Determining the necessity of human intervention when migrating models of an evolved DSL

Hauke Wittern
Department of Computer Science
HAW Hamburg
Hamburg, Germany
hauke.wittern@haw-hamburg.de

Abstract—This paper presents an approach for generating a migration script for existing models of an evolved DSL. The approach has two major benefits: first, unnecessary migration actions are eliminated automatically. Second and even more important, the necessity of human intervention during migration is determined automatically. The necessity of migration is determined by analyzing a DSL’s metamodel and evaluating pre- and postconditions of transformation operator sequences.

Keywords—DSL evolution; migration; human intervention;

I. INTRODUCTION

A domain-specific language (DSL) is a computer programming language that is focused on a particular domain [1], [2]. In contrast to general purpose languages (GPLs), the expressiveness of DSLs is limited [1]. Because of these characteristics, DSLs make it easy to understand certain parts of a system, and implementing system artifacts with a DSL is potentially easier, faster and less prone to errors than implementing the same functionality with a GPL [1], [2]. In this paper programs expressed by means of a DSL are called models.

DSLs evolve throughout their lifetime [1], [3] for manifold reasons. The domain itself [4], [5], [6], technology [7] and system requirements [5], [8], [9] are subject to evolution. These changes must be reflected by the DSL and thus require to adapt it [1], [4], [5], [7], [9], [10].

As a consequence of the evolution of a DSL, models created with the old version of the DSL might not conform to the new version [5], [11], [12]. In order to keep models consistent with the latest DSL version, they have to be migrated [12], [13], [14]. This coupling of DSL transformation and model adaption is called co-evolution [12] or coupled evolution [14], [15]. As manual migration is laborious and prone to errors, automated migration is preferred [12], [13]. While there is a huge potential for automation [15], sometimes migration cannot be automatically resolved from DSL transformations so that human intervention is required [12], [16]. Human intervention means that a human has to provide missing information required during migration of models. For example, when a string attribute is split into two attributes, a human has to specify how the string values are split. The metamodel does not indicate how to split values since it does not contain any information about the contents of the string. After a human has specified how the strings are split, migration can be performed automatically.

One approach for co-evolving DSLs and models is specifying the evolution of a DSL as a sequence of metamodel-transformation-operators (change trace) that are coupled with migrations [14]. Becker et al. [16] have classified primitive operators by whether human intervention is necessary or not.

Currently, no method is known for determining the necessity of human intervention either in a composition of primitive operators to complex ones or in a sequence of transformations of an entire evolution step. A natural approach is to determine the necessity of human intervention for each primitive operator separately. But the result of that approach includes some situations in which human intervention is actually not necessary. In this paper we present an approach based on using pre- and postconditions of coupled operators. By means of analyzing the metamodel and pre- and postconditions we omit migrations for metamodel elements that cannot be instantiated in the model.

The paper is organized as follows: Section II shows how this work is related to existing research. Section III explains our approach, which is illustrated by an exemplary evolution in Section IV. Some important implementation aspects are discussed in Section V. In Section VI we conclude the paper and provide an outlook for future work.

II. RELATED WORK

When a DSL is evolved models that have been created with the DSL might become invalid [1], [5], [7], [11], [12], [13], [15], [16], [17], [18], [19]. One approach to address this problem is changing a DSL in a manner that keeps the new DSL version downward compatible with the old version [13]. However, this limits flexibility in adapting a DSL [13]. In this paper another approach, which does not limit flexibility and allows the removal of legacy constructs [13], is followed: models are adapted to the new DSL version. This is called migration [1], [5], [7], [12], [13], [15], [16], [18].
Changes to a DSL can be either determined by comparing two DSL versions [12], [16] or specified as a sequence of operators that transform the metamodel [7], [13], [19]. This paper follows the approach of specifying the evolution of a DSL as a sequence of transformation operators. An operator sequence can be specified manually [19] or can be recorded as a change trace [13], [16]. Our approach does not make any assumptions about how an operator sequence is created.

In the operator based approach, metamodel and models are usually co-evolved by coupling metamodel transformations with corresponding migrations [7], [12], [13], [14], [19], [20]. Then, an operator is called a coupled operator [7] and evolution with coupled operators is called coupled evolution [7], [13], [14], [20]. Our approach follows the principle of coupled operators, too.

Coupled operators can be independent or dependent on a DSL’s metamodel [15]. Operators can also be specific for a particular model if specific migration is required [15]. Metamodel-independent coupled operators can be reused for evolving any DSL and migrating its models. A study by Herrmannsdoerfer et al. revealed considerable potential for reuse as two third of the operators that were used in their study were metamodel-independent [15]. A number of generic operators have been listed in some papers [7], [12], [13], [16], [20]. Our work is based on the catalog by Herrmannsdoerfer et al. [7].

Primitive operators are atomic operators. That means they perform metamodel transformations that cannot be divided into two or more transformations [7]. Complex operators on the other hand, can be composed as a sequence of primitive operators [7]. However, the migration of a complex operator might have different effects on the model [7]. While our primitive operators are metamodel-independent, complex operators can also be metamodel-specific. Which primitive operations for transforming a DSL’s metamodel are possible depends on the metamodel (i.e. the metamodel of the DSL’s metamodel). As we use the operator catalog by Herrmannsdoerfer et al. we use the same metamodel which they presented in [7]. It is shown in Figure 1.

Several classifications of metamodel transformations have been used in [7], [12], [13], [16], [19], [20], [21], among others. Most relevant in this paper is whether a metamodel transformation requires migration of models and whether the corresponding migration requires human intervention. Transformations that do not require migration affect the metamodel only [15] and are called non-breaking [12], [16] or model-preserving [7]. Otherwise a transformation is called breaking [12], [16] or model-migrating [7]. Breaking transformations whose corresponding migrations require input by a human are called unresolvable (or non-resolvable) otherwise they are called resolvable and can be automatically migrated [12], [16]. The identification of unresolvable breaking metamodel transformations is a central point in our approach. We use the classification of coupled operators in [7], [12], [16] to identify which operators potentially require input by a human.

III. APPROACH FOR MIGRATION SCRIPT DETERMINATION

This section presents our approach for generating a model migration script for the evolution of a DSL’s metamodel. The generated migration script allows us to identify when human intervention is necessary during migration. In our approach, the evolution of a DSL is seen as a sequence of transformations. A transformation is a particular change to a particular metamodel. These changes are performed by applying transformation operators. A transformation operator specifies a recurring change to metamodels. Primitive transformation operators can be composed to complex ones [7]. While complex operators ease the evolution for the user, the underlying primitive operators are relevant at some points in the approach for migration script determination.

A migration script is a sequence of instructions for reconciling existing models with an evolved DSL version. So a migration script can be thought of as a small imperative computer program. Using the principle of coupled operators [7], [14] each instruction in a migration script is coupled to the transformation operator in a transformation sequence that necessitates the migration instruction. That is a logical mapping. But it does not mean parallel actions at the same time.

The reconciling actions for a specific transformation operator are encapsulated in migration strategies. A migration script is basically a sequence of calls to migration strategies and parameterizes them with metamodel and model elements. A migration script can also contain conditional statements and iterations over metamodel/model elements. For resolvable transformations predefined migration strategies are used. Otherwise, if a transformation is unresolvable, a human has to implement a custom migration strategy. In our approach the implementation of custom migration strategies embodies human intervention.

A. Benefits of the approach

The benefit of our approach is that the necessity of migrations can be detected. This is especially useful for detecting the necessity of migrations that require the human intervention. Resources can be saved when unnecessary custom migration strategies are detected and not implemented.

With our approach it can be determined which model elements have to be migrated, which of these model elements can be migrated automatically with a reusable migration strategy and which have to be migrated with custom migration strategies. This migration determination is done regardless of any existing models. Model instances are neither required nor used during migration script generation. The resulting script is applicable to any model, though. This is beneficial since a DSL is essentially a published interface [1] and models might therefore be unavailable to
the DSL Developer. Moreover, migration can be performed lazily when a model is instantiated [16]. However, in rare cases migration can require model specific information. Then the user either has to do some specific migration steps manually or has to add the missing information or migration implementation that is needed for automatic migration execution. Fortunately, model specific migration is rarely needed in practice. Hermannsdoerfer et al. did not find any case of model specific migrations in their study about the evolution of two different metamodels although numerous transformations had been carried out [15]. Anyway, our approach delivers a framework of a migration that shows the user at which points he must complete it manually.

B. Determining the necessity of human intervention

The type of instructions in a migration script allows to identify when human intervention is required during migrations. Those aspects that can be migrated automatically are migrated with predefined migration strategy objects. Otherwise, when human intervention is required, the script contains a call to a new custom migration stub. This stub has to be implemented by a human and is a specific custom migration for a particular transformation of a DSL’s metamodel.

C. Determining the necessity of migration

The generated migration script has to contain only migrations that are really necessary. A migration is unnecessary if it creates non-mandatory model elements or if it is related to a metamodel element of which instances cannot exist in the model. Conversely, migration instructions must be included for model migrating transformations if there is a possibility that an instance of the affected metamodel elements exists in the model. We identified three complementary methods for determining the necessity of migration:

- Analysis of the properties of the DSL’s metamodel with respect to the recent transformation operator.
- Computation and evaluation of pre- and postconditions of the transformation operator sequence.
- Analysis of the model.

The last of these is not applicable during migration script generation. By analyzing a model it can be identified which metamodel elements are instantiated. This allows us to exclude migrations for metamodel elements that are not instantiated. However, this applies to a particular model and is done during execution of a migration script. Therefore, we do not use this method. But the first two methods are well suited to identify the potential necessity of migration in any model during migration script generation and are both used in our approach.

By analyzing the DSL’s metamodel, the potential necessity of migration is determined for each single transformation. This includes determination of the extent of migration: the metamodel elements are identified for which model instances are affected by the migration. Some properties of the metamodel limit the extent of migration. For example, abstract classes in the metamodel cannot be instantiated in the model. Some properties of the metamodel even totally exclude the necessity of migration. For example non-mandatory features never have to be instantiated during migration. Existing instances of non-mandatory feature, however, might have to be migrated depending on the metamodel transformation.

The resulting set of potentially necessary migrations is further reduced by examining the pre- and postconditions of the transformation sequence up to the current transformation. The pre- and postconditions allow conclusions to be made about the existence of instances in the model. If the absence of any instance of a particular metamodel element can be

---

Figure 1. Our metamodel for which the transformation-operators are defined (see [7]).
concluded, some potential migrations might be excluded. For example, if a class is made abstract all of its instances must be migrated (abstract classes cannot be instantiated). But if it is concluded that instances of that class cannot exist, there is nothing to migrate. Furthermore, the absence of any instance of a particular metamodel element implies in some cases the absence of instances of another element’s instances. Attributes, for example, can only be instantiated for instances of their owning class. Hence, a migration script must include migration instructions for mandatory attributes only if instances of their owning class might exist.

D. Algorithm for determining \textit{PRE} and \textit{POST} for transformation sequences

Pre- (\textit{PRE}) and postconditions (\textit{POST}) of transformation operator sequences are calculated iteratively. The operators are iterated in the order in which they are executed. In each iteration \textit{i} the entire preconditions \textit{PRE\textsubscript{i}} and postconditions \textit{POST\textsubscript{i}} up to the sub-transformation \textit{t\textsubscript{i}} are determined. \textit{PRE\textsubscript{i}} is the set of preconditions that must be fulfilled to be able to execute the sequence of sub-transformations \textit{t\textsubscript{1}} to \textit{t\textsubscript{i}}. \textit{POST\textsubscript{i}} is the set of postconditions that are fulfilled after executing the sequence of sub-transformations \textit{t\textsubscript{1}} to \textit{t\textsubscript{i}}.

Initially, the sets of pre- and postconditions (\textit{PRE\textsubscript{0}} and \textit{POST\textsubscript{0}}) are empty. During iterating the sub-transformations, pre- and postconditions are added or excluded successively as each sub-transformation can have an effect on the entire pre- and post-conditions.

In each iteration \textit{i} those postconditions of the previous iteration \textit{i−1} whose negations are implied by postconditions of \textit{t\textsubscript{i}} (\textit{POST\textsubscript{i}}) are not included in \textit{POST\textsubscript{i}}. The remaining postconditions in \textit{POST\textsubscript{i−1}} are united with the postconditions of \textit{t\textsubscript{i}}. Similarly, the previous preconditions \textit{PRE\textsubscript{i−1}} are united with those preconditions of \textit{t\textsubscript{i}} (\textit{PRE\textsubscript{i}}) that are not implied by either previous preconditions (\textit{PRE\textsubscript{i−1}}) or previous postconditions (\textit{POST\textsubscript{i−1}}). The determination of \textit{PRE} and \textit{POST} can be formally defined by recursively defining the sets:

\begin{align}
\text{PRE\textsubscript{0}} &= \emptyset \\
\text{PRE\textsubscript{i}} &= \text{PRE\textsubscript{i−1}} \cup \{x | x \in \text{PRE\textsubscript{i−1}} \wedge \neg y \in \text{PRE\textsubscript{i−1}} : y \Rightarrow x \}
&\quad \wedge \neg z \in \text{POST\textsubscript{i−1}} : z \Rightarrow x \} \\
\text{PRE} &= \text{PRE\textsubscript{n}} \\
\text{POST\textsubscript{0}} &= \emptyset \\
\text{POST\textsubscript{i}} &= \text{POST\textsubscript{i−1}} \cup \{x | x \in \text{POST\textsubscript{i−1}} \wedge \neg y \in \text{POST\textsubscript{i−1}} : y \Rightarrow \neg x \} \\
\text{POST} &= \text{POST\textsubscript{n}}
\end{align}

The pre- and postconditions of each single transformation (\textit{PRE\textsubscript{t\textsubscript{i}}} and \textit{POST\textsubscript{t\textsubscript{i}}}) are determined by parameterizing the generic specification of the corresponding transformation operator’s pre- and postconditions. Each operator has been formally specified by us based on the informal specification in [7], [12], [16]. The specification of some operators is shown later in the example in Section IV.

Our algorithm can also be used to compute pre- and postconditions of complex transformation operators. Then the underlying operators build the transformation sequence. However, the resulting preconditions might be incomplete. Depending on its semantics, a complex operator might have additional preconditions. These express relations between those metamodel elements that are used as input for the complex operator. The operator \textit{PullupFeature}, for example, is a composition of \textit{DeleteFeature} and \textit{CreateFeature}. When a feature is pulled up from class \textit{a} to \textit{b}, \textit{b} must be a super class of \textit{a}. This condition is not contained in the single operators’ preconditions and thus not contained in the computed preconditions of the complex operator. Such additional preconditions are important since a complex operator must not be applied when any precondition is not fulfilled.

E. Algorithm for migration script generation

The algorithm in Listing 1 is used to generate a model migration script for a given transformation operator sequence. The \textit{CreateMigrationScript} function iterates the operators and uses the algorithm from the previous section to determine pre- and postconditions (call to \textit{DeterminePreAndPost}). In each iteration migration instructions for the current operator are collected by evaluating the migration definition that is coupled to the operator.

Listings:

Listing 1. Algorithm for migration script generation

\begin{verbatim}
func CreateMigrationScript(transformations) {
  PRE := {} // empty set
  POST := {} // empty set
  script := new List;
  for each t in transformations {
    tempTransformations := new List;
    tempTransformations.AppendAll(GetMigrationInstructions(t));
    DeterminePreAndPost(tempTransformations);
    script.AppendAll(
      GetMigrationInstructions(t));
  }
  return script;
}
\end{verbatim}
parameter values, (2) the metamodel as it was constituted before the operator has been executed and (3) the pre- and postconditions of the transformation sequence. It returns a list of those migration instructions that are appropriate for the given parameterization and necessary for valid migration. The necessary instructions are determined as described in Section III-C. First the conditions in a migration definition are evaluated, which embodies the analysis of the properties of a DSL’s metamodel. Afterward, preconditions are evaluated within the Evaluate function to conclude whether instances of the elements referenced in the given migration definition could exist. If the absence of an element for which a migration instruction is specified is concluded, the related migration instruction is left out. For example “migrate each instance of attribute a with new custom migration” contains a reference to an attribute a. If a cannot have been instantiated the custom migration stub is not created and no instruction is appended to the migration script.

IV. EXPLANATION BY EXAMPLE

In this section we explain our approach by evolving a simple DSL. In the course of the example we present the specifications of the operators used. Due to lack of space, we cannot present the specifications of all operators that we are using.

Listing 2 shows the metamodel of the DSL that will be evolved in this section. Listing 3 shows a model that is written with the DSL. The metamodel is evolved with the transformation sequence in Listing 4. It consists of creating a class, adding attributes to the class, making the class the super class of two other classes and finally deleting an attribute of a previously existing class. The evolved metamodel is shown in Listing 5. In the following subsections we explain how the corresponding migration script is generated. Migration definitions are presented as we implemented them in C#.

Listing 2. Initial metamodel of the example DSL
abstract Class DataNode { Attribute Name : String [lower=1, upper=1]; }  
Class LogicalNode : DataNode { Reference ChildNodes : DataNode [lower=0, upper=1]; }  
Class DataModule : DataNode { Reference DataPoints : DataPoint [lower=0, upper=1]; }  
Class DataPoint { Attribute Unit : String [lower=1, upper=1]; Attribute Name : String [lower=1, upper=1]; Attribute Comment : String [lower=0, upper=1]; }

Listing 3. Initial example model
logical node CHPStation01 { data module AnalogValues { data point ActivePower { Unit="W" Comment="unit: Watt" } } }

Listing 4. Executed transformations
CreateClass name=DataObject, isAbstract=true;  
CreateAttribute class=DataObject, name=Id, type=String, lower=1, upper=1;  
CreateAttribute class=DataObject, name=Description, type=String, lower=0, upper=1;  
AddSuperType sub=DataNode, super=DataObject;  
AddSuperType sub=DataPoint, super=DataObject;  
DeleteFeature class=DataPoint, feature=Comment;

Listing 5. Evolved metamodel of the example DSL
abstract Class DataObject { Attribute Id : String [lower=1, upper=1]; Attribute Description: String [lower=0, upper=1]; Attribute Name: String [lower=1, upper=1]; }  
abstract Class DataNode : DataObject { }  
Class LogicalNode : DataNode { Reference ChildNodes : DataNode [lower=0, upper=1]; }  
Class DataModule : DataNode { Reference DataPoints : DataPoint [lower=0, upper=1]; }  
Class DataPoint : DataObject { Attribute Unit : String [lower=1, upper=1]; }

A. Create Class DataObject

The first operator that is executed during evolution is CreateClass. We have specified the pre- and postconditions of the CreateClass operator as follows:

\[ \text{PRE}_{\text{CreateClass}} = \{\neg \text{classExists}(c)\} \]  
\[ \text{POST}_{\text{CreateClass}} = \{\text{classExists}(c)\} \]

The operator has a parameter that qualifies the class with its package path and name. So the pre- and postconditions of this particular transformation are:

\[ \text{PRE} = \{\neg \text{classExists(DataObject)}\} \]  
\[ \text{POST} = \{\text{classExists(DataObject)}\} \]

As this is the first operator of the sequence, the above pre- and postconditions temporarily become the pre- and postconditions of the sequence (Line 8 in Listing 1). Afterwards, the corresponding migration instructions for the first transformation are determined. As CreateClass is a model-preserving operator [7] nothing has to be migrated. Therefore the migration definition for CreateClass (in our implementation represented by the EvaluateMigrationDefinition method) is empty:

class CreateClass : Transformation { TypeQualifier Class; override void EvaluateMigrationDefinition(MigrationContext context) { } }

Consequently, no instructions are added to the migration script (Line 9 in Listing 1).
B. Create attribute Id for class DataObject

Next, an attribute Id is added to the previously created class. The pre- and postconditions of the CreateAttribute operator are defined as follows:

\[
\begin{align*}
\text{PRE}_{\text{CreateAttribute}} & = \{ \text{classExists}(c), \\
& \quad \neg \text{featureExists}(c,a), \\
& \quad \text{isExistingPrimitiveType}(t) \}\ \\
\text{POST}_{\text{CreateAttribute}} & = \{ \text{ownedAttributeExists}(c,a) \}
\end{align*}
\]

The pre- and postconditions of the operator sequence up to this transformation are:

\[
\begin{align*}
\text{PRE} & = \{ \neg \text{classExists}(\text{DataObject}), \\
& \quad \text{isExistingPrimitiveType}(\text{String}) \} \\
\text{POST} & = \{ \text{classExists}(\text{DataObject}), \\
& \quad \text{featureExists}(\text{DataObject,Id}) \}
\end{align*}
\]

The attribute Id is not included in the preconditions according to Equation 1b as it is implied by \neg \text{classExists}(\text{DataObject}).

In accordance with [7] our migration definition for CreateAttribute specifies that migration is not required unless the attribute is mandatory (i.e. lowerBound > 0):

\[
\begin{align*}
\text{class CreateAttribute : Transformation} & ( \\
& \text{FeatureQualifier AttributeQualifier;} \\
& \text{Cardinality LowerBound;} \\
& \text{Cardinality UpperBound;} \\
& \text{override void EvaluateMigrationDefinition}(
\text{MigrationContext context}) \{
& \quad \text{if (LowerBound > 0) \{ \\
\quad & \quad \text{var cvp = context.NewCustomFeatureValueProvider();} \\
\quad & \quad \text{// fetch the attribute from} \\
\quad & \quad \text{// the evolved metamodel:} \\
\quad & \quad \text{var attribute =} \\
\quad & \quad \text{(Attribute)context.New[AttributeQualifier];} \\
\quad & \quad \text{context.MigrateEachInstance(attribute,} \\
\quad & \quad \text{MigrationAction.SetValue(cvp));} \\
& \quad \} \})
\end{align*}
\]

For this particular transformation migration is not necessary although the attribute Id is mandatory. When the migration definition is evaluated (Line 17 in Listing 1), it is reasoned from the precondition \neg \text{classExists}(\text{DataObject}) that no instance of Id can exist in any model. We implemented this class existence check in the MigrateEachInstance method of the MigrationContext class:

\[
\begin{align*}
\text{void MigrateEachInstance(Feature feature,} \\
& \text{FeatureMigrationAction action)} \{
& \quad \text{if (!Preconditions.Implies(new} \\
& \quad \text{ClassExists(feature.Owner).Not()) \{ \\
& \quad \quad \text{var instruction = new} \\
& \quad \quad \text{MigrateEachFeatureInstance(feature,} \\
& \quad \quad \text{action);} \\
& \quad \quad \text{AppendInstruction(instruction);} \\
& \quad \})
\end{align*}
\]

As migration of the attribute Id is not necessary, the implementation of a custom value provider for this attribute is saved.

C. Create attribute Description for class DataObject

Adding the attribute Description to class DataObject does not require migration, either, as the attribute is not mandatory. Hence, the implementation of a custom value provider for the attribute Description is saved.

D. Add super-type DataObject to DataNode

The fourth transformation is making DataObject a super class of DataNode. We have specified the pre- and postconditions of the AddSuperType operator as follows:

\[
\begin{align*}
\text{PRE}_{\text{AddSuperType}} & = \{ \text{classExists}(s) \}, \text{classExists}(c), \\
& \quad \neg \text{isSuperType}(s,c), \\
& \quad \text{isSuperType}(c,s) \} \\
\text{POST}_{\text{AddSuperType}} & = \{ \text{isSuperType}(\text{DataObject, DataNode}) \}
\end{align*}
\]

The pre- and postconditions up to this transformation are:

\[
\begin{align*}
\text{PRE} & = \{ \neg \text{classExists}(\text{DataObject}), \\
& \quad \text{isExistingPrimitiveType}(\text{String}), \\
& \quad \text{classExists}(\text{DataNode}) \} \\
\text{POST} & = \{ \text{classExists}(\text{DataObject}), \\
& \quad \text{featureExists}(\text{DataObject,Id}), \\
& \quad \text{featureExists}(\text{DataObject,Description}), \\
& \quad \text{isSuperType}(\text{DataObject,DataNode}) \}
\end{align*}
\]

Mandatory features that the sub-class inherits from its new super-class have to be migrated:

\[
\begin{align*}
\text{class AddSuperType : Transformation} & ( \\
& \text{TypeQualifier SubClass;} \\
& \text{TypeQualifier SuperClass;} \\
& \text{override void EvaluateMigrationDefinition}(
\text{MigrationContext context}) \{
& \quad \text{var superClass = (Class)context.New[SuperClass];} \\
& \quad \text{var subClass = (Class)context.New[SubClass];} \\
& \quad \text{var featuresToMigrate =} \\
& \quad \text{from f in superClass.AllFeatures() where f.LowerBound > 0 select f;} \\
& \quad \text{foreach (var feature in featuresToMigrate) \{ \\
\quad & \quad \text{var cvp = context.NewCustomFeatureValueProvider();} \\
\quad & \quad \text{context.MigrateEachInstance(feature,} \\
\quad & \quad \text{SubClass,} \\
\quad & \quad \text{MigrationAction.SetValue(feature, cvp));} \\
& \quad \})
\end{align*}
\]

For each instance of the sub-class (including its subclasses) the mandatory features inherited from the new super-class must be set. This has to be done with a custom migration strategy: the values for the features are provided by custom value providers. Each value provider is specific for a particular feature. For each custom value provider a stub is generated. A programmer has to complete these stubs with logic for computing the values.

In the example model LogicalNode and DataModule are the only (indirect) sub-classes of DataObject that can be instantiated. Their super-class DataNode however, is abstract
and thus cannot be instantiated in a model. Id is the only inherited mandatory attribute. For this attribute a custom value provider class CustomValueProvider_DataObjectId is generated. The generated migration instructions reference this class:

```
migrate each instance of LogicalNode with
    SetValue(Id,
        CustomValueProvider_DataObjectId);
migrate each instance of DataModule with
    SetValue(Id,
        CustomValueProvider_DataObjectId);
```

E. Add super-type DataObject to DataPoint

Making DataObject a super type of DataPoint is similar to the previous transformation. The migration script is extended with the instruction below and a custom value provider stub CustomValueProvider_DataObjectId_2 is generated:

```
migrate each instance of DataPoint with
    SetValue(Id,
        CustomValueProvider_DataObjectId_2);
```

The custom migration logic in CustomValueProvider_DataObjectId_2 might be the same as in CustomValueProvider_DataObjectId. However, both migrations relate to different transformations. Therefore, separate stubs are generated. A developer can reuse migration logic, though. Actually, we also implemented an operator that accepts a list of sub-classes as input. That operator could be used once instead of using AddSuperType two times as in this example. That would require only one custom value provider stub.

F. Delete feature Comment from DataPoint

At last the attribute Comment is removed from DataPoint. The pre- and postconditions of DeleteFeature are:

```
PRE_{DeleteFeature} = \{\text{classExists}(c),
    \text{featureExists}(c, f)\}
POST_{DeleteFeature} = \{\text{¬featureExists}(c, f)\}
```

The resulting pre- and postconditions of the complete transformation sequence are as follows:

```
PRE = \{\text{¬classExists(DataObject)},
    \text{isExistingPrimitiveType(String)},
    \text{classExists(DataNode)},
    \text{classExists(DataPoint)},
    \text{featureExists(DataPoint, Comment)}\}
POST = \{\text{classExists(DataObject)},
    \text{featureExists(DataObject, Id)},
    \text{featureExists(DataObject, Description)},
    \text{isSuperType(DataObject, DataNode)},
    \text{isSuperType(DataObject, DataPoint)},
    \text{¬featureExists(DataPoint, Comment)}\}
```

During migration, feature instances must be removed from models:

```
class DeleteFeature : Transformation {
    // ...
    FeatureQualifier Feature;
    override void
    EvaluateMigrationDefinition(MigrationContext context) {
        var feature = context.Old[Feature];
        context.MigrateEachInstance(feature,
            MigrationAction.DropFeatureValue);
    }
}
```

As instances of the feature become obsolete, an instruction for removing them from models is appended to the migration script:

```
migrate each instance of DataPoint with
    DropFeatureValue(Comment);
```

G. Result

The resulting migration script is shown in Listing 6. From the script can be gathered automatically that a human has to implement three custom value providers. These specify how the values of the Id attribute are computed for each LogicalNode, DataModule and DataPoint existing in a model. This should be supported by appropriate tool support. Due to lack of space, we left out statements from the script that enable tracing back migration instructions to the operators. Listing 7 shows the resulting model. The values of the Id attributes have been generated by the custom value provider implementations by simply incrementing an instance counter.

```
Listing 6. The resulting migration script

migration script for Evolution1:
    migrate each instance of LogicalNode with
        SetValue(Id,
            CustomValueProvider_DataObjectId);
    migrate each instance of DataModule with
        SetValue(Id,
            CustomValueProvider_DataObjectId);
    migrate each instance of DataPoint with
        SetValue(Id,
            CustomValueProvider_DataObjectId_2);
    migrate each instance of DataPoint with
        DropFeatureValue(Comment);
```

```
Listing 7. Migrated example model

logical node CHPStation01 {
    ID="1"
    data module AnalogValues {
        ID="2"
        data point ActivePower { ID="3 Unit="W" } }
}
```

In the previous example the implementation of one custom value provider has been saved due to an unfulfilled precondition (Section IV-B). The implementation of two further custom value providers has been saved on the basis of the analysis of the DSL’s metamodel properties (Sections IV-C and IV-D).
V. IMPLEMENTATION ASPECTS

In this section two important aspects of our implementation are presented. These aspects are (1) how the generic migration engine is able to migrate DSL model with arbitrary concrete syntax and (2) how it is determined whether certain conditions imply other conditions.

A. Generic migration engine

Our approach allows to migrate models of any DSL. It can also be used for evolving DSLs with different metametamodels than ours. An implementation of our approach is based on a certain metametamodel, though. We use the metametamodel in Figure 1. It can be used to specify any DSL. The general applicability of our approach saves the implementation of a migration mechanism for each DSL. However, a DSL is usually not explicitly specified with our metametamodel in the first place. When the metametamodels differ, the DSL developers have to redefine the DSL's metametamodel in order to use our implementation. In any case the developers also have to implement a model adapter. Depending on the tools used, they might have to implement an unparsing mechanism as well.

1) Metamodel redefinition: The grammar of a DSL is specified with a metasyntax like EBNF (Extended Backus-Naur Form) or SDF [22] (Syntax Definition Format). The metamodel of these metalanguages differs from our metametamodel. A parse tree of a DSL model can be translated to a semantic model. This is a data structure representing the subject described with the model [1]. Usually, the metametamodel of such semantic model is not our metametamodel, either. When the semantic model is specified with an object-oriented programming language the metametamodel is the metamodel of that programming language.

When the metametamodels differ, models cannot be migrated with our approach, initially. Then DSL developers have to define the DSL using our metametamodel in addition to the existing definition. The metamodel redefinition can potentially be generated from the metametamodel. However, we have not implemented such a generator, yet.

2) Model adapter: The implementation component of our implementation takes a model as input and outputs a migrated model. To be independent from both the abstract and concrete syntax of a DSL, we defined a generic representation of models. The input of the migration component must be in this format. In order to migrate a model of an initial DSL version, the model has to be translated to our generic representation. The output of the migration component is in the generic format as well. It has to be translated to a model in the evolved DSL’s syntax. Both translations are done by a model adapter component. Model adapters are DSL specific and have to be implemented by DSL developers. Currently, we implement model adapters manually. However, we plan to generate them from annotated syntax definitions.

A model adapter can be implemented based on the parse tree or the semantic model (see Figure 2). A combination of both strategies is possible as well. Either way a model is always parsed into a parse tree first. When the DSL implementation uses a semantic model, the parse tree is afterwards translated to the semantic model. The latter translation is implemented by a DSL user. The model adapter takes the parse tree or the semantic model. It transforms model elements of type Class to entity objects. Model elements of type Feature are transformed to property values and linked with the entities.

The other way round, a migrated model in the generic representation has to be translated back to DSL code. As an intermediate step, the model adapter translates entities and property values either to a semantic model or a parse tree complying with the new DSL version. Afterwards, this semantic model or the parse tree, respectively, is translated to DSL code. From a semantic model DSL code is generated using a code generator. From a parse tree DSL code is generated using an unparsen. Both code generator and unparsen could be generated from the DSL’s syntax definition.

Which strategy for implementing a model adapter is best depends primarily on the development effort. The development effort, in turn, depends on tool support. Unparser generation, for example, is not supported by each parser generator. A model adapter can potentially be generated from a DSL’s syntax and semantic model specifications. This would minimize manual development effort to extending the syntax and semantic model specifications with the information required for generating a model adapter. However, the means used for syntax and semantic model specifications might not allow us to attach these information.

Currently we use the Spoofax language workbench [23]. It provides an unparsing mechanism. Therefore we implement model adapters based on syntax trees (abstract syntax trees to be precise). Due to lack of tool support we implement model adapters manually. We plan to generate model adapters in the future, though.

3) Consideration of development effort: Anyway our approach is associated with some effort for the DSL developers. As mentioned above it requires to redefine a DSL’s metametamodel using our metametamodel and the implementation of a model adapter. Additionally, custom migrations have to be implemented manually. But programming the custom migrations is not more effort than programming the whole migration script manually. With proper tool support the effort associated with our approach could be minimized to extending the DSL definition. The redefined metametamodel and model adapters can potentially be generated from it. However, this is not supported by current tools.

Even if it does not follow our approach, a generic migration script generator requires some adaption mechanism, unless there is an explicit metametamodel provided by the development tools. To our knowledge our approach is the
In this paper we presented an approach for generating a migration script for an evolved DSL. The approach was illustrated by an example evolution of a DSL’s metamodel. The example has shown that it can be determined easily from a migration script where a human has to implement custom migrations, which represent human intervention.

VI. CONCLUSION AND FUTURE WORK

In [7], [12], [16] we found specifications and classifications of coupled operators that enabled us to identify when human intervention is potentially necessary during migration. However, these specifications are informal and the conditions under which human intervention is necessary have not always been noted completely. Therefore, we specified the migration operations formally and in more detail, in order to be able to automatically determine when human intervention is really necessary. Moreover, we specified pre- and postconditions of metamodel transformation operators, which has not been done before. Due to space limitations, we did not present the specifications of all operators. But they can be reconstructed by the reader easily. The operators we use apply only to our metametamodel anyway. Our approach, however, is applicable to any metametamodel.

Our approach is unique as we can identify the necessity of human intervention automatically. Hence, we believe that our approach simplifies migration and is associated with less effort than other approaches. We are currently evaluating our approach and implementation as part of the project Smart Power Hamburg\(^1\) within the energy and smart grid domain. We will evaluate how our approach performs in comparison to generating migration scripts without it and creating migration scripts manually.

Currently we are also improving our tool support. Especially, we are automating the metamodel redefinition and model adaption as mentioned in Section V.

The presented approach could be further improved. As presented in this paper unnecessary migration actions are eliminated on the basis of preconditions. Postconditions are only required to compute preconditions. It seems also possible to further reduce the list of migration actions on the basis of postconditions. If a metamodel element is deleted those migrations for instances of this element that are related to previous transformation actions are not required. However, this cannot be easily put to practice. Removing

\(^1\)http://www.smartpowerhamburg.de, funded by BMWi via EnEff:Wärme
previous migration actions might lead to side effects that affect the final result of the migration. It must be ensured that the result of migration without them is the same as it would be with them.

REFERENCES


