A Semi-automatic Transformation Approach for Semantic Interoperability in MDE

Sixuan Wang[1], Brice Morin[2], Dumitru Roman[2], and Arne J. Berre[2]
SINTEF ICT
Forskningsveien 1, Oslo, Norway

[1]sam.wangsixuan@gmail.com [2]{firstname.lastname}@sintef.no

ABSTRACT

As data exchange and model transformation become ubiquitous nowadays, it is a key requirement to improve interoperability of enterprise systems at the semantic level. Many approaches in Model-driven Architecture (MDA) and Model-driven Interoperability (MDI) emerge to fulfil the above requirement. However, most of them still demand significant user inputs and provide a low degree of automation, especially when it comes to finding the mappings. A generic approach that can easily handle both semantic interoperability and automatic transformation is currently missing.

This paper presents AutoMapping, a semi-automatic model transformation architecture. This approach focuses on two aspects: 1) semi-automatic mapping between data models expressed as class diagrams by involving minimal user interactions at design-time; 2) generation of executable mappings. Particularly at design-time, a semantic engine that solves various kinds of semantic attribute mismatches is devised, such as type, scale, synonym, homonym, granularity, etc. Furthermore, a heuristic-based similarity analysis between each pair of classes is proposed, which takes all relations of classes into account, such as inheritance, reference, etc. Finally, a method is given to match fragments and then generate mappings specification that conforms the proposed mapping metamodel for solving existing semantic mismatches.

The main contribution of this paper is to create a generic platform-independent approach for semi-automatic model transformation towards semantic interoperability, with tool-based implementation and motivating case experiment, showing the feasibility of using MDA and MDI techniques for semantic interoperability.

***Keywords**: data exchange, model transformation, Model-driven Architecture (MDA), semantic interoperability, semi-automatic mapping*

1. INTRODUCTION

Nowadays, demands of integration between enterprise systems have changed from data exchange and model transformation, to interoperability among enterprise systems, especially when it comes to various semantic issues. Besides, improving the level of automation of data exchange between B2B systems is widely regarded as a key enabler for agile interoperability and scalability in B2B collaborations [1]. A generic approach that can easily handle both semantic interoperability and automatic model transformation is currently missing. In this paper we propose a generic and semi-automatic model transformation architecture towards semantic interoperability (called AutoMapping). Before we give a brief overview of AutoMapping, let us define the data exchange problem in more details.

Figure 1 provides an overview of the elements involved in a typical data exchange between two *companies X* and *Y* (adapted from [2]). The main challenge is how to transform data manipulated in *Source Instance* by *Company X* (conforming to *Source Metamodel*) to data manipulated in *Target Instance* by *Company Y* (conforming to *Target Metamodel*). A *Transformation Layer* is usually designed to address this challenge by providing means to map the *Source Schema* to the *Target Schema* at design-time, and by providing an environment that implements the schema mappings at run-time when the *Source Instance* needs to be transformed to *Target Instance*.

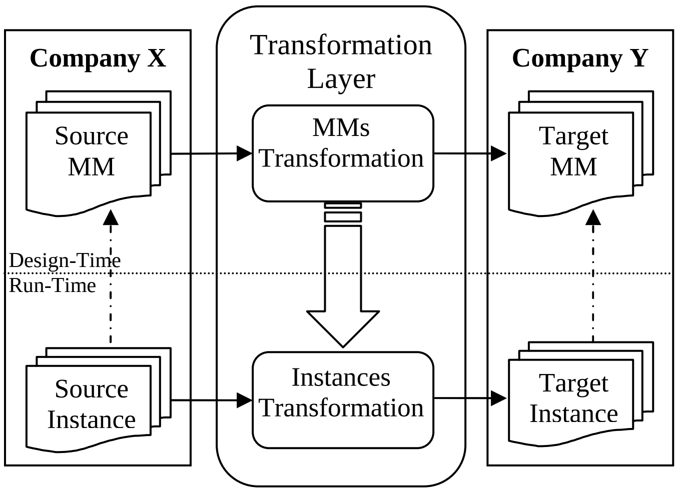


Figure 1: General design-time and run-time data exchange

When addressing the above problem of data exchange, interoperability issues arise due to the lack of consensus on the common standards to conform to and the shortage of proper approaches and supporting tools. Among these issues, semantic mismatches identified in [3], such as type, scale, precision, synonym, homonym, granularity and overage, are typical conflicts appearing during data exchanging. In this paper, these semantic mismatches are described with certain modification in table 1, with examples from a common supplier/seller scenario in Enterprise Resource Planning (ERP) systems. Therefore it is a key requirement to improve interoperability by solving these semantic mismatches when data exchanging.

Table 1: Semantic Interoperability Mismatches

Name	Description	Example	Functionality Need
Type	different data types	*Salary* in source is *Float*. *Salary* in target is *Integer*.	Conversions between different data types require type casting.
Scale	different measurements	*Currency* in source is *euro*. *Currency* in target is *dollar*.	An agreement between sources and targets, and conversions.
Precision	different accuracies	*Amount* in source is *2* decimals. *Amount* in target is *3* decimals.	An agreement between sources and targets, and conversions.
Synonym	different names	Abbreviate in source Short Name in target	Rename to same shared name.
Homonym	different contents	*Note* in source means *supplier descriptions*. *Note* in target means remark of *supplier companies*.	Rename to distinguished different names.
Granularity	different structures	*Address* information in source has three elements: *Address*, *Province*, and *Place*. *Address* information in target has a single element.	Rename using a agreement structure, to merge, split, etc.
Coverage	different ranges and intersections	Supplier Information in source has *name, group, note, location,* and *code*. *Supplier* Information in target contains less information: *name, group*, and *code*.	An agreement between sources and targets. Conversions using the intersection of sources and targets.

Many approaches to model transformation and mapping in Model-driven Engineering (MDE) and semantic annotation with ontology techniques are emerging to address the aforementioned requirement. However, most of them still demand significant user inputs and efforts, and lack automation, so the need for automatic model transformation is much high, especially in the following aspects:

1. **Mappings Finding**. The source and target metamodels share common concepts but might propose different ways to represent these concepts. At design-time, most of the mappings should thus be as much as possible automatically identified and not written by hand (time consuming, error-prone).

2. **Transformation Execution**. For the identified mappings specification to be usable at run-time for instance model transformation, executable transformations need to be automatically generated and executed.

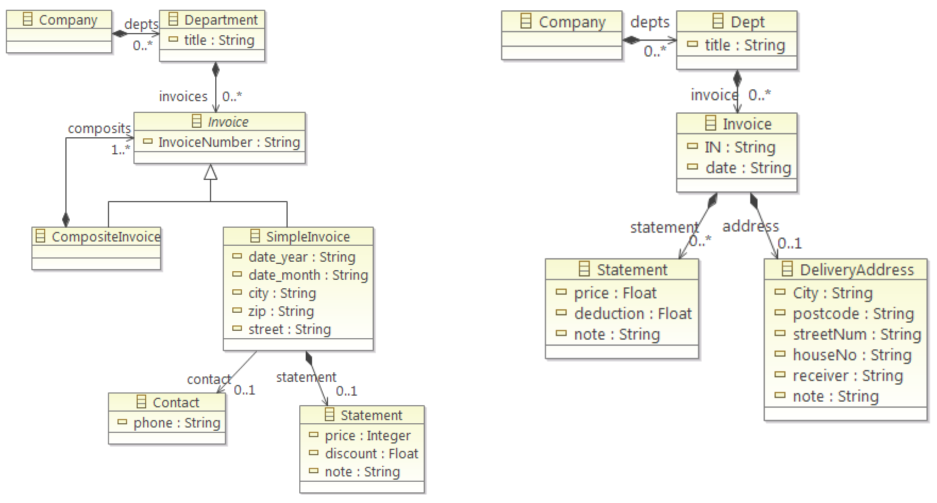
3. **Semi-automatic Transformation**. Because of specific standards and formats between enterprise systems, the fully automatic transformation is unrealistic; therefore, user interaction is needed to customize and control the mapping process, and semi-automatic transformation is such a solution in which necessary user interaction is kept at a minimum.

The remaining of this paper is organized as follow: Section 2 presents a motivating case derived from a practical industrial scenario. Section 3 details our semi-automatic model transformation architecture. Section 4 presents our AutoMapping approach that is based on the proposed architecture towards semantic interoperability. Section 5 introduces current implementation and some experiments of the proposed AutoMapping. Section 6 concludes this paper, together with some relevant related work and potential extensions.

2. MOTIVATING CASE

This section presents a case scenario for model transformation between two companies where invoices are sent from a source system to a target system. This example is derived from a practical industrial example of REMICS project (see http://www.remics.eu/), and it covers most semantic interoperability mismatches mentioned in table 1, and also requires high automation level for the specification and transformation of mappings. The rest work in the paper is based on this case, to show how AutoMapping approach works to solve the above requirements.

Both company invoice schemas are abstracted as ECore metamodel (see http://www.eclipse.org/modeling/emf/), which provides concepts like classes, attributes, inherence, relations, etc. The source company metamodel is depicted in figure 2. *Company* contains some *Department* (with *title*), and *Departement* has some *Invoice* (with *InvoiceNumber*). *Invoice* can be either *SimpleInvoice* (with *data_year*, *data_month*, *city*, *zip*, *street*) or *CompositeInvoice*. *SimpleInvoice* contains zero or one *Statement* (with *price*, *discount*, *note*) and a single reference of *Contact* (with *phone*), while *CompositeInvoice* contains at least one *Invoice* (*SimpleInvoice* or *CompositeInvoice*). Particularly in *Statement*, *price* is *Integer* with *Euro* unit, and *discount* is *Float* with *3 decimal* precision. For the target company metamodel, it is depicted in figure 2-5. *Company* also contains some *Dept* (with *title*), and *Dept* has some *Invoice* (with *IN* and *date*). *Invoice* contains some *Statement* (with *price*, *deduction*, *note*) and *DeliveryAddress* (with *City*, *postcode*, *streetNum*, *houseNo*, *receiver*, *note*), Particularly in *Statement*, *price* is *Float* with *USD* unit, and *deduction* is *Float* with *2 decimal* precision.



Source Company MM **Target Company MM**

Figure 2: Source Metamodel and Target Metamodel

The two companies want to exchange invoice data and improve the interoperability between them. Though the two metamodels are very aligned (e.g. *title* in *Department* of Source metamodel and *title* in *Dept of* Target metamodel), they have some semantic mismatches showed in table 2. The first challenge of this case is how to identify these semantic mismatches and solve them in certain automation level.

Table 2: Corresponding Semantic Mismatches

Name	Corresponding in the case
Type	? Data type of *price* in *Statement* of source is *Integer*, ? Data type of *price* in *Statement* of target is *Float*.
Scale	? Unit of *price* in *Statement* of source is *Euro*, ? Unit of *price* in *Statement* of target is *USD*.

Precision	? *discount* in *Statement* of source is calculated in *3 decimals*, ? *reduction* in *Statement* of target is calculated in *2 decimals*.
Synonym	? *title* in *Department* of source, *title* in *Dept* of target. ? *price* in *Statement* of source, *price* in *Statement* of target. ? *invoiceNumber* in *Invoice* of source, *IN* in *Invoice* of target. ? city in *SimpleInvoice* of source, *City* in *DeliveryAddress* of target. ? *note* in *Statement* of source, *note* in *Statement*. of target.
Homonym	? *note* in *Statement* of source, ? *note* in *DeliverAddress* of target.
Granularity	? Merge: *date_year* and *date_month* in *SimpleInvoice* of source; *date* in *Invoice* of target. ? Split: *street* in *SimpleInvoice* of source; *steetNum* and *houseNo* in *DeliveryAddress* of target.
Coverage	? *contect* in *Simpleinvoice* of source. ? *receiver* in *Invoice* of target

To improve the level of automation, the second challenge of this case is how to find the similarity of each class pair of source metamodel and target metamodel. And then find possible fragments (various combination patterns, initial mappings) and mappings specification of them. Besides, for the semi-automatic transformation, user interaction would be involved in the specification and execution of mappings but should be kept at a minimum.

3. GENERIC ARCHITECTURE OF SEMI-AUTOMATIC TRANSFORMATION

The ultimate goal is to create a generic approach of semantic mapping and similarity analysis at a platform-independent level for solving semantic mismatches and generating mapping rules with minimal user involvement. To achieve this goal, we present a generic semi-automatic model transformation architecture, which is evolved from ExchangeMap [4]. Figure 3 shows its architecture. The shadowed box in the center of the figure (which refers as *AutoMapping Framework*), is the core part of the architecture, where all the semantic engine, mappings specification and transformations take place at the platform-independent level. AutoMapping focuses on semi-automatically generating mappings between class diagrams by involving minimal user interactions at design-time, and then running executable mapping rules on concrete instances (such as XML files) at run-time. This approach provides a detailed solution for solving semantic mismatches and improving automation of transformation by using similarity analysis and mapping fragments discovering. The detailed design of AutoMapping Framework is described in section 4. The elements outside the shadowed box (*source* and *target schemas*, their *transformation*, the *source instance*, and the generated *target instance*) represent platform-specific models and transformations.

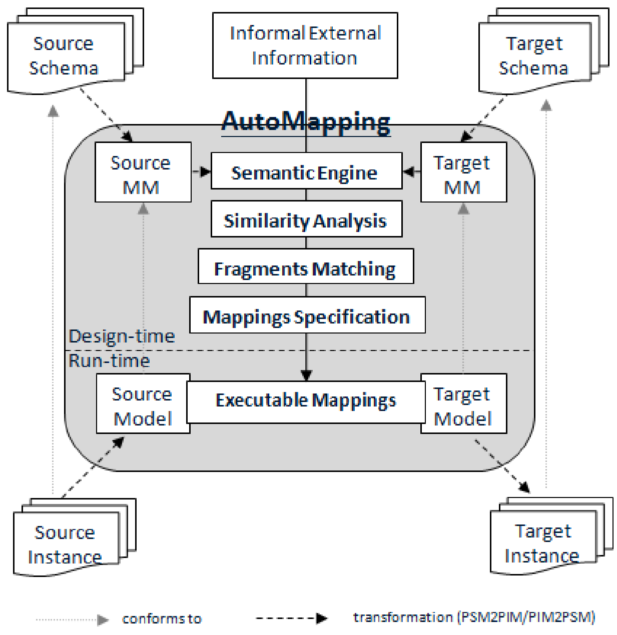


Figure 3: Generic Architecture of Semi-automatic Transformation

Another way of looking at the architecture is through its design-time and run-time elements. The following process takes place when using the architecture for model transformation:

At design-time:

1. The platform-specific *source* and *target schemas* are abstracted into platform-independent *source* and *target metamodels* through a given transformation (*PSM2PIM*) specific to the concrete technologies used at the platform-specific level.

2. By using *AutoMapping*, the mappings between the source and target metamodels are found and specified, based on *Semantic Engine*, *Similarity Analysis*, *Fragments Matching* and *Mappings Specification*.

3. *Executable mappings* are generated from the mappings specified in the previous step, and will be used during the run-time data exchange.

At run-time:

4. The platform-specific *source instance* is abstracted into a source model through a given transformation (*PSM2PIM*) specific to the concrete technologies used at the platform-specific level.

5. The *executable mapping rules* from step 2 are executed for the source model and a target model corresponding to the source model is generated.

6. The target model is serialized into a platform-specific *instance target* through a given transformation (*PIM2PSM*) specific to the concrete technologies used at the platform-specific level.

4. AUTOMAPPING APPROACH

The focus of this paper is to design and implement AutoMapping (the core part of Generic Architecture of

Semi-automatic Transformation proposed in figure 3.

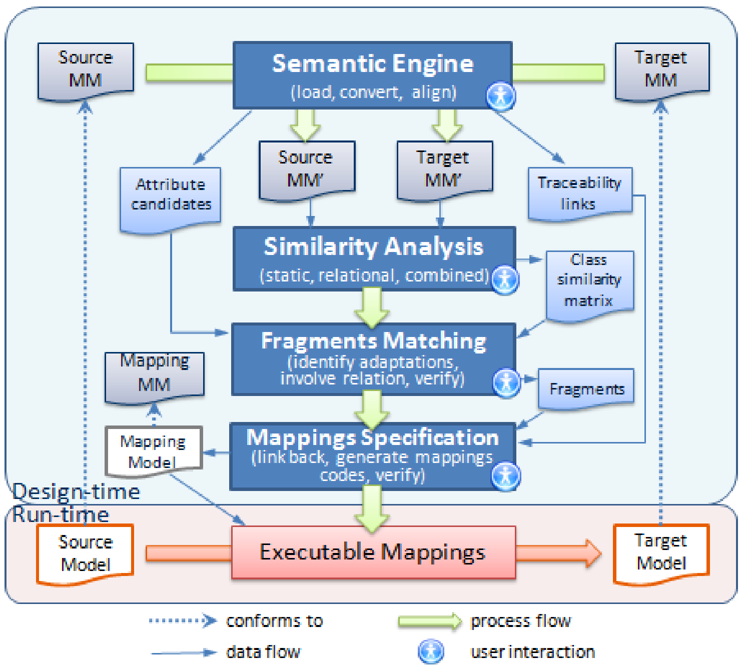


Figure 4: Framework of AutoMapping

Figure 4 illustrates the framework of AutoMapping. In AutoMapping, source and target metamodels (e.g. XSDs, database schemas, etc.) are abstracted as platform-independent ECore models (*Source MM* and *Target MM* in the figure). ECore provides powerful object-oriented mechanisms to data modeling, such as inheritance and composition; at the same time, it is much easier to be transformed to/from platform-dependent metamodel due to its widely applicability and coverage. Similarly, the source instance model that conforms to *Source MM* is abstracted to platform-independent model (*Source Model* in the figure), which could be processed by the generated executable mappings code at run-time to generate a corresponding platform-independent target model (*Target Model* in the figure). In the current version of AutoMapping, the instance models are represented as XML files, but they can also be other format as long as conforming to metamodels.

The main steps of AutoMapping are the following five steps:

1. **Semantic Engine**: based on its algorithm and *informal external information* from user, can identify most semantic mismatches between the attributes of metamodels, and then provides solution suggestions by adding new information or renaming operations.

2. **Similarity Analysis**: uses modified heuristic bi-similarity algorithm to analyze similarity matrix of each class pair, then get the class pairs with higher similarities.

3. **Fragments Matching**: finds possible fragments based on analyzed information of above two steps by refactoring inheritances and references.

4. **Mappings Specification**: using the *fragments* analyzed above, it generates a concrete *mapping model* that conforms to the proposed *mapping metamodel* and *executable mappings* for run-time transformation.

5. **Run-time Transformation**: based on *executable mappings* generated from *Mapping Specification*,

source model instance is transformed to *target model instance.*

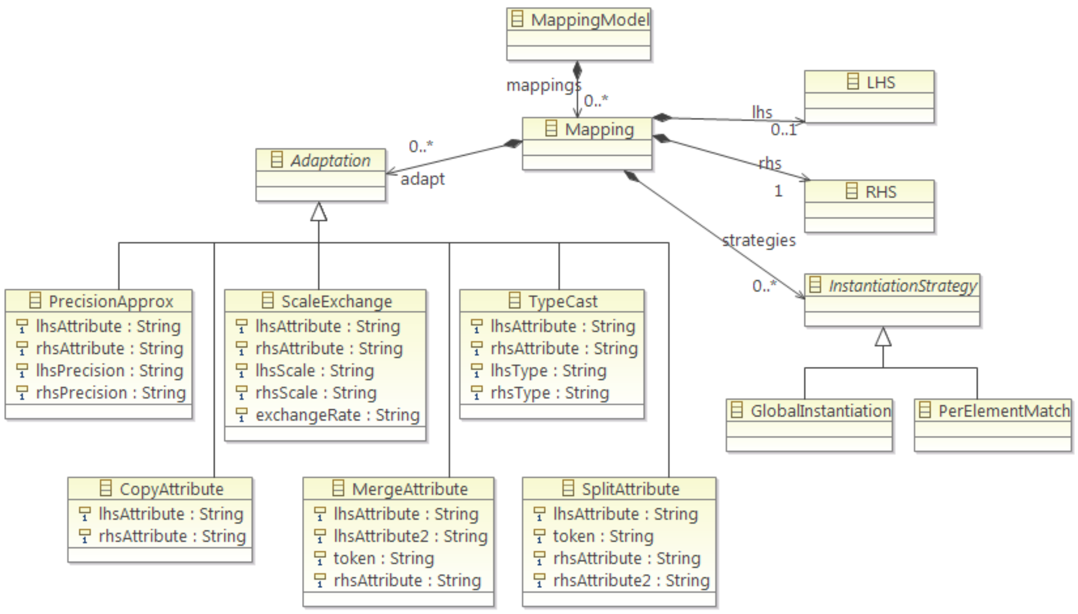


Figure 5: Mapping Metamodel

Before introducing the detailed design of each step, let us see the devised mapping metamodel that AutoMapping relies on. At design-time, the purpose is to find the mappings that conform to the mapping metamodel. Though mappings are mainly generated in the step of mapping specification, its concepts and elements are used in the other previous steps. The mapping metamodel, depicted in figure 5, is inspired by graph-based approaches [5] and Aspect-Oriented Modeling approaches [6]. This mapping metamodel is simplified and expended for AutoMapping purpose.

The basic idea is to describe mappings between metamodels, each mapping has a Left-Hand-Side (*LHS*, represented classes fragment of the *Source MM*) and a Right-Hand-Side (*RHS*, represented classes fragment of the *Target MM*). Also, each mapping has a set of mapping *Adaptations*, which represents the mapping attributes behaviors of *LHS* and *RHS*. In the current version of this mapping metamodel, it contains six concrete *Adaptations* which are to be applied in *Semantic Engine* and *Fragment Matching*, including *CopyAttribute*, *PrecisionApprox*, *ScaleExchange*, *TypeCast*, *MergeAttribute*, and *SplitAttribute*. The mapping metamodel is easily extensible and users can customize it as needed (e.g. add new adaptation). Since the *LHS* can match multiple times, instantiation strategies are added that introduced in [6]. This allows that to control the way the elements of the *RHS* should be instantiated: every time the *LHS* matches, once, etc. By default, all the elements of the *RHS* are instantiated every time the *LHS* matches. More details on these strategies can be found in [6].

Figure 6 illustrates a simple example to show how the mapping would be like when using the mapping metamodel. As discussed above, the mappings should be the combination of *FirstName* and *LastName of Company X* corresponds *name* of *Company Y*, and *salary* of *Company X* corresponds salary *of Company Y*. In figure 6, here is actually only one mapping (*Mapping1*). Its LHS are *Employee* and *FullName*, RHS is *Person*, which means if found *Employee* and *FullName* structure in the LHS, then generates *Person* in RHS. In this mapping, there are two adaptations: 1) *MergeAttribute* with default linking *token* “ ” from *firstName* and *lastName* in *FullName* to *name* in *Person*; 2) *ScaleExchange* with default *exchangeRate 1.4* (from € to $) from *salary* of *Company X* to salary *of Company Y*. Finally, at run-time, the instance model transformation executes according to the mapping, which is shown in the *Instance Models* part.

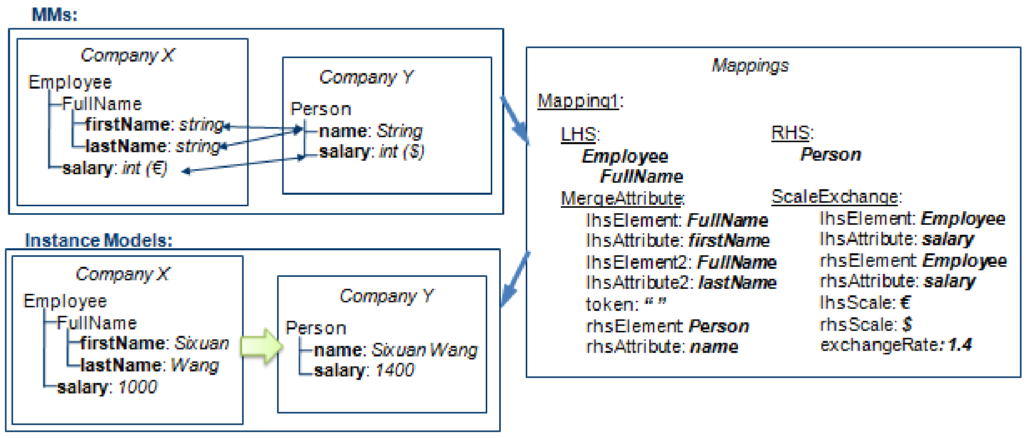


Figure 6: Simple Example of Mapping Metamodel

4.1 Semantic Engine

The first step of AutoMapping is *Semantic Engine*, it focuses on finding and giving solutions for semantic mismatches of metamodels on the attributes level, by using *alignment* and *conversion* semantic operations, generating new metamodels (*MM'*) as well as maintaining related *traceability links* and *attribute candidates*.

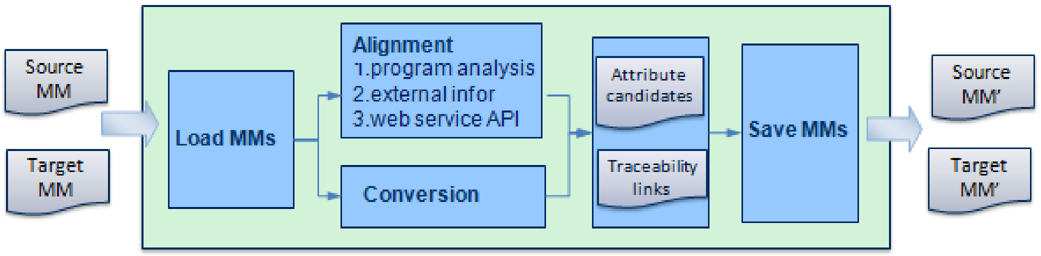


Figure 7: Semantic Engine Process

As depicted in figure 7, *Semantic Engine* has the following four main sub-steps:

1) **Load Metamodels**: This step is to load metamodels using available API in ECore.

2) **Alignement**: *Alignment* is one of semantic mismatches operations. The purpose is to analyze the attribute candidates for *identical* (exactly the same), *synonym* (same meaning but different names), *homonym* (same name but different meaning), *granularity* (need merge/split). Three methods are devised for finding possible attribute candidates for alignment:

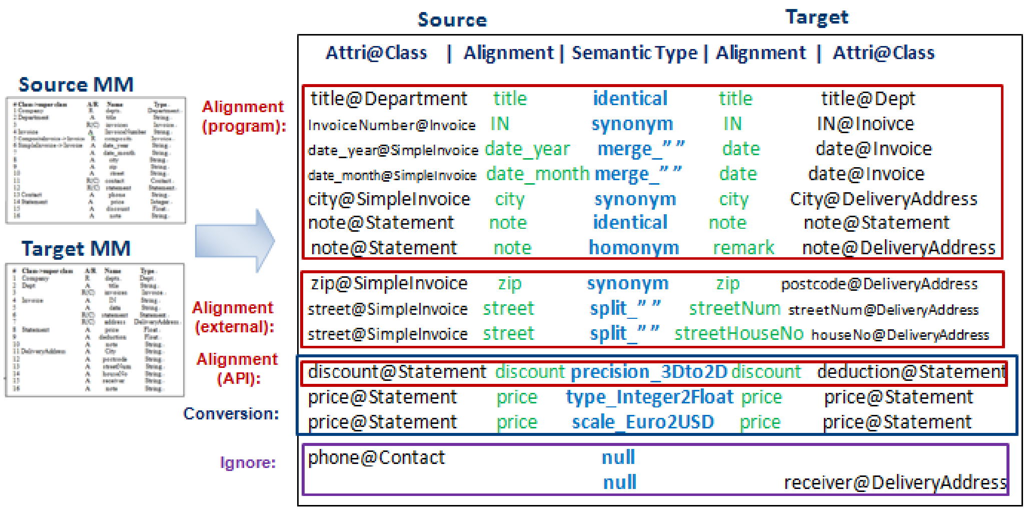
- Program analysis: program uses itself algorithm to automatically align the possible attribute candidates, (e.g. *identical, capital, abbreviation, substring,* etc).
- External information: since some companies maybe have their own specific standard between the metamodels, user can simply input his informal external information file (e.g. *xls*, *xml*, *database*, ect.) to enable the program know particular agreements between the metamodels (e.g. *synonym list*, *granularity list*, etc).
- Web service API: several on-line web services support user to access available resources to support semantic interoperability, e.g. *Big Huge Thesaurus of Princeton University* (see http://words.bighugelabs.com/about.php/), *Abbreviations* (see http://www.abbreviations.com/)

and *Acronym Finder* (see http://www.acronymfinder.com/).

3) **Conversion**: *conversion* is one of semantic mismatches operations. The purpose is to identify the attribute candidates with specify semantic type for type, scale, and precision of semantic mismatches. Additionally, the semantic type would be attached with some information to indicate the conversion detail. For example, the semantic type for scale conversion from *Euro* to *USD* would be specified to *type_euro2usd*, similarly *type_Integer2Float* for *type* and *precision_3Dto2D* for *precision*.

4) **Save new metamodel**s: the last step of semantic engine is to save the aligned and converted attributes in new metamodels (Source MM' and Target MM') for supporting the following other steps in AutoMapping.

For the motivating case, after above *alignment* and *conversion* operation, the possible attribute candidates are found, with *attribute name*, *owner class*, *new aligned name*, and *semantic type*. The example result of this part is illustrated in figure 8. For the clarity purpose, this figure show both of *attribute candidates* with semantic type and the *traceability links* (original attribute and its aligned attribute) each row.


Source MM
Target MM
Source
Target

	Attri@Class	Alignment	Semantic Type	Alignment	Attri@Class
Alignment (program):	title@Department	title	identical	title	title@Dept
	InvoiceNumber@Invoice	IN	synonym	IN	IN@Inoivce
	date_year@SimpleInvoice	date_year	merge_" "	date	date@Invoice
	date_month@SimpleInvoice	date_month	merge_" "	date	date@Invoice
	city@SimpleInvoice	city	synonym	city	City@DeliveryAddress
	note@Statement	note	identical	note	note@Statement
	note@Statement	note	homonym	remark	note@DeliveryAddress
Alignment (external):	zip@SimpleInvoice	zip	synonym	zip	postcode@DeliveryAddress
	street@SimpleInvoice	street	split_" "	streetNum	streetNum@DeliveryAddress
	street@SimpleInvoice	street	split_" "	streetHouseNo	houseNo@DeliveryAddress
Alignment (API):	discount@Statement	discount	precision_3Dto2D	discount	deduction@Statement
Conversion:	price@Statement	price	type_Integer2Float	price	price@Statement
	price@Statement	price	scale_Euro2USD	price	price@Statement
Ignore:	phone@Contact		null		
			null		receiver@DeliveryAddress

Figure 8: Example of Alignment and Conversion after Semantic Engine

4.2 Similarity Analysis

After getting new metamodels with aligned attribute names from semantic engine, most semantic mismatches are identified and given possible solutions. However, the semantic engine cannot answer the coverage mismatch, which should be analyzed in class level rather than attribute level. For automation requirement, it is necessary to find their *similarity degrees* of pairs of classes, by combining the similarity percentage of *Static 1-to-1 Similarity Analysis* (which focuses on single classes with their names of attributes) and *Relational Iteration Analysis* (which focuses on bi-similarity of the relations of subclass, super class, composition, etc.), so that by filtering with certain threshold, the pairs of classes with higher similarity can be used for following fragment matching for solving coverage mismatch.

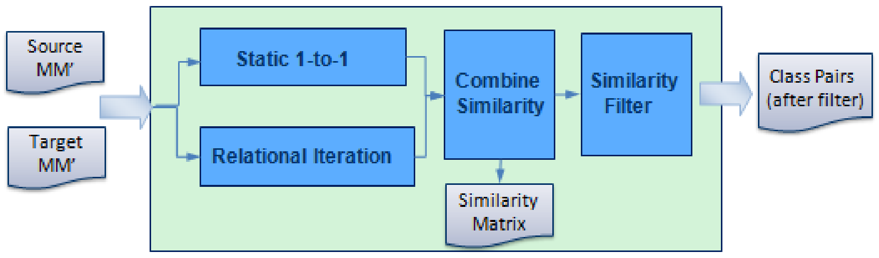


Figure 9. Similarity Analysis Process

As illustrated in figure 9, *Similarity Analysis* has the following four main sub-steps:

1) **Static 1-to-1 analysis**: to calculate the similarity degree of each class pair-to-pair, and then save them in a *Matrix S* (represents each class pair similarity value between 0 and 1). Besides, static 1-to-1 does not only consider attribute name, but also take class name and reference name into account. Static 1-to-1 analysis devises the following formula:

$$S(X_i, Y_j) = 2 \times \frac{|set(x_i) \cup set(y_j)|}{|set(x_i)| + |set(y_j)|}$$

- X_i or Y_j represents one class in *source MM'* or *target MM'*. $S(X_i,Y_j)$ is similarity value of class pair X_i and Y_j.
- $set(X_i)$ or $set(Y_j)$ is the set of names of X_i or Y_j, which includes class name, attributes (including inherited attributes) and reference class names, without the reduplicate names.
- $|set(X_i)|+|set(Y_j)|$ is the size of the set of names.
- $|set(X_i)\ U\ set(Y_j)|$ is the size of the intersection with the same names (or one is substring of the other) of the two sets.

2) **Relational iteration similarity**: this paper applies a variation of the bi-similarity heuristic algorithm described in [7], which includes new heuristic algorithm for analyzing similarity of class diagrams. The basic idea is when calculate the similarity of a pair of classes X_i and Y_j, all the related neighbor classes (super class, subclass, reference, and itself) of X_i and Y_j should be taken into account. The similarity value considers the highest similarity value of each neighbor of X_i with all neighbor of Y_j, and vice versa. In detail, the matrix is stored in similarity *Matrix R* ranging from 0 to 1. The first initial R^0 is equal to the static 1-to-1 similarity *Matrix S*, then iterate k times from R^0 to R^K based on the following formula algorithm (the iteration would be finished by several times or some value reaches certain threshold):

$$R^K(X_i, Y_j) = \frac{(M+N)/T) + R^{K-1}(x_i, y_j)}{2}$$

$$T = |rela(X_i)| + |rela(Y_j)|$$

$$M = \sum_{X_i \to X_{i'}} \max_{Y_j \to Y_{j'}} W(x,y) \times R^{k-1}(X_{i'}, Y_{j'})$$

$$N = \sum_{Y_j \to Y_{j'}} \max_{X_i \to X_{i'}} W(x,y) \times R^{k-1}(X_{i'}, Y_{j'})$$

- X_i or Y_j represents one class in *source MM'* or *target MM'*. $R^K(X_i,Y_j)$ is similarity value of pairs of classes X_i and Y_j in iteration K, and $R^{K-1}(X_i,Y_j)$ is the value in iteration K-1.
- $|rela(X_i)|$ or $|rela(Y_j)|$ is number of relations of neighbor classes of X_i or Y_j.
- X_i' or Y_j'. is a neighbor (super class, subclass, reference, or itself) of X_i or Y_j.
- $W(x,y)$ is a given value of relation weight, x or y means the relation of X_i->X_i' or Y_j->Y_j', the $W(x,y)$ value conforms to *Weight Matrix W* shown in table 3, and also can be customized.

Table 3: Weight Matrix (W) in Relational Iteration

	Self	Inheritance	Reference

Yj->Yj' \ Xi->Xi'			Super class	Subclass	Composition	Aggregation	Single
Self		1	0.9	0.9	0.9	0.8	0.7
Inheritance	Super class	0.9	0.9	0.8	0.8	0.7	0.6
	Subclass	0.9	0.8	0.9	0.8	0.7	0.6
Reference	Composition	0.9	0.8	0.8	0.9	0.7	0.6
	Aggregation	0.8	0.7	0.7	0.7	0.8	0.6
	Single	0.7	0.6	0.6	0.6	0.6	0.7

3) Combination similarity: *Matrix S* represents the static 1-to-1 semantic similarity; while *Matrix* R^K represents the relational semantic similarity after *K* time iterations. Though in calculation of relational semantic similarity, the static 1-to-1 has been already considered (by $R^0 = S$), for highlight the different weights of static and relational similarities, it is necessary to combine them as one similarity matrix, we can define their weighted averages of *S* and R^K, so the final combined similarity degree *Matrix C* is like:

$$C(X_i, Y_j) = \text{weight}(S)\times S(X_i, Y_j) + \text{weight}(R)\times R^K(X_i, Y_j)$$

4) **Similarity filter**: after getting combined similarity matrix, *similarity filter* could allow user to use certain threshold to filter the pairs of classes with lower similarity values, so let the following fragment matching get possible and credible class pair candidates.

For motivating case, after *static 1-to-1 similarity analysis* and 2 iterations of *relational iteration analysis*, then through combination ratio of 2-to-1, we can get the following similarity matrix in table 4, the class pairs in shadow are the ones with similarities that are higher than 0.4 filter threshold.

Table 4: Combined Similarity Matrix (C)

X \ Y	Company	Dept	Invoice	Statement	DeliveryAddress
Company	0.51	0.09	0.02	0.0	0.0
Department	0.11	0.64	0.25	0.01	0.01
Invoice	0.09	0.44	0.55	0.28	0.50
CompositeInvoice	0.10	0.64	0.66	0.01	0.01
SimpleInvoice	0.05	0.22	0.68	0.22	0.42
Contact	0.0	0.0	0.0	0.0	0.0
Statement	0.01	0.05	0.30	1.0	0.0

4.3 Fragments Matching

After getting *attribute candidates* from semantic engine and *pairs of classes* with higher similarity values from similarity analysis, in order to find the mappings, it is necessary to group most similar fragments of *MM's* with their references and related adaptation solutions for attributes (the adaptation reflects the fragment detail, e.g. *CopyAttribute*, *MergeAttributes*, *TypeCast*, etc.)

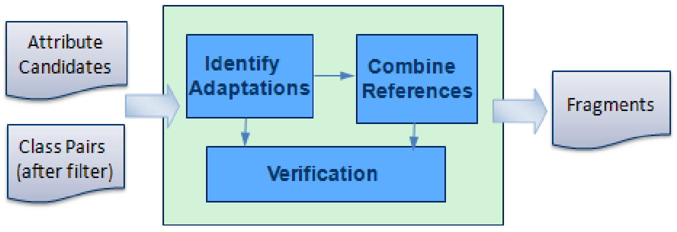


Figure 10: Fragment Matching Process

As illustrated in Figure 10, *Fragment Matching* has the following three main sub-steps:

1) **Identify adaptations**: from previous steps, the *attribute candidates* and *pairs of classes* after filtering are available. This step is to identify initial *fragments* with *adaptations*, by putting *attributes* into pairs of classes. Each *Adaptation* is identified by the *SemanticType* of *AttributeCandidate*. For instances, *CopyAttribute* (corresponds to identical, synonym), *MergeAttribute/SplitAttribute* (corresponds to merge/split), *TypeCase* (corresponds to type), *ScaleExchange* (corresponds to scale), and *PresicionApprox* (corresponds to precision). Additionally, when identifying adaptations for each fragment, the inherited attributes of LHS and RHS would be considered.

2) **Combine References**: because the basic idea of mapping and run-time transformation is like this, firstly to check the LHS classes of each mapping whether it appears in source instance model, if LHS classes appear then generate the classes of RHS classes of found mapping, by using the adaptations and strategies for the transformation of attributes. Therefore, it is significant to combine the fragments according to their references (*composition*, *aggregation*, *single reference*) for effectiveness and correctness. The combination has two methods as follows:

- Combine the fragments that LHS classes are same and RHS classes are reference relation. So when run-time transformation, after matching the LHS classes, only the combined fragment would be used so that it is more effective. Figure 11 shows the idea of this process, both *(X1, Y1)* and *(X1, Y2)* are two initial fragments with higher similarities, *Y1* has a composition of *Y2*, after combination, the two fragments combine into one *(X1, Y1&Y2)*.

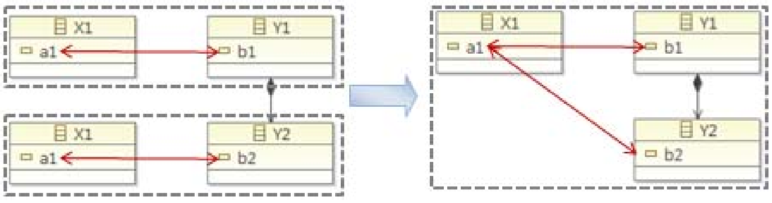


Figure 11: References Combination (RHS)

- Combine the fragments that both LHS classes and RHS classes have same reference relation. So when run-time transformation, the position of structure could be specified (e.g., where to generate the new class to *RHS*). Figure 12 shows the idea of this process, both *(X1, Y1)* and *(X2, Y2)* are two initial fragments with higher similarities, *X1* has a composition of *X2*, *Y1* has a composition of *Y2*, after combination, the two fragments combine into one *(X1&X2, Y1&Y2)*.

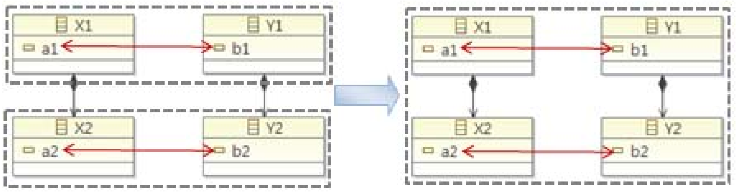
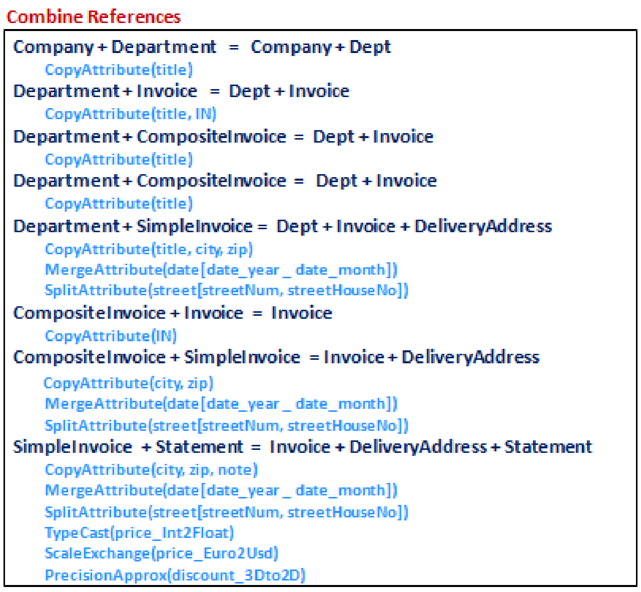


Figure 12: References Combination (*LHS&RHS*)

3) **Verification**: the whole process of fragments matching has a high-level automation to find the fragments. It can run automatically with certain default values. However, user interaction is also allowed, user can verify each process, for example, in the step of identify adaptations, user can select the attribute candidates and pairs of classes arbitrarily, and also can customize specific adaptation.



```
Combine References
Company + Department  =  Company + Dept
     CopyAttribute(title)
Department + Invoice  =  Dept + Invoice
     CopyAttribute(title, IN)
Department + CompositeInvoice  =  Dept + Invoice
     CopyAttribute(title)
Department + CompositeInvoice  =  Dept + Invoice
     CopyAttribute(title)
Department + SimpleInvoice  =  Dept + Invoice + DeliveryAddress
     CopyAttribute(title, city, zip)
     MergeAttribute(date[date_year _ date_month])
     SplitAttribute(street[streetNum, streetHouseNo])
CompositeInvoice + Invoice  =  Invoice
     CopyAttribute(IN)
CompositeInvoice + SimpleInvoice  = Invoice + DeliveryAddress
     CopyAttribute(city, zip)
     MergeAttribute(date[date_year _ date_month])
     SplitAttribute(street[streetNum, streetHouseNo])
SimpleInvoice  + Statement  =  Invoice + DeliveryAddress + Statement
     CopyAttribute(city, zip, note)
     MergeAttribute(date[date_year _ date_month])
     SplitAttribute(street[streetNum, streetHouseNo])
     TypeCast(price_Int2Float)
     ScaleExchange(price_Euro2Usd)
     PrecisionApprox(discount_3Dto2D)
```

Figure 13: Example of fragments after Fragments Matching

For the motivating case, figure 13 shows the example of fragments results after references combination. For clarity purpose, the LHS classes and RHS classes of one fragment are described in one line, and the adaptations of one fragment just with the adaptation type and the aligned attribute names.

4.4 Mappings Specification

After getting fragments, the mapping model that conforms to the mapping metamodel can be finally generated. Also, this step links the fragments back to original metamodels according to the *traceability links*. After the verification of user, the executable mappings code would be generated.

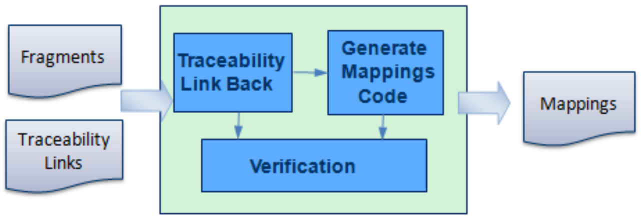


Figure 14: Mapping Specification Process

As illustrated in figure 14, mapping specification has the following three main sub-steps:

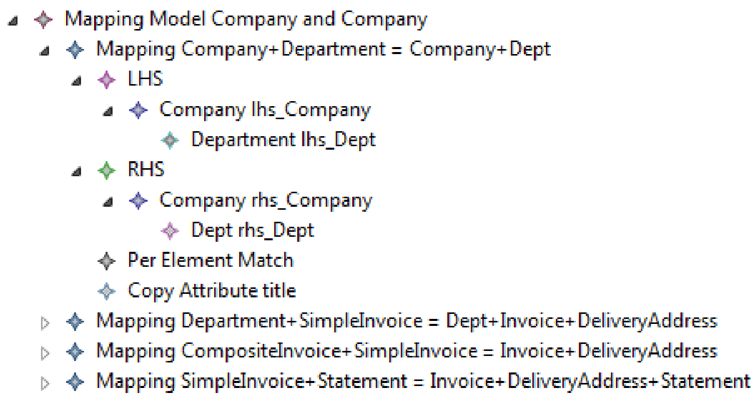
1) **Traceability Link Back**: this step is to rename the names of fragments found from fragments matching to the original names based on the traceability link. After linking back, the *initial mappings* that conform the mapping metamodel are available.

2) **Generate Mappings Code**: to serialize the mappings into concrete XML model that conforms to the mapping metamodel, using ECore metamodel API.

3) **Verification**: user can verify each initial mapping and generated code text. For example, optimize, modify or customize the mappings for specific purpose, and modify the executable mapping code as the

user wants.

For the motivating case, figure 15 gives a quick look of the mapping xml file, which has the found four mappings, including *LHS* and *RHS classes*, *adaptations* and *strategies* (most are *PerElement* type because that if a *RHS* element has already been created and associated to a *LHS* element in a previous mapping, then it will be reused and not duplicated).

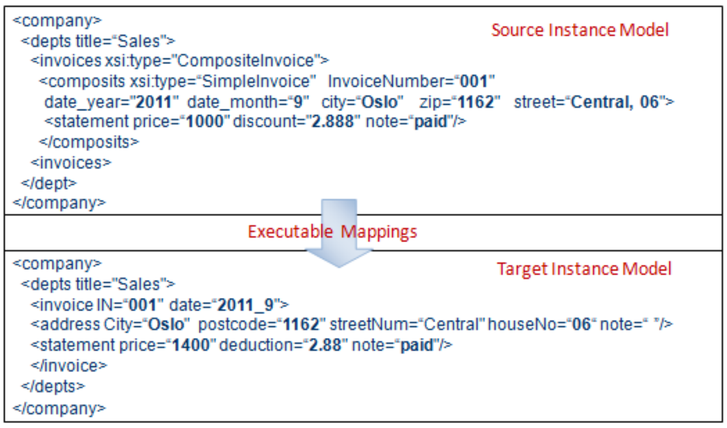


```
◢ Mapping Model Company and Company
  ◢ Mapping Company+Department = Company+Dept
    ◢ LHS
      ◢ Company lhs_Company
          Department lhs_Dept
    ◢ RHS
      ◢ Company rhs_Company
          Dept rhs_Dept
      Per Element Match
      Copy Attribute title
  ▷ Mapping Department+SimpleInvoice = Dept+Invoice+DeliveryAddress
  ▷ Mapping CompositeInvoice+SimpleInvoice = Invoice+DeliveryAddress
  ▷ Mapping SimpleInvoice+Statement = Invoice+DeliveryAddress+Statement
```

Figure 15: Mapping Model of Example

4.5 Run-time Transformation

Finally, using the found mappings generated at the design-time, the *source instance model* can be transformed into the *target instance model* in run-time transformation environment. AutoMapping reuses our run-time transformation environment of ExchangeMap[4], which is based on Drool Expert (see http://www.jboss.org/drools/drools-expert.html/) techniques to execute the mappings and then get *target instance model*.



Source Instance Model

```
<company>
 <depts title="Sales">
  <invoices xsi:type="CompositeInvoice">
   <composits xsi:type="SimpleInvoice"  InvoiceNumber="001"
    date_year="2011"  date_month="9"  city="Oslo"  zip="1162"  street="Central, 06">
    <statement price="1000" discount="2.888" note="paid"/>
   </composits>
  <invoices>
 </dept>
</company>
```

Target Instance Model

```
<company>
 <depts title="Sales">
  <invoice IN="001" date="2011_9">
  <address City="Oslo"  postcode="1162" streetNum="Central" houseNo="06" note=" "/>
  <statement price="1400" deduction="2.88" note="paid"/>
  </invoice>
 </depts>
</company>
```

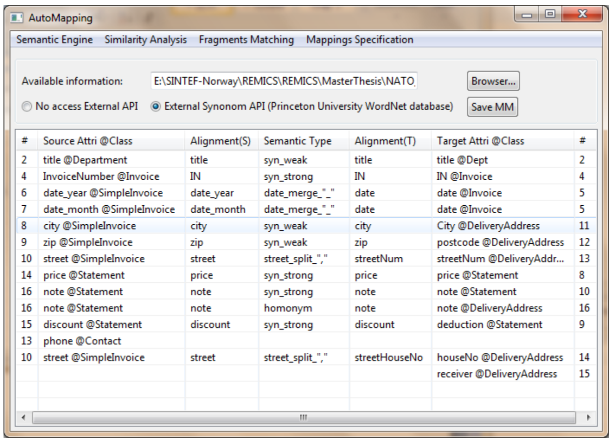
Figure 16: Example of Run-time Transformation

For the motivating case, figure 16 shows an example of run-time transformation from source instance model to target instance model in the motivating case.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

AutoMapping is currently implemented in Java/EMF programming language and KerMeta meta-modeling language (see http://www.kermeta.org/). For the design-time, it uses SWT (see http://www.eclipse.org/swt/) for the Graphical User Interface (GUI). For the run-time transformation, AutoMapping reuses the available project of ExchangeMap [4], which is our previous tool devised for data exchange of Model-driven Interoperability.

For design-time finding mappings of the motivation case, figure 17 shows several important program results of AutoMapping.

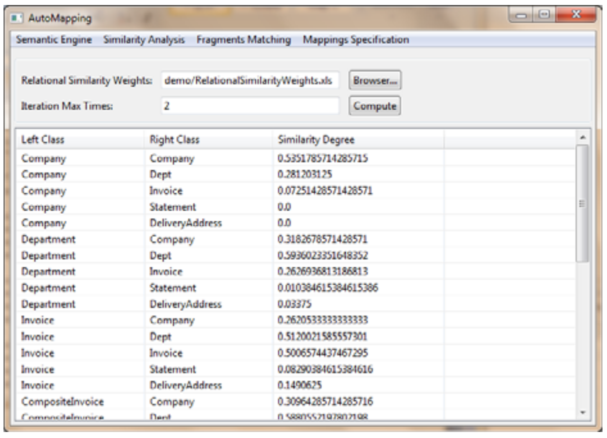


Available information: E:\SINTEF-Norway\REMICS\REMICS\MasterThesis\NATO

No access External API [✓ External Synonom API (Princeton University WordNet database)]

#	Source Attri @Class	Alignment(S)	Semantic Type	Alignment(T)	Target Attri @Class	#
2	title @Department	title	syn_weak	title	title @Dept	2
4	InvoiceNumber @Invoice	IN	syn_strong	IN	IN @Invoice	4
6	date_year @SimpleInvoice	date_year	date_merge_"_"	date	date @Invoice	5
7	date_month @SimpleInvoice	date_month	date_merge_"_"	date	date @Invoice	5
8	city @SimpleInvoice	city	syn_weak	city	City @DeliveryAddress	11
9	zip @SimpleInvoice	zip	syn_weak	zip	postcode @DeliveryAddress	12
10	street @SimpleInvoice	street	street_split_","	streetNum	streetNum @DeliveryAddr...	13
14	price @Statement	price	syn_strong	price	price @Statement	8
16	note @Statement	note	syn_strong	note	note @Statement	10
16	note @Statement	note	homonym	note	note @DeliveryAddress	16
15	discount @Statement	discount	syn_strong	discount	deduction @Statement	9
13	phone @Contact					
10	street @SimpleInvoice	street	street_split_","	streetHouseNo	houseNo @DeliveryAddress	14
					receiver @DeliveryAddress	15

Step1- Semantic Engine (alignment)

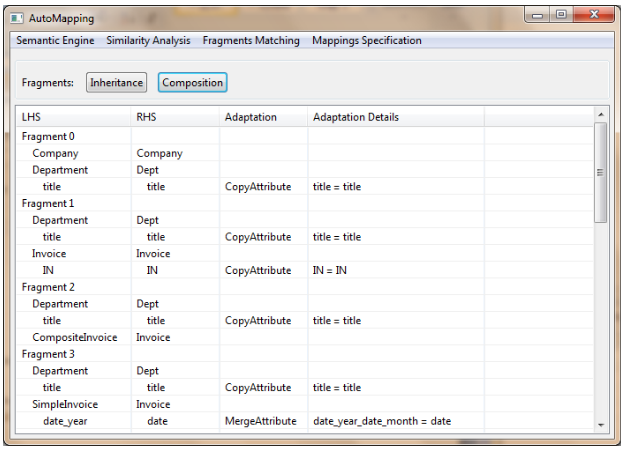


Relational Similarity Weights: demo/RelationalSimilarityWeights.xls

Iteration Max Times: 2

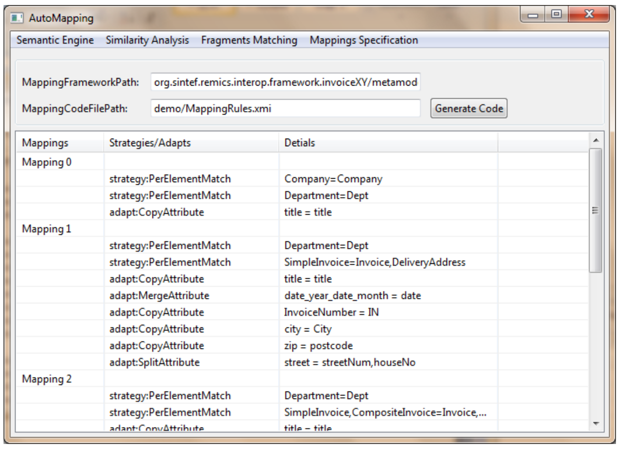
Left Class	Right Class	Similarity Degree
Company	Company	0.5351785714285715
Company	Dept	0.281203125
Company	Invoice	0.07251428571428571
Company	Statement	0.0
Company	DeliveryAddress	0.0
Department	Company	0.3182678571428571
Department	Dept	0.5936023351648352
Department	Invoice	0.2626936813186813
Department	Statement	0.010384615384615386
Department	DeliveryAddress	0.03375
Invoice	Company	0.2620533333333333
Invoice	Dept	0.5120021585557301
Invoice	Invoice	0.5006574437467295
Invoice	Statement	0.08290384615384616
Invoice	DeliveryAddress	0.1490625
CompositeInvoice	Company	0.3096428571428716
CompositeInvoice	Dept	0.5880557197802198

Step2- Similarity Analysis (relational iteration)



LHS	RHS	Adaptation	Adaptation Details
Fragment 0			
Company	Company		
Department	Dept		
title	title	CopyAttribute	title = title
Fragment 1			
Department	Dept		
title	title	CopyAttribute	title = title
Invoice	Invoice		
IN	IN	CopyAttribute	IN = IN
Fragment 2			
Department	Dept		
title	title	CopyAttribute	title = title
CompositeInvoice	Invoice		
Fragment 3			
Department	Dept		
title	title	CopyAttribute	title = title
SimpleInvoice	Invoice		
date_year	date	MergeAttribute	date_year_date_month = date

Step3- Fragments Matching (combination)



MappingFrameworkPath: org.sintef.remics.interop.framework.invoiceXY/metamod

MappingCodeFilePath: demo/MappingRules.xmi

Mappings	Strategies/Adapts	Detials
Mapping 0		
	strategy:PerElementMatch	Company=Company
	strategy:PerElementMatch	Department=Dept
	adapt:CopyAttribute	title = title
Mapping 1		
	strategy:PerElementMatch	Department=Dept
	strategy:PerElementMatch	SimpleInvoice=Invoice,DeliveryAddress
	adapt:CopyAttribute	title = title
	adapt:MergeAttribute	date_year_date_month = date
	adapt:CopyAttribute	InvoiceNumber = IN
	adapt:CopyAttribute	city = City
	adapt:CopyAttribute	zip = postcode
	adapt:SplitAttribute	street = streetNum,houseNo
Mapping 2		
	strategy:PerElementMatch	Department=Dept
	strategy:PerElementMatch	SimpleInvoice,CompositeInvoice=Invoice,...
	adapt:CopyAttribute	title = title

Step4- Mappings Specification (code generation)

Figure 17: Design-time SWT program results

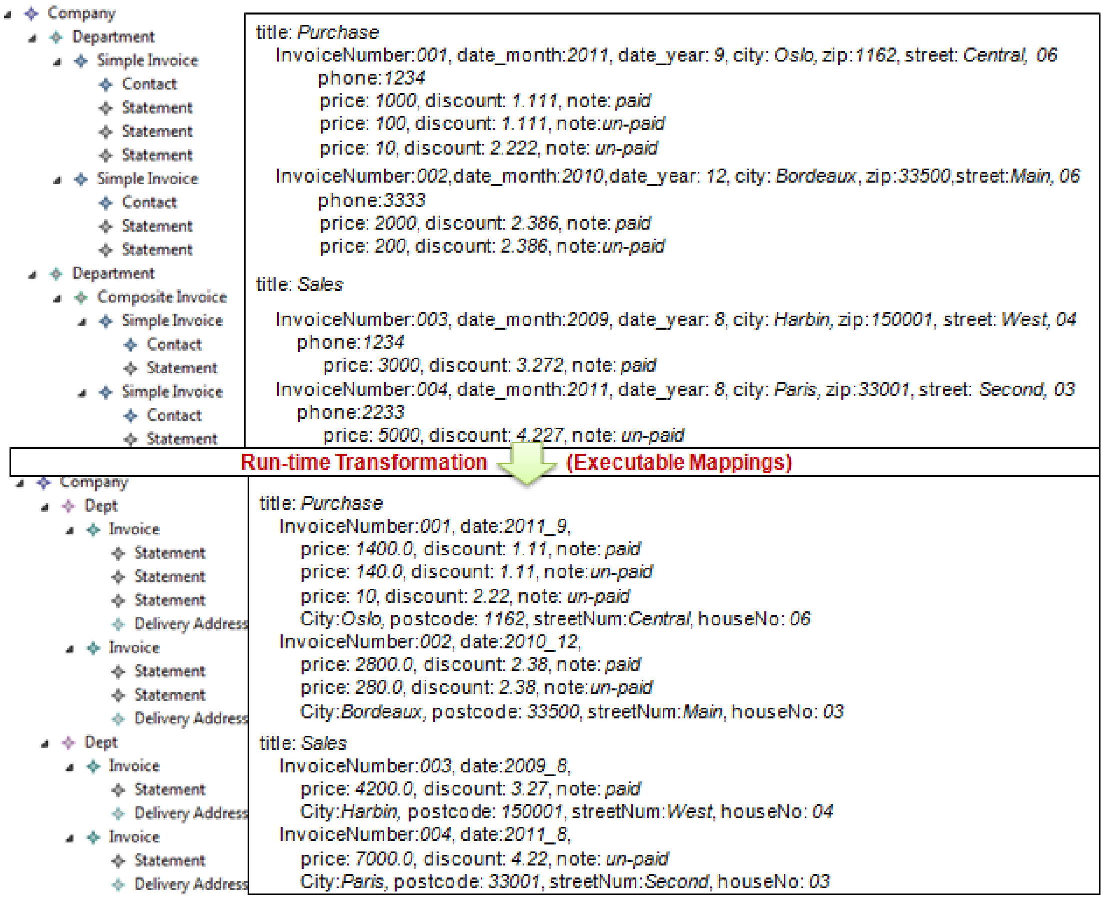
Based on the mapping model, AutoMapping reuses existed technique in ExchangeMap[8] to generate executable mappings. The following script illustrates the result of the first mapping of the executable mappings. This technique is proposed for automatically compile executable code (Java and Drools Expert, see http://www.jboss.org/drools/drools-expert.html/) from the specification of mappings, using a 2-pass visitor implemented in Kermeta.

```
rule "Company+Department = Company+Dept"
when
        // Code dealing with the relations between classes: compostion
        lhs__Company: companyx.Company(this == lhs__CompanyDecl, depts contains lhs__DepartmentDecl)
        lhs__Department: companyx.Department(this == lhs__DepartmentDecl)
then
        companyy.Company rhs__Company = null;
```

```
    companyy.Dept rhs__Dept = null;
    // Code dealing with the instantiation of the RHS elements, depending on:
    // strategies: PerElement (make sure just one instatntiation of RHS classes)
    rhs__Company.getDepts().add(rhs__Dept);
    //adaptations: CopeAttribute (title = tilte)
    rhs__Dept.setTitle(lhs__Department.getTitle());
end
// other three mappings' rule code
```

The first line is the name of the mapping. The *when* clause of the script corresponds to the *LHS* (from line 2 to line 6), they are the time this rule happens, which specifies that the rule is looking for any composition of *Company* and *Department of LHS*. The *then* clause of the script corresponds to the *RHS*. All the elements of the *RHS* are first declared and set to null. Then that are properly instantiated according to their associated strategies, as described in [6]. In the motivating example, the *PerElement* of strategies are mainly used, which makes sure that the *RHS* classes just be instantiated once, because several mappings may correspond to the same class of *RHS*. Lastly, the mapping adaptations are compiled into *set* primitives that properly manage the details of the attributes and references of the *RHS* elements.

Finally, using the *executable mappings* generated from reusable part of ExchangeMap, the *source instance model* can be transformed at the run-time into the *target instance model*. Automapping reuses our run-time transformation environment of ExchangeMap[6], which is based on Drool Expert techniques to execute the mappings and then get *target instance model*. To roundly test the implementation performance of AutoMaping, using the executable mappings generated from design-time, figure 18 shows multiple examples of run-time transformation from s*ource instance model* to *target instance model* in the motivating case, including multiple *departments*, *composite invoices*, s*imple invoices* and *statements*. We can see from the results that *target instance models* conform to the *target ECore metamodel* and the data of *target instance models* are compatible and correct.



```
title: Purchase
  InvoiceNumber:001, date_month:2011, date_year: 9, city: Oslo, zip:1162, street: Central, 06
        phone:1234
        price: 1000, discount: 1.111, note: paid
        price: 100, discount: 1.111, note:un-paid
        price: 10, discount: 2.222, note: un-paid
  InvoiceNumber:002,date_month:2010,date_year: 12, city: Bordeaux, zip:33500,street:Main, 06
        phone:3333
        price: 2000, discount: 2.386, note: paid
        price: 200, discount: 2.386, note:un-paid

title: Sales
  InvoiceNumber:003, date_month:2009, date_year: 8, city: Harbin, zip:150001, street: West, 04
     phone:1234
        price: 3000, discount: 3.272, note: paid
  InvoiceNumber:004, date_month:2011, date_year: 8, city: Paris, zip:33001, street: Second, 03
     phone:2233
        price: 5000, discount: 4.227, note: un-paid
```

```
title: Purchase
  InvoiceNumber:001, date:2011_9,
     price: 1400.0, discount: 1.11, note: paid
     price: 140.0, discount: 1.11, note:un-paid
     price: 10, discount: 2.22, note: un-paid
     City:Oslo, postcode: 1162, streetNum:Central, houseNo: 06
  InvoiceNumber:002, date:2010_12,
     price: 2800.0, discount: 2.38, note: paid
     price: 280.0, discount: 2.38, note:un-paid
     City:Bordeaux, postcode: 33500, streetNum:Main, houseNo: 03
title: Sales
  InvoiceNumber:003, date:2009_8,
     price: 4200.0, discount: 3.27, note: paid
     City:Harbin, postcode: 150001, streetNum:West, houseNo: 04
  InvoiceNumber:004, date:2011_8,
     price: 7000.0, discount: 4.22, note: un-paid
     City:Paris, postcode: 33001, streetNum:Second, houseNo: 03
```

Figure 18: Multiple test results of Run-time Transformation

6. RELATED WORK, CONCLUSIONS AND OUTLOOK

The problem of data exchange has been extensively studied for decades, and model transfromation is well established in MDA and MDI domain. Nevertheless, a generic approach that can easily handle both semantic interoperability and automatic model transformation has not yet been widely investigated in the community.

For solving semantic mismatches, existed approaches propose some possible solutions. The schema translation operations in [8] could provide valuable ideas. The principle behind it is to use different operations to add or change to the attribute, so that the semantic mismatches can be identified or modified without confusions. The solution for semantic operations in AutoMapping is inspired from [8], including *alignment*, *conversion*, *similarity filter*, *reference combination*, etc.

For improving automation transformation, this paper applies the bi-similarity heuristic algorithm described in [9]. Bi-similarity is a recursive notion and can be calculated in two ways, forward and backward which mean it considers not only the similarity each other elements in the forward direction but also when going backward in their history [10]. Though [9] and [10] are addressing for state machines diagram, this paper significantly changed its algorithm and devises new heuristic algorithm for analyzing similarity values of class diagrams.

Several works are related to our approach. For example, Atlas Transformation Language (ATL) [11] is a model transformation engine and is supported by abundant standard material. However, ATL does not support the semi-automatic transformation very well, the mapping rules are required hard coding by user;

also, its support to graphical interface is not well satisfactory. Besides, Spicy System [12] provides a solution for finding a mapping selection mechanism in certain automation level with some user interaction. However, it cannot solve all semantic interoperability mismatches, and the similarity analysis of mapping pairs of classes is not supported, which are well discussed in this paper.

The AutoMapping is an initial proof-of-concept that shows that semi-automatic transformation for semantic interoperability is feasible. However, there are still some directions that can be considered to further enhance AutoMapping.

- **Verify Mappings**: mapping verification in the current AutoMapping approach focuses on user manually level. However, it lacks of mapping verification module, which is used to check candidate mappings and choose the ones that represent better transformations of the source into the target. Several ideas of mapping verification approaches are inspired to further enhance, for example, the spicy verification module in [13] can achieve high precision in mapping selection.
- **Leverage additional knowledge**: If additional knowledge is available, such as a common ontology between the source and target data model, AutoMap should leverage this knowledge. However, it should also be able to identify mappings if such an ontology does not exists. Also not that AutoMap could be used to identify mapping between data models and ontologies.
- **Extend more Metamodels/Schemas**: current AutoMapping approach is designed for finding mappings between ECore metamodels and executing transformation between XML instance models. Though the approach works at an expressive model level, it should be fairly simple to extend it to handle other types of schemas or metamodels such as rule-based schemas [2], OWL metamodel (see http://www.w3.org/TR/owl-guide/). This would widen the instance models to be transformed that conform to different schematic representation.

7. REFERENCES

[1] C. Bussler. *Business-to-Business (B2B) Integration.* Springer: ISBN-3540434879. 2003.

[2] Y. Liao, D. Roman, and A. J. Berre. *Model-driven Rule-based Mediation in XML Data Exchange.* MDI 2010. Proceedings of the First International Workshop on Model-driven Interoperability. ACM 2010: 89–97.

[3] M. Missikoff and F. Taglino. *An Ontology-based Platform for Semantic Interoperability.* Handbook on Ontologies, Springer-Verlag. 2004: 617-634.

[4] D. Roman, B. Morin, S. Wang, and A. J. Berre. *A Model-driven Approach to Interoperability in B2B Data Exchange.* Advanced results in MDI/SOA innovation Workshop, MDI 2010, IWEI. 2011.

[5] R. Grønmo, S. Krogdahl, and B. Møller-Pedersen. *A Collection Operator for Graph Transformation.* ICMT'09: 2nd International Conference on Theory and Practice of Model Transformations, Berlin, Heidelberg. Springer 2009: 67–82.

[6] B. Morin, J. Klein, J. Kienzle, and J-M. Jézéquel. *Flexible model element introduction policies for aspect-oriented modeling.* ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway. October, 2010.

[7] S. Nejati, M. Sabetzadeh, M. Chechik, et al. *Matching and Merging of State charts Specifications.* ICSE. 2007: 54-64.

[8] L. Lehto. *Schema translations in a Web service based SDI.* 10th AGILE International Conference on

Geographic Information Science, the European Information Society: Leading the way with geo-information, Denmark. 2007

[9] S. Nejati, M. Sabetzadeh, M. Chechik, et al. *Matching and Merging of State charts Specifications.* ICSE. 2007: 54-64.

[10] R. D. Nicola, U. Montanari, and F. Vaandrager. *Back and forth bisimulations.* CONCUR. 1990:152–165.

[11] F. Jouault, F. Allilaire, J. Bézivin, et al. *ATL: a QVT-like Transformation Language.* OOPSLA'06, Portland, Oregon, USA. ACM 2006: 719-720.

[12] A.Bonifati, G.Mecca, A.Pappalardo, et al. *The Spicy system: towards a notion of mapping quality.* SIGMOD'08, Vancouver, Canada. June, 2008.

[13] A. Bonifati., G. Mecca, A. Pappalardo, et al. *Schema Mapping Verification: The Spicy Way.* EDBT'08. ACM 2008: 85-96.

ACKNOWLEDGMENT

This work is partly funded by the FP7 projects "Reuse and Migration of legacy applications to Interoperable Cloud Services (REMICS, http://www.remics.eu/).