A Comparison of Rule Inheritance in Model-to-Model Transformation Languages*

M. Wimmer[1], G. Kappel[1], A. Kusel[2],
W. Retschitzegger[2], J. Schönböck[1], W. Schwinger[2],
D. Kolovos[3], R. Paige[3], M. Lauder[4], A. Schürr[4], and D. Wagelaar[5]†

[1] Vienna University of Technology, Austria
[2] Johannes Kepler University Linz, Austria
[3] University of York, United Kingdom
[4] Darmstadt University of Technology, Germany
[5] AtlanMod (INRIA & École des Mines de Nantes), France

Abstract. Although model transformations presumably play a major role in Model-Driven Engineering, reuse mechanisms such as inheritance have received little attention so far. In this paper, we propose a comparison framework for rule inheritance in declarative model-to-model transformation languages, and provide an in-depth evaluation of three prominent representatives thereof, namely ATL, ETL (declarative subsets thereof), and TGGs. 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 constraints, only.

Key words: Rule Inheritance, Model Transformations, 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" paradigm to the "everything is a model" paradigm [5]. In this context, model transformations are crucial for the success of MDE, being comparable in role and importance of compilers for high-level programming languages. Support for large transformation scenarios is still in its infancy, since reuse mechanisms in model transformations such as inheritance have received little attention so far [10], although the concept of inheritance plays a major role in metamodels (MMs) (as revealed, e.g., by the evolution of the UML standard [13]). As inheritance is

* This work has been funded by the FWF under grant P21374-N13.
† The author's work is funded by a postdoctoral research grant provided by the Institute for the Promotion of Innovation by Science and Technology in Flanders.

employed in MMs to reuse feature definitions from previously defined classes, inheritance between transformation rules is indispensable in order to avoid code duplication and consequently maintenance problems. Although this need has already been recognized by developers of several transformation languages, the design rationales underlying individual realizations are unclear. This makes it more difficult to understand and explain how these constructs are to be used.

Therefore, we propose a comparison framework for rule inheritance in declarative model-to-model transformation languages that makes explicit the hidden design rationales. The proposed framework categorizes the comparison criteria along three different dimensions analogous to the three primary building blocks of programming languages [2]. The first two dimensions comprise *static criteria*: (i) the *syntax* a transformation language defines with respect to inheritance and (ii) *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 (ATL [9], ETL [11], TGGs (MOFLON) [10]) are compared. The results show that the inheritance semantics of these languages differ, which has profound consequences for the design of transformation rules.

Outline. Section 2 provides the rationale of this work, and Section 3 presents the comparison framework with its three dimensions. In Section 4, we compare the inheritance mechanisms of ATL, ETL and TGGs and present lessons learned. Finally, Section 5 gives an overview of related work, and Section 6 concludes.

2 Motivation

When developing a framework for comparing rule inheritance in transformation languages, it is natural to look at the well-known model transformation pattern (cf. Fig. 1) and examine where the introduction of inheritance plays a role. Obviously, a transformation language must define syntactic concepts (cf. question 1 in Fig. 1), which leads to the first dimension of our comparison framework, namely the *syntax*. In this respect, 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 a transformation behavior, that is purely inherited?

In addition to the 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 can be interpreted in a meaningful way?
- *When is a set of inheriting rules defined unambiguously?* Are there sets of rule definitions that do not allow deciding for a single rule?

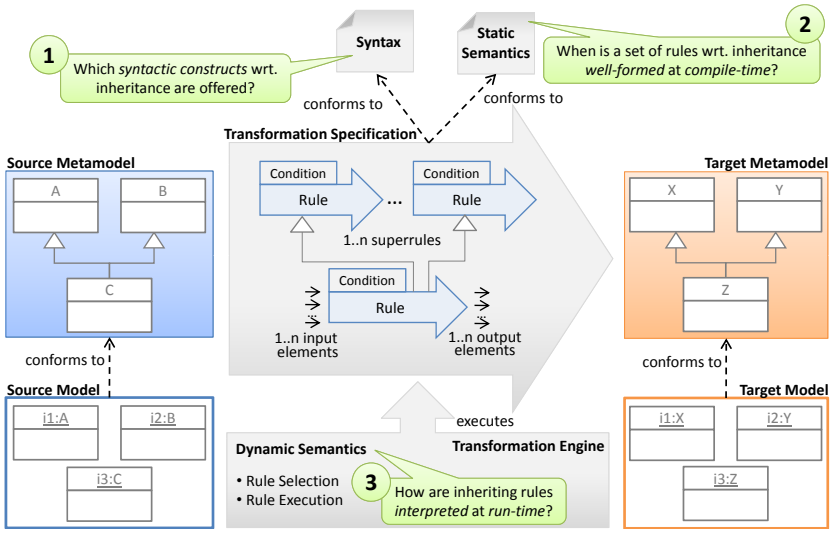

Fig. 1. Model-to-Model Transformation Pattern

A transformation specification is usually compiled into executable code, which is interpreted by a transformation engine that takes a source model and tries to select and execute rules in order to generate a target model. 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 subtype also affected by this rule?
- *How are inheriting rules executed?* From the top down or from the bottom up of a rule inheritance hierarchy?

3 Comparison Framework

This section presents our framework for comparing inheritance support in declarative transformation languages which are used to describe transformations between object-oriented MMs, conforming to, e.g., Ecore or MOF2. Please note, that although metamodeling languages such as MOF2 support refinements between associations (e.g., subsets, redefines), these are out of scope of this paper. As shown in Fig. 2, the criteria can be divided into the three dimensions of (i) *syntax*, (ii) *static semantics*, and (iii) *dynamic semantics*. These dimensions and the corresponding criteria are described in the following.

3.1 Syntax

This subsection provides criteria for comparing transformation languages in terms of syntactic concepts that they support. We consider both, general criteria

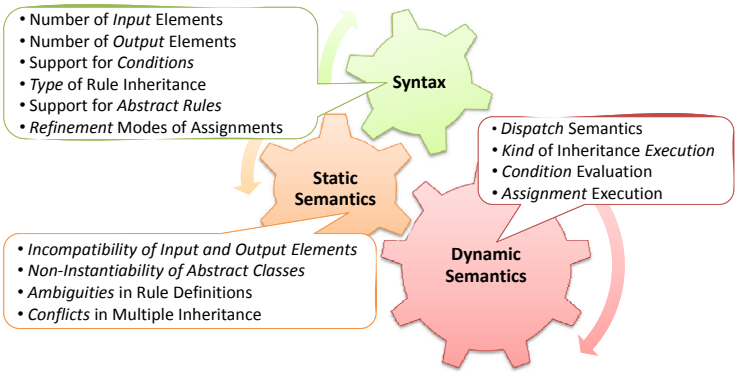

Fig. 2. Overview on the Comparison Framework

(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).

To identify the criteria for comparison, we analyzed (i) the features of transformation languages and (ii) the classification of model transformation approaches presented in [7]. The identified features are expressed in a MM (shown in Fig. 3) illustrating the core concepts of transformation languages. A `Transformation` typically consists of several `TransformationRules`, including an `InPattern`, referring to `InputElements` of the source MM, and an `OutPattern`, referring to `OutputElements` of the target MM. 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 first 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). Finally, they provide the possibility of setting the values for target features by means of `Assignments`.

In the context of inheritance-related aspects, three criteria are relevant. First, a `TransformationRule` may inherit from 1 or 1..n 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 can be reused in subrules. Finally, one can distinguish between different refinement modes by which

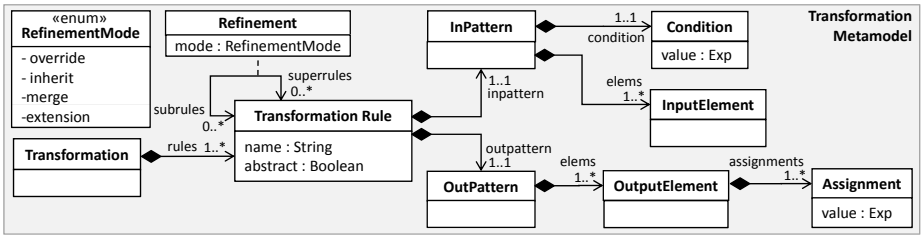

Fig. 3. Inheritance-Related Concepts of Transformation Languages

inherited parts are incorporated into inheriting rules (modeled by the 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 overridden assignments are executed, and then the overriding assignment can alter the resulting intermediate result (such as by initializing some state by a supercall and then altering this intermediate result accordingly). Third, *merge* means that again both assignments are executed, but first the assignment of the subrule and then the overridden assignments are executed. Finally, the refinement mode *extension* induces that inherited assignments may not be changed at all. For consistency reasons, all assignments in a rule should follow the same refinement mode (cf. class `Refinement`).

3.2 Static Semantics

In the previous subsection, we identified criteria targeting the comparison of syntactic concepts. Here we elaborate on criteria relevant for checking the static semantics of 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) ambiguities in rule definitions, and (iv) conflicts in multiple inheritance.

Incompatibility of Input and Output Elements. In the context of transformation rules, both feature assignments and conditions should be inheritable by subrules. Thus, it has to be ensured that the *types* of the input and output elements of subrules have at least the features of the types of the elements of the superrule. Thus, 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 therefore have the same structure as the inheritance hierarchy of the MMs. This means that co-variance for input and output elements is demanded, conforming to the principles of *specialization inheritance* in object-oriented programming. Please note that 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, i.e., *specification inheritance* [12]. Additionally, the *number* of input and output elements should be extensible. Therefore, four cases of potential variations of input elements in type and number can be distinguished:

- **Same number, different types (a).** As an example, Fig. 4(a) shows two rules, `A2X` and `B2Y`, that are bound to the source base classes `A` and `B` and to the target base classes `X` and `Y`, where both rules simply copy the contained features. Since class `C` inherits from both classes `A` and `B`, the rule `C2Z` inherits from the rules `A2X` and `B2Y`. Thus, the feature assignments of the superrules are reused (cf. grey assignments in Fig. 4(a)).
- **Same number, equal types (b).** This case (cf. Fig. 4(b)) may be counter-intuitive, since inheritance is usually used to specialize some core behavior for subsets of instances, and subtypes are typically used to build subsets. In

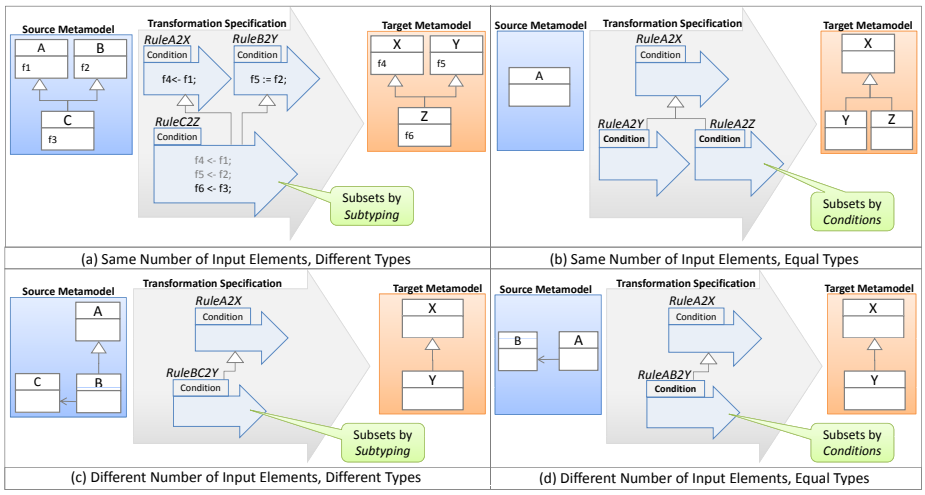

Fig. 4. Rule Compatibility

this case – at first sight – no subsets (according to *specialization inheritance*) are built, and it is unclear which rule should be executed for a combination of instances. Therefore, in such a configuration the subsets needed must be built by applying corresponding disjoint conditions to the subrules.

- **Different number, different types (c).** Here, the subsets needed are built through the specialization of at least one input element (cf. Fig. 4(c)).
- **Different number, equal types (d).** In this case, the same problem as in case (b) arises, where the subsets must be realized by means of conditions which may require certain relationships between the matched input elements (cf. Fig. 4(d)).

One interesting question 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, we discuss this further in Section 3.3.

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.

Ambiguities in Rule Definitions. An ambiguity between inheriting transformation rules may arise if a rule requires multiple input elements, and if 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. [1, 6]), since choosing a method requires the run-time type not of a *single* input element, but of a *set* of input elements.

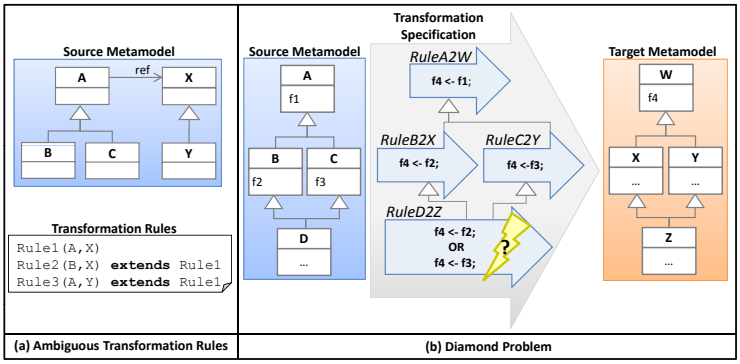

Fig. 5. Examples of Static Constraints: (a) Rule Ambiguity and (b) Diamond Problem

Thus, the method whose run-time types most closely match the statically specified types should be dispatched at run-time. A simple example of such a problem is depicted in Fig. 5(a). Three transformation rules are specified all of them take two input elements of different MM types. Now, suppose that a pair of objects `(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 can be determined, since `Rule2` and `Rule3` are equally good matches. Thus, the set of defined transformation rules is ambiguous.

Conflicts in Multiple Inheritance. The *diamond problem* [16], also referred to as *fork-join* inheritance [15], arises, when contradicting assignments are inherited from different inheritance paths. Consider, for instance the common superrule `A2W` in Fig. 5(b), which contains an assignment for copying a feature value. 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, unless assistance is given by the transformation designer.

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 is 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 [7], potential strategies and scheduling variations of rules were discussed, but without any focus on inheritance. Thus, the literature does not indicate, whether *type substitutability* should be considered. This principle 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 [12]. Type substitutability for transformation rules would, thus, mean that if a rule can be applied to all instances of class T, then

this rule can 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.

Another interesting point in the context of dispatching is, how the evaluation of the condition is incorporated. Thereby, two main strategies can be followed. 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. 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.

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 [16]: (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) the *direction of the message lookup*, i.e., whether a method lookup starts in the subclass (*descendant-driven*) or in the superclass (*parent-driven*).

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 MMs, and source models, can be downloaded from our project homepage[6].

Comparison Setup. For the comparison we considered common model-to-model transformation languages which offer dedicated inheritance support and allow relationships between source and target models to be specified in a declarative way. We examined the declarative subsets of the hybrid transformation languages ATL (version 3.1.0) and ETL (version 0.9.0). 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 [10]. In order to compare the bidirectional TGG-based model transformation approach with the unidirectional languages ATL and ETL, we considered only the unidirectional forward

[6] http://www.modeltransformation.net

translation. Although the QVT standard specifies the declarative transformation language QVT Relations, it is not included in this survey, since QVT Relations support only redefinition of whole rules (i.e., it does not allow reuse of original rule definitions) and not inheritance between rules, as is the focus of our framework. Actual mapping refinement is only mentioned in the QVT Core part, which leaves the transfer to QVT Relations open. Fig. 6 shows an example of the differences between the languages when transforming UML Statemachines into Petri Nets. The rule `State2Place` transforms `State` instances that are not of the kind `initial` into corresponding `Place` instances, while inheriting from the rule `ModelElem2Element`, which specifies the `name` assignment.

4.1 Comparison of Syntax

When comparing the supported language features (cf. Table 1), differences in the *number of allowed input elements* can be detected. Whereas ATL (multiple elements in `from` pattern) and TGGs (source object graph) allow several input elements to be bound to a rule, this is not possible in ETL (cf. single variable after `transform` keyword in Fig. 6). However, all of the languages evaluated support multiple *output elements* (multiple elements in `to` pattern in ATL and ETL, target object graph in TGGs). Finally, all transformation languages allow for the specification of *conditions* (OCL expressions in ATL and TGGs or a `guard` in ETL). ETL and TGGs support multiple inheritance, whereas ATL is restricted to single inheritance (keyword `extends` in ATL and ETL, inheritance arrow in type level of TGGs). All languages provide means to define abstract rules (keyword `abstract` in ATL, annotation `@abstract` in ETL, property `abstract` in TGGs). Finally, concerning potential *refinement modes of assignments*, none of the approaches evaluated provide specific keywords for explicitly choosing the semantics to be applied. Instead, ATL and ETL implicitly assume *override* semantics, and TGGs support the refinement mode *extension* since only new assignments may be added, but existing ones must not be modified.

In summary, all of the approaches evaluated support similar syntactic concepts in terms of inheritance. The main differences lie in the type of inheritance supported and the implicitly assumed refinement mode of assignments.

Table 1. Comparison of Syntax

Rule Part	Permitted Parameter Values	ATL	ETL	TGGs
Input Elements	1 | 1…n	1..n	1	1..n
Output Elements	1 | 1…n	1..n	1..n	1..n
Condition	Yes | No	Yes	Yes	Yes
Type of Rule Inheritance	Single | Multiple	Single	Multiple	Multiple
Abstract Rules	Yes | No	Yes	Yes	Yes
Refinement Modes of Assignments	Override | Inherit | Merge | Extension	Override (implicit)	Override (implicit)	Extension

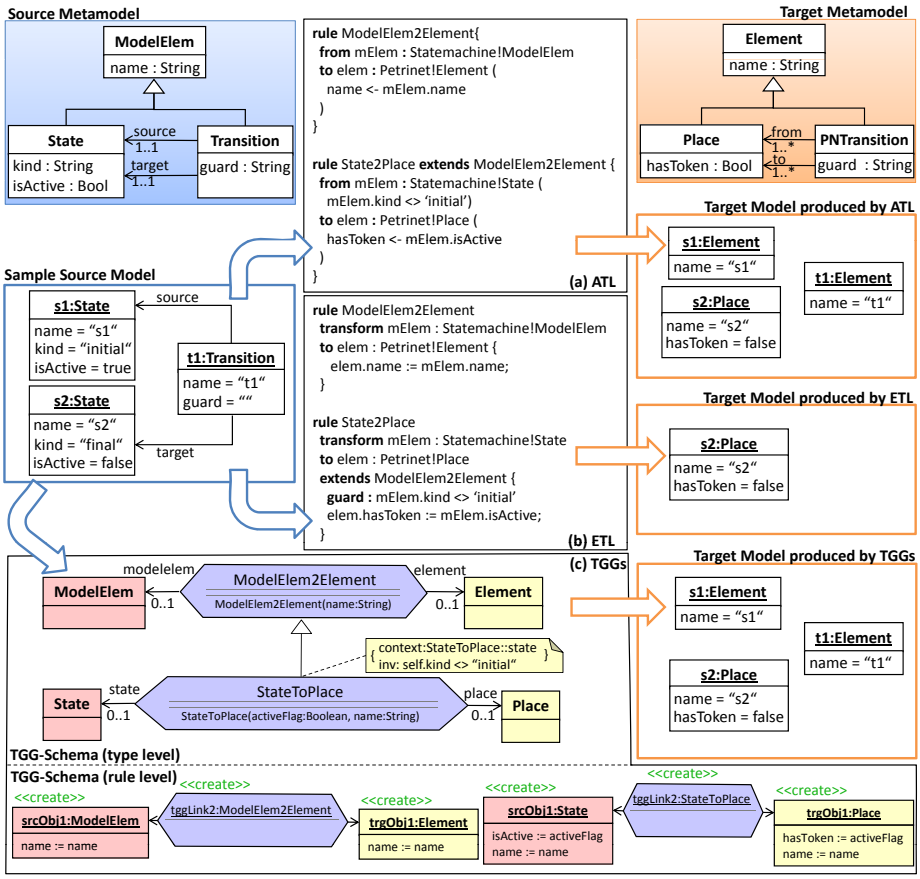

Fig. 6. Transformation example in ATL, ETL and TGGs

4.2 Comparison of Static Semantics

This part of the comparison evaluates in how far the static semantics of inheritance is checked in each transformation language (cf. Table 2). Concerning *input and output elements*, in ATL a violation of co-variance is detected at run-time, since missing features result in a "feature not found" exception. In ETL no error is reported, which leaves the detection of the resulting erroneous instances to the transformation designer or another model management operation executed after the transformation. In TGGs this results in a compile-time error in the upcoming implementation, since the main principle is that applying the subrule should guarantee the existence of the subgraph created by the superrule. Concerning the number of input elements, in ATL a run-time error also occurs, if the number is changed in any way (including name changes). Thus, ATL requires that the number of input elements is not increased. ATL does not raise any exception if the number of output elements is restricted, since they are produced even if they are not respecified. In ETL, the restriction of the number of input

Table 2. Comparison of Static Semantics with respect to Inheritance

Verification Target	Fault	Permitted Parameter Values	ATL	ETL	TGGs
Input Elements	Non-co-variant Type Change	[Compile-Time|Run-Time|No] Error	Run-Time Error	No Error (erroneous instances result)	Compile-Time Error
	Restriction in Number	[Compile-Time|Run-Time|No] Error	Run-Time Error (also with extension)	n.a. (cf. syntax)	Compile-Time Error
Output Elements	Non-co-variant Type Change	[Compile-Time|Run-Time|No] Error	Run-Time Error	No Error (erroneous instances result)	Compile-Time Error
	Restriction in Number	[Compile-Time|Run-Time|No] Error	n.a. (output elements are still produced even if not specified again)	Run-Time Error	Compile-Time Error (except of output to input modification)
Abstract Target Classes	Concrete Rules for Abstract Target Classes	[Compile-Time|Run-Time|No] Error	Run-Time Error	Run-Time Error	Run-Time Error (application fails)
Rule Ambiguity		[Compile-Time|Run-Time|No] Error	No Error (first matching rule in file wins)	n.a. (cf. syntax)	Run-Time Error
Diamond Problem		[Compile-Time|Run-Time|No] Error	n.a. (cf. syntax)	Compile-Time Error	Compile-Time Error

elements is not applicable, since ETL restricts the number of input elements to exactly one anyway. In ETL a run-time error ("index out of bound" exception) is raised if the number of output elements is restricted. In TGGs, 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.

None of the languages evaluated detect *concrete rules referencing abstract classes* at compile-time, but run-time errors are thrown. ATL does not throw exceptions for *ambiguous rule definitions* – neither at compile-time nor at run-time. Instead, the first matching rule defined in the file is executed. In ETL, the problem of ambiguous rule definitions cannot arise, since multiple input elements are not supported. In TGGs, a run-time error is thrown. It must be noted that, in the area of multi-methods, there are approaches for explicit disambiguation (e.g., [3] proposes a minimal set of method redefinitions necessary for disambiguation) which could be reused in transformation languages. The *diamond problem in multiple inheritance* does not apply to ATL, since multiple inheritance is not supported. Although the diamond problem is detected in ETL at compile-time, it is checked on a coarse-grained level, i.e., diamonds that do not include ambiguous assignments also cause errors. In TGGs, this problem is checked statically. Analogously to the explicit disambiguation of ambiguous rule definitions, the transformation designer could be supported by proposals which assignments must be overridden in rules in order to achieve unambiguous assignment definitions.

In summary, static inheritance checks are poorly supported by ATL and ETL. In ATL, none of the static semantics are checked statically. The same is true for ETL with the exception of the diamond problem. In contrast, the TGG-related publication lists quite a number of static checks that will be considered in the upcoming implementation of rule inheritance.

4.3 Comparison of Dynamic Semantics

In order to compare the dynamic semantics, the *dispatch semantics* and the *execution semantics* are investigated (cf. Table 3). Considering the *dispatch semantics*, one can see that the output models produced by ATL and TGGs (Fig. 6(a) and (c)) include only one `Place` instance, since only the `State s2` fulfills the specified condition in the subrule. As ATL and TGGs support *type substitutability* and *rule applicability* semantics for conditions, instance `s1` is matched by the more general superrule `ModelElem2Element`, and therefore creates the target `Element s1`. Due to *type substitutability*, the indirect instance `t1` is matched by the superrule, and therefore the target `Element t1` is created. In contrast, ETL does not support *type substitutability* by default. Thus, although the specifications in ETL and ATL are syntactically very similar, the target models produced differ. ETL's target model contains only a `Place s2` produced by the rule `State2Place`. The dispatch semantics may be modified by annotating rules with `@greedy` in ETL. This means that such rules also match indirect instances, but the interpretation is different than in ATL and TGGs, since the superrule still regards all instances irrespective of whether the instances have already been matched by subrules or not. Adding the `@greedy` annotation to the rule `ModelElem2Element` in our example would therefore create four instances in total: three `Elements s1, s2, t1` produced by the superrule `ModelElem2Element`, and one `Place s2` produced by the subrule `State2Place`. 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, disregarding specialized subrules. Thus, the condition semantics is evaluated as not applicable in ETL.

Regarding the inheritance support in the engine, in ATL inherited rules are flattened during compilation and can thus use optimization strategies, i.e., the ATL compiler inlines the assignments of a superrule. In contrast, ETL supports inheritance in the execution engine, which reduces the amount of code generated.

Table 3. Comparison of Dynamic Semantics of Inheritance

Criterion		Subcriterion	Permitted Parameter Values	ATL	ETL	TGGs
Dispatch semantics		Type Substitutability	Yes | No	Yes	User-Definable	Yes
		Condition Semantics	Filter | Rule Applicability	Rule Applicability	n.a.	Rule Applicability
Execution Semantics	Inheritance Support	-	Flattened | Direct engine support	Flattened	Direct engine support	n.a. (since flattened in patterns already)
	Condition	Completion of lookup	Asymmetric | Composing	Composing	Composing	Composing (by copy)
		Direction of lookup	Parent-driven | Descendent-driven	Parent-driven	Descendent-driven	n.a.
	Assignments	Completion of lookup	Asymmetric | Composing	Optimized Composing	Composing	Composing (by copy)
		Direction of lookup	Parent-driven | Descendent-driven	Descendant-driven	Parent-driven	n.a.

In TGGs, this criterion is not applicable, since an inheriting TGG rule contains a copy of the superrules, which causes code duplication. Concerning the evaluation of the conditions, all transformation languages we compared exhibit a *composing* completion of the lookup, i.e., an instance processed by a subrule must fulfill all the specified conditions up the inheritance hierarchy (i.e., *and* conjunction). The actual evaluation is parent-driven in ATL and descendent-driven in ETL. In TGGs, this criterion is not applicable, since a subrule lists all its inherited conditions. All approaches execute *all* assignments along the inheritance hierarchy (i.e., *composing* completion of the lookup). Finally, the direction of lookup in assignments occurs descendent-driven in ATL, whereas parent-driven in ETL. Thus, in ATL (i) the assignments of the superrule, which are not overridden, (ii) the overridden assignments, and (iii) new assignments specified in the subrule are executed realizing the optimization strategy. In contrast, in ETL, (i) the assignments of the superrule and (ii) the assignments of the subrule are executed. In TGGs this is again not applicable. More specifically, TGGs enforce composition already in the syntax, which causes code duplication.

In summary, the main difference in terms of dynamic semantics lies in the application of type substitutability, which is user-definable in ETL, but interpreted in a different way than in ATL and TGGs. ETL has the disadvantage that several target instances for a single source instance are created when a superrule is annotated with `@greedy`. Moreover, all of the transformation languages implement a *composing* behavior for conditions and assignments. Thus, the lookup direction does not influence the result of the transformation.

4.4 Lessons Learned

This subsection presents lessons learned from our comparison.

Similar Syntax, Different Semantics. As the example in Fig. 6 reveals, similar syntax (cf. ATL and ETL) 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. 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.

Fixed Dynamic Semantics. As introduced above, different kinds of refinement modes may be desirable. The evaluation of the languages has shown, that each of them assumes a certain refinement mode, but none of them allow the transformation designer to choose between different options. Thus, the languages support only fixed dynamic semantics for rule inheritance. Since different dynamic semantics 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 would enable the transformation designer to express different refinement modes.

5 Related Work

This section considers two 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.

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 [9], and that in ETL in [11], but rather on a syntactical level, while the actual execution semantics are left open. A detailed discussion of semantic constraints that must be considered in TGG rule inheritance can be found in [10]. For graph transformations in general, Bardohl et. al [4] 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 [14] detailed semantics with respect to inheritance is defined only for QVT Operational.

Reuse Facilities in Model Transformations. General work has been done in composing transformations. Wagelaar et. al. [18] proposed a superimposition mechanism of transformations to build the union of all transformation rules. Thereby rules can be added and redefined (i.e., replacing a rule by a new one), whereby it is impossible to refer to the original rule. This is similar to the mechanism in QVT Relations, in which a transformation can extend another transformation and redefine existing rules [14].

Another reuse mechanism is to provide predefined transformations that can be adapted to specific MMs. Varró et al. [17] introduced generic transformations in VIATRA2 which in fact resembles the concept of templates in programming languages. Another approach to generic transformations was proposed in [8], where transformations are designed between generic "concepts models". These transformations can then be bound to concrete MMs, but only if they have the same structure. Finally, Wimmer et. al. [19] presented mapping operators which allow model transformations to be specified by means of reusable components, which is similar to mappings known in data engineering.

In summary, only preliminary approaches for reuse are available, which confront the transformation designer with code duplication and consequently maintenance problems.

6 Conclusion and Future Work

In this paper, we have presented a systematic comparison of inheritance support in the transformation languages ATL, ETL, and TGGs. We (i) identified syn-

tactic concepts required for inheritance, (ii) elaborated on semantic constraints (i.e., 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 have been made explicit. Since we considered only declarative model-to-model transformations, we intend to investigate inheritance support in imperative transformation languages, too, including also the imperative parts of hybrid transformation languages.

References

1. R. Agrawal, L. G. Demichiel, and B. G. Lindsay. Static Type Checking of Multi-Methods. In *Proc. of OOPSLA'91*, pages 113–128, 1991.
2. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques,and Tools*. Addison-Wesley, 1986.
3. E. Amiel and E. Dujardin. Supporting explicit disambiguation of multi-methods. In *Proc. of ECOOP'96*, pages 167–188, 1996.
4. R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In *Proc. of FASE'04*, pages 214–228. 2004.
5. J. Bézivin. On the Unification Power of Models. *SoSyM Journal*, 4(2), 2005.
6. C. Chambers. Object-Oriented Multi-Methods in Cecil. In *Proc. of ECOOP'92*, pages 33–56, 1992.
7. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
8. J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *Proc. of MoDELS'10*, pages 16–30. 2010.
9. F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proc. of the Model Transformations in Practice Workshop*, 2005.
10. F. Klar, A. Königs, and A. Schürr. Model transformation in the large. In *Proc. of ESEC-FSE'07*, pages 285–294, 2007.
11. D. Kolovos, R. Paige, and F. Polack. The epsilon transformation language. In *Proc. of ICMT'08*, pages 46–60, 2008.
12. B. Liskov and J. M. Wing. A new definition of the subtype relation. In *Proc. of ECOOP'93*, pages 118–141, 1993.
13. H. Ma, W. Shao, L. Zhang, Z. Ma, and Y. Jiang. Applying OO metrics to assess UML meta-models. In *Proc. of UML'04*, 2004.
14. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. http://www.omg.org/spec/QVT/1.1/Beta2/PDF/, 2009.
15. M. Sakkinen. Disciplined Inheritance. In *Proc. of ECOOP'89*, pages 39–56, 1989.
16. A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.
17. D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *Proc. of UML'04*, pages 290–304, 2004.
18. D. Wagelaar, R. Van Der Straeten, and D. Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *SoSyM Journal*, 9:285–309, 2010.
19. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In *Proc. of ICMT'10*, pages 260–275, 2010.