Heterogeneous Coupled Evolution of Software Languages

Sander Vermolen and Eelco Visser

Software Engineering Research Group
Delft University of Technology, The Netherlands
s.d.vermolen@tudelft.nl, visser@acm.org

Abstract. As most software artifacts, meta-models can evolve. Their evolution requires conforming models to co-evolve along with them. Coupled evolution supports this. Its applicability is not limited to the modeling domain. Other domains are for example evolving grammars or database schemas. Existing approaches to coupled evolution focus on a single, homogeneous domain. They solve the co-evolution problems locally and repeatedly. In this paper we present a systematic, heterogeneous approach to coupled evolution. It provides an automatically derived domain specific transformation language; a means of executing transformations at the top level; a derivation of the coupled bottom level transformation; and it allows for generic abstractions from elementary transformations. The feasibility of the architecture is evaluated by applying it to data model evolution.

1 Introduction

Data models are an integral part of software development. They define the structure of data that is processed by an application and the schema of a database. Running applications produce and store data that conforms to the data model.

Due to changing requirements or maintenance, data models need to evolve. This process is known as *format evolution* [15]. As a consequence of evolution, stored data no longer conforms to the evolved data model and can thereby become useless to the evolved application. To continue using existing data, the data needs to be transformed to reflect the evolution, which is an instance of coupled evolution.

Coupled evolution does not only apply to data transformation, but is a reoccurring problem in computer science [14]. Models need to be transformed to reflect evolution in their meta-models [7,6,23]. Programs need to be transformed when the programming languages (or domain specific language) they have been written in evolves [18]. And a data model itself needs to be transformed to reflect evolution in the data modeling language (e.g. UML). We unify these scenarios by considering evolving software languages and transformation of sentences in these languages.

Current approaches to support coupled evolution of software languages are homogeneous. They solve the problem in a specific domain. They repeatedly implement the coupled evolution structure and solve problems common to coupled evolution locally. Instead, we would like a systematic approach to realize heterogeneous coupled evolution for any scenario of software language evolution.

K. Czarnecki et al. (Eds.): MoDELS 2008, LNCS 5301, pp. 630–644, 2008.

```
User {
  name    :: string
}
Web {
  admin   :: set of User
  topic   :: string
}
Page {
  content :: string
  date    :: date
  author  :: User
  web     :: Web
}
```

Fig. 1. Data model 1

```
User {
  name    :: string
}
Web {
  admin   :: set of Group
  topic   :: string
}
Page {
  content :: string
  date    :: date
  author  :: User
  web     :: Web
  topic   :: string
}
Group {
  name    :: string
  members :: set of User
}
```

Fig. 2. Data model 2

In this paper, we present two generalizations over coupled software language evolution scenarios and introduce the concept of heterogeneous coupled evolution. To enable the generalization, we present an architecture to support heterogeneous coupled evolution of software languages. We have implemented a tool to support the architecture. It generates a domain specific transformation language (DSTL) for an arbitrary software language domain. It generates an interpreter of transformations defined in the DSTL. And it supports generic abstraction from the basic transformations that are defined in the DSTL. We illustrate the architecture and tool by elaborating their application to coupled data model evolution.

The paper is structured as follows: In Section 2 we briefly introduce data model evolution and its context. In Section 3 we elaborate on coupled data evolution by defining data model transformations and deriving data transformations to reflect these. In Section 4 we generalize over the different scenarios of coupled software language evolution. In Section 5, we discuss the architecture to support heterogenous coupled evolutions. Section 6 discusses related work. Section 7 concludes.

2 Data Model Evolution

Data models describe the structure of data that is processed and stored by an application. As example application we consider a Wiki. It consists of web pages, users to edit these and webs, which are collections of pages that cover a similar topic. The corresponding data model is shown in Figure 1.

Changing requirements and maintenance cause data models to evolve along with the application they are set in, a process known as format evolution [15]. Consider for example the shift from a user-based to a group-based access control security mechanism and the addition of page topics. The new data model to support these is shown in Figure 2.

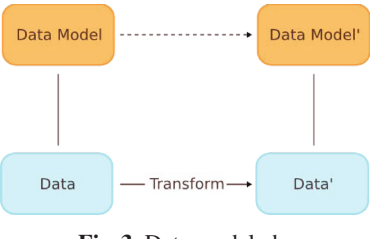

Fig. 3. Data model changes

Since the Wiki is a running application during evolution, it has stored pages, users and webs. Such data conforms to some version of the data model. To prevent the loss of data when the data model changes, the data needs to be transformed to reflect these changes. Figure 3 shows the process graphically. At the top level, we see two versions of the data model. At the bottom level we see the stored data and the transformation needed to reflect the data model change. The vertical lines indicate conformance. The dashed arrow indicates the changes applied to the data model and is usually performed manually by editing the data model. The transform arrow on the other hand requires tools for database transformations, as it is usually too much of an effort to reenter all data manually.

3 Coupled Data Evolution

When a data model evolves, stored data may no longer conform. In practice the data is usually no longer usable. To continue to use the data, we need to reflect the data model changes in a transformation of the stored data. Supporting a single data model change, requires a significant effort. Supporting data model changes in an evolution process, requires repeated data transformations. If these transformations are defined manually, this becomes costly and holds back the development process.

Coupled data evolution automates the data transformation process. It is based upon the assumption that the data model transformation and the data transformation are related. The concept of coupled data evolution is shown in Figure 4. Coupled evolution consists of two components, which are represented by the two new arrows: (1) A definition of the data model transformation (the evolve arrow) and (2) a mapping from data model transformation to data transformation (the dashed arrow). The first needs to be specified for each change of the data model, whereas the second is typically defined once for our data models.

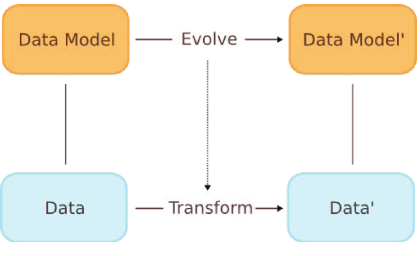

Fig. 4. Coupled data evolution

The questions that remain are: How to define the data model transformation and how to derive a data transformation. In the literature we can find various approaches to formalize both [15,2,4]. In this section, we take a closer look at the two, using the Wiki data model as a running example. We introduce a language for defining data model transformations and show a mapping that targets a broad set of databases.

3.1 Defining Data Model Transformations

We distinguish two methods to formalize transformations for coupled evolution [7]: Specify the difference between the two versions of the data model or specify a trace of elementary transformations defining how the new version is obtained from the old version. Both have advantages and disadvantages. We choose the second because it allows us to define the mapping, as we will see in the next section.

Our data models are relatively basic. They consist of entities with a name and properties. Each property has a name and a type. Entities, properties, types and names are the constructs of our data model. A fairly limited set of constructs. Consequently, the number of elementary transformations we can perform on our model (on our constructs) is fairly small. We identify:

- adding or removing entities
- changing the name of an entity
- adding or removing properties
- changing the name of a property
- substituting the type of a property
- substituting the type of a set

We define for example the addition of a new entity "Group" as follows:

```
add Group {
  name        :: string
  description :: string
  members     :: set of User   }
```

Although the addition is in itself a valid transformation, in general the elementary actions above do not have sufficient meaning on their own. One cannot change the name of an entity without knowing which entity is being referred and a substitution of a type should not only specify the new type, but also the old type that is being replaced. The transformations above are local and need a location to make them executable to a specific model.

The representation of a data model is tree-structured. The root node is the model itself. Its direct children are entities, which have in turn properties as their children and so on. We define a unique location in a data model by specifying a path from the root node. For example: "Entity *Group* - Property *members* - Type" indicates the type of the `members` collection in the `Group` entity that we have just added. In a similar way, we define locations for our transformations using the APath [13] notation, which is based on the XPath language [3]. APath expressions consist of a /-separated list of construct names. The above would be written as:

```
Entity [Id="Group"] / Property [Id="members"] / Type
```

The `[...]`-part indicates a predicate on the node that is being evaluated. If we would have written `Entity/Property/Type`, we would have got all types, of all properties, of all entities. The predicates restrict this by only allowing those with the right id's.

We define a transformation to be a combination of an APath and a local transformation. The two are separated by a ::-sign. As examples, we specify the removal of the 'description' property and the substitution of the type in a set:

```
Entity[Id="Group"] / Property[Id="description"] :: remove
Entity[Id="Web"] / Property[Id="admin"] / Type / Set / Type :: substitute with Group
```

We also need more complex transformations, such as copying properties over an association, or merging entities. Although these can be modeled as separate transformations or transformation patterns [12], we recognize them to be similar to the already defined transformations, with the addition of being able to use other data model elements as input. So copying the topic from a Web to all of its pages is similar to adding a topic property to every page, with its web topic as a source:

```
Entity[Id="Page"] / Property :: add Web/Property[Id="topic"]
```

Elementary transformations are combined by sequential composition indicated by a semi-colon. Figure 2 shows the result of applying the above transformations to the Wiki data model in Figure 1.

3.2 Deriving Data Migrations

We have defined the data models and the data model transformation. The last step is therefore to specify a 'data model transformation' to 'data transformation' mapping, as indicated by the dashed arrow in Figure 4. The implementation of the mapping depends on numerous factors, such as how the data is stored, what platform is available to execute the data transformation on and the quantity of the data. The implementation is therefore driven by the context. We have implemented the mapping using Stratego/XT [21] and a data model to Java classes mapping from the WebDSL project [22]. It maps the data model transformations as shown above to a data migration program in Java. The migration program loads objects from a database, transforms them to conform to the new data model and stores the new objects. It follows a so-called Extract-Transform-Load (ETL) process.

The migration mainly uses two libraries, namely an object to relational mapping and an object transformation library. The first library provides functionality for loading and storing Java Objects in a relational database. The migration program is based upon the Java Persistence API (JPA) [20], which provides an interface to accessing these types of libraries. Consequently, any JPA compliant library is suitable. An example of such a library is Hibernate [11]. The combination of JPA and for example Hibernate supports a large number of database systems. The second library provides functionality for transforming Java objects and managing these transformations. It supports transformations such as adding attributes, changing attribute types and changing attribute names, but due to Java restrictions, the set of transformations does not directly cover the elementary transformation set we have seen above. We have written the transformation library specifically for the mapping, but it could also be used in different settings.

In the remainder of this section, we introduce the mapping using the examples we have presented above. Although the mapping directly refers to the transformation library, we use a domain specific language (DSL) for the library to abstract away from the underlying Java and JPA details. Nevertheless, the DSL can directly be mapped onto executable Java code.

Basic concepts. The group addition, introduces most of the basic concepts. It is mapped to the following:

```
transform () to (Group)  {
   EmptyObject();
   AddAttributeByValue("name", "Group Name");
   AddAttributeByValue("description", "...");
   AddAttributeByValue("members", null)
}
```

The `transform` directive defines a transformation as follows:

```
transform INPUT-TYPES to OUTPUT-TYPE
   {   TRANSFORMATION-DEFINITION   }
```

In the Group addition, `INPUT-TYPES` is empty and `OUTPUT-TYPE` is a Group. The transformation itself starts with an empty object (an object with no attributes) and subsequently adds the various attributes of Group. The `AddAttributeByValue` directive has the new attribute's name as first parameter and the attribute's value as second.

Similarly, the description removal is mapped to:

```
transform (Group) to (Group)  {
   DropAttribute("description")
}
```

Annotations. In the data transformations above, we carelessly introduced values for each of the attributes (`"Group Name"`, `"..."` and `null`). These are required in a data transformation, but unknown in the data model or data model transformation. Such information can be considered to be a separate input of the mapping, yet at the same time, storing it separately from the data model transformation would be impractical.

As a solution, we allow data model transformations to be annotated. When transforming data models, these annotations can be ignored, but when looking at data transformations, annotations provide the information we were missing. Instead of writing:

```
Entity[Id="User"] :: add age :: int
```

to add an attribute `age`, we therefore write:

```
Entity[Id="User"] :: add age :: int       defaultValue(25)
```

Using a similar approach we specify the value of a newly added group.

Data-level computations. Copying the topic property from a web to a page is done by an attribute addition. The attribute addition by constant value we have seen above is not sufficient. We need an attribute addition by computed value here. The computation itself is a parameter to the attribute addition:

```
transform (Page) to (Page)  {
  AddAttribute("topic", getWeb().getTopic() )
}
```

In Java, the computation is represented by an anonymous class.

The substitution from the previous section indicates a type substitution. At the data level this is reflected by a conversion to a value of the new (substituted) type. There are various type substitutions that have a standard value conversion. Examples are int to string, string to int, but also set of int to set of string. We have explicitly included the conversions for these substitutions in our mapping.

The substitution from the previous section is a set of User to a set of Group substitution. Such a substitution does not have a standard value conversion. To still be able to execute the data transformation, the user is required to explicitly specify the desired conversion by means of an annotation[1]. An example conversion would be to convert our set of user to a set of singleton groups in which each group holds exactly one user. This is mapped to:

```
transform (Web) to (Web)  {
  AttributeSetConversion("admin", new Group( getUser().getName(), { getUser() } ) )
}
```

[1] Not by means of a parameter, as this computation only influences the data, not the data model.

Note that the name `AttributeSetConversion` indicates that the conversion (the second parameter) is applied to each of the elements in the admin set, not to the set as a whole, which would be the functionality of the `AttributeConversion` transformation.

Types. Each of the above transformations refers to types. They have a set of source types and a target type. Since the code above is directly mapped to Java, these should represent actual Java types. The final step of the mapping is therefore to construct a Java type base to support the transformations. To establish the type base, we use a Java type generator from the WebDSL project, that takes a data model as input and produces the corresponding Java classes as output. Without investigating the transformations, we generate all types for the data model before transformation as well as all types in the data model after transformation. The resulting types are stored in different Java packages to prevent name clashes.

In addition to the source and target types, we also sometimes need types half-way through the transformation (e.g. when using the attribute addition by computation). These are generated when performing the mapping.

4 Heterogeneous Coupled Transformation

When data models evolve, conforming data needs to be transformed to reflect these changes. Although a frequently reoccurring approach is to define data transformations manually, it requires a significant effort and can hold back a development or maintenance process. We have shown that it can be performed automatically. In this section we step away from the detailed look on data model transformations and take a broader look at the problem from a higher level of abstraction.

4.1 Horizontal Generalization

Recall Figure 4. It shows the outline of the coupled data evolution problem. We have looked at Object Oriented data models describing data in a data base. If we would have described our data by means of a database schema (e.g. SQL schema), we would have an evolving database schema and data that has to be transformed to reflect these changes [4,2,8], as shown in Figure 5. The problem of coupled data evolution therefore reoccurs when using a different formalism for describing our data.

Similarly, we could use XML to store our data and DTD's to describe it. Again our DTDs evolve to satisfy changing requirements and our XML data needs to be transformed to reflect these changes [15], as shown in Figure 6.

We also see the same problem reoccurring in different domains. When programming languages evolve, the programs written in it have to be migrated to the new version of the languages. The programs have to conform to the grammar of the programming languages [18] (Figure 7). Similarly, when meta-models evolve, conforming models need to be transformed to reflect the evolution [7,24,6,23,12,9].

Coupled evolution is a reoccurring phenomenon. Naming conventions for the coupled evolution problem vary in the different areas between co-evolution, two-level

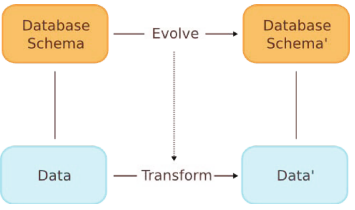

Fig. 5. Schema evolution

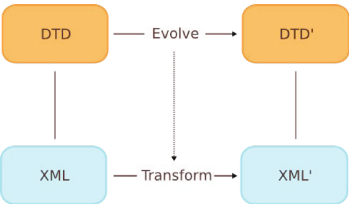

Fig. 6. DTD evolution

data transformations, coupled transformation and simply synchronization or adaptation. But they effectively address the same problem of coupled evolution. Lämmel discusses this for a subset of the above in [14], naming it the ubiquity of coupled transformation problems. Identifying the coupled evolution problem in different domains is a form of *horizontal* generalization.

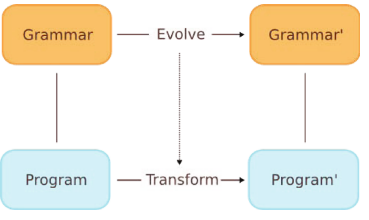

Fig. 7. Grammar evolution

4.2 Vertical Generalization

We introduced the data model language on the fly in the previous sections. We thereby implicitly defined its syntax. The syntax is formalized by the grammar found in Figure 8. It is written in SDF [**?**] format. For simplicity, it only shows the context-free production rules. The lexical syntax definitions and the start symbol (`DataM`) definition have been left out. On the left-hand-side of each production rule the construction of the specific sort is defined, on the right-hand-side the produced sort. Each of the rules are annotated, which is indicated by the `{...}` text at the end of each rule. We ignore these annotations for now.

```
Entity*          -> DataM {cons("Model")}
Id "{" Prop* "}"-> Entity{cons("Entity")}
Id "::" Type     -> Prop  {cons("Prop")}
"int"            -> Type  {cons("Int")}
"bool"           -> Type  {cons("Bool")}
Id               -> Type  {}
"set of" Type    -> Type  {cons("Set")}
Name             -> Id    {cons("Id")}
```

Fig. 8. Data model grammar

As data conforms to a data model, data models conform to the data model grammar. The grammar describes the structure of the data model and the data model describes the structure of the data. The data model grammar itself is rather limited. In future, it may for example be useful to add support for more attribute types, inheritance, or support for uniqueness of property values. So, in practice, the grammar is far from fixed and is itself subject to expansion and modification. No different from the data model scenario, if the grammar changes, data models that originally conformed to it are invalidated and need to be migrated along with the grammar. In other words, we have a second scenario of coupled evolution in the single context of data models. Figure 9 shows the extension with the additional evolution scenario.

We cover another conformance level by saying that also the SDF syntax may be subject to change, at which point, the grammar defined above has to be migrated along

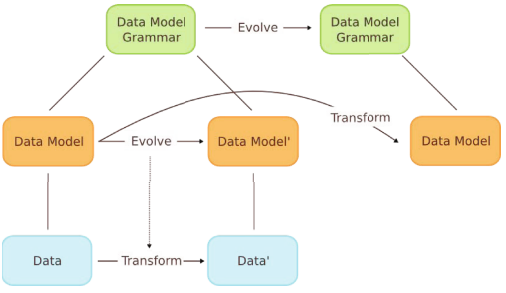

Fig. 9. Vertical generalization

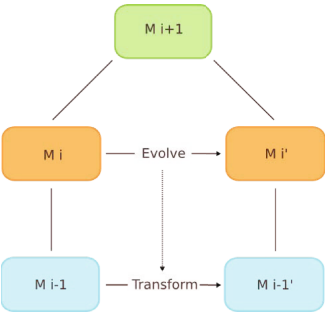

Fig. 10. Software language evolution

with the changing SDF definition. From which we see that the same coupled software transformation problem reoccurs over different conformance levels, which is a vertical generalization of the problem. In MDA [19] terms the vertical generalization can be phrased as coupled evolution on the levels M_1-M_0 (data model - data), M_2-M_1 (data model grammar - data model), M_3-M_2 (SDF - data model grammar), or even higher if M_3 is not defined in itself.

To abstract away from a specific conformance level and from specific areas of application, we will from now on use a generalized representation of the problem as shown in Figure 10. In this generalized view, we see the common aspects of coupled software language evolution:

- An evolving software language (M i)
- Software that is subject to transformation to reflect the evolving language (M i-1)
- A means to define software languages (M i+1)

For the case of data model evolution, we have automated the transformation process. To do the same in the generic case, we need a way to formalize the evolution for an arbitrary software language. Furthermore, we need a mapping from the language evolution to a concrete transformation.

5 Generic Architecture

In this section we propose and outline a generic architecture for coupled software evolution. Its goal is to reduce the manual effort involved in traditional coupled evolution. Furthermore, it structures the evolution process, increases the transformation abstraction levels and allows for common problems to be solved once instead of repeatedly.

Traditional approaches to coupled evolution are usually based on architectures similar to the one in Figure 4. The generic solution is based on the generalized and extended architecture displayed in Figure 11. The main component in the architecture is the definition of the transformation language used to formalize the evolution (named DSTL). In earlier work, the transformation language is usually fixed and considered to be an assumption of the approach. We assume it to be variable and consider it an artifact in coupled evolution.

Although the transformation language stands out most in the figure, the key concept of the architecture lies in the added arrows. The dashed arrow denotes a transformation defined by the user. The solid arrows denote automatic transformation, these do not require human interaction.

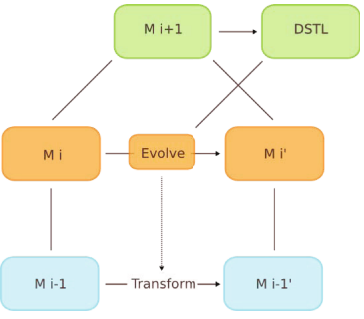

Fig. 11. Generic architecture

Input to the architecture is a coupled evolution scenario as explained in the previous section and a mapping from a top-level transformation to a bottom-level transformation. The first should come for free (either implicit or explicit), since it is merely defining what is to be evolved. Without it, the coupled evolution problem does not exist. The second is also part of most domain-specific approaches and may to that respect be reused in these specific domains.

The mapping, or dashed line in the figure, is defining a semantic link between the two levels, which is by definition sufficient to allow for coupled evolution. Since it is indirectly based upon M i+1, it is generic over any software (or model) being evolved within the same domain. We therefore have fixed mappings for the domain of data model evolution, or the domain of SDF evolution. In practice, the evolution scenario is therefore the main input, varying most frequently.

Based upon the two inputs, the architecture provides a structured approach to software language evolution, consisting of:

- Automatic derivation of a transformation language for each domain
- Automatic derivation of an interpreter for transformations in the transformation language
- Automatic software migration along a specified transformation

In practice there is a non-obvious ordering in these items, different from the ordering above. One needs the transformation language to define the mapping. Yet, since the transformation language derivation is automatic, the ordering will not be a problem in practice.

The following subsections focus on the aspects of the architecture individually.

5.1 Deriving Domain Specific Transformation Languages

The first and most central component of the architecture is a transformation language specific to the M i+1 definition. We will refer to it as the Domain Specific Transformation Language (DSTL). The transformation language cannot be generic, as we cannot construct a complete mapping from a generic language. Generic languages contain by definition concepts that are not part of the domain[2].

[2] Having a partial mapping is similar to using an implicit domain specific language. The language is defined by the domain of the mapping.

Many of the traditional transformation languages for coupled evolution define a large set of elementary transformations and an extensive mapping. In contrast to this, we focus on a transformation language that is as small as possible, but still covers all transformations. This makes defining the mapping as easy as possible. Usability of the language is achieved through abstractions. We have implemented the DSTL derivation in Stratego/XT and assume M i+1 to have been defined in SDF. The DSTL syntax is again defined in SDF.

Input to the derivation is M i+1, the software language grammar (in SDF). The derivation produces elementary transformations from the production rules in the grammar. It starts at the productions of the start symbol and traverses the grammar recursively. We distinguish different types of production rules, for which different types of elementary transformations are generated.

Lists. The top-most production rule in the data model grammar (Figure 8) defines a data model to be a list of entities:

```
Entity*          -> DataM
```

In the transformation language, the list is reflected by two list operations, namely addition and removal of entities. The syntax for these transformations is defined by the context-free productions:

```
"add" Entity     -> Transformation
"remove"         -> Transformation
```

In the same way, the addition and removal of properties are generated when considering the Entity production recursively. Furthermore, we generate transformations for optional symbols in a similar way (these are set and unset transformations).

Lexical Syntax. When a symbol is defined to be lexical, it has no more productions and can thus not be decomposed further. The recursion therefore stops and a transformation is generated to substitute its value. An example of a lexical symbol is `Name`, for which the following transformation is generated:

```
"substitute with" Name -> Transformation
```

Multiple Productions. The symbols that are considered above are either lexical, or produced by a single production. The `Type` symbol inside a property can be produced in multiple ways (`"int"`, `"bool"`, `"set of Type"`, or `Id`). Consequently, we must allow it to be substituted by one of these:

```
"substitute with" Type -> Transformation
```

We import the original data model grammar into the DSTL definition to reuse the `Type` symbol that was defined in the original grammar.

Type checking. A software language defines groups of software elements. For data models we have entities, properties, ids, but also ids inside sets, or ids inside a property. The local transformations defined above are only applicable to some of the element groups, which make up the domain of a local transformation. For example, the addition of entities can only be applied to data models and the substitution of types only to properties or sets.

We use APath expressions to indicate where a local transformation is to be applied. Each APath expression results in certain groups of software elements. To make sure a local transformation is applied within its domain, we need to verify that the APath expression to which it is connected can only result in elements that are in the domain of the local transformation, which is a form of type checking.

We have implemented type checking for any DSTL. It primarily consists of three components: A generation of domains for each of the local transformations. A (generic) type derivation for APath expressions and (generic) functionality that checks whether the result of the APath will indeed fall inside the domain of the local transformation. The type checking is complicated by the use of recursive productions: the `set of int` should fall in the same group as the set inside `set of set of int`.

Larger grammars. The presented data model grammar is small. We have used a much larger data model grammar, which was developed as part of the WebDSL project. Although the principles above can be applied to all the rules in a larger grammar, in practice, one does not want to be able to transform every group of software elements. In the small grammar, we could for example leave out the type substitution within sets if we would not be able to map it to a data transformation.

By means of annotations on the production rules of a grammar, the user can indicate which rules (and thereby what symbols) should be transformed and which should not be transformed. There are two possible annotations: A 'transform' annotation, which tells the tool to generate transformations for a production rule and a 'constant' annotation, which tells the tool to take the production rule into account by recursively generating transformations for each of the symbols on the left-hand-side (in SDF), but not generating transformations for the production rule itself. No annotation on a rule means that it is ignored during DSTL derivation.

5.2 Automated Transformation

The DSTL syntax we have defined allows us to write transformations. The next step is to execute these transformations. For this purpose, we have defined an interpreter generator. Similar to the syntax generator, it takes the software language definition as input, but instead produces an interpreter for the associated DSTL as output. The interpreter mainly consists of:

- A mapping of the elementary DSTL transformations onto generic transformations
- A generic transformations library (build on top of Stratego/XT)
- Implementation of generic DSTL constructs such as composition and abstraction
- An APath evaluation library

The first is specific for the DSTL and therefore generated. It mainly consists of production rules that denote the specific to generic mapping. These look like:

```
transform(|mmodel, path):
   AddProperty(newValue) -> <addAtLocator(|path, newValue)> mmodel
```

The last three items are generic over all DSTLs, so defined once. Their definition is in most ways straightforward and is therefore not discussed here.

6 Related Work

Coupled evolution plays a significant role in computer science and has been treated in various areas. Earlier research has primarily focused on constructing coupled evolution support for specific domains. We discuss related work in the most important domains of coupled evolution: model evolution, domain specific language evolution and schema evolution.

Coupled evolution for the meta modeling domain is introduced by Gruschko [7]. As is the case for most publications on coupled evolution for models, Gruschko models evolution using small elementary transformation steps. A classification of these steps is proposed: non-breaking changes, breaking and resolvable changes and breaking and unresolvable changes. A classification, which is frequently reused in later work and is also applicable to our work, yet not directly relevant to the proposed architecture. In his paper, Gruschko also identifies different steps in coupled evolution, although these steps are generic, they mainly consider what we have called 'the mapping' and are in that sense only applicable to a subset of what has been discussed here. The only step, which does not fall inside the scope of this mapping is a change detection, to determine the evolution steps that have occurred between two given models. If automated, the change detection may provide a useful addition to our work as it reduces the number of required inputs. Yet existing change detections are usually specific to a certain domain and are only applicable within these domains.

Wachsmuth [23] introduces a set of transformations specific to MOF [16] compliant meta models. The set is very similar to the set of elementary transformations for data models as we have introduced in Section 3.1 and which is derived automatically in our approach. Different to the transformations we have derived is their distinction between two type changes, namely generalization and restriction, yet they do not provide a specification on how these should be mapped to concrete types. Furthermore, they have transformations to support changes to inheritance and inlining of classes. Both concepts were not included in our input data model grammar and are therefore not reflected in the output. Wachsmuth proposes a mapping to model migrations implemented in QVT [17].

Similar to Wachsmuth, Herrmannsdoerfer [10] considers coupled model evolution based on small evolution steps. He focuses on the Eclipse Modeling Framework (EMF) [5], in which ECore, the meta-meta model implements a subset of MOF. In his approach, named COPE, Herrmannsdoerfer distinguishes two types of evolution steps: open and closed coupled evolution. The first is what we have named the elementary transformations and the second are transformations based upon these. In contrast to other works, this view does provide a way of abstracting from meta-model specific transformations. However, the derivation of the elementary transformations as well as the definition of these transformations are left to the user. This does not only require additional effort, it also prevents structured abstractions as are possible in our approach. Herrmannsdoerfer provides a prototypical editor based on Eclipse.

In the area of domain specific languages, Pizka et al. discuss the evolution of DSLs. They claim three obstacles in DSL development: (1) Stepwise bottom-up generalization is required, which is a special case of the evolution we have been looking at. (2) DSLs should be layered, which is specific to DSLs and not directly related to coupled

evolution. (3) Automated co-evolution is required for DSLs, which is what we generically solve in our work. As we have seen in [23], Pizka's work is focused on a single domain, namely DSLs, it is limited to the discussion of a transformation definition and mapping specific to this domain.

With respect to the data variants of coupled evolution (schema evolution) and the related two-level data transformations, numerous approaches have been found to solve these problems [15,4,2,8,1]. These mainly focus on the schema to data mapping, frequently taking different types of complicating concepts into account, such as data restrictions and performance optimization. These are typically aspects that may also be solved generically, such that they can be used in any domain. In current work, we have not focused on this, but it may be interesting as future work.

7 Conclusion

In this paper, we presented two directions of generalizing coupled software language evolution scenarios and introduced the concept of heterogeneous coupled evolution. We presented an architecture to automate coupled evolution on an arbitrary software domain (e.g. programming languages, modeling or data modeling). The architecture requires as input: a coupled software evolution scenario and a mapping from software language transformations to software transformations. The outputs are: Automatic derivation of a domain specific transformation language (DSTL) to formalize the software language evolution; automatic derivation of an interpreter for transformations conforming to the DSTL; and automatic software migration along the evolving software language.

Using Stratego/XT, we have implemented a coupled evolution tool to support the architecture. It is based on software languages defined in SDF. We have succesfully applied the tool to the domain of data modeling in the web modeling language WebDSL [22], where we have used it to create a tool for automatic database migration along an evolving data model, which targets a broad set of databases.

Acknowledgments. This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

1. Alves, T., Silva, P., Visser, J.: Constraint-aware schema transformation. In: Ninth International Workshop on Rule-Based Programming (Rule 2008) (2008)
2. Berdaguer, P., Cunha, A., Pacheco, H., Visser, J.: Coupled schema transformation and data conversion for XML and SQL. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 290–304. Springer, Heidelberg (2006)
3. Clark, J., DeRose, S., et al.: XML Path Language (XPath). W3C Recommendation 16 (1999)
4. Cunha, A., Oliveira, J., Visser, J.: Type-safe two-level data transformation. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 284–299. Springer, Heidelberg (2006)
5. Eclipse Foundation. Eclipse Modeling Framework Project (EMF) (2008), `http://eclipse.org/emf`

6. Favre, J.: Meta-model and model co-evolution within the 3D software space. In: Evolution of Large-scale Industrial Software Applications (ELISA 2003), pp. 98–109 (2003)
7. Gruschko, B., Kolovos, D.S., Paige, R.F.: Towards synchronizing models with evolving metamodels. In: Workshop on Model-Driven Software Evolution at CSMR 2007 (2007)
8. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: International conference on management of data (SIGMOD 1993), pp. 157–166. ACM, New York (1993)
9. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)
10. Herrmannsdörfer, M.: Metamodels and models. Master's thesis, München University of technology, München, Germany (July 2007)
11. Hibernate. Relational persistence for Java and.NET (2008), `http://www.hibernate.org`
12. Hoßler, J., Soden, M., Eichler, H.: Coevolution of models, metamodels and transformations. In: Bab, S., Gulden, J., Noll, T., Wieczorek, T. (eds.) Models and Human Reasoning, Berlin, pp. 129–154. Wissenschaft und Technik Verlag (2005)
13. Janssen, N.: Transformation tool composition. Master's thesis, Institute of Information and Computing Sciences Utrecht University, Utrecht, The Netherlands (March 2005)
14. Lämmel, R.: Coupled Software Transformations (Extended Abstract). In: First International Workshop on Software Evolution Transformations (November 2004)
15. Lämmel, R., Lohmann, W.: Format Evolution. In: Reverse Engineering for Information Systems (RETIS 2001). books@ocg.at, vol. 155, pp. 113–134. OCG (2001)
16. Object Management Group (OMG). Meta Object Facility (MOF) Core Specification - Version 2.0 (January 2006)
17. Object Management Group (OMG). MOF QVT Final Adopted Specification (March 2007)
18. Pizka, M., Jurgens, E.: Tool-supported multi-level language evolution. In: Tomi Männistö, E.N., Raatikainen, M. (eds.) Software and Services Variability Management Workshop, Helsinki, Finland. Helsinki University of Technology Software Business and Engineering Institute Research Reports, vol. 3, pp. 48–67 (April 2007)
19. Soley, R., et al.: Model driven architecture. OMG white paper 308 (2000)
20. Sun. Java persistence api (2008), `http://java.sun.com/javaee/technologies/persistence.jsp`
21. Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 216–238. Springer, Heidelberg (2004)
22. Visser, E.: WebDSL: A case study in domain-specific language engineering. In: Lammel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2007. LNCS, vol. 5235. Springer, Heidelberg (2008)
23. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
24. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: Automated Software Engineering (ASE 2007), pp. 164–173. ACM, New York (2007)