An Implementation of Entity-Relationship Diagram Merging

Wentao He
Department of Computer Science
University of Toronto
Toronto, ON, Canada
wentao.he@mail.utoronto.ca

Abstract

Entity-Relationship model is an abstract way to describe database. During large scale development, each database designer often focuses on designing a particular part of the system, and the model created by each designer may have overlaps. How to efficiently merge these models to construct a larger model that describes the whole system is an interesting problem.

This report explains the implementation of an entity-relationship model merge operator that I created for solving this issue. The implementation is based on an algorithm that was proposed earlier. This operator is currently at its first release (release 1.0). Further improvement is required in terms of visualization and inconsistency validation.

Keywords

Entity-Relationship Diagram, Merge, Operator, EMF, MMTF.

1. Introduction

Entity-Relationship diagram is a visual representation of different data using conventions that describe how these data are related to each other. While being able to describe almost any system, ER diagrams are most associated with complex databases that are used in software engineering and IT networks. In particular, ER diagrams are frequently used during the design stage of a development process in order to identify different system elements and their relationships with each other.

In large scale model-based development, a key problem is to integrate a collection of models into a larger specification. This problem exists in the domain of entity-relationship model as well. In this report, I will explain how I created a merge operator that merge two entity-relationship models. This operator implements an algorithm proposed by Mehrdad in his PHD thesis [1]. In fact, Mehrdad had also implemented this in a tool called TReMer+ [3] back in 2008. However, the merge operator I created has different features and run in a different environment: MMTF, an Eclipsed based tool framework for model management.

In the following, I will introduce the background of entity-relationship diagram and model merge, followed by explanation to the merge algorithm that I implemented. In section 3, I will explain how I implemented this merge operator. In section 4, I will illustrate an example and validate the merge operator. In section 5, I will identify the areas where future improvements are required for this merge operator. At last, in section 6, I will close with some concluding remarks.

2. Background

2.1 Entity-Relationship Diagram

The concept for Entity-Relationship diagram (also called ER diagram) was introduced in a 1976 paper by Peter Chen six years after E.F. Codd published his seminal work defining the relational model of data. Chen's notation provided a way to graphically show relationships between data models.

There are three basic elements in an ER diagram: entity, attribute, relationship. On top of these, there are various types of these elements such are weak entity, multi-valued attribute, derived attribute, identifying relationship, etc. Figure 2.1 shows the standard symbols for all of them. All the types of elements in figure 2.1 have been implemented in the merge operator that I developed.

Assuming the reader has the basic knowledge of ER diagrams, introduction to each of these elements will not be covered in this paper.

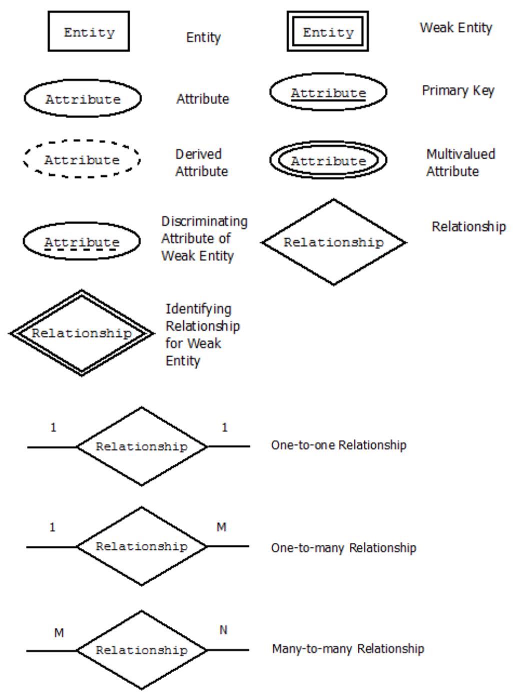


Figure 2.1 ER Diagram Symbols

2.2 Merge

In model-based development, models are usually constructed and manipulated by distributed teams, each work on a partial view of the overall system. A key problem is to integrate these separate but interrelated models into one single, larger model representing a larger view of the system.

Marsha distinguished three key integration operators – merge, composition and weaving, and provided a detailed analysis at the factors that one must consider in defining a merge operator, particularly the way in which the relationships should be captured during merge [2]. Marsha also identified a list of common criteria that merge result is expected to meet:

- ✓ Completeness
- ✓ Non-redundancy
- ✓ Minimality
- ✓ Totality
- ✓ Soundness

All these factors and criteria apply to the ER diagram merging. In my implementation of the ER diagram merge operator, I have considered all these factors.

2.3 Merge Algorithm

Mehrdad proposed an algorithm in his PHD thesis [1] for merging models. The algorithm can be adapted to any graph-based modeling language. It treats the mappings between models in terms of mapping between nodes and edges in the underlying graphs. The algorithm contains two major parts as described below.

2.3.1 Merging Sets

A system of interrelated sets is given by an interconnection diagram whose objects are sets and whose mappings are functions. Each function is considered as such a mapping: each element of the domain set is mapped to a corresponding element in the co-domain set. For example, in a three-way merge, the mappings would show how the set C is embedded in each of A and B.

To describe the algorithm for merging sets, I need to introduce the concept of *disjoint union* first. The disjoint union of a given family of sets $S_1, S_2, \ldots, S_n$, denoted $S_1 \uplus S_2 \uplus \ldots \uplus S_n$, is (isomorphic to) the following set: $S_1 \times \{1\} \cup S_2 \times \{2\} \cup \ldots \cup S_n \times \{n\}$. For conciseness, construct the disjoint union by subscripting the elements of each given set with the name of the set and then taking the union. For example, if $S_1 = \{x, y\}$ and $S_2 = \{x, t\}$, write $S_1 \uplus S_2$ as $\{x_{S1}, y_{S1}, x_{S2}, t_{S2}\}$.

To merge a system of interrelated sets, start with the disjoint union as the largest possible merged set, and refine it by grouping together elements that get unified by the interconnections. To identify which elements should be unified, construct a unification graph *U*, a graphical representation of the symmetric binary relation induced on the elements of the disjoint union by the interconnections. Then combine the elements that fall in the same connected component of *U*. Below shows the merge algorithm for an interconnection diagram whose objects are sets $S_1, \ldots, S_n$ and whose mappings are functions $f_1, \ldots, f_k$.

Algorithm: Set-Merge

```
Input:  Sets $S_1, \ldots, S_n$
        Functions $f_1, \ldots, f_k$
Output: Merged set $P$
➢ Let $U$ be an initially discrete graph with node-set
  $S_1 \uplus S_2 \uplus \ldots \uplus S_n$;
➢ For every function $fi$ $(1 \le i \le k)$:
    ➢ For every element $a$ in the domain of $fi$:
        ➢ Add to $U$ an undirected edge between the
          elements corresponding to $a$ and $fi(a)$;
➢ Let $P$ be the set of the connected components of $U$;
➢ Return $P$ as the result of the merge operation.
```

Figure 2.2 shows an example of set merge. (a) shows the interconnection diagram; (b) shows the induced unification graph and its connected components; (c) shows the merged set.

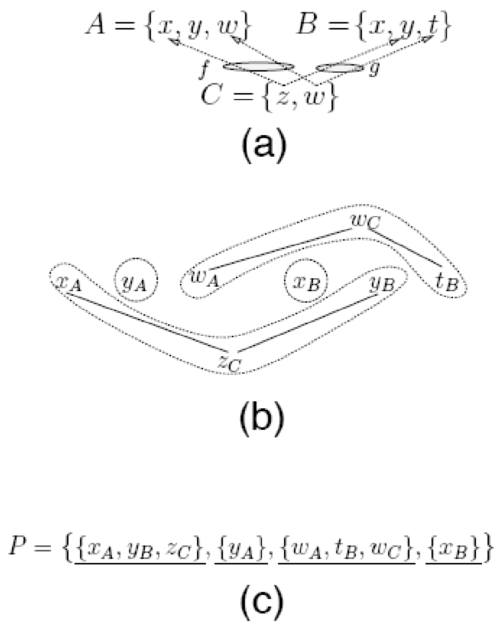

$A = \{x, y, w\}$
$B = \{x, y, t\}$
f
g
$C = \{z, w\}$
(a)
x_A
y_A
w_A
x_B
y_B
t_B
w_C
z_C
(b)
$P = \{\underline{\{x_A, y_B, z_C\}}, \underline{\{y_A\}}, \underline{\{w_A, t_B, w_C\}}, \underline{\{x_B\}}\}$
(c)

Figure 2.2 Three-way merge example for set [1]

2.3.2 Merging Graph

The notion of graph used here is a specific kind of directed graph used in algebraic approaches to graph-based modeling and transformation (Ehrig & Taentzer, 1996).

Definition: Graph

A (directed) graph is a tuple $G = (N, E, \text{source}, \text{target})$ where *N* is a set of nodes, *E* is a set of edges, and source, target: $E \rightarrow N$ are functions respectively giving the source and the target of each edge.

To interconnect graphs, mapping needs to be defined. Mehrdad uses homomorphism (a structure-preserving map describing how a graph is embedded into another) to denote the mapping.

A system of interrelated graphs is given by an interconnection diagram whose objects are graphs and whose mappings are homomorphisms. Merging is done component-wise for nodes and edges. For a graph interconnection diagram with objects $G_1, \ldots, G_n$ and mappings $h_1, \ldots, h_k$, the merged object *P* is computed as follows: The node-set (resp. edge-set) of *P* is the result of merging the node-sets (resp. edge-sets) of $G_1, \ldots, G_n$ with respect to the node-map (resp. edge-map) functions of

$h_1, \ldots, h_k$.

To determine the source (resp. target) of each edge *e* in the edge-set of the merged graph *P*, pick among $G_1, \ldots, G_n$, some graph G_i that has an edge *q* which is represented by *e*. Let *s* (resp. *t*) denote the source (resp. target) of *q* in G_i; and let *s′* (resp. *t′*) denote the node that represents *s* (resp. *t*) in the node-set of *P*. We set the source (resp. target) of *e* in *P* to *s′* (resp. *t′*).

Figure 2.3 shows an example of graph merge. In the figure, each homomorphism has been visualized by a set of directed dashed lines. In addition to the homomorphisms of the interconnection diagram, i.e., *f* and *g*, we have shown the homomorphisms δ_A and δ_B specifying how *A* and *B* are represented in *P*. The homomorphism from *C* to *P* is implied and has not been shown.

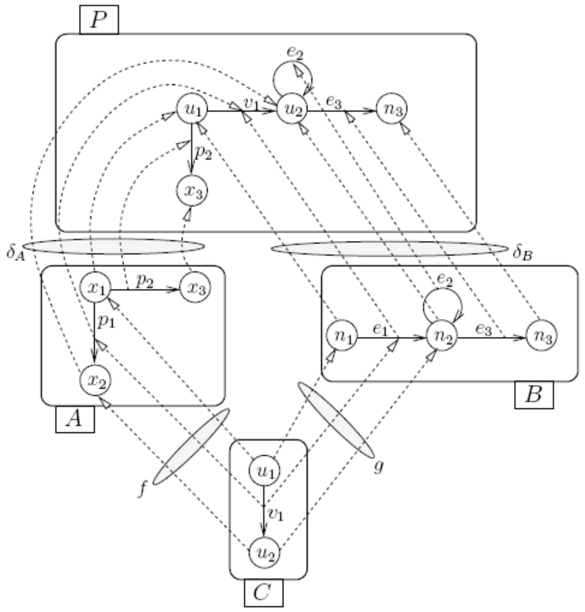


Figure 2.3 Three-way merge example for graphs [1]

To compute the merged graph *P* in figure 2.3, first merge the note-sets and edge-sets of *A*, *B* and *C*, using the algorithm described in the section 2.3.1. This generates two sets: *N*= {u_1, u_2, x_3, n_3}, *E*= {v_1, p_2, e_2, e_3}, representing the node-set and edge-set of *P*. Then, to determine the source and target of each edge in *E*, use the method described earlier in this section. For example, for edge v_1 in *E*, we get p_1 in *A* and e_1 in *B* that are mapped to this edge, any source (resp. target) of these 3 edges that is contained in *N* will be the source (resp. target) of v_1.

3. Implementation

In this section, I will first list the tools and frameworks I used, followed by the explanation of my implementation. At the end, I will summarize the overall features of the operator.

3.1 Tools and Frameworks

Eclipse: a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. I used Java language for this implementation.

EMF: a modeling framework and code generation facility for building tools and other applications based on a structured data model.

MMTF: an Eclipsed based tool framework for model management.

3.2 Create meta-model (eCore)

Meta-model describes the underlying structure of the model while a model is then the instance of this meta-model. Below I describe how I create the meta-model (eCore) to model ER diagram:

- Create an *EClass* named *ERDiagram* as the root.
- An ER diagram can contain multiple entities and relationships, so create *EClass Entity* and *EClass Relationship*, create *EReference* between *ERDiagram* and *Entity*, *ERDiagram* and *Relationship*, set the upper bound to "*".
- Create two references between *Entity* and *Relationship* indicating that a relationship must have one "from" entity and one "to" entity.
- Each entity and relationship can have multiple attributes, hence create *EClass Attribute*, create *EReference* between Entity and Attribute, Relationship and Attribute, set the upper bound to "*".
- An attribute can have multiple sub attributes if it is a composite attribute, so create *EReference* from Attribute to Attribute, and set the upper bound to "*".
- Create *EAttributes* for *EClass Entity*, *Relationship* and *Attribute*.

Figure 3.1 shows the ER diagram eCore model that I created.

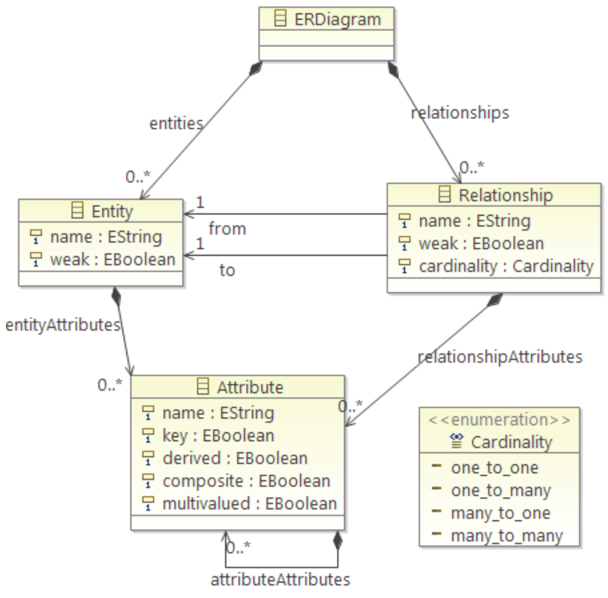
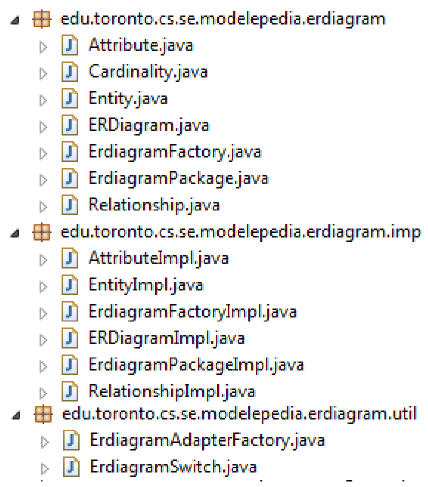


Figure 3.1 ER Diagram eCore Model

Use the eCore file created above to produce the corresponding *genmodel* which is then used to auto generate the Java implementation of the EMF model.



```
edu.toronto.cs.se.modelepedia.erdiagram
    Attribute.java
    Cardinality.java
    Entity.java
    ERDiagram.java
    ErdiagramFactory.java
    ErdiagramPackage.java
    Relationship.java
edu.toronto.cs.se.modelepedia.erdiagram.imp
    AttributeImpl.java
    EntityImpl.java
    ErdiagramFactoryImpl.java
    ERDiagramImpl.java
    ErdiagramPackageImpl.java
    RelationshipImpl.java
edu.toronto.cs.se.modelepedia.erdiagram.util
    ErdiagramAdapterFactory.java
    ErdiagramSwitch.java
```

Figure 3.2 Auto Generated Classes

Figure 3.2 shows the auto generated classes. The generated classes consists of the following:

- *edu.toronto.cs.se.modelepedia.erdiagram*

  Interfaces and the Factory to create the Java classes
- *edu.toronto.cs.se.modelepedia.erdiagram.impl*

  Concrete implementation of the interfaces defined in model
- *edu.toronto.cs.se.modelepedia.erdiagram.util*

  The AdapterFactory

3.3 Create merge operator

Now the ER diagram meta-model has been created. The merge operator is expected to take two instances of this meta-model and a mapping between the two model instances as input, output a merged model instance.

The main flow of the implementation can be summarized below:

1) Create class *ERDiagramMerge.java* that extends MMTF framework class *OperatorExecutableImpl.java* and override *execute()* method. The input parameter of *execute()* method is a list of models, in my case containing two ER models that are to be merged and a relationship between the two models.

2) Convert the two input ER models to ERDiagram typed objects. This type is defined in the eCore model;

   Convert the relationship model to a *HashMap*. Key of the hashmap represents the element in the left ER diagram while value represents the element in the right ER diagram (For the two input ER diagrams, I call the first one as left ER diagram and the second one as right ER diagram).

   While converting the model relationship to a *HashMap*, validation happens if the user chooses to do close-world merge, this makes sure that the user identified mapped elements do not have inconsistencies. For example, if user maps a strong entity in the left ER diagram to a weak entity in the right ER diagram, validation will catch this and notify the user of this inconsistency. However, if user chooses to do open-world merge, such validation will not happen. For the mapped elements, the merged ER diagram will take either left or right elements' naming based on user's preference.

3) Create a connector based on mappings between the two input ER diagrams and populate mappings between connector to each input ER diagram.

4) Now start merging set. I categorize the elements in ER diagram into four different types (or levels), they are:

   - Entity
   - Relationship
   - Entity Attribute
   - Relationship Attribute

   Create a disjoint union for each element type based on the left ER diagram, the right ER diagram and the connector. Use the algorithm described in section 2.3.1 to populate a merged set *P* for each element type.

   P is a list of element list. For each element list in *P*, take only one element according to the naming preference specified by the user, and construct another list of element called $P_{distinct}$, so this list contains the elements for the merged ER diagram.

5) Now start merging graph. Create an empty ER diagram D_{merged}, and then add merged elements to this ER diagram type by type, in the following order:

   - Entity type: get all the entities from $P_{distinct}$, add them to D_{merged}.
   - Relationship type: get all the relationships from $P_{distinct}$, add them to D_{merged}. For each relationship, check the existence of *from* entity and *to* entity in D_{merged}, if not exists, join *P* to get it and set it to the relationship.
   - Entity Attribute type: for each entity attribute in $P_{distinct}$, check the existence of its associated entity in D_{merged}, if exists, associate it to that entity; else, join *P* to get the attribute and associate it to the corresponding entity in D_{merged}.
   - Relationship Attribute type: for each relationship attribute in $P_{distinct}$, check the existence of its associated relationship in D_{merged}, if exists, associate it to that relationship; else, join *P* to get the attribute and associate it to the corresponding relationship in D_{merged}.

3.4 Unit Testing

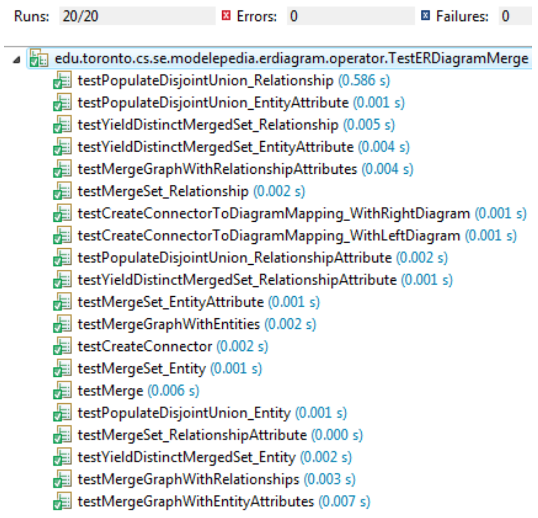
I have seen many applications where there is no robust or even no unit testing which I see it a big risk as the application keeps growing (it appears MMTF does not have unit test cases accompanied by its source code).

For the ER diagram merge operator, I used JUnit framework to create test cases and did throughout testing. As majority of my core logic sits in private methods, I used java reflection to access these methods from JUnit class. Figure 3.3 shows the unit test cases I have created.

3.5 Features of the operator

Overall, here are the features that this ER diagram merge operator supports:

- Capable of merging mapped entities, relationships, entity attributes and relationship attributes.
- User has the option to specify the merge approach: open-world merge or close-world merge.
- User has the option to specify naming preference: use naming of the shared elements from left ER diagram or right ER diagram.



```
Runs: 20/20        Errors: 0        Failures: 0

edu.toronto.cs.se.modelepedia.erdiagram.operator.TestERDiagramMerge
    testPopulateDisjointUnion_Relationship (0.586 s)
    testPopulateDisjointUnion_EntityAttribute (0.001 s)
    testYieldDistinctMergedSet_Relationship (0.005 s)
    testYieldDistinctMergedSet_EntityAttribute (0.004 s)
    testMergeGraphWithRelationshipAttributes (0.004 s)
    testMergeSet_Relationship (0.002 s)
    testCreateConnectorToDiagramMapping_WithRightDiagram (0.001 s)
    testCreateConnectorToDiagramMapping_WithLeftDiagram (0.001 s)
    testPopulateDisjointUnion_RelationshipAttribute (0.002 s)
    testYieldDistinctMergedSet_RelationshipAttribute (0.001 s)
    testMergeSet_EntityAttribute (0.001 s)
    testMergeGraphWithEntities (0.002 s)
    testCreateConnector (0.002 s)
    testMergeSet_Entity (0.001 s)
    testMerge (0.006 s)
    testPopulateDisjointUnion_Entity (0.001 s)
    testMergeSet_RelationshipAttribute (0.000 s)
    testYieldDistinctMergedSet_Entity (0.002 s)
    testMergeGraphWithRelationships (0.003 s)
    testMergeGraphWithEntityAttributes (0.007 s)
```

Figure 3.3 JUnit Test Cases

In addition, this operator can also work with Name Match Operator that Alessio (the core MMTF developer) has created. If the user wants to simply consider that elements with the same names are same elements, run the Name Match Operator to create a mapping relationship between the two input ER diagrams, and then select the two ER diagrams and the mapping relationship to produce a merged ER diagram.

4. Evaluation

4.1 Integration

In this section, I will run an end-to-end example to verify if the operator works as expected.

Jack and Tom work in the same company, both of them have created an ER model that describes their organization database. See figure 4.1, the diagram at the top is created by Jack while the diagram at the bottom is created by Tom. As we can see, these two diagrams have some overlaps. For example, entity “Employee” is in both of the diagrams; entity “Unit” in Jack’s diagram seems to represent the same thing as entity “Department” in Tom’s diagram though their names are different. After Jack and Tom sit down together and discussed, they come up with a mapping between the two diagrams, the mapping is depicted by the middle diagram (connector) in figure 4.1.

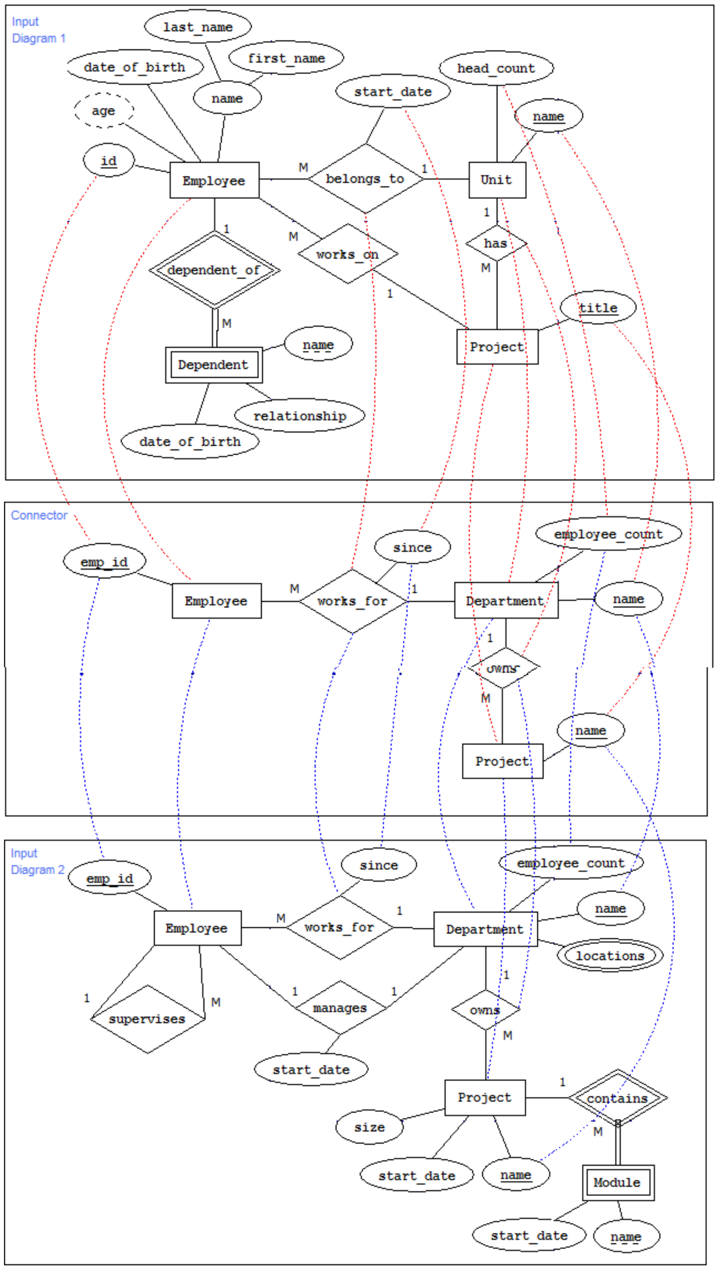


Figure 4.1 Mappings between the connector and the two ER diagrams to be merged

Now, to merge Jack and Tom’s diagrams, I first convert their diagrams to erdiagram meta-model instances, I name them: *Company_Input1.erdiagram* and *Company_Input2.erdiagram*. Then create a mid-diagram in MMTF, import the two instances to the mid-diagram, and create a mapping relationship between the two instances (figure 4.2).

Double click on the mapping, MMTF will take me to the relationship diagram where I specify the element mappings.

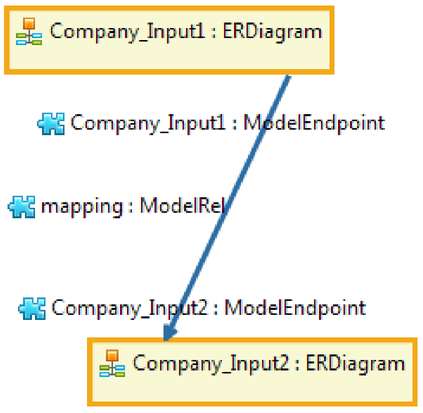

Company_Input1 : ERDiagram
Company_Input1 : ModelEndpoint
mapping : ModelRel
Company_Input2 : ModelEndpoint
Company_Input2 : ERDiagram

Figure 4.2 Two ER models and their mapping in MMTF mid-diagram

The last preparation step is to specify the merge preferences, i.e. open-world or close-world, left diagram naming or right diagram naming. They are controlled by two flags in a properties file. In this example, I use close-world approach and take right diagram naming.

Now come back to the mid-diagram, select the two ER models and the mapping, select "Run Operator -> ERDiagramMerge" through right click menu, a merged ER model is created and displayed in the mid-diagram. Convert the merged erdiagram model to ER diagram (figure 4.3).

Through manual verification, we can see that all the overlapped elements have been eliminated; right diagram naming is taken; there is no element lost as well as no new element got introduced. The merged diagram is exactly the same as expected.

In addition to the example above, I have also run several other examples to verify the operator (refer to ERDiagramMerge-Examples folder in the project submission package), all got expected results.

4.2 Performance

To evaluate the performance in terms of how fast the merge respond to heavy input ER models, I wrote a short program that automatically create two ER models containing a pre-defined number of entities, relationships and attributes, as well as a relationship between the two models with half of the elements mapped. I start the number from 50 (i.e. each input ER model contain 50 entities, 50 relationship and 50 attributes), and keep increasing it 50 by 50 till 500. Figure 4.6 shows the time spent for each run.

In fact, in real world barely there is company with so complex business requirement that would require to design an ER model involving hundreds of entities/relationships. Even if there is, it is strongly recommended that such huge ER model should be broken down into modules. Otherwise it's too difficult for human to interpret. Given this, the test result in figure 4.4 can prove that the performance of this ER merge operator is fairly acceptable.

5. Future Improvements

What has been implemented so far is basedlined as release 1.0. Due to the time constraint of this project, there are several features not implemented or not implemented maturely. They are important features for making this merge operator a "sellable" product:

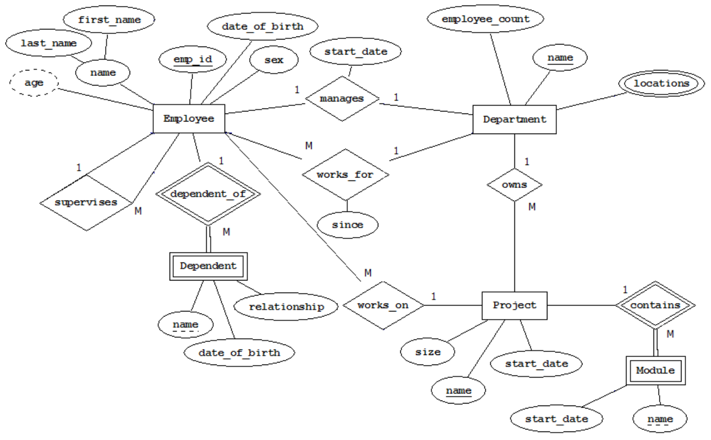


Figure 4.3 Merged ER diagram

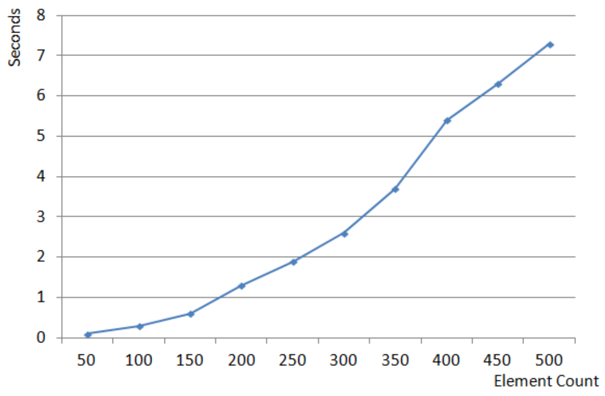


Figure 4.4 Performance Testing Result

a. Visualization

Currently conversion between ER model in MMTF and ER diagram is not implemented. User has to manually do the conversion.

In fact, I had tried using Eclipse GMF. Displaying a simple diagram with only very basic entity or relationship icon is easily doable, however visualizing a rich and heavy typed ER diagram (containing strong/weak entity, strong/weak relationship, multi-valued/derived/primary attribute etc) like in figure 4.3 take a lot more effort. Due to the time constraint I decided not to include this in release 1.0.

b. Validations

Current operator is not mature in catching all possible validation errors. The open-world vs. close-world approach implementation is at initial stage.

For example, if user drags a relation between an *entity* in left ER diagram and an *attribute* in right ER diagram in MMTF, neither the operator nor the MMTF framework will be able to catch this error (the mapped elements are of different types) and the merge will fail. I did not implement this validation in my operator as I believe this should be done at MMTF framework level. Further analysis and discussion is required here.

6. Conclusion

In this report, I explained the implementation of an entity-relationship diagram merge operator that I have created. This merge operator is a type of endogenous model transformation. It merges two entity-relationship models to an entity-relationship model that preserves all the behaviors expressed in both input models. This operator supports both open-world and close-world approaches (at conceptual stage), and also supports taking naming from either of the two input models into the merged model.

I also walked through the unit testing, integration testing and performance testing that I have conducted. The test results all come as expected and performance look fairly acceptable. Furthermore, I have identified some future improvements that are required for this merge operator - visualization and validation in order to make it a mature product.

In industry, database designer often focuses on designing a particular part of a large system and then merge with other designers' models. In most companies, even today, almost all the merge work is done manually. Manual work cost big effort and tend to high chance of making mistakes. I believe the entity-relationship diagram merge operator that I have created can significantly help in solving these problems.

Acknowledgements

I would like to thank Marsha Chechik, my professor, and Rick Salay who instructed me in finalizing the directions of the project and approaches. And I would like to thank Alessio Di Sandro for giving me the session of MMTF, and the technical support when I run into issues with MMTF.

References

[1] Mehrdad Sabetzadeh. Merging and Consistency Checking of Distributed Models. A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy. 2008

[2] Marsha Chechik, Shiva Nejati, Mehrdad Sabetzadeh. A Relationship-Based Approach to Model Integration. 2012

[3] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik. Global consistency checking of distributed models with TReMer+. In ICSE'08: Proceedings of the 30th International Conference on Software Engineering, pages 815–818, 2008.