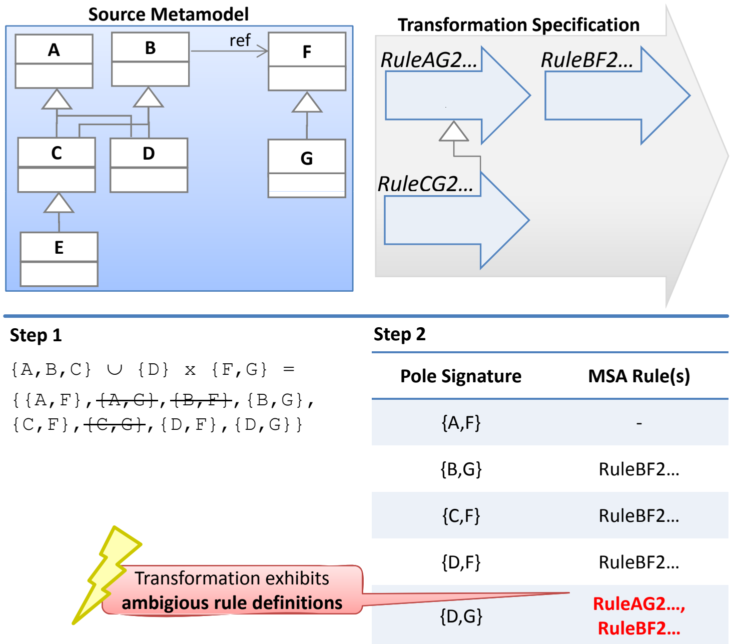

Figure 8 – Rule Disambiguation Algorithm by Example

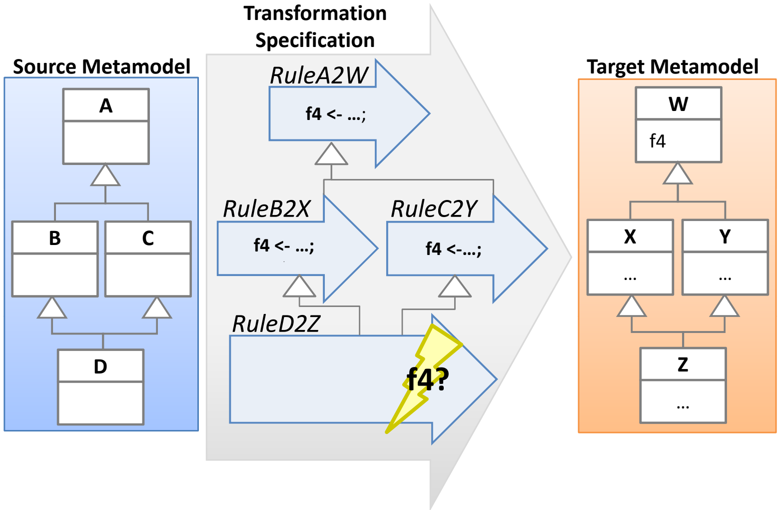

Figure 9 – Example of Diamond Problem

five different pole signatures in this example. For this, the MSA rules are calculated for each pole signature and if more than one MSA rule exists, an ambiguity arises, as is the case for the pole signature {D,G} in the example since `RuleAG2...` and `RuleBF2...` are both applicable and none of them is more specific than the other one.

3.2.5 Conflicts in Multiple Inheritance

The *diamond problem* [Tai96], also referred to as *fork-join* inheritance [Sak89], arises, when contradicting assignments are inherited via different inheritance paths. Consider, for instance, the common superrule `A2W` in Fig. 9, which contains an assignment for feature `f4`. This assignment is overridden within the transformation rules `B2X` and `C2Y`. Thus, it cannot be decided in the rule `D2Z` which assignment should be applied for feature `f4`, unless assistance is given by the transformation designer, e.g., either by selecting one of the inherited assignments or by specifying a specific assignment for feature `f4` in rule `D2Z`.

Specification in OCL. For detecting conflicts in multiple inheritance, Listing 4 depicts the corresponding OCL invariant.

```
context TransformationRule inv NoDiamondProblem:
 self.allEffOE() −> forall(oe | oe.allInhAssignments()
    −> reject(ib | oe.assignments−>collect(feature) −> includes(ib.feature))
    −> forall(ib1,ib2 | ib1 <> ib2 and ib1.feature = ib2.feature implies
      ib1.distance <> ib2.distance or ib1.rule = ib2.rule)
 )

−− OCL operation to compute all inherited assignments.
−− this operation should be equivalent to the operation used for executing transformations
context TransformationRule : def allInhAssignments() : Set(TupleType(feature : Feature,
   rule : TransformationRule, distance : Int)) = ...

−− OCL operation to compute all effective output elements
−− this operation should be equivalent to the operation used for executing transformations
context TransformationRule : def allEffOE() : Set (OutputElement) = ...
```

Listing 4 – Invariant to Check Conflicts in Multiple Inheritance

By means of a helper function (cf. `allEffOE()`), the set of all effective output elements for a transformation rule is calculated first. For each element of this set, the set of all inherited assignments is calculated by means of another helper function (cf. `allInhAssignments()`), whereby for each assignment, the *affected feature*, the *rule* as well as the *distance* from the context rule to the rule setting the feature is stored. The output of this function for a concrete example is illustrated on the right hand side of Fig. 10. The resulting set of assignments is reduced by those assignments, which are overridden in the current context rule (cf. line 3). This is done, since those assignments may not result in a conflict anymore (cf. case ② in Fig. 10). The remaining assignments are then examined for occurring conflicts (cf. lines 4-5). A conflict exists, if two assignments target the same feature, occur at the same distance and originate from different rules. This is, since assignments originating from the same rule exhibit the same specification and thus, no conflict might arise (cf. case ① in Fig. 10). Furthermore, if the distance is different, the most specific assignment, i.e., the one, which is closer to the context rule, wins (cf. case ③ in Fig. 10).

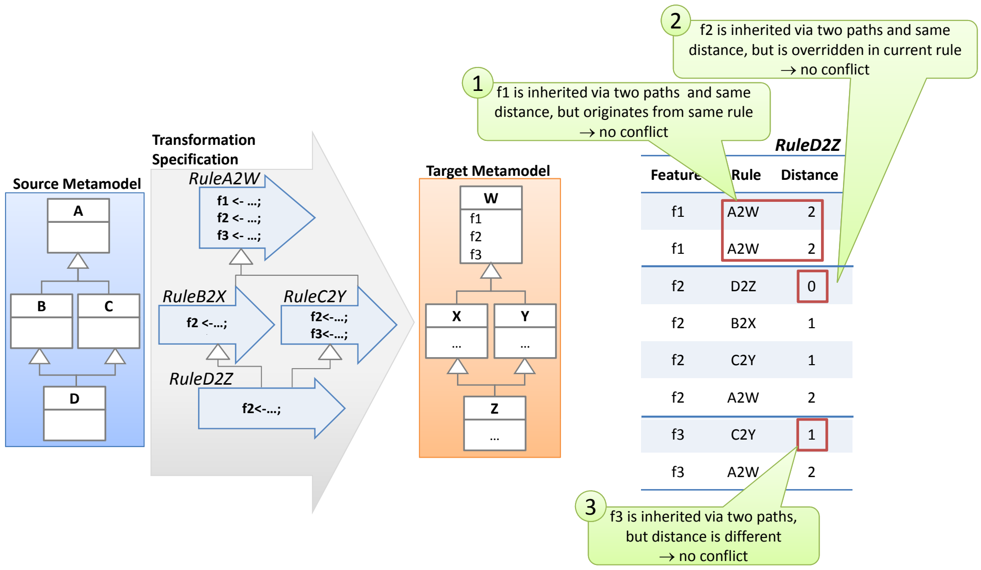

Figure 10 – Example for Diamond Check

3.3 Dynamic Semantics

Now we shift our focus from static to dynamic semantics, i.e., how transformation specifications may be interpreted at run-time. In this context, two main aspects are investigated: (i) which rules apply to which instances, i.e., *dispatch semantics*, and (ii) how a set of inheriting rules gets executed, i.e., *execution semantics*.

Dispatch Semantics. In order to execute transformation specifications, it must be determined which rules apply to which instances, i.e., transformation rules must be dispatched for source model instances. In [CH06], potential strategies and scheduling variations of rules were discussed, but without any focus on inheritance. Thus, literature does not indicate, whether *type substitutability* should be considered in the context of model transformations – instead there exists work on model typing [SJ07], i.e., when a whole model is substitutable by another model, only. The principle of type substitutability is well-known in object-oriented programming and states that if S is a subtype of T, objects of type T may be safely replaced by objects of type S [LW93]. Type substitutability for transformation rules would thus, mean that if a rule may be applied to all instances of class T, then this rule may also be applied to all instances of all subclasses of T. Consequently, if no specific subrule is defined for instances of a subclass, then these instances of the subclass may be transformed by the rule defined for the superclass. However, as already stated, if an object is matched and transformed by a specific rule, more general rules should not match and transform this element again.

Concerning the *evaluation of the conditions* three main strategies may be followed during dispatching. First, the condition is part of the matching process, i.e., if the condition fails, the rule is not applicable, but a superrule might be applied (*rule applicability* semantics). Second, the condition is not part of the matching process, i.e., the matching takes only place on the specified types of the input elements and thus, those elements, which do not fulfill the condition, are filtered, but never matched by a superrule anymore (*filter* semantics). Finally, a condition may represent a *precondition* on the source instances, i.e., instances that do not fulfill the condition are considered to be erroneous and therefore, an exception should be raised.

Execution Semantics. After having determined which rules are applicable to which source model instances, the question arises, how a set of inheriting rules is executed. A first distinguishing criterion is, whether the concept of inheritance is directly supported by the execution engine or whether it is first flattened to ordinary transformation code in a pre-processing step. Independent of whether the inheritance hierarchy is flattened or not, various strategies may be applied to evaluate conditions and to execute assignments. This raises questions such as "Are conditions of a superrule also evaluated?" and "Are the assignments of a superrule executed before the assignments of a subrule?". Hence, we investigated the main characteristics of executing methods in an inheritance hierarchy in object-oriented programming [Tai96]: (i) the *completion of the message lookup*, i.e., whether only the first matching method is executed (*asymmetric*) or all matching methods along the inheritance hierarchy are executed (*composing*), and (ii) provided that a composing completion of the message lookup is given, the *direction of the message lookup*, i.e., whether a method lookup starts in the subclass (*descendant-driven*) or in the superclass (*parent-driven*). Please note that the execution of the assignments may be influenced by the transformation designer in case that a transformation language offers different refinement modes, as discussed in Section 3.1.

4 Comparison of Transformation Languages

In this section, we use the criteria introduced in the previous sections to compare inheritance support in model-to-model transformation languages. The results are based on a carefully developed test set, which includes at least one test case for each criterion. These documented test cases including the example code, the metamodels, and source models may be downloaded from our project homepage[5].

4.1 Comparison Setup

Before delving into the details of the comparison, the chosen set of transformation languages as well as a running example are introduced.

[5] `http://www.modeltransformation.net`

4.1.1 Chosen Transformation Languages

For the comparison, model-to-model transformation languages with dedicated inheritance support have been considered. In order to provide an extensive survey, representatives of the three common paradigms of imperative, declarative, and hybrid transformation languages have been chosen [CH06]. In this context, we examined the imperative transformation languages Kermeta[6] (version 1.4.0) and QVT-O[7] (version 3.1.0), the declarative transformation languages TGGs[8] and TNs [Sch11], and the hybrid transformation languages ATL[9] (version 3.1.0) and ETL[10] (version 0.9.1). Please note that there are different implementations of TGGs, whereby our comparison bases on the one of MOFLON. Although MOFLON's current implementation of the execution engine of TGGs (MOFLON 1.5.1) does not yet support inheritance, TGGs were included, since specific literature concerning inheritance support exists [KKS07]. In order to compare the bidirectional TGG-based model transformation approach with unidirectional languages, we considered only the unidirectional forward translation.

Besides the imperative transformation language QVT-O, the QVT standard specifies additionally the declarative transformation language QVT Relations and a low-level language for specifying the semantics of QVT Relations, i.e. QVT Core. However, QVT Relations is not included in this survey, since QVT Relations supports the redefinition of whole rules, only, i.e., they do not allow the reuse of original rule definitions, and thus, no inheritance between rules is offered.

4.1.2 Running Example

In order to demonstrate the inheritance-related differences in the investigated languages, a running example is introduced. The example has been chosen to be as simple as possible to foster comprehensibility but nevertheless, complex enough to evaluate the key criteria of the framework. As may be seen in Fig. 11, the example aims at transforming UML Statemachine models into according Petri Net models. For achieving this, the actual transformation specification includes three transformation rules, whereby the first transformation rule `Statemachine2Petrinet` is responsible for transforming the according root container objects of the models. The two remaining transformation rules `ModelElem2Element` and `State2Place`, inheriting from each other, should achieve the goal of transforming `State` instances into `Place` instances, whereby only `State` instances, whose kind is unequal "initial" and whose name is not null, should be transformed into according `Place` instances as defined by corresponding conditions. Another inheriting transformation rule `Transition2PNTransition`, which may transform `Transition` instances into `PNTransition` instances has been consciously omitted for demonstration purposes, e.g., to check if a superrule also matches for elements of subtypes if no specific rule has been defined.

In the following, the chosen transformation languages are evaluated according to the comparison framework. Consequently, first a comparison concerning syntactic constructs is performed (cf. Section 4.2), followed by the evaluation concerning the static semantic constraints (cf. Section 4.3). Finally, the dynamic semantics is investigated (cf. Section 4.4), in order to verify if different target models are produced by the transformation languages.

[6] `http://www.kermeta.org`
[7] `http://www.eclipse.org/projects/project.php?id=modeling.m2m.qvt-oml`
[8] `http://www.moflon.org`
[9] `http://www.eclipse.org/atl`
[10] `http://www.eclipse.org/epsilon/doc/etl0`

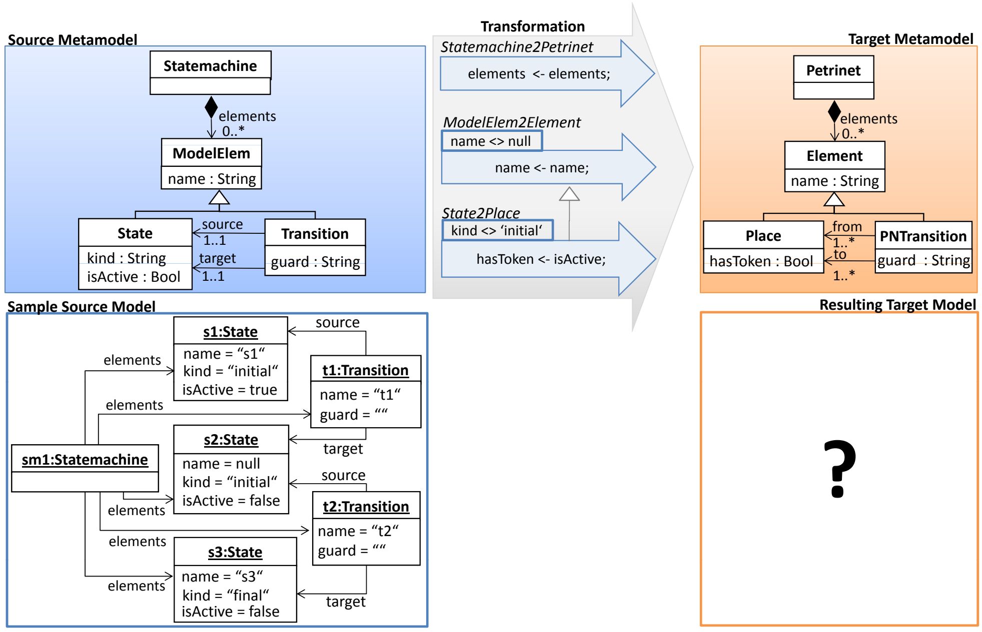

Figure 11 – Running Example

4.2 Comparison of Syntax

As already mentioned, transformation languages allow to specify relationships between source metamodel elements and target metamodel elements by means of dedicated rules. For realizing a model transformation, the specified rules must be applied including recurring tasks. These tasks comprise (i) reading of the source model, (ii) dispatching of rules for source model elements, (iii) execution of the dispatched rules including the instantiation of target model elements and the establishing of a trace model between source model elements and target model elements, and (iv) writing of the target model. Depending on the style of the transformation language and the provided support, some of these tasks may happen transparent to the transformation designer, i.e., no syntactical elements have to be specified for them. For example, in declarative transformation languages the dispatching of rules for source model elements is transparent to the transformation designer, i.e., an abstraction from control flow is achieved. Although the support for these recurring tasks does not directly influence the syntactical elements provided for inheritance, it determines how the presented syntactical solutions for the running example look like and thus, they were shortly discussed beforehand.

In the following, the syntactic elements with respect to inheritance provided by the inspected transformation languages are discussed. To illustrate them, exemplary solutions for the running example in the according languages are presented.

4.2.1 Imperative Languages

In imperative transformation languages, the transformation designer has to take care of the dispatching of rules. This gives full control over the transformation execution, but comes with additional efforts for orchestrating the rules in an appropriate way. Depending on the design of the imperative transformation language, other tasks may be transparent to the transformation designer such as building a trace model between the source and the target model automatically. In the following, the two imperative languages Kermeta and QVT Operational are evaluated according to the comparison framework.

Kermeta. Kermeta allows not only to specify model transformations, but also metamodels including OCL constraints and executable operations. Consequently, Kermeta may be described as a comprehensive environment for metamodel engineering [MFJ05]. Due to this general nature, Kermeta does neither support an explicit rule concept for model transformations nor any support for the recurring tasks of a model transformation out-of-the-box as described above, e.g., dispatching of rules for source model elements, tracing, etc. Although the missing rule concept might be simulated by classes with according methods, the missing support for the recurring tasks leads to verbose transformation specifications that must be defined again and again. Nevertheless, Kermeta is considered to be a dedicated imperative transformation language [CH06] and has thus, been included in this comparison.

In order to make Kermeta comparable with rule-based transformation languages, each transformation rule is implemented by one class and one additional class is responsible for rule dispatching (cf. Listing 5 and Listing 6). Consequently, to implement the running example, one ends up with three classes implementing the transformation rules `Statemachine2PetriNet` (cf. lines 2-18 in Listing 5), `ModelElem2Element` (cf. lines 21-33 in Listing 5), and `State2Place` (cf. lines 36-51 in Listing 5) as well as a third class `StateMachine2Petrinet_Dispatcher` (cf. Listing 6) for the dispatching of the transformation rules.

In this context, each class, which implements a transformation rule follows a certain style, i.e., implements three specific methods. The first method is responsible for implementing conditions (cf. method `conditionFulfilled`), the second one is responsible for implementing attribute assignments (cf. method `attAssignments`) and the third one is responsible for implementing reference assignments (cf. method `refAssignments`). Attribute assignments have been separated from reference assignments, since the assignment of references demands for the availability of the to be referenced objects and may thus, be performed after object creation, only, i.e., in a potential second pass by querying the established trace model.

If transformations in Kermeta are specified in this manner, it is possible to let the according transformation classes inherit from each other (cf. keyword `inherits` in line 36 in Listing 5), since each class follows the same style. However, some specifics must be regarded. First, Kermeta does neither support contra-variance of input parameters nor co-variance of output parameters. Consequently, all methods of classes in an inheritance hierarchy must exhibit exactly the same signatures. Thus, the methods in the class `State2Place` are also typed to `ModelElems` and `Elements` instead of `States` and `Places`, resulting in cast operations (cf. line 45 in Listing 5) for accessing specific attributes and references in subrules. Second, to actually override methods, the keyword for specifying methods must be `operation` in the superclass and `method` in all subclasses.

```
1 //transformation code for Statemachine2PetriNet
2 class Statemachine2PetriNet{
3
4   operation conditionFulFilled(s : Statemachine) : kermeta::standard::Boolean is do
5     result := true
6   end
7
8   operation attAssignments(s : Statemachine, p : PetriNet) is do
9   end
```

```

  operation refAssignments(s : Statemachine, p : PetriNet, trace: Trace<Object, Object>) is do
    s.elements.each{ e |
      if trace.getTargetElem(e) != void then
        p.elements.add(trace.getTargetElem(e).asType(Element))
      end
    }
  end
}

//transformation code for ModelElem2Element
class ModelElem2Element{

  operation conditionFulFilled(m : ModelElem) : kermeta::standard::Boolean is do
    result := and m.name != void
  end

  operation attAssignments(m : ModelElem, e : Element) is do
    e.name := m.name
  end

  operation refAssignments(m : ModelElem, e : Element, trace: Trace<Object, Object>) is do
  end
}

//transformation code for State2Place
class State2Place inherits ModelElem2Element{

  method conditionFulFilled(m : ModelElem) : kermeta::standard::Boolean is do
    result := super(m)
    result := result and (m.asType(State)).kind != "initial"
  end

  method attAssignments(m : ModelElem, e : Element) is do
    super(m,e)
    (e.asType(Place)).hasToken := (m.asType(State)).isActive
  end

  method refAssignments(m : ModelElem, e : Element, trace: Trace<Object, Object>) is do
    super(m,e,trace)
  end
}
```

Listing 5 – Transformation Rules in Kermeta

Furthermore, the class `Statemachine2Petrinet_Dispatcher` has been realized. This class is responsible for iterating the source model elements and ensuring that the corresponding rules are dispatched for them. Additionally, the target model elements are instantiated and trace links are established. To follow the design rationale that the most specific rule is dispatched for a certain source model element, the transformation rules must be called in right order, i.e., from specific to general. For example the rule `State2Place` must be called before the rule `ModelElem2Element` to correctly transform `State` instances with the specific rule. To prevent multiple matches, i.e., to check whether a certain source model element has already been transformed by another rule, first always a query on the trace model is performed. After a first pass for object creation and attribute assignments, a second pass for reference assignments is done (cf. lines 57-76 in Listing 6).

```
1  class Statemachine2Petrinet_Dispatcher{
2
3    //global variable for the trace model
4    reference SM2PN_Trace : Trace<Object, Object>
5
6    //global variables for rules
7    reference SM2PN_Rule : Statemachine2PetriNet
8    reference S2P_Rule : State2Place
9    reference ME2E_Rule : ModelElem2Element
10
```

```
  //main entry point for the transformation
  operation transform(sm : Statemachine) : PetriNet is do

    //initialize the trace model
    SM2PN_Trace := Trace<Object, Object>.new
    SM2PN_Trace.create

    //instantiation of rules
    SM2PN_Rule := Statemachine2PetriNet.new
    S2P_Rule := State2Place.new
    ME2E_Rule := ModelElem2Element.new

    //first pass for object creation + attribute assignments
    //call the rules in right order (from specific to general)
    //transformation for the root object Statemachine
    if SM2PM_Trace.getTargetElem(sm) == void and
      SM2PN_Rule.conditionFulFilled(sm) then
      //initialize the target element
      var pn : PetriNet init PetriNet.new
      SM2PN_Rule.attAssignments(sm, pn)
      //create the trace entry
      SM2PN_Trace.storeTrace(sm, pn)
      //set the root target element
      result := pn
    end

    //transformation for the specific type State
    getAllStates(sm).each{s|
      if SM2PM_Trace.getTargetElem(s) == void and
        S2P_Rule.conditionFulFilled(s) then
        var p : Place init Place.new
        S2P_Rule.attAssignments(s, p)
        SM2PM_Trace.storeTrace(s, p)
      end
    }

    //transformation for the general type ModelElem
    getAllModelElements(sm).each{me|
      if SM2PM_Trace.getTargetElem(me) == void and
        ME2E_Rule.conditionFulFilled(me) then
        var e : Element init Element.new
        ME2E_Rule.attAssignments(me, e)
        SM2PM_Trace.storeTrace(me, e)
      end
    }

    //second pass for reference assignments
    if SM2PN_Trace.getTargetElem(sm) != void then
      SM2PN_Rule.refAssignments(sm, SM2PN_Trace.getTargetElem(sm).asType(PetriNet),
       SM2PN_Trace)
    end

    getAllStates(sm).each{s|
      if SM2PN_Trace.getTargetElem(s) != void then
        S2P_Rule.refAssignments(s,SM2PN_Trace.getTargetElem(s).asType(Element),
          SM2PN_Trace)
      end
    }

    getAllModelElements(sm).each{e|
      if SM2PN_Trace.getTargetElem(e) != void then
        ME2E_Rule.refAssignments(e, SM2PN_Trace.getTargetElem(e).asType(Element),
          SM2PN_Trace)
      end
    }
  end

  //helper to query all States
  operation getAllStates(sm : Statemachine) : State[0..*] is do...
  //helper to query all ModelElements
  operation getAllModelElements(sm : Statemachine) : ModelElem[0..*] is do...
}
```

Listing 6 – Rule Dispatching in Kermeta

Concerning the evaluation of Kermeta according to the criteria posed in the evaluation framework with respect to syntax, one may now conclude that a transformation rule may reference an *arbitrary number of source model elements as well as target model elements*, since the methods `conditionFulFilled`, `attAssignments`, as well as `refAssignments` may exhibit an arbitrary number of parameters. However, please note that the number of parameters must not be changed in subrules – neither in type nor in number. Furthermore, although no explicit concept for specifying *conditions* is provided, this may be simulated by an according method returning a boolean value (cf. method `conditionFulFilled`). Since Kermeta is an imperative language the criterion rule type is evaluated as *lazy*. Additionally, in Kermeta a class may inherit from multiple other classes, i.e., *multiple inheritance* is supported. It is also possible to mark classes as being *abstract* in order to simulate abstract rules. Since Kermeta does not restrict the access to super classes, it is possible to simulate all the introduced *refinements modes*. For example, it is possible to realize the refinement mode *inherit* by first calling the `attAssignments` method of the base class and then altering some attribute values.

QVT Operational. QVT Operational (QVT-O), as the imperative protagonist of the QVT language family, represents a dedicated model-to-model transformation language. Consequently, QVT-O keeps most of the recurring tasks of a model transformation transparent to the transformation designer, e.g., automatically building a trace model during execution. However, since QVT-O is an imperative language, the dispatching of the transformation rules is not automatically achieved, i.e., the transformation designer must take care of the control flow by herself.

Transformation rules are denoted as *mappings* in QVT-O and consist of a *single input element* and an arbitrary number of *output elements* (cf. Listing 7). Additional input elements may be specified as according parameters of the mapping. *Conditions* restricting the applicability of a certain mapping may be specified in terms of OCL within a so-called *when-clause* (cf. lines 17 and 22). QVT-O allows to specify multiple supermappings, i.e., *multiple inheritance* is supported. Although *abstract* mappings may be syntactically specified, they are ignored at run-time, i.e., they are nevertheless executed[11]. In the context of inheritance, different *refinement modes of assignments* are provided depending on the keyword used for specifying the inheritance relationship between rules, e.g., the keyword `inherit` used in the example in Listing 7 implements the *inherit* semantics. Please note that a QVT-O transformation starts automatically with the main method, but mappings have to be explicitly invoked such as done in lines 7, 12, and 13 (cf. rule type *lazy*). In this context, the rules must be called in right order again, i.e., from specific to general.

```
1  modeltype pn uses 'petrinet_1';
2  modeltype sm uses 'statemachine_1';
3
4  transformation testTrafo(in inModel : sm, out outModel : pn);
5
6    main() {
7      inModel.rootObjects()[Statemachine] −> map Statemachine2PetriNet();
8    }
9
10   mapping Statemachine::Statemachine2PetriNet() : PetriNet {
11     //please note that specific rules must be called first!
12     elements := self.elements[State] −> map State2Place();
13     elements += self.elements[ModelElem] −> map ModelElem2Element();
14   }
15
```

[11] However, what can be concluded from the OMG standard document, abstract mappings should not be executable.

```
mapping ModelElem::ModelElem2Element() : Element
when{self.name != null}{
  name := self.name;
}

mapping State::State2Place() : Place inherits ModelElem::ModelElem2Element
when{self.kind != 'initial'}{
  result.hasToken := self.isActive;
}
```

Listing 7 – Transformation Example in QVT-O

4.2.2 Declarative Languages

In contrast to imperative model-to-model transformation languages, declarative transformation languages release the transformation designer from the burden of specifying the control flow, since this is handled by the underlying execution engine. As a consequence, the resulting specifications for TGGs and TNs as depicted in Fig. 12 and in Fig. 13 do not contain any control sequences, but solely define declarative relations between source metamodel elements and their corresponding target metamodel elements (cf. rule type *non-lazy* in Table 1).

Triple Graph Grammars. Triple Graph Grammars (TGGs) exhibit two different levels for the specification of model transformations – first a so-called *type level* to specify high-level correspondence nodes, i.e., *trace links*, and second a so-called *rule level* to specify the actual implementation of correspondence nodes, e.g., attribute assignments. In this context, TGGs allow for 1:1 correspondences on the type level, only, i.e., a *single input and a single output element* is supported. This is in contrast to the rule level, where an arbitrary number of input and output elements may be specified. For the specification of *conditions*, TGGs allow to attach OCL constraints to correspondence nodes on the type level, as may be seen in Fig. 12. Inheritance itself is also specified between correspondence nodes, whereby *multiple inheritance* is supported. Furthermore, it is possible to specify *abstract* correspondence nodes, i.e., abstract transformation rules. However, TGGs do not provide means to specify different *refinement modes* – instead the refinement mode *extend* is assumed, i.e., assignments may be added, only, but assignments of a superrule must not be altered. On investigating the rule level view, one may further see that the inherited assignments are duplicated (e.g., `name:=name` in rule `State2Place`).

Transformation Nets. Transformation Nets are a declarative model-to-model transformation language, forming a DSL on top of Colored Petri Nets (CPNs) [JK09]. They may not only be used to specify model transformations, but may also act as compilation target for other declarative, rule-based model-to-model transformation languages for debugging. This is, since the explicit representation of all the ingredients of a model transformation – i.e., metamodels, models and transformation logic – makes them especially suitable for debugging. In this context, metamodel elements are represented by places, whereby a corresponding place exists for each class, each attribute and each reference. Additionally, model elements are represented by tokens, which are put into the according places. Finally, the actual transformation logic is represented by a system of transitions, which employ graphical patterns in order to describe the matching and producing of tokens (cf. Fig. 13). Further details of Transformation Nets may be found in [Sch11].

In Transformation Nets, it is allowed to match for an arbitrary *number of input elements* (left side of a transition) and to produce an arbitrary *number of output*

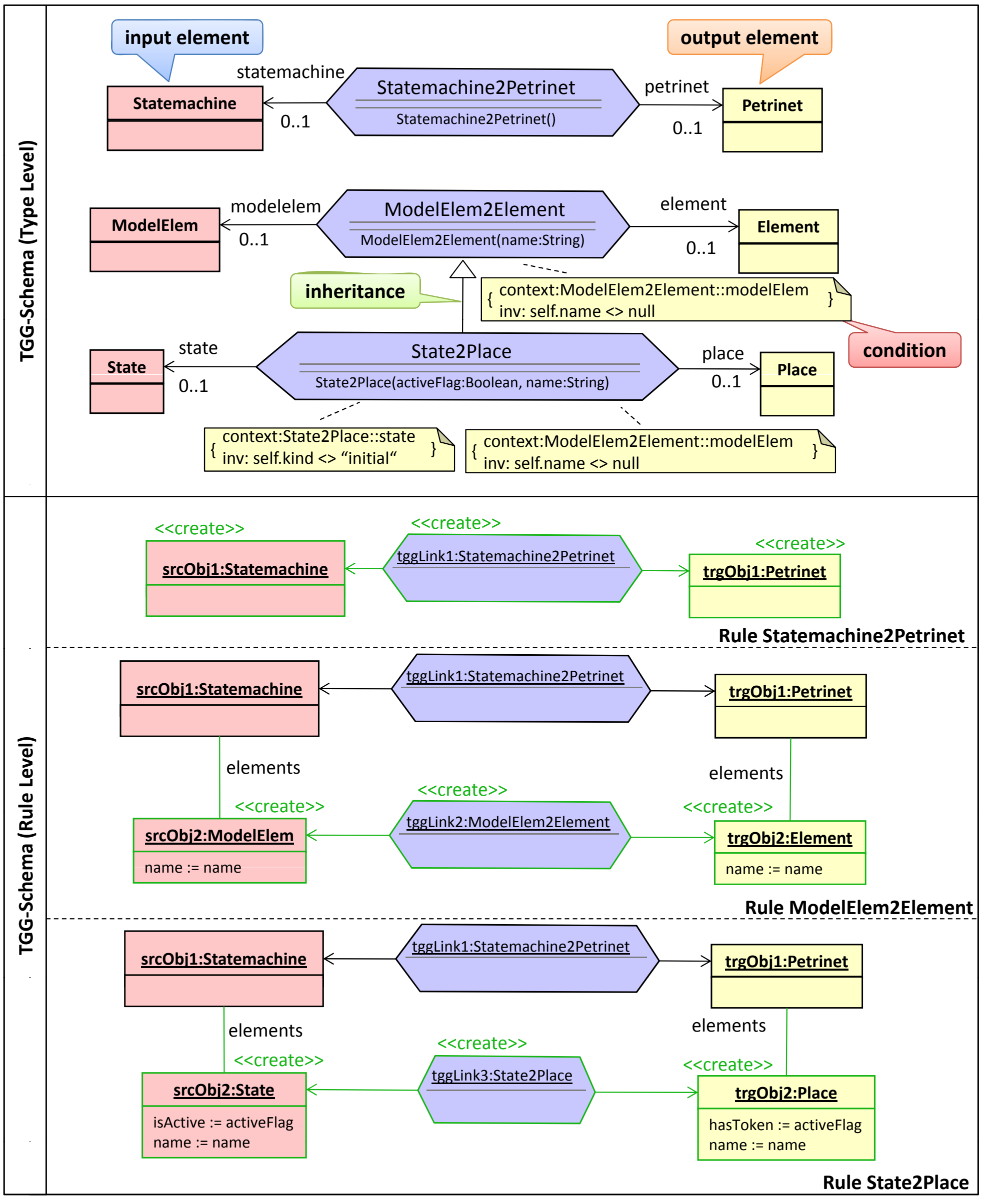

Figure 12 – Transformation Example in TGGs

elements (right side of a transition). It is further possible to add *conditions* to transitions in terms of OCL constraints (extended with the capability to refer to the variables of a certain pattern with the symbol @ – cf. Fig. 13). Transformation Nets allow for *multiple inheritance* between transitions. Additionally, transitions may be marked as being *abstract*. Finally, no means are provided to define different *refinement modes of assignments*. Instead, the refinement mode *override* is implicitly assumed, whereby patterns of subtransitions may override patterns of supertransitions, if they exhibit the same color (variable), e.g., the pattern querying for `State` instances in

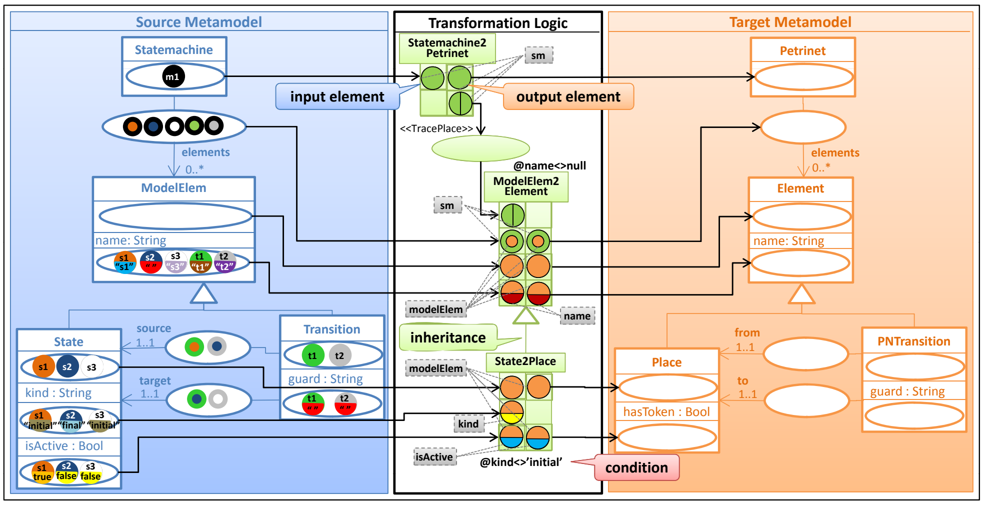

Figure 13 – Transformation Example in TNs

the transition `State2Place` overrides the pattern querying for `ModelElem` instances in the transition `ModelElem2Element`, since both exhibit the same color (variable). In contrast to TGGs, TNs do not enforce syntactical duplication.

4.2.3 Hybrid Languages

Hybrid languages allow to specify transformations by mixing up declarative constructs with imperative ones. For example in ATL and ETL, it is possible to specify declarative transformation rules that are executed by an underlying transformation engine – similar to declarative approaches. Additionally, these rules may incorporate imperative expressions for conditions and assignments. Moreover, both ATL and ETL allow to explicitly call transformation rules which are not automatically activated by the transformation engine, i.e., *lazy* rules. In order to make explicit the associated semantics for inheritance for the declarative parts as well as for the imperative parts, we present for each hybrid language two solutions for the running example. The first solutions ($_{d}$ATL and $_{d}$ETL) are using non-lazy rules, only, whereas the second solutions ($_{i}$ATL and $_{i}$ETL) are employing lazy rules. The resulting specifications are shown for $_{d}$ATL and $_{i}$ATL in Listing 8 and in Listing 9, respectively. Listing 10 shows the $_{d}$ETL solution, while Listing 11 provides a solution in $_{i}$ETL.

ATL. ATL allows for multiple *input elements* in the `from` pattern as well as *multiple output elements* in the `to` pattern. Furthermore, *conditions* may be specified in ATL in terms of OCL conditions after the last input element in the `from` clause (cf. lines 9 and 16 in Listing 8). Additionally, one rule may inherit from one single other rule by means of the keyword `extends`, i.e., *single inheritance* is supported, only. Please note that since the preceding version of this paper, an experimental version of ATL supporting multiple inheritance has been proposed [Wag11]. Nevertheless, this feature is not publicly available in the official distribution yet, and thus, has been neglected in the further comparison. *Abstract* rules are supported by means of the keyword `abstract`. Finally, concerning potential *refinement modes of assignments*, ATL does not provide specific keywords for explicitly choosing a specific semantics to be applied. Instead, *override* semantics is implicitly assumed.

```
rule Statemachine2Petrinet {
  from sm: Statemachine!Statemachine
  to pn: Petrinet!Petrinet(
    elements <− sm.elements
  )
}

rule ModelElem2Element {
  from mElem : Statemachine!ModelElem (mElem.name <> null)
  to elem : Petrinet!Element (
    name <− mElem.name
  )
}

rule State2Place extends ModelElem2Element {
  from mElem : Statemachine!State (mElem.kind <> 'initial')
  to elem : Petrinet!Place (
    hasToken <− mElem.isActive
  )
}
```

Listing 8 – Transformation Example in $_d$ATL

The mentioned language features are available for automatically matched rules (cf. Listing 8) as well as for lazy rules (cf. Listing 9). Lazy rules have to be additionally marked by the `lazy` keyword and are called by using `thisModule.rule_name(input_elements)` explicitly. The return value of a lazy rule is per default its first output element. Please note that the subsequent processing of the produced collection is necessary, because for elements which are not processed by any subrule, a trace entry is created and returned.

```
rule Statemachine2Petrinet {
  from sm: Statemachine!Statemachine
  to pn: Petrinet!Petrinet(
    elements <− sm.elements −> collect(e|thisModule.ModelElem2Element(e))
      −> reject(e|e.oclType().toString().startsWith('TransientLink'))
  )
}

lazy rule ModelElem2Element {
  from mElem : Statemachine!ModelElem (mElem.name <> null)
  to elem : Petrinet!Element (
    name <− mElem.name
  )
}

lazy rule State2Place extends ModelElem2Element {
  from mElem : Statemachine!State (mElem.kind <> 'initial')
  to elem : Petrinet!Place (
    hasToken <− mElem.isActive
  )
}
```

Listing 9 – Transformation Example in $_i$ATL

ETL. ETL allows for a single *input element*, only – cf. single variable after `transform` keyword (cf. lines 2, 8 and 15 in Listing 10). However, an arbitrary number of *output elements* may be specified in the `to` pattern. *Conditions* may be defined in ETL by means of OCL after the keyword `guard` (cf. lines 10 and 18). Additionally, ETL allows for `multiple inheritance`, i.e., several superrules may be specified after the `extends` keyword. Furthermore, *abstract* rules may be specified with the annotation `@abstract`. Finally, ETL again implicitly assumes *override* semantics instead of providing several *refinement modes of assignments*.

```
1 rule Statemachine2Petrinet
2   transform sm: Statemachine!Statemachine
3   to pn : Petrinet!Petrinet {
```

```
    pn.elements ::= sm.elements;
  }

rule ModelElem2Element
  transform mElem : Statemachine!ModelElem
  to elem : Petrinet!Element {
    guard : mElem.name <> null
    elem.name := mElem.name;
  }

rule State2Place
  transform mElem : Statemachine!State
  to elem : Petrinet!Place
  extends ModelElem2Element {
    guard : mElem.kind <> 'initial'
    elem.hasToken := mElem.isActive;
  }
```

Listing 10 – Transformation Example in $_d$ETL

As for ATL, in ETL there are the same possibilities for lazy rules as for automatically matched rules. Lazy rules are marked by a special annotation given before the rule and are called by using the `equivalent` operation. This operation searches for all matching rules for the given context element.

```
rule Statemachine2Petrinet
  transform sm: Statemachine!Statemachine
  to pn : Petrinet!Petrinet {
    pn.elements.addAll(sm.elements.equivalent());
  }

@lazy
rule ModelElem2Element
  transform mElem : Statemachine!ModelElem
  to elem : Petrinet!Element {
    guard : mElem.name <> null
    elem.name := mElem.name;
  }

@lazy
rule State2Place
  transform mElem : Statemachine!State
  to elem : Petrinet!Place
  extends ModelElem2Element {
    guard : mElem.kind <> 'initial'
    elem.hasToken := mElem.isActive;
  }
```

Listing 11 – Transformation Example in $_i$ETL

4.2.4 Synopsis

In summary, one may detect that the languages evaluated offer similar syntactic constructs for the specification of inheritance between transformation rules. Although some languages allow for *single input or output elements*, only, this does not necessarily influence the expressivity of the languages, since typically other means are provided to add further elements, e.g., multiple elements at the rule level in TGGs. Furthermore, all languages allow for the specification of *conditions*. In addition to that, all of the languages except ATL support *multiple inheritance*. *Abstract rules* are also possible in all languages. Finally, a main difference lies in the supported *refinement modes for assignments* as may be seen in Table 1.

Table 1 – Comparison of Syntax

Rule Part	Values	Imperative		Declarative		Hybrid	
		Kermeta[1]	QVT-O	TGGs	TNs	ATL	ETL
Input Elements	**1 | 1…n**	1..n	1[2]	1[4]	1..n	1..n	1
Output Elements	**1 | 1…n**	1..n	1..n	1[4]	1..n	1..n	1..n
Condition	**Yes | No**	Yes	Yes	Yes	Yes	Yes	Yes
Rule Types	**Lazy | Non-lazy**	Lazy	Lazy	Non-lazy	Non-lazy	Lazy | Non-lazy	Lazy | Non-lazy
Type of Rule Inheritance	**Single | Multiple**	Multiple	Multiple	Multiple	Multiple	Single[5]	Multiple
Abstract Rules	**Yes | No**	Yes	Yes[3]	Yes	Yes	Yes	Yes
Refinement Modes of Assignments	**Override | Inherit | Merge | Extend**	Override | Inherit | Merge	Override | Inherit | Merge	Extend	Override (implicit)	Override (implicit)	Override (implicit)

[1] No rule concept in the language available, but may be simulated by dedicated classes and methods
[2] Rule exhibits a single input element only, but may contain several parameters
[3] Although foreseen in the language, they are nevertheless, executed such as concrete rules by the QVT-O implementation
[4] N elements are allowed on rule-level, only
[5] Multiple inheritance has been proposed in [Wag11], but is not publicly available yet

4.3 Comparison of Static Semantics

This part of the comparison evaluates to which extent the static semantics of inheritance is checked in each transformation language, being summarized in Table 2.

4.3.1 Imperative Languages

In this subsection, the evaluation of the imperative languages Kermeta and QVT Operational is performed.

Kermeta. Concerning the *input elements and output elements*, Kermeta does neither support contra-variance of input parameters nor co-variance of output parameters. Consequently, no changes in the types of input and output elements are allowed, i.e., the parameter types of the subrule need to be exactly the same types as those of the superrule, which is enforced at compile-time (cf. Table 2). Furthermore, Kermeta does not allow to alter the number of parameters between inheriting methods, which is again checked at compile-time. In contrast, concrete rules, which target *abstract target classes*, are not recognized at compile-time. However, a run-time error is thrown instead. Furthermore, Kermeta reports no warning, if an abstract class, which represents a rule, is never refined by a *concrete class*, since Kermeta does not implement the rule concept and thus, is not aware of such a problem. With respect to *ambiguous rule definitions*, this criterion is not applicable in Kermeta, since rules are dispatched according to a single type, only, i.e., multiple dispatching of methods is not supported. Finally, the *diamond problem* is detected at compile-time, since the problem of inheriting several equally named methods is recognized. To resolve such ambiguities, the user is asked to explicitly decide for a certain method by the usage of the `from` keyword.

QVT Operational. As demanded by inheriting transformation rules, QVT-O allows for type changes of *input elements and output elements* in a co-variant manner. If this design principle is disregarded, i.e., if the types are changed in a non co-variant manner, an according error message is shown at compile-time – "`Mapping operation has non-conformant signature for inherits`". It is not allowed to change the number of parameters in a mapping rule – neither by extension nor by restriction. Consequently, these criteria have been evaluated as not applicable in Table 2. Regarding concrete rules targeting *abstract target classes*, QVT-O recognizes this at compile-time by the error message "`Result and out parameters of abstract types must be explicitly instantiated in the init-section`". In contrast, abstract rules, which are never refined by any *concrete rule* are not detected – neither at compile-time nor at run-time. With respect to *rule ambiguities*, this criterion is not applicable, since each rule allows for a single input element, only, which is used for dispatching, i.e., single dispatching is applied. Finally, QVT-O does not recognize the *diamond problem*. Instead, the latest specified inherited rule is executed.

4.3.2 Declarative Languages

In the following, the static semantics of the declarative languages TGGs and TNs is evaluated.

Triple Graph Grammars. Regarding *input elements and output elements*, TGGs allow for a co-variant type change of parameters, whereby non co-variant changes are detected at compile-time. To conform to the main principle that applying the subrule should guarantee the existence of the subgraph created by the superrule, only an extension of the number of input and output elements is allowed, which is again ensured statically. With respect to concrete rules, which target *abstract classes*, this problem is not detected at compile-time, but a run-time error is thrown. Concerning a potential warning for abstract rules which are never refined by any *concrete rule*, TGGs do not provide any support. Regarding, *rule ambiguities*, this criterion is not applicable in TGGs, since one input element is allowed at the type level of a TGG rule, only, which is responsible for dispatching the according transformation rule. In case that several rules match for the same input element on the type level, the rule level is investigated. Thereby, it is required that the rules match for disjoint sets of source objects, e.g., built by according conditions. Finally, the *diamond problem* is detected at compile-time.

Transformation Nets. With respect to *input elements and output elements*, like TGGs, also TNs allow for a co-variant type change of parameters, whereby non co-variant changes are again detected at compile-time. Furthermore, an extension of the number of input and output elements is allowed, whereas a restriction is not possible. This is, since elements, which are not re-specified in the syntax, are nevertheless, inherited. Consequently, it is required that inherited patterns still match in case of input elements (denoted as LHS patterns). In case of output elements (denoted as RHS patterns), the same principle is followed, i.e., although patterns need not to be re-specified in the subrule, they are produced anyway. Concrete rules targeting *abstract classes* are detected at compile-time by the error message "`Only abstract transitions may target abstract target classes`". Furthermore, TNs provide a warning, if an abstract rule is never refined by any *concrete rule*. In contrast to the previous approaches, the *rule ambiguity* problem may arise in TNs, since multiple input elements are considered for dispatching, i.e., multiple dispatching is performed. However, potentially arising rule ambiguity problems are detected at compile-time. In order to resolve ambiguity problems, the transformation designer may add priorities to transitions in order to manually influence dispatching. Finally, the *diamond problem* is also recognized at compile-time in TNs. Thereby, a fine-grained check is performed, i.e., an error is only reported, if conflicting assignments are inherited. However, TNs do not provide any means for automatic resolution.

Table 2 – Comparison of Static Semantics

Verification Target	Fault	Values	Imperative		Declarative		Hybrid	
			Kermeta	QVT-O	TGGs	TNs	ATL	ETL
Input Elements	Non-co-variant Type Change	[Compile-Time|Run-Time|No] Error	n.a. (signature must not be changed)	Compile-Time Error	Compile-Time Error	Compile-Time Error	Run-Time Error	No Error (invalid target model)
	Restriction in Number	[Compile-Time|Run-Time|No] Error	n.a. (signature must not be changed)	n.a. (signature must not be changed)	Compile-Time Error	n.a. (input elements are still matched even if not specified again)	Run-Time Error (also with extension)	n.a. (cf. syntax)
Output Elements	Non-co-variant Type Change	[Compile-Time|Run-Time|No] Error	n.a. (signature must not be changed)	Compile-Time Error	Compile-Time Error	Compile-Time Error	Run-Time Error	No Error (invalid target model)
	Restriction in Number	[Compile-Time|Run-Time|No] Error	n.a. (signature must not be changed)	n.a. (signature must not be changed)	Compile-Time Error (except of output to input modification)	n.a. (output elements are still produced even if not specified again)	n.a. (output elements are still produced even if not specified again)	Run-Time Error
Abstract Target Classes	Concrete Rules for Abstract Target Classes	[Compile-Time|Run-Time|No] Error	Run-Time Error	Compile-Time Error	Run-Time Error (application fails)	Compile-Time Error	Run-Time Error	Run-Time Error
Missing Concrete Rule for Abstract Rule		[Compile-Time|Run-Time|No] [Warning|Error]	n.a. (no rule concept supported)	No Warning	No Warning	Compile-Time Warning	Run-Time Error	No Warning
Rule Ambiguity		[Compile-Time|Run-Time|No] Error	n.a. (no multi-dispatch)	n.a. (cf. syntax)	n.a. (cf syntax)	Compile-Time Error	No Error (first matching rule in file wins)	n.a. (cf. syntax)
Diamond Problem		[Compile-Time|Run-Time|No] Error	Compile-Time Error	No Error (latest specified inherited rule wins)	Compile-Time Error	Compile-Time Error	n.a. (cf. syntax)	Compile-Time Error

4.3.3 Hybrid Languages

Finally, this subsection surveys static semantics with respect to the hybrid languages ATL and ETL.

ATL. Concerning *input and output elements*, in ATL a violation of co-variance is detected at run-time, only, resulting in a "`feature not found`" exception. Regarding the number of input elements, in ATL a run-time error occurs, if the number is changed in any way, i.e., ATL prohibits to extend the number of input elements. ATL does not raise any exception if the number of output elements is restricted, since they are produced, even if they are not re-specified. Concrete rules targeting *abstract classes* are not detected at compile-time – instead a run-time error is thrown. The same applies for abstract rules which are never refined by any *concrete rule*. Instead of providing a warning at compile-time, the run-time error "`Operation not found`" is thrown. In ATL, where also multiple input elements are allowed and thus, multiple dispatching is considered, no exceptions for *ambiguous rule definitions* are thrown – neither at compile-time nor at run-time. Instead, the first matching rule defined in the file is executed. Finally, in ATL the *diamond problem* does not apply, since multiple inheritance is not supported.

ETL. With respect to *input and output elements*, in ETL no error is reported, if the types are changed in a non-covariant way. Instead, a target model with invalid features is created. The restriction of the number of input elements is not applicable, since ETL restricts the number of input elements to exactly one, anyway. If the number of output elements is restricted, a run-time error ("`index out of bound`" exception) is raised. Concrete rules targeting *abstract classes* are not detected at compile-time, but run-time errors are thrown instead. In contrast, abstract rules, which are never refined by any *concrete rule* are not detected at all – neither at compile-time nor at run-time. The problem of *ambiguous rule definitions* may not arise in ETL, since multiple input elements are not supported. Finally, the *diamond problem* results in a compile-time error. However, it is recognized at a coarse-grained level, only, since an error is reported in any case, even if no conflicting assignments exist.

4.3.4 Synopsis

In summary, one may see that the checking of the static semantics is still limited in the various transformation languages (cf. Table 2). Regarding *input and output elements*, Kermeta is most restrictive, since no changes in types and number are allowed. In contrast, all the other languages allow for co-variance of input elements and output elements, which is typically ensured at compile-time – the only exceptions are ATL and ETL. In contrast, a potential restriction in number is either not allowed (cf. Kermeta and QVT-O), not possible due to syntactical constraints (cf. ETL), statically checked (cf. TGGs) or not applicable, since the elements, which are not re-specified are nevertheless inherited (cf. TNs and ATL). With respect to concrete rules targeting *abstract classes*, most of the languages detect this not before run-time, except QVT-O and TNs. Concerning abstract rules which are never refined by any *concrete rule*, only TNs provide support. The rule *ambiguity problem* is in four of the six languages rated as not applicable, since a single element is employed for dispatching, only. The only exceptions are TNs and ATL. Finally, the *diamond problem* is in all languages that support multiple inheritance statically detected, except in QVT-O.

4.4 Comparison of Dynamic Semantics

Fig. 14 shows the resulting target models after having executed the specified transformations. It may be seen that the languages produce partly different target models, although syntactically equal transformation logic has been specified. Consequently, the investigated transformation languages exhibit different dynamic semantics. In order to compare the dynamic semantics, the *dispatch semantics* as well as the *execution semantics* are investigated in the following (cf. Table 3).

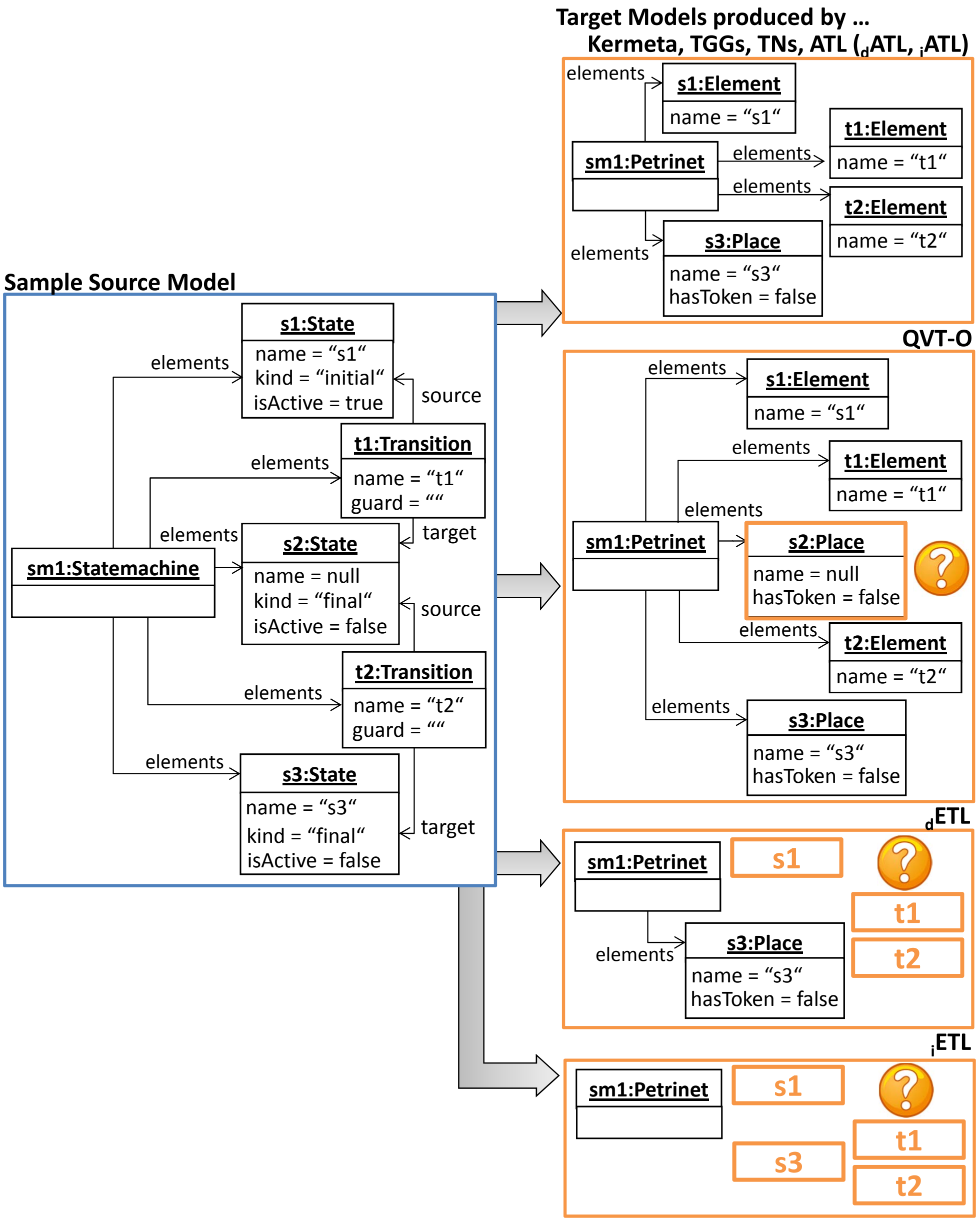

Figure 14 – Resulting Target Models of the Specified Transformations

4.4.1 Imperative Languages

In this subsection, the evaluation of the imperative languages Kermeta and QVT Operational is performed.

Kermeta. Since in Kermeta no explicit rule concept is available, the dynamic semantics of a transformation incorporating inheritance depends on the concrete realization of the rule concept as decided by the transformation designer. For example, type substitutability may be realized, if rules are called accordingly, but may also not be supported, if the transformation designer decides differently. Consequently, the criteria for *dynamic semantics* have been rated as not applicable. Kermeta provides direct inheritance support in the engine, since it is compiled into Java code.

QVT Operational. In contrast to Kermeta, QVT-O exhibits an explicit rule concept, for which dedicated behavior is defined. The only decision, which is left to the transformation designer is the dispatching of the transformation rules, since in imperative languages, control flow is typically user-defined. Consequently, the subcriteria of the category *dispatch semantics* have been rated as not applicable. By default, rule applicability semantics for conditions is applied. However, a specifics of QVT-O is that conditions may also be interpreted as *preconditions*, if a transformation is executed in the so-called *strict mode*. In case that a condition is not fulfilled in this mode, the transformation throws an exception and terminates execution. Thus, in this mode, conditions act as preconditions that need to be fulfilled by any input model. Regarding *execution semantics*, inheritance support in the engine in QVT-O is unknown, but it is assumed that direct support in the engine is available, since the implementation builds on Java. Concerning the execution of conditions, surprisingly, an asymmetric completion of the lookup is performed, i.e., conditions are *not* inherited along the inheritance hierarchy. Consequently, the output model produced by QVT-O deviates from the other output models, since it exhibits an additional `Place` instance `s2`, which results from the fact that the `State` instance `s2` fulfills the condition of the subrule, but needs not to fulfill the condition of the superrule. Finally, regarding the execution of assignments, a composing completion of the lookup is performed, i.e., all assignments along the inheritance hierarchy are executed. The actual direction of the lookup is parent-driven, i.e., assignments of superrules are executed before the assignments of the subrule.

4.4.2 Declarative Languages

In the following, the dynamic semantics of the declarative languages TGGs and TNs is evaluated.

Triple Graph Grammars. When regarding the *dispatch semantics* of TGGs, one might detect that type substitutability is supported. Furthermore, rule applicability semantics for evaluating conditions is employed as may be concluded from the resulting instances in the output model. Concerning the realization of inheritance in the engine (cf. sub-criterion of *execution semantics*), TGGs do not need an explicit support for the concept, since inheritance is flattened in the syntax already – thus, this criterion has been rated as not applicable. Finally, regarding the execution of conditions and assignments, a composing behavior is automatically given due to the flattening in the syntax. This is also the reason, why direction of the lookup is not applicable.

Transformation Nets. Like TGGs, TNs also support type substitutability as well as rule applicability for conditions when analyzing the *dispatch semantics.* When taking a look at the *execution semantics*, one may find that TNs support the concept of inheritance in the engine by flattening, i.e., all inherited patterns along the inheritance hierarchy are collected and result in a single transition on the CPN level. Conditions as well as assignments exhibit a composing completion of the lookup, i.e., all inherited conditions and assignments are applied in a transformation rule. Whereas the direction of the lookup in case of conditions happens descendent-driven, i.e., the most specific condition is evaluated first, the direction of the lookup in case of assignments has been evaluated as not applicable, since the assignments occur in a single firing step on the CPN level due to the flattening.

4.4.3 Hybrid Languages

Finally, this subsection surveys dynamic semantics with respect to the hybrid languages ATL and ETL.

ATL. As may be seen in Fig. 14, the outputs for the $_d$ATL and $_i$ATL solutions are the same. With respect to *dispatch semantics*, ATL employs type substitutability as well as rule applicability for evaluating conditions. When taking a look at the sub-criteria making up the *execution semantics*, one may find that the concept of inheritance is not directly supported in the engine, but flattened before execution. Furthermore, conditions as well as assignments exhibit both a composing completion of the lookup. Nevertheless, conditions are evaluated parent-driven, whereas assignments are evaluated descendent-driven. The descendent-driven evaluation of assignments is surprising, since typically assignments must be executed parent-driven in order to achieve *override* semantics, which is implicitly assumed in ATL. However, ATL accomplishes *override* semantics by the descendent-driven evaluation of assignments though, since it pursues an optimized composing strategy, i.e., in the flattening process the overridden assignments are removed and thus, for each feature a single assignment remains, only. Please note that only assignments specified within the declarative part of ATL are inherited, whereas the imperative parts, i.e., statements in the `do`-block, are ignored.

ETL. Concerning *dispatch semantics*, one may see that ETL does not support type substitutability by default – neither for lazy rules nor for non-lazy rules. This may be inferred from the fact that no `Element` instances have been created. Instead, $_d$ETL's target model includes a single `Place s3`, only and $_i$ETL's target model includes no `Place` instance at all, since no dynamic dispatching for source model instances is supported (cf. Fig. 14). The dispatch semantics may be modified by annotating rules with `@greedy`. Such rules also match indirect instances, but the interpretation is different though, since the superrule still regards all instances irrespective of whether the instances have already been matched by subrules or not. Consequently, the application of the rule `ModelElem2Element` annotated with `@greedy` would result in both cases, i.e., $_d$ETL and $_i$ETL, in six instances in total: four `Element` instances `s1`, `s3`, `t1`, and `t2` produced by the superrule `ModelElem2Element`, one `Place` instance `s3` produced by the subrule `State2Place`, and finally, one `Petrinet sm1` stemming from the rule `Statemachine2Petrinet`. Thus, even if type substitutability is enabled in ETL, the result of the condition evaluation does not influence the dispatch semantics because the superrule always matches all direct and indirect instances, i.e., disregards subrules. Consequently, the criterion condition semantics has been rated as not applicable. Regarding the sub-criteria of the *execution semantics*, one may first find that ETL provides direct support for the concept of inheritance in the engine. With respect to the execution of conditions and assignments, one may see that both are performed by a composing completion of lookup. However, conditions are evaluated descendent-driven, whereas assignments are evaluated parent-driven, whereby the implicitly assumed *override* semantics is achieved. Please note that in contrast to ATL, ETL does not exhibit an explicit section for imperative parts and consequently, also imperative parts are inherited to subrules.

4.4.4 Synopsis

In summary, one main difference with respect to *dispatch semantics* is the application of type substitutability in the different languages as may be seen in Table 3. Whereas ATL, TGGs, and TNs provide support by default, ETL allows the transformation designer to interfere. However, type substitutability is interpreted differently in ETL anyhow, as discussed above. Furthermore, the imperative languages QVT-O and Kermeta also allow the transformation designer to interfere, since the calling of rules is performed by the transformation designer. Provided that type substitutability is supported, all the languages evaluated provide rule applicability semantics for conditions. This may be inferred from the fact that the output models produced by ATL, TGGs and TNs include an `Element` instance `s1`, which has been produced by the superrule `ModelElem2Element`, since `s1` does not fulfill the condition of the specific rule `State2Place`. Regarding *execution semantics*, one may find that the concept of inheritance is rarely supported in the engine – typically, inheritance is flattened before execution. Furthermore, conditions are evaluated by a composing completion of the lookup - the only exception thereof is QVT-O, which implements an asymmetric completion of lookup. Finally, all of the transformation languages implement a *composing* behavior for assignments.

5 Lessons Learned

This subsection presents lessons learned from our comparison.

Similar Syntax, Different Semantics. As especially the examples in Listing 8 and in Listing 10 reveal, similar syntax does not necessarily lead to the same results, which implies different dynamic semantics. This is undesirable, since the dynamic semantics is not made explicit by any syntactical elements to the transformation designer. Thus, the transformation designer must know the design decisions taken in each transformation language in order to obtain the desired result. Therefore, the current situation concerning rule inheritance is comparable to the situation in the early stages of object-oriented programming, where no common agreements on the dynamic semantics of inheritance had been reached.

Limited Support for Static Semantics. Currently, support for checking the static semantics is limited except in TGGs and TNs. This gives rise to run-time errors or – even worse – to erroneous target instances with no error message. Thus, the tedious task of checking the static semantics is left entirely to the transformation designer. The OCL invariants defined in Section 3.2 for the generic transformation metamodel, may act as a blueprint for developing static checks for specific transformation languages. In particular, transformation languages which are equipped by a metamodel may adopt the present OCL constraints to their specific structures.

Fixed Dynamic Semantics. As introduced above, different kinds of refinement modes may be desirable. The evaluation of the languages has shown, that most of them assume a certain refinement mode, but only QVT-O allows the transformation designer to choose between different options. Thus, most of the languages support only fixed dynamic semantics for rule inheritance. Since different dynamic semantics

Table 3 – Comparison of Dynamic Semantics

Criterion		Subcriterion	Values	Imperative		Declarative		Hybrid	
				Kermeta	QVT-O	TGGs	TNs	ATL	ETL
Dispatch semantics		Type Substitutability	Yes | No	n.a. (determined by programmer)	n.a. (depends on rule calling order)	Yes	Yes	Yes	User-Definable
		Condition Semantics	Filter | Rule Applicability | Precondition	n.a. (determined by programmer)	Rule Applicability (or Precondition in case of strict mode)	Rule Applicability	Rule Applicability	Rule Applicability	n.a. (due to different interpretation of type substitutability)
Execution Semantics	Inheritance Support	-	Flattened | Direct engine support	Direct engine support (building on Java)	Unknown (Direct support assumed since building on Java)	n.a. (since flattened in patterns already)	Flattened	Flattened	Direct engine support
	Condition	Completion of lookup	Asymmetric | Composing	n.a. (determined by programmer)	Asymmetric	Composing (by copy)	Composing (by flattening)	Composing	Composing
		Direction of lookup	Parent-driven | Descendent-driven	n.a. (determined by programmer)	n.a. (due to asymmetric behaviour)	n.a. (due to copy in syntax)	Descendent-driven	Parent-driven	Descendent-driven
	Assignments	Completion of lookup	Asymmetric | Composing	n.a. (determined by programmer)	Composing	Composing (by copy)	Optimized Composing (by flattening)	Optimized Composing	Composing
		Direction of lookup	Parent-driven | Descendent-driven	n.a. (determined by programmer)	Parent-driven	n.a. (due to copy in syntax)	n.a. (single firing step)	Descendent-driven	Parent-driven

are suitable for different transformation scenarios, the transformation designer should be enabled to alter the dynamic semantics. The introduction of a `super` reference as in object-oriented programming languages and already supported in Kermeta would enable the transformation designer to express different refinement modes.

Missing Access Modifiers. None of the transformation languages evaluated provide any means to restrict accessability of assignments of a superrule. Instead, all assignments may be accessed by the subrules. However, such access modifiers could be employed to prohibit overriding of assignments in subrules. Furthermore, modifiers to prohibit the overriding of rules are missing, e.g., a keyword with the semantics of `final` in Java – especially in the context of inheriting whole transformations.

Consequences for Transformation Design. The discovered differences in the interpretation of inheritance lead to profound consequences for transformation design. Concerning ATL, the main restriction is the support for single inheritance, only. Although multiple inheritance may be achieved by simulation, this leads to code duplication reducing the advantages of the concept of inheritance significantly. In contrast, although TGGs and ETL allow for multiple inheritance, they exhibit other intricacies. In case of TGGs, assignments are duplicated in any case. ETL provides a different interpretation of type substitutability, leading to redundant instances. Thus, transformation development demands for a detailed knowledge to achieve exactly the same outcome of a transformation expressed in different transformation languages.

Imperative Languages Provide Freedom but Little Support. When taking a look at the imperative languages evaluated, one might detect that concerning the dynamic semantics much freedom is provided due to the user-defined specification of control flow. This exhibits the advantage that the actual execution semantics of inheritance may be influenced by the transformation designer. Maximal freedom in writing transformations may be achieved by employing general purpose programming languages like Java. To include the concept of transformation rules therein, frameworks for writing transformations have been proposed [ABE^{+}06]. However, the main drawback of this is that the transformation designer must take care thereof by herself. Especially, how to properly orchestrate lazy rule calls without having dynamic bining seems to be challenging in large model transformations.

6 Related Work

This section considers three threads of related work. First, we focus on inheritance support in transformation languages, and second, since inheritance is mainly a reuse mechanism, we broaden the scope to other reuse facilities. Finally, inheritance support in rule-based languages beyond the scope of MDE is highlighted.

6.1 Inheritance Support in Transformation Languages

Although inheritance plays a vital role in object-oriented modeling, and thus, also in model transformations, no dedicated survey exists to the best of our knowledge. Only a small number of publications mention inheritance, explicitly. Inheritance support in ATL is briefly described in [JK05], and that in ETL in [KPP08], but rather on a syntactical level, while the actual execution semantics are left open. A detailed discussion of static semantics that must be considered in TGG rule inheritance may be found in [KKS07]. For graph transformations in general, Bardohl et. al [BEdLT04] introduced type substitutability when executing graph transformation rules, i.e., (abstract) supertypes may be used in patterns which are then applicable to subtypes at run-time. Finally, in the QVT standard [OMG09] detailed semantics with respect to inheritance is defined for QVT Operational, only.

6.2 Reuse Facilities in Model Transformations

Proposed reuse facilities in the area of model transformations target at different scopes, e.g., within/across transformations or between the same/different metamodels. Consequently, reuse facilities may be divided into five different scopes ranging from *reuse in the small* to *reuse in the large*, as detailed in the following.

Scope 1. To avoid code duplication, reuse of logic within *a single transformation* is needed, i.e., the scope is to reuse the *same transformation logic* between the *same metamodels* in the *same transformation*. Proposed reuse facilities for this scope include *functions* as well as *inheritance*, whereby functions are supported by nearly all transformation languages, being in contrast to inheritance as already detailed above.

Scope 2. To realize similar transformation logic, reuse of transformation logic between the *same metamodels* in *different transformations* is needed. In this context, two main reuse facilities have been proposed - namely *superimposition* for ATL [WVDSD10] or QVT Relations (cf. redefinition of whole rules in an inheriting transformation) and so-called *transformation product lines* [KGKG09, Sij10]. Thereby, superimposition allows to build the union of transformation rules from different transformations. Rules may be redefined, i.e., a rule is replaced by a new one if their signatures are identical, and added, whereby it is impossible to reuse the original rule. To deal with variabilities in model transformations, approaches arose that allow transformation designers to explicitly specify potential variabilities in model transformations, which we call transformation product lines (inspired by software product lines). These approaches typically use some variability model, e.g., feature models, to guide the generation of a specific transformation.

Scope 3. Thirdly, reuse of transformation logic may be required across the boundaries of metamodels. In this respect, *generic transformations* and *domain-specific languages* (DSLs) have been proposed, whereby genericity allows to parameterize transformation logic with types to *abstract* from concrete metamodels. Thereby, approaches have been proposed for *fine-grained* genericity [LAKS09, VP04], i.e., on the level of rules or functions, and *coarse-grained* genericity [CGdL11, WKR$^+$11], i.e., on the level of transformations. DSLs provide means to simplify specification of recurring problems in transformations. Two different kinds of DSLs may be distinguished: (i) *external* DSLs, i.e., the DSL may be used independently from the underlying transformation language, and (ii) *internal* DSLs, i.e., DSL constructs are embedded in a transformation language. External DSLs have been proposed by [DFV09] and [WKK$^+$10], which focus on the resolution of structural heterogeneities. In order to execute a DSL-based specification, it has to be translated into a certain executable transformation language. Internal DSLs follow the same principles but differ in the fact that DSL constructs are tightly integrated in a certain transformation language. A representative for internal DSLs is the High Level Navigation Language (HNL) [CJGMB09], which hides complex OCL navigation expressions using ATL as host language. Furthermore, modularization concepts in the area of graph transformations have been proposed, which allow to encapsulate graph transformation rules in so-called units [KHKK04]. These units may then be imported in a graph transformation and by this reused.

Scope 4. Since cross-cutting concerns, e.g., debugging or tracing, should be reusable throughout transformations, mechanisms are needed that allow to reuse *logic* irrespective of *metamodels* and *transformations*. To reuse such cross-cutting concerns, several mechanisms have been proposed, including *higher-order transformations* [TJF+09], *aspect-orientation*, e.g., supported in Kermeta, and *reflection* [Kur10].

Scope 5. Finally, to achieve reuse in the large, whole transformations might be reused *without adaptations*. Thus, mechanisms exist to orchestrate model transformations, e.g., describing sequential or conditional executions of model transformations. *Orchestration languages* have been proposed to replace low-level descriptions, e.g., in terms of Ant[12] tasks. Basically, they may be divided into approaches allowing to orchestrate model transformations written in different languages [Kle06, Old05, VAB+07] or in a specific language, only (Wires* [RRGLR+09], ATLFlow[13], QVT-O[14]).

Synopsis. Although a large number of reuse mechanisms including inheritance has been proposed, several shortcomings might be identified, which still represent major barriers to efficient reuse in model transformations. These barriers include, e.g., insufficient abstraction from metamodels, since most of the proposed reuse mechanisms do not allow to decouple transformation logic and concrete metamodels. Furthermore, although mechanisms for reuse have been presented, corresponding repositories of reusable artifacts are still missing. This is in contrast to software engineering, where different kinds of repositories of reusable artifacts exist, ranging from fine-grained class-libraries (being delivered with any programming language) over components to coarse-grained frameworks. Finally, the specialization of reusable artifacts is often challenging. For example, in case of inheritance, specialization has potential for improvement, since none of the approaches allows to define reuse policies, e.g., to disallow rule inheritance (cf. `final` keyword in Java) or to define some access rights (cf. keywords `private`, `protected` or `public`).

6.3 Inheritance in Rule-Based Languages Beyond MDE

Beyond the area of MDE, rule-based languages may be found in the area of data engineering as well as ontology engineering. Concerning the former, rule-based languages are employed in active databases in the form of event-condition-action constructs, i.e., triggers. In this context, static semantics for inheriting triggers has been defined in [CMR99]. This work has been complemented in [BGM00] by also discussing the dynamic semantics of inheriting triggers. When regarding XML, the concept of inheritance is considered in the XML Schema[15] standard by type derivation. However, corresponding transformation languages like XSLT[16] or XQuery[17] neglect the concept of rule inheritance, e.g., XSLT supports the reuse of templates only by delegation. Regarding the area of ontology engineering, an extensive survey on inheritance in rule-based frame systems may be found in [YK06].

[12] `http://ant.apache.org/`
[13] `http://opensource.urszeidler.de/ATLflow/`
[14] `http://www.omg.org/spec/QVT/1.1/`
[15] `http://www.w3.org/XML/Schema`
[16] `http://www.w3.org/TR/xslt`
[17] `http://www.w3.org/TR/xquery`

7 Conclusion and Future Work

In this paper, we have presented a systematic comparison of inheritance support in model-to-model transformation languages. In particular, we considered Kermeta, QVT-O, TGGs, TNs, ATL, and ETL. We (i) identified syntactic concepts required for inheritance, (ii) elaborated on static semantics that should be checked between inheriting rules, and (iii) investigated potential dynamic semantics of rule inheritance. Thus, the design rationales behind the realizations of rule inheritance in different languages have been made explicit.

Since this paper focussed on inheritance in model-to-model transformation languages, future work includes a survey of inheritance in model-to-text transformation languages, e.g., in Xtend[18]. Furthermore, since we considered functional requirements of inheritance only, an investigation of non-functional requirements, e.g., performance measures between transformations employing inheritance and transformations neglecting inheritance may also be an interesting point for future work. Finally, a user study with transformation designers would be of interest to find out, to which extent inheritance is applied in real world transformation examples.

References

[ABE+06] David H. Akehurst, Behzad Bordbar, M. J. Evans, W. Gareth J. Howells, and Klaus D. McDonald-Maier. SiTra: Simple Transformations in Java. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *LNCS*, pages 351–364. Springer, 2006. doi:10.1007/11880240_25.

[AD96] Eric Amiel and Eric Dujardin. Supporting explicit disambiguation of multi-methods. In Pierre Cointe, editor, *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *LNCS*, pages 167–188. Springer, 1996. doi:10.1007/BFb0053061.

[ADL91] Rakesh Agrawal, Linda G. Demichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. In Andreas Paepcke, editor, *Proceedings of the 6th International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, pages 113–128. ACM, 1991. doi:10.1145/117954.117963.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BEdLT04] Roswitha Bardohl, Hartmut Ehrig, Juan de Lara, and Gabriele Taentzer. Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In Michel Wermelinger and Tiziana Margaria, editors, *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE'04)*, volume 2984 of *LNCS*, pages 214–228. Springer, 2004. doi:10.1007/978-3-540-24721-0_16.

[18] http://www.eclipse.org/workinggroups/oaw

[Béz05] Jean Bézivin. On the Unification Power of Models. *SoSyM Journal*, 4(2):171–188, 2005. doi:10.1007/s10270-005-0079-0.

[BGM00] Elisa Bertino, Giovanna Guerrini, and Isabella Merlo. Trigger Inheritance and Overriding in an Active Object Database System. *IEEE Trans. on Knowl. and Data Eng.*, 12(4):588–608, 2000. doi:10.1109/69.868909.

[CGdL11] Jesús Cuadrado, Esther Guerra, and Juan de Lara. Generic Model Transformations: Write Once, Reuse Everywhere. In Jordi Cabot and Eelco Visser, editors, *Proceedings of the 4th International Conference on Theory and Practice of Model Transformations (ICMT'11)*, volume 6707 of *LNCS*, pages 62–77. Springer, 2011. doi:10.1007/978-3-642-21732-6_5.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006. doi:10.1147/sj.453.0621.

[Cha92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92)*, volume 615 of *LNCS*, pages 33–56. Springer, 1992. doi:10.1007/BFb0053029.

[CJGMB09] Jesús Cuadrado, Frédéric Jouault, Jesús García Molina, and Jean Bézivin. Experiments with a High-Level Navigation Language. In Richard F. Paige, editor, *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT'09)*, volume 5563 of *LNCS*, pages 229–238. Springer, 2009. doi:10.1007/978-3-642-02408-5_16.

[CMR99] Naumann Chaudry, James Moyne, and Elke Rundensteiner. Rule Inheritance and Overriding in Active Object-Oriented Databases. In *Current trends in data management*, pages 153–178. IGI Publishing, 1999.

[DFV09] Marcos Del Fabro and Patrick Valduriez. Towards the Efficient Development of Model Transformations using Model Weaving and Matching Transformations. *SoSyM Journal*, 8(3):305–324, 2009. doi:10.1007/s10270-008-0094-z.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, 2008. doi:10.1016/j.scico.2007.08.002.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the Model Transformations in Practice Workshop @ MoDELS'05*, 2005.

[JK09] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets - Modeling and Validation of Concurrent Systems*. Springer, 2009.

[KGKG09] Amogh Kavimandan, Aniruddha Gokhale, Gabor Karsai, and Jeff Gray. Templatized Model Transformations: Enabling Reuse in Model Transformations. Technical report, Vanderbilt University, 2009.

[KHKK04] Renate Klempien-Hinrichs, Hans-Jörg Kreowski, and Sabine Kuske. Typing of Graph Transformation Units. In Hartmut Ehrig, Gregor

Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Proceedings of the 2nd International Conference on Graph Transformations (ICGT'04)*, volume 3256 of *LNCS*, pages 203–206. Springer, 2004. doi:10.1007/978-3-540-30203-2_10.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model transformation in the large. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 285–294. ACM, 2007. doi:10.1145/1287624.1287664.

[Kle06] Anneke Kleppe. MCC: A Model Transformation Environment. In Arend Rensink and Jos Warmer, editors, *Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'06)*, volume 4066 of *LNCS*, pages 173–187. Springer, 2006. doi:10.1007/11787044_14.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT'08)*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008. doi:10.1007/978-3-540-69927-9_4.

[Kur10] Ivan Kurtev. Application of Reflection in a Model Transformation Language. *SoSyM Journal*, 9(3):311–333, 2010. doi:10.1007/s10270-009-0138-z.

[LAKS09] Elodie Legros, Carsten Amelunxen, Felix Klar, and Andy Schürr. Generic and Reflective Graph Transformations for Checking and Enforcement of Modeling Guidelines. *Journal of Visual Language and Computing*, 20(4):252–268, 2009. doi:10.1016/j.jvlc.2009.04.005.

[LW93] Barbara Liskov and Jeannette M. Wing. A New Definition of the Subtype Relation. In Oscar Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, volume 707 of *LNCS*, pages 118–141. Springer, 1993. doi:10.1007/3-540-47910-4_8.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In Lionel C. Briand and Clay Williams, editors, *Proceedings od the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005. doi:10.1007/3-540-47910-4_8.

[MG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006. doi:10.1016/j.entcs.2005.10.021.

[MSZJ04] Haohai Ma, Weizhong Shao, Lu Zhang, and Yanbing Jiang. Applying OO metrics to assess UML meta-models. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *Proceedings of the 7th International Conference on the Unified Modelling Language (UML'04)*, volume 3273 of *LNCS*. Springer, 2004. doi:10.1007/978-3-540-30187-5_2.

[Old05] Jon Oldevik. Transformation Composition Modelling Framework. In Lea Kutvonen and Nancy Alonistioti, editors, *Proceedings of the 5th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS'05)*, volume 3543 of *LNCS*, pages 108–114. Springer, 2005. doi:10.1007/11498094_10.

[OMG09] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. http://www.omg.org/spec/QVT/1.1/Beta2/PDF/, 2009.

[RRGLR+09] José E. Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating ATL Model Transformations. In *Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL'09)*, 2009.

[Sak89] Marko Sakkinen. Disciplined Inheritance. In Stephen Cook, editor, *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, pages 39–56. Cambridge University Press, 1989.

[Sch11] Johannes Schönböck. *Testing and Debugging of Model Transformations*. PhD thesis, Vienna University of Technology, Business Informatics Group, 2011.

[Sij10] Marten Sijtema. Introducing Variability Rules in ATL for Managing Variability in MDE-based Product Lines. In *Proceedings of the 2nd International Workshop on Model Transformation with ATL MtATL'10*, 2010.

[SJ07] Jim Steel and Jean-Marc Jézéquel. On model typing. *SoSyM Journal*, 6(4):401–413, 2007. doi:10.1007/s10270-006-0036-6.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003. doi:10.1109/MS.2003.1231150.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996. doi:10.1145/243439.24344.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09)*, volume 5562 of *LNCS*, pages 18–33. Springer, 2009. doi:10.1007/978-3-642-02674-4_3.

[VAB+07] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. UniTI: A Unified Transformation Infrastructure. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *LNCS*, pages 31–45. Springer, 2007. doi:10.1007/978-3-540-75209-7_3.

[VP04] Dániel Varró and András Pataricza. Generic and Meta-Transformations for Model Transformation Engineering. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *Proceedings of the 7th International Conference on the Unified Modelling Language (UML'04)*, volume 3273 of *LNCS*, pages 290–304. Springer, 2004. `doi:10.1007/978-3-540-30187-5_21`.

[Wag11] Dennis Wagelaar. A Revised Semantics for Rule Inheritance and Module Superimposition in ATL. In *Proceedings of the 3rd International Workshop on Model Transformation with ATL (MtATL'11)*, 2011.

[WKK+10] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In Laurence Tratt and Martin Gogolla, editors, *Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations (ICMT'10)*, volume 6142 of *LNCS*, pages 260–275. Springer, 2010. `doi:10.1007/978-3-642-13688-7_18`.

[WKK+11] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitris Kolovos, Richard Paige, Marius Lauder, Andy Schürr, and Dennis Wagelaar. A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In Jordi Cabot and Eelco Visser, editors, *Proceedings of the 4th International Conference on Theory and Practice of Model Transformations (ICMT'11)*, volume 6707 of *LNCS*, pages 31–46. Springer, 2011. `doi:10.1007/978-3-642-21732-6_3`.

[WKR+11] Manuel Wimmer, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Jesús Sánchez Cuadrado, Esther Guerra, and Juan De Lara. Reusing Model Transformations across Heterogeneous Metamodels. In *Proceedings of the 5th International Workshop on Multi-Paradigm Modeling @ MoDELS'11 (MPM'11)*, 2011.

[WVDSD10] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *SoSyM Journal*, 9:285–309, 2010. `doi:10.1007/s10270-009-0134-3`.

[YK06] Guizhen Yang and Michael Kifer. Inheritance in Rule-Based Frame Systems: Semantics and Inference. In Stefano Spaccapietra, editor, *Journal on Data Semantics VII*, volume 4244 of *LNCS*, pages 79–135. Springer, 2006. `doi:10.1007/11890591_4`.

About the authors

M. Wimmer is post-doc researcher at the Business Informatics Group of the Vienna University of Technology. His research interests comprise Web engineering and model engineering; in particular model transformations based on formal methods, generating transformations by-example as well as applying model transformations to deal with model (co-)evolution. Currently, he is on leave working as visiting researcher at the Software Engineering Group of the University of Málaga (Spain). For further information about his research activities, please visit http://www.big.tuwien.ac.at/staff/mwimmer or contact him at wimmer@big.tuwien.ac.at.

G. Kappel is a full professor in the Institute for Software Technology and Interactive Systems at the Vienna University of Technology, heading the Business Informatics Group. Her current research interests include model engineering, Web engineering, as well as process engineering. For further information about her research activities, please visit http://www.big.tuwien.ac.at/staff/gkappel or contact her at gerti@big.tuwien.ac.at.

A. Kusel is a postdoctoral researcher at the Cooperative Information Systems Group at the Johannes Kepler University, Linz. Her current research interests include model engineering; in particular the specification of model transformations. She received a PhD in Computer Science from the Johannes Kepler University in 2011 in the area of model engineering entitled "Reusability in Model Transformations – Resolving Recurring Heterogeneities by Composite Mapping Operators." For further information about her research activities, please visit http://www.tk.jku.at/people or contact her at Angelika.Kusel@jku.at.

W. Retschitzegger is associate professor at the Johannes Kepler University (JKU) Linz, Austria and scientific head of the Department of Cooperative Information Systems (CIS). In 2002, he was appointed a temporary full professorship for business informatics at TU Vienna. From 2003 to 2006 he was chair of the Institute of Bioinformatics at JKU, from 2008 to 2009 he held a guest professorship for workflow management at the University of Vienna. He has published more than 140 papers in international refereed journals and conference proceedings, along with a series of books. His research interests focus on the model-driven and semantic-based engineering of cooperative information systems in domains like road traffic management, workflow management and social media. Contact him at werner@ifs.uni-linz.ac.at, or visit http://www.bioinf.jku.at/people/wr/.

J. Schönböck is postdoctoral researcher at the Business Informatics Group of the Vienna University of Technology and the Cooperative Information Systems Group at the Johannes Kepler University Linz. He received a PhD in Computer Science from the Vienna University of Technology in 2012 for his thesis "Testing and Debugging of Model Transformations". His research comprise model transformations, model transformation testing and debugging, model (co-)evolution. For further information about his research activities, please visit http://www.bioinf.jku.at/people/schoenboeck/ or contact him at schoenboeck@big.tuwien.ac.at.

W. Schwinger is associate professor at the Department of Cooperative Information Systems (CIS) at the Johannes Kepler University (JKU) Linz, Austria and jointly is acting as scientific head of department. Prior to that, he was working as a senior researcher and project manager of strategic research projects at the Software Competence Center Hagenberg, Austria. He was involved in several national and international projects in the areas of context and situation aware systems, model engineering and web engineering resulting in more than 90 publications in international refereed journals and conference proceedings. Contact him at wieland@schwinger.at, or visit http://www.tk.uni-linz.ac.at/people/.

D. Kolovos Dimitris Kolovos is a Lecturer in Enterprise Systems at the Department of Computer Science of the University of York, United Kingdom, and the project leader of the Epsilon project (http://www.eclipse.org/epsilon). He can be reached at dimitris.kolovos@york.ac.uk. See also http://www-users.cs.york.ac.uk/?dkolovos.

R. Paige is Professor of Enterprise Systems at the University of York, where he leads research on modelling, domain-specific languages, agile development and software engineering. He is director of the Engineering Doctorate Centre in Large-Scale Complex IT Systems. He can be reached at richard.paige@york.ac.uk, followed on Twitter at @richpaige, and on the web at http://www.cs.york.ac.uk/~paige.

M. Lauder is a Ph.D. student at the Graduate School of Computational Engineering and the Real-Time Systems Lab (ES) at the Technische Universität Darmstadt, Germany. His interests include bidirectionality of graph transformations, triple graph grammars, and application scenarios concerning incremental and bidirectional model-to-model synchronization (round trip engineering, DSL development and evolution, tool integration). He received his master's degree in Computer Science from the Technische Universität Darmstadt in 2008 and is currently one of the core developers of the meta-CASE tool eMoflon (`www.moflon.org`). Contact him at `marius.lauder@es.tu-darmstadt.de`, or visit `http://www.es.tu-darmstadt.de/mitarbeiter/marius-lauder/`.

A. Schürr holds the Real-Time System chair of the Electrical Engineering and Information Technology Department of the Technische Universität Darmstadt. His main research interests are related to model-based development of embedded systems with a special emphasis on automotive software development and automation engineering. Besides others, his focus is on bidirectional graph transformations and model-based testing of software product lines. His research group develops the meta modeling tool eMOFLON which offers integrated support for Eclipse-based visual metamodeling and graph-transformation-based model transformation techniques. A. Schürr is a member of the Steering Committees of the international ECMFA, ICGT, and MODELS conference series as well as a co-founder of the German Computer Society Joint Interest Group on Modeling. Contact him at `andy.schuerr@es.tu-darmstadt.de`, or visit `http://www.es.tu-darmstadt.de/mitarbeiter/andy-schuerr`.

D. Wagelaar is a post-doctoral researcher at the Software Languages Lab of the Vrije Universiteit Brussel in Belgium. He received a Ph.D. in Science at the Vrije Universiteit Brussel for his dissertation "Platform Ontologies for the Model Driven Architecture" and holds a MSc. degree in Computer Science from the University of Twente (The Netherlands). His research interests are model-driven engineering, model transformation, software product lines, and using knowledge-based techniques in the field of software engineering. His research expertise is on Platform Variability and platform dependency management. Contact him at `dennis.wagelaar@vub.ac.be`, or visit `http://soft.vub.ac.be/soft/members/denniswagelaar/start`.

Acknowledgments This work has been partially funded by the FWF under grant P21374-N13 and J3159-N23.