A Feature-based Approach for Variability Exploration and Resolution in Model Transformation Migration

Davide Di Ruscio[1], Juergen Etzlstorfer[2], Ludovico Iovino[3], Alfonso Pierantonio[1], and Wieland Schwinger[2]

[1] University of L'Aquila - L'Aquila, Italy
{davide.diruscio,alfonso.pierantonio}@univaq.it
ORCiD: 0000-0002-5077-6793, 0000-0002-5231-3952
[2] Johannes Kepler University Linz - Linz, Austria
{juergen.etzlstorfer,wieland.schwinger}@jku.at
ORCiD: 0000-0002-8497-2266, 0000-0002-7895-3165
[3] Gran Sasso Science Institute - L'Aquila, Italy
ludovico.iovino@gssi.it - ORCiD: 0000-0001-6552-2609

Abstract. The key to success with Model-Driven Engineering is the ability to maintain metamodels and their related artifacts consistent over time. Metamodels can evolve under evolutionary pressure that arises when clients and users express the need for enhancements. However, metamodel changes come at the price of compromising metamodel-related artifacts, including model transformations, necessitating their migration to again conform to the evolved metamodel. Restoring conformance of transformations is intrinsically difficult since a multitude of possible migration alternatives exist, which are unfeasible to be inspected manually. In this paper, we present an approach to explore variability in model transformation migration. Employing a feature-based representation of several possible transformation migrations, the approach permits modelers to explore and explicitly discover differences and conflicts among them. Once the desired migration alternatives are selected, the actual migration program is generated and executed by exploiting the EMFMigrate platform.

1 Introduction

As the complexity of software systems escalates, there is an increasing consensus on the need to leverage abstraction. In Model-Driven Engineering [20] (MDE) this is usually accomplished by formalizing domains by means of metamodels that are at the core of this software discipline. As a consequence, complete modeling environments, which consist of a multitude of artifacts including models and model transformations, are formally defined in accordance with their reference metamodels [7]. Similarly to other software artifacts, metamodels can evolve under evolutionary pressure that arises when clients and users express a need for enhancements. Changing a metamodel might break conformance to its dependent artifacts because of the existing dependencies among them [6]: conformance restoring migrations are therefore necessary to re-establish the conformance in the modeling environment. Model transformations are no exception and urge to be *migrated* whenever metamodels they are based on undergo modifications [8].

Analogously to the well-known update view problem in relational databases [1] there are multiple ways of propagating metamodel changes, i.e., there are many alternatives to *migrate* a transformation. The problem is how to choose one i.e., *how is it possible to identify a migration alternative reflecting both the modeler intents and the rationale behind the metamodel refactoring among the viable alternatives?* Existing approaches (e.g., [10,16]) typically start from a formalization of the metamodel changes to automatically derive *a single* migration. However, these techniques offer a prefixed solution only, which must be used in any context and regardless of the reasons behind the occurred metamodel evolution, entailing the drawback that potential solutions which better fit the modeler intents are left unexplored. However, since multiple solutions are possible, each leading to a differently migrated transformation, it is of utmost importance to identify the one that best fits developers' needs. In particular, small changes in a given metamodel typically correspond to a large number of migration alternatives. Unsupported manual inspection and detection of those is prone to errors, because alternatives might overlap each other, hampering a successful transformation migration.

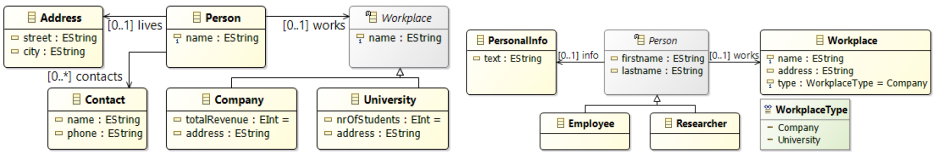
This paper proposes an approach to represent a set of possible model transformation migration alternatives in response to metamodel evolution to support the user in inspection and detection of migration alternatives. As a result, migration solutions can be better compared as differences and potential conflicts between migration alternatives are denoted by variability points without the necessity of manually inspecting each of them. In this context, the user is supported in choosing the desired migration alternative by means of a feature model [2]. EMFMigrate [23] rules are automatically generated and executed with respect to the selected migration alternative, to migrate the initial transformation to recover its conformance with the evolved metamodel.

Outline. Next section presents a motivating scenario, while Sect. 3 introduces a notation for managing variability in an intensional way and its application on an example. A prototypical implementation is presented in Sect. 4 and related work is discussed in Sect. 5. Finally, Sect. 6 draws conclusions and outlines future work.

2 Motivating Scenario

In this section, we present an explanatory metamodel evolution and its effects on a model transformation. Despite its simplicity, it is able to show the large number of migration alternatives and, thus, the multitude of different migrations a user is confronted with.

Figure 1a shows the *Simple Workplace* metamodel acting as the source metamodel of a transformation, comprising metaclasses for the specification of persons and their



(a) Initial version (b) Evolved version

Fig. 1: An explanatory Workplace metamodel evolution

corresponding workplaces. According to the metamodel, a `Person` works optionally in an (abstract) `Workplace`, which can be a `Company` or `University`. The specification of `Person`s can include the corresponding `Address` and `Contact` data. `Company` elements consist of the specification of the corresponding `address`es and `total revenue`s. The definition of `University` includes also information about the number of students.

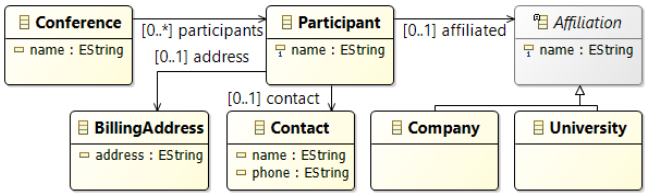


Fig. 2: The conference management metamodel

Figure 2 shows the metamodel of a simple conference management system: a `Conference` can have a set of `participants` that can be affiliated with either a `University` or a `Company`. In order to complete the registration to a conference, each participant has to provide the corresponding organizers with a `BillingAddress` and a `Contact`.

```
1  rule Person2Participant {
2   from s: WORKPLACE!Person
3   to t: CONFERENCE!Participant (
4   name <- s.name,
5   affiliated <- s.works,
6   contact <- s.contacts->first(),
7   address <- s.lives
8   )
9  }
10 rule University2University {
11  from s: WORKPLACE!University
12  to t: CONFERENCE!University (
13  name <- s.name
14  )
15 }
16 rule Company2Company {
17  from s: WORKPLACE!Company
18  to t: CONFERENCE!Company (
19  name <- s.name
20  )
21 }
22 rule Address2Billing {
23  from s: WORKPLACE!Address
24  to t: CONFERENCE!BillingAddress
25  (
26  address <- s.street + ',␣' + s.city
27  )
28 }
29 rule Contact2Contact {
30  from s: WORKPLACE!Contact
31  to t: CONFERENCE!Contact (
32  name <- s.name,
33  phone <- s.phone
34  )
35 }
```

Listing 1.1: Snippet of *SimpleWorkplace2ConferenceManagement*

Listing 1.1 shows an ATL transformation [14] generating models conforming to the metamodel in Fig. 2 out of workplace models conforming to the metamodel in Fig. 1a. Therefore, the rule `Person2Participant` generates a `Participant` instance for each instance of the `Person` metaclass. Analogously, the rules `University2University` and `Company2Company` create instances of the corresponding metaclasses. The rule `Address2Billing` generates a `BillingAddress` instance for each `Address` instance, concatenating the source values `street` and `city` for the target `address` value. The rule `Contact2Contact` generates `Contact` instances.

In order to address unforeseen requirements or to better represent the considered application domain, metamodels can evolve. For instance, the workplace metamodel shown in Fig. 1a is modified to obtain the new version in Fig. 1b by applying the following refactorings:

- *R1. Introduction of subclasses:* the `Employee` and `Researcher` metaclasses are introduced as subtypes of `Person`, which in turn becomes abstract;
- *R2. Split attribute:* the attribute `name` of the metaclass `Person` is split in two attributes with the same type, named `firstname` and `lastname`;
- *R3. Flatten Hierarchy:* the hierarchy between the metaclass `Workplace` and the subclasses `Company` and `University` is reduced to the new version of the `Workplace` metaclass. It contains a new attribute of type `WorkspaceType`, which is a new enumeration representing the pruned subclasses, whose default value is `Company`;
- *R4. and R5. Replace metaclass*: The metaclass `Address` is replaced by the `PersonalInfo` metaclass, and `Contact` is replaced by the `PersonalInfo` metaclass.

Because of these applied metamodel changes, the model transformation in Listing 1.1 has lost its *domain conformance* [13] to the metamodel, and thus, has to be migrated. Migration of model transformations is difficult and can easily give place to inconsistencies and omissions [8]. Moreover, multiple migrations are possible [21], each providing a different solution. Thus, Table 1 shows possible migration alternatives for each applied refactoring. It is worth noting how even simple, *non-breaking changes* [10] induce multiple options of migration according to developer's expertise and goals, which is the case of R1. In particular, if the rule `Person2Participant` (cf. line 1–9 of Listing 1.1) is left unmodified the transformation remains valid, since `Person` instances will be matched by the rule. However, developers might still decide to change the input pattern of the transformation with one of the subclasses, i.e. `Employee` or `Researcher`.

Refactoring R2 involves the split of the attribute `name` (cf. line 4). The pattern `s.name` in the right hand side of the binding can not be longer queried and thus, needs to be adapted. The corresponding migrations shown in Table 1 are not exhaustive since the use of OCL in ATL transformations increases complexity and gives place to many different solutions. However, possible migration alternatives for the right hand side of the binding can be for instance at least the following expressions: (i) `s.firstname`, (ii) `s.lastname` or (iii) `s.firstname + ' ' + s.lastname`.

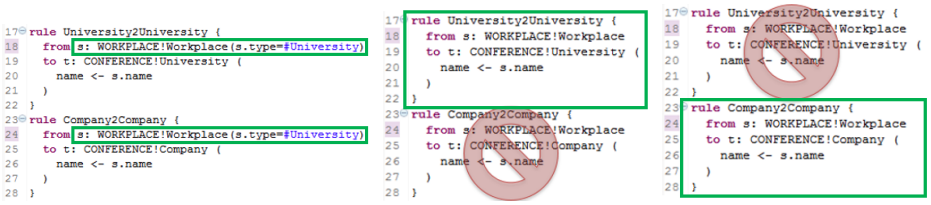
Concerning refactoring R3, the rules in lines 10–21 are no longer valid since the types of the input patterns (i.e., `University` and `Company`) have been removed from the initial version of the source metamodel. One possible migration is to change the input patterns of the affected rules by adding conditions based on the new `type` attribute (cf. Fig. 3a), e.g., `s:WORKPLACE!Workplace(s.type=#University)`. Such a "filter" is

Metamodel change	Possible migration alternatives
R1. Introduce subclasses	*R1a1*. Leave transformation unchanged
	R1a2. Change in-pattern to `Employee`
	R1a3. Change in-pattern to `Researcher`
R2. Split attribute	*R2a1*. Use `firstname`
	R2a2. Use `lastname`
	R2a3. Use concatenation of `firstname` and `lastname`
	R2a4. Delete the affected binding (it is assumed it is not mandatory in the target metamodel)
R3. Flatten hierarchy	*R3a1*. Change input pattern of the affected rule to the remaining class `Workplace` and introduce guards to produce instances of `University` and `Company`
	R3a2. Change input pattern of the affected rule `University2University` to `Workplace` and delete the other rule `Company2Company`
	R3a3. Change input pattern of the rule `Company2Company` to `Workplace` and delete the other rule `University2University`
	R3a4. Delete both rules `Company2Company` and `University2University`
R4. Replace metaclass `Address` with `PersonalInfo`	*R4a1*. Delete rule `Address2Billing`
	R4a2. Change input pattern of the rule `Address2Billing` to the class `PersonalInfo`
	R4a3. Change input pattern of the rule `Address2Billing` to match the class `PersonalInfo`. In addition, add another output pattern to produce also target `Contact` instances
R5. Replace metaclass `Contact` with `PersonalInfo`	*R5a1*. Delete rule `Contact2Contact`
	R5a2. Change input pattern of the rule `Contact2Contact` to be class `PersonalInfo`
	R5a3. Change input pattern of the rule `Contact2Contact` to match the class `PersonalInfo`. In addition, add another output pattern to produce also target `BillingAdress` instances

Table 1: Possible migration alternatives for the motivating example

necessary since in ATL each source model element can match with one rule only [14] and, consequently, the input pattern `Workplace` can not be used in two different rules without any guard. Alternatively, it is possible to drop one of the affected rules and change the input pattern type of the kept rule to `Workplace` (cf. Fig. 3b and Fig. 3c). Another option can be dropping both rules. However, although this would be a syntactically valid option, no instances would be transformed, resulting in a loss of information.

Among the possible ways to resolve refactoring R4, Table 1 shows three alternatives consisting of dropping the rule `Address2Billing` (*R4a1*), and change the type of its input pattern to `PersonalInfo` (*R4a2*). This would be enough to run the transformation



(a) Migration alternative (a) (b) Migration alternative (b) (c) Migration alternative (c)

Fig. 3: Possible migration alternatives related to refactoring R3

without errors. However, an additional output pattern can be added in order to generate also `Contact` instances (***R4a3***). Similarly to R4, Table 1 shows three alternatives for adapting the sample ATL transformation because of refactoring R5.

When migrating model transformations, which have been compromised by metamodel refactoring actions, developers have to combine different migration alternatives, one for each metamodel refactoring, to obtain a *migration solution*. This represents a major difficulty because alternatives must be combined causing a combinatorial explosion of cases: for instance, the 5 refactorings presented above can give place to

$$3 \times 4 \times 4 \times 3 \times 3 = 432$$

migration alternatives. Although this is an over-approximation since conflicts might occur between migration options as discussed later in the paper, it is highly impractical for the modeler to sort out a myriad of individual alternatives. The problem can be even more complex if the affected transformation has several source and target evolving metamodels[4]. In the remainder of the paper we consider the management of one-to-one model transformations with only the source metamodel evolving, while the target metamodel remains unchanged.

3 Proposed approach

In this section, we propose an approach to represent, explore, and select migration alternatives for ATL transformations in response to an evolved *source* metamodel. The approach permits to represent all migration alternatives in a single model with variability. Besides having an *intensional* representation of the solution space, i.e., all valid migrations of the transformation, the approach permits the identification of the differences among the alternatives by means of variation points originated from each metamodel refactoring. Moreover, the approach permits to highlight conflicting alternatives, which will be discussed in more detail later.

The approach is outlined in Fig. 4, where weaving model m_{WMM} represents possible migration alternatives to be applied on the affected transformation T. The weaving model[5] conforms to the the *Variability Weaving Metamodel* explained in detail in Sect. 3.1 and Sect. 3.2. To allow developers exploring the alternatives represented in m_{WMM}, the *WMM2FM* transformation automatically generates a feature model, a common mean to represent variability [2] (Sect. 3.3). This is further used to easily determine a valid combination of migration alternatives (Sect. 3.4) and to select a configuration to generate EMFMigrate migration programs, which can be executed to migrate the affected transformations.

[4] In order to give more evidence of the difficulties related to the extensional treatment of transformation migrations, which might be required because of metamodel evolutions, our online appendix discusses a list of metamodel changes borrowed from existing catalogues, e.g., [5,12]: `http://www.emfmigrate.org/wp-content/uploads/2017/04/appendix.pdf`. Such changes are organized with respect to the impact they might have on existing transformations.

[5] Currently, the weaving model m_{WMM} is manually specified even though an automatic creation is feasible as discussed later in the paper. Such a relevant automation step is an important work that we plan to do in the future.

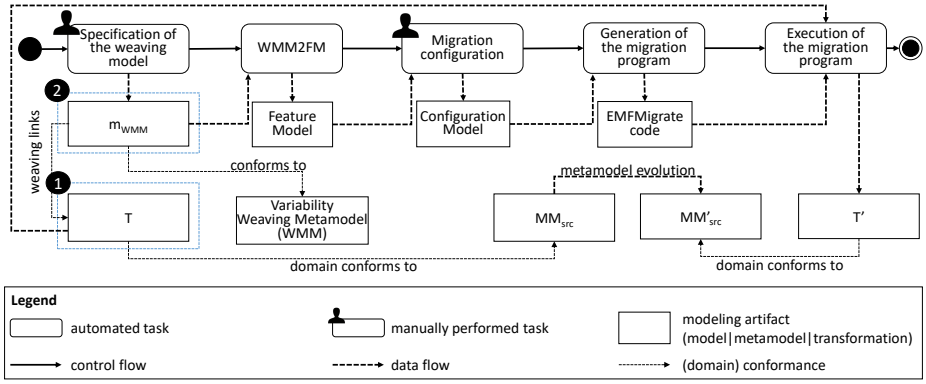


Fig. 4: Overview of the proposed approach

3.1 Variability weaving metamodel for representing different migration solutions

The *Variability Weaving Metamodel (WMM)* has been designed in order to be independent from the model transformation language in use. As a result, it can be used without loss of generality for any rule-based transformation language. To this end, a generalization step has been employed to abstract from language dependent concepts in order to define a simplified transformation language very much aligned with the notations given in [11,24], collecting common concepts of model transformation languages (cf. Fig. 5). According to the simplified transformation metamodel, `Module` holds one or more `Rules` that might have `superrules` which are composed of `InPatterns` and `Outpatterns`. An `InPattern` is further composed of `InputElements` and an optional `Guard`. The `OutPattern` is composed of one or more `OutputElements` which have optional `bindings`. It is worth noting that the actual bindings, guards as well as the input and output patterns are expressed as strings in the current version, but are planned to be replaced by including an OCL metamodel, e.g., as done in [18].

As previously said, the variability weaving metamodel relies on the simplified transformation metamodel to deal with all kind of transformation language specificities. Thus, by applying the approach presented in [3], for each metaclass *MC* in the simplified transformation metamodel, corresponding *AddedMC*, *DeletedMC*, and *ChangedMC* metaclasses, e.g., `AddedRule`, are defined in WMM as shown in Fig. 6. WMM permits to represent `Solutions` that are considered as the counterpart for the applied metamodel changes. As shown in Fig. 6, each `Solution` is composed of `Alternatives`, which are disjunct and represent migrations which have to be performed to co-evolve the affected transformation. Each `Alternative` consists of `DiffElements`. A `DiffElement` can be in turn a `DiffRule` or a `DiffPattern`, i.e., the changes that a rule can undergo or that affect a pattern of a rule, respectively. `AddedRule`, `ChangedRule` and `DeletedRule` are provided for added, updated or deleted rules, while `CopyRule` allows the rule to be copied without actions, which might be a valid choice for some refactorings, e.g., introduction of superclasses. The same concept is replicated for other transformation constructs like patterns that are composed of bindings containing an

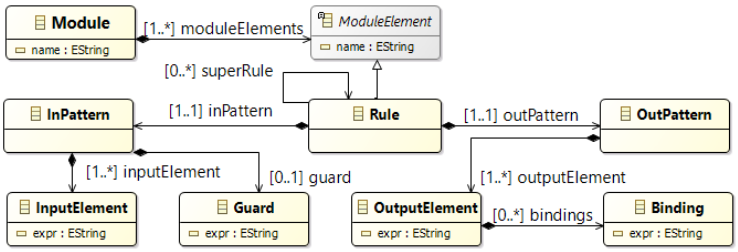


Fig. 5: The Simple Transformation Language (STL) Metamodel

OCL expression, to be held in the attribute `expr`. Finally, input patterns can have guards to match only certain input patterns, and also in this case is done using an expression. The references `applicationElement` from `DiffElement` link to the abstract class `SimpleTransformationElement`, which can be specialized for each concept shown in Fig. 5 in order to refer concrete elements of the transformation to be migrated.

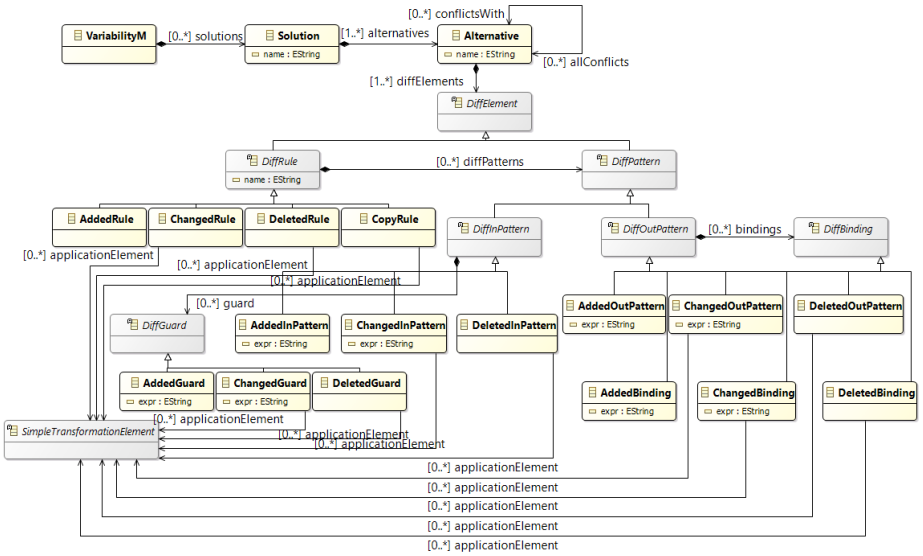


Fig. 6: The Variability Weaving Metamodel (WMM)

An important characteristics of WMM is the specification of *conflicts*, i.e., disjunct choices that must not occur in the same solution. For instance, a migration solution can not contain choices that contribute to the generation of different rules defined on the same input pattern without any guard, thus causing run-time errors due to multiple rules matching the same model element. As another example, a migration solution must not contain choices that refer to an element which has been deleted by another alternative.

As previously mentioned, WMM can be used for managing model transformations specified in different rule-based transformation languages. Thus, a transformation written in a language, such as ATL or ETL [15], can be mapped into its simplified version

as shown in Fig. 7. This enables a simpler specification of the linkage between the (abstract) transformation and the migration variants. Once the modeler has selected the desired alternatives (cf. Sect. 4), the transformation can be migrated by projecting the modification from the abstract model to the original transformation. Therefore, the trace information produced when executing the $ATL2STL$ or $ETL2STL$ transformation is used.

3.2 Specification of variability weaving models

Figure 8 presents a variability model conforming to the WMM and related to the running example. The weavings have been labeled with the same numbers as in Fig. 4, so the left panel is the simplified version of the transformation in Listing 1.1, and the right panel is the associated variability model containing all migration alternatives listed in Table 1.

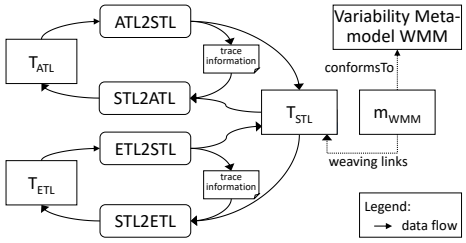


Fig. 7: Generation of simplified transformation models

Please note that the graphical overlay by means of dashed and dotted lines is for presentation purposes only and not visualized in this way in the tool, instead we provide a feature model as graphical decision support which is a common and widely used mean to manage variability in software product lines [2] (cf. Sect. 4). The weaving links have been highlighted as dashed (green) lines, while the link shown as dotted (red) line identifies a conflict in the solutions. The weaving link denoted by the left (green) tooltip maps the `OutPattern` from the simplified transformation model to a `Changed-OutPattern` in the `Alternative R4a1` as part of the `Solution R4`. It comprises a `DeletedBinding` element, which is in turn linked to the affected binding in the simplified transformation model in rule `Person2Participant`. The corresponding action in Table 1 is denoted by *R4a1*.

Furthermore, a conflict between `Alternative R4a2` and `R5a2` has been identified, using the proposed conflict detection algorithm (cf. Sect. 3.4). In fact, having both alternatives in the solution model would give place to an invalid transformation with two rules matching the same input metaclass, which is forbidden since the input pattern has to be unique. Possible conflicts that can occur when migrating transformations are discussed in Sect. 3.4.

In the following, we show how a feature-based representation of the migration alternatives, as those represented in Fig. 8, can be automatically generated and how it is beneficial in the variability management.

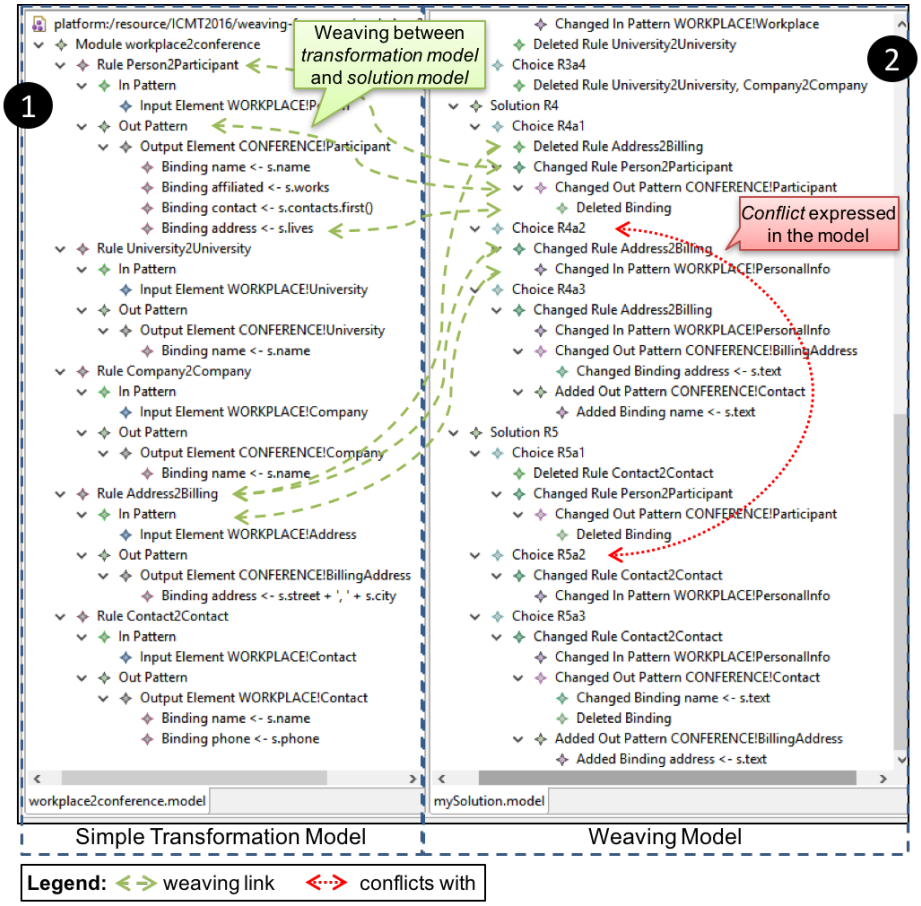


Fig. 8: Excerpt of the variability model for the transformation example

3.3 Feature model as representation for managing variability

Fig. 9 shows a generated feature model[6] specifying the alternatives for migrating the transformation to cope with the source metamodel evolution. In the context of this paper, the feature model is a compact representation of all migration alternatives and possible conflicts between them, supporting the exploration of the desired migration alternatives.

As shown in Table 1, we consider five refactorings R1–R5 that entail five solutions, each of those has possible alternatives for migration that satisfy the domain conformance relationship. Also, the automatically identified conflict between the alternatives is reflected by means of a constraint in the feature model. Thus, the constraint $R4a2 \Rightarrow \neg R5a2$ defines that if alternative *R4a2* is chosen, the alternative *R5a2* is no longer valid and can not be chosen by the modeler.

The feature model is automatically generated starting from the weaving model by employing the transformation *WMM2FM* shown in Listing 1.2. The transformation

[6] In this work we employed the Eclipse FeatureIDE plugin [22] for specifying feature models.

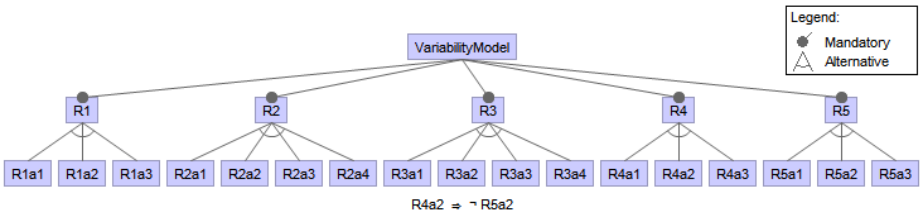


Fig. 9: Feature model for the transformation example

employing the Epsilon Generation Language (EGL) [19]. The model-to-code transformation targets XML as technical space and starts storing in the `variabilityM` variable the instance of the variability model and in `allSolutions` the solutions specified (cf. lines 1–2). The transformation iterates the solutions and for each alternative a feature labeled with the alternative's name is created (cf. lines 7–13). Then the script generates constraints stemming from the defined conflicts (cf. lines 17–30). Those constraints correspond with the following expression $a \Rightarrow \neg a_1 \wedge \neg a_2 \wedge \ldots \wedge \neg a_n$ defining that if alternative a is chosen, the alternatives a_1, a_2, ... a_n are no longer valid.

```
1  [% var variabilityM := VariabilityM.allInstances().at(0);
2     var allSolutions := variabilityM.solutions; %]
3  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
4  <featureModel chosenLayoutAlgorithm="1">
5    <struct>
6     <and abstract="false" mandatory="true" name="VariabilityModel">
7  [% for (s in allSolutions ) { %]
8       <alt mandatory="true" name="[%=s.name%]">
9         [% for (a in s.alternatives) { %]
10          <feature mandatory="true" name="[%=a.name%]" />
11        [% } %]
12      </alt>
13 [%  }   %]
14    </and>
15   </struct>
16   <constraints>
17 [% for (s in allSolutions) {
18           for (a in s.alternatives) {
19             if (a.allConflicts.size() > 0) { %]
20               <rule>
21                <imp>
22                  <var>[%=a.name%]</var>
23                  <conj>
24                    [% for (conflict in a.allConflicts) { %]
25                      <not><var>[%=conflict.name%]</var></not>
26                    [% } %]
27                  </conj>
28                </imp>
29               </rule>
30 [%  } } }  %]
31   </constraints>
32   ...
33 </featureModel>
```

Listing 1.2: Fragment of the *WMM2FM* transformation

3.4 Automated identification of conflicting alternatives

In order to further automate the presented approach, we propose an algorithm that automatically identifies conflicts between alternative solutions and sets the corresponding conflict relations in the variability model (cf. Fig. 8). A conflict can be defined as a situation where two different alternatives can not co-exist, since they could raise errors at run-time. The algorithm shown in Algorithm 1 is able to identify the following conflicts, however, the algorithm can be extended or changed if needed, to account for language-specific conflict detections[7]:

c1) Rules are missing guards avoiding multiple matches for a same model element;
c2) A rule that has been deleted in one alternative is used in another alternative;
c3) A binding that has been deleted in one alternative is used in another alternative;

Algorithm 1 Detection of Conflicting Alternatives

```
1: for all solutions s in model do
2:   for all alternatives a in s do
3:     for all rules r in a do
4:       doubleMatch ← HASDOUBLEMATCH(m, r)
5:       if doubleMatch <> null then
6:         ADDCONFLICT(a, doubleMatch)
7:       end if
8:       deletedRule ← ISRULEDELETED(r)
9:       if deletedRule <> null then
10:        ADDCONFLICT(a, deletedRule)
11:      end if
12:      for all bindings b in rule r do
13:        deletedBinding ← ISBINDINGDELETED(b, r)
14:        if deletedBinding <> null then
15:          ADDCONFLICT(a, deletedBinding)
16:        end if
17:      end for
18:    end for
19:  end for
20: end for
```

In particular, all the solutions in the input solution model are queried, and all their alternatives and rules are iterated to check if two rules can potentially match a same model element. Since in ATL this must not occur, it has to be ensured that rules violating such a constraint are marked as conflict. In line 4 of Algorithm 1 it is therefore checked if the current metamodel element is matched by any other rule, by calling the auxiliary function HASDOUBLEMATCH. If a double match is detected, then a conflict is added in the model by means of the ADDCONFLICT auxiliary function (cf. line 6). Analogously, in

[7] The auxiliary functions HASDOUBLEMATCH, ISRULEDELETED, ISBINDINGDELETED, and ADDCONFLICT used in Algorithm 1 are reported online: `http://www.emfmigrate.org/wp-content/uploads/2017/04/appendix.pdf`.

case elements are accessed in one alternative, but deleted in another one a conflict has to be declared. To this end, in line 8 and line 13 the auxiliary functions IsRuleDeleted and IsBindingDeleted are executed, respectively and depending on their outcomes the AddConflict function is executed accordingly.

4 Configuration and execution of the feature model

In order to enable the execution of the migration solution obtained by selecting the alternatives in the considered feature model, we exploit the EMFMigrate migration platform[8]. EMFMigrate provides modelers with languages and tools supporting the coupled evolution of any kind of modeling artifacts. An EMFMigrate specification consists of migration rules as shown in Fig. 10. In particular, a migration program, usually specified by the modeler, is able to migrate artifact *A*, conforming to the metamodel *MM*, according to the metamodel differences represented in the model *Delta*, conforming to the difference metamodel proposed in [3] already applied to other co-evolution cases (e.g., [4]).

A migration program consists of a sequence of migration rules mr_i. Each rule is applied on artifact *A* if the corresponding $guard_i$ evaluated on the difference model *Delta* holds. The body of a migration rule consists of a sequence of rewriting rules like the following

$$s[guard] \rightarrow t_1[assign_1]; t_2[assign_2]; \ldots t_n[assign_n]$$

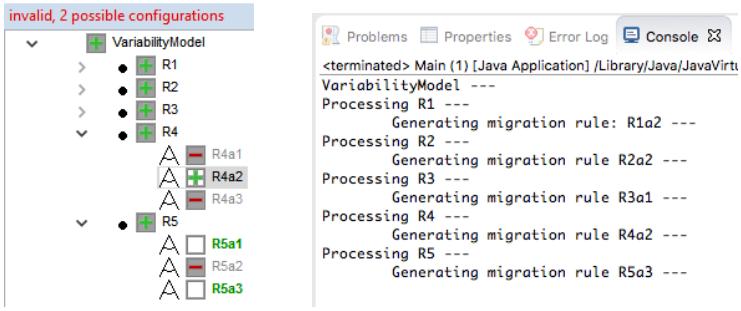
where s, t_1, ..., t_n refer to metaclasses of *MM*, and $guard$ is a boolean expression which has to be *true* in order to rewrite s with t_1, t_2, and t_n. It is possible to specify the values of the target term properties by means of assignment operations (see $assign_i$).

migration $migrationName$
include $\{library^*\}$
migrate $A : MM$ **with** $Delta$ {
 rule mr_1
  [$guard_1$] $\{rewritingRule^*$ }
 rule mr_2
  [$guard_2$] { $rewritingRule^*$ }
...
 rule mr_n
  [$guard_n$] { $rewritingRule^*$ }
}

Fig. 10: EMFMigrate syntax

Figure 11a shows the building of migration solution by selecting migration alternatives represented by means of a feature model as discussed in the previous section. The

[8] A detailed discussion of EMFMigrate is outside the scope of this paper. Interested reader can refer to [6,23] for a detailed presentation of the approach.



a) Selection of migration alternatives b) Log of EMFMigrate code generation

Fig. 11: Generation of migration solution code

desired options for migration are selected and shown as green crosses. For each selected migration alternative corresponding EMFMigrate migration rules are generated, that can be then executed on the input transformation in order to obtain the migrated one. As one might see, the option *R5a2* is already grayed out by the tool, since this option is in conflict with the selected one *R4a2*, thus, not available anylonger. Fig. 11b shows the generation of the selected migration alternatives and a fragment of the generated EMFMigrate code is shown in Listing 1.3. The shown code is related to the management of the metamodel refactoring *R1* by means of the selected alternative *R1a2* (cf. Table 1). Lines 5–17 represent the guard of the rule and thus the metamodel changes that have to match in order to execute the migration specification in line 21. The reported guard corresponds to the metamodel refactoring *R1* involving the metaclasses `Person`, `Employee`, and `Researcher`. The application of the migration in line 21 induces the adaptation of the affected transformation by changing the input patterns typed `Person`, which are all replaced with the class `Employee`.

```
1  migration Workplace2ConferenceManagement-Transformation;
2  migrate SimpleWorkplace2ConferenceManagement.atl : ATL
3    with WorkplaceMM0-WorkPlaceMM2.delta{
4    ...
5    rule migrationR1a2[
6      class person= changeClass(oldperson: class){
7        set abstract=true;
8        ...
9      }
10     class employee=addClass("Employee"){
11       set name="Employee";
12       set eSuperType=person;
13     }
14     class researcher=addClass("Researcher"){
15       set name="Researcher";
16       set eSuperType=person;
17     }
18   ]
19   {
20     -- input patterns and helper contexts will be assigned to the R1a2 choice
21     o1: OclModelElement where [name = oldperson.name] -> o2: OclModelElement [ name
         = employee.name ]
22   }
23 ...
```

```
24 }
```

Listing 1.3: Fragment of the generated EMFMigrate code

With the help of this approach and by taking potential conflicts into account, the number of valid migration alternatives is significantly reduced, resulting in

$$3 \times 4 \times 3 \times 2 \times 2 = 144$$

migration alternatives for the refactoring presented in Table 1, which are significantly less than those that can be obtained without the proposed approach. Furthermore, migration alternatives are reduced even more once some of them are selected, as shown in Fig. 11, thus, helping the user in finding the most appropriate solution. Moreover, the generated EMFMigrate code consists only of those migration rules related to the alternatives that are selected by means of the feature model.

5 Related Work

In this section, we report on work (i) closely related to model transformation migration, and (ii) more widely related with respect to variability in co-evolution in MDE.

Recent approaches tackling the problem of transformation migration mostly aim at providing a unique, predefined, and possible over-writable solution, thus, variability is not supported, but has to be manually considered after the migration, entailing the drawback that generated solutions have to be modified by hand, regardless if the solution has been generated by means of a higher-order transformation [10], predefined migration actions [12,16], or mapping operators [25]. In [9] the authors propose the usage of transformation chains, which are chosen by the user, thus, different solutions can be generated, but having the drawback that the modeler has to be familiar with transformation chains. In [17] a comprehensive set of metamodel changes is proposed, each accompanied with a migration action for models and transformations. Since for the same metamodel evolution, different semantic changes entailing different migration solutions are proposed, variability is slightly considered in the sense, that the evolution designer can incorporate the intention of the evolution when applying the changes. However, support for exploration of different options for migration is not provided, i.e., an intensional representation is not provided.

As a more widely related work, in [21] an approach for the generation of multiple, ranked solutions for model migration (in contrast to transformation migration as proposed in this paper) is presented. Based on the formalization of the conformance relationship, the authors employ logic programming to generate a set of ranked solutions for model migration. However, to the best of our knowledge, an approach supporting variability in the context of transformation migration has not been proposed yet. As a result, one may see that the presented approach is unique in the respect that alternative solutions can be explored and selected by having a suitable representation in terms of a feature model easing the burden of exploring and selecting the ultimately desired solution.

6 Conclusion and Future Work

In this paper, we proposed an approach for exploring and resolving variability during model transformation migration, which is inevitable as a response to metamodel evolution. The approach builds upon an *intensional* representation to explore variability by representing each migration alternative as a dedicated element in a weaving model. Furthermore, potentially arising conflicts between solutions that are not compatible to each other can be explicitly highlighted. In order to support the user not only in exploring but also in resolving variability, a suitable representation in terms of a feature model has been proposed, which is automatically generated from the weaving model. Additionally, EMFMigrate migration actions have been attached to the migration alternatives, which allow for a (semi-)automatic co-evolution of transformations. Besides the possibility of representing alternatives in a compact way, the method provides means for detecting variations among the different solutions. Detection that otherwise should be performed by manually comparing models, which are greatly overlapping one with another.

There are several lines of future work. As already mentioned, we plan on automating the generation of the weaving model, starting form our previous work on generating multiple solutions for model co-evolution [21] in order to create the intensional representation of multiple migration alternatives. Furthermore, we plan on an evaluation in terms of a user study to identify and highlight the major benefits and possible drawbacks of the proposed approach especially from a usability point of view.

Acknowledgment

This work has been partly funded by the Austrian Science Fund (FWF) under grant P 28519-N31 and the OeAD under grant WTZ AR18/2013 and WTZ AR10/2015.

References

1. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Transactions on Database Systems (TODS) 6(4), 557–575 (1981)
2. Beuche, D., Papajewski, H., Schröder-Preikschat, W.: Variability management with feature models. Science of Computer Programming 53(3), 333 – 352 (2004)
3. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology 6(9), 165–185 (October 2007)
4. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Model Conflicts in Distributed Development. In: MoDELS '08. pp. 311–325. Springer (2008)
5. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: Proc. of EDOC. pp. 222–231. IEEE (2008)
6. Di Ruscio, D., Iovino, L., Pierantonio, A.: Coupled evolution in Model-Driven Engineering. IEEE Software 29(6), 78–84 (2012)
7. Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems. In: ICGT. vol. 7562. Springer (2012)
8. Di Ruscio, D., Iovino, L., Pierantonio, A.: A methodological approach for the coupled evolution of metamodels and atl transformations. In: ICMT. Springer (2013)

9. Garcés, K., Vara, J.M., Jouault, F., Marcos, E.: Adapting transformations to metamodel changes via external transformation composition. Software & Systems Modeling (2013)
10. García, J., Diaz, O., Azanza, M.: Model transformation co-evolution: A semi-automatic approach. In: Proc. of SLE. pp. 144–163. Springer (2013)
11. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: Engineering model transformations with transml. Software and System Modeling 12(3), 555–577 (2013)
12. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Cope - automating coupled evolution of metamodels and models. In: Proc. of ECOOP. pp. 52–76. Springer (2009)
13. Iovino, L., Pierantonio, A., Malavolta, I.: On the Impact Significance of Metamodel Evolution in MDE. JoT 11(3), 3:1–33 (Oct 2012)
14. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming 72(1–2), 31 – 39 (2008)
15. Kolovos, D.S., Paige, R.F., Polack, F.: The Epsilon Transformation Language. In: ICMT. pp. 46–60 (2008)
16. Kruse, S.: On the Use of Operators for the Co-Evolution of Metamodels and Transformations. In: Int. Workshop on Models and Evolution 2011 (2011)
17. Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schwinger, W., Schönböck, J.: Consistent Co-Evolution of Models and Transformations. In: MODELS. IEEE (10 2015)
18. Richters, M., Gogolla, M.: «UML»'99 — The Unified Modeling Language: Beyond the Standard, chap. A Metamodel for OCL, pp. 156–171. Springer (1999)
19. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.: The Epsilon Generation Language. LNCS, vol. 5095, pp. 1–16. Springer (2008)
20. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. Computer 39(2), 25–31 (Feb 2006)
21. Schönböck, J., Kusel, A., Etzlstorfer, J., Kapsammer, E., Schwinger, W., Wimmer, M., Wischenbart, M.: CARE – A Constraint-Based Approach for Re-Establishing Conformance-Relationships. In: Proc. of the APCCM (2014)
22. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: An extensible framework for feature-oriented software development. Science of Computer Programming 79, 70 – 85 (2014)
23. Wagelaar, D., Iovino, L., Di Ruscio, D., Pierantonio, A.: Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In: ICMT, LNCS, vol. 7303. Springer (2012)
24. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D., Paige, R., Lauder, M., Schürr, A., Wagelaar, D.: Surveying Rule Inheritance in Model-to-Model Transformation Languages. JOT 11(2), 3:1–46 (Aug 2012)
25. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Surviving the heterogeneity jungle with composite mapping operators. In: ICMT. pp. 260–275. Springer (2010)