Schema Reintegration Using Generic Schema Manipulation Operators

by

Xun Sun

B.Sc., Concordia University, 2004
B.E., Jilin University, 1997

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

September, 2006

Abstract

Schemas are very likely to be changed from time to time because of requirement changes, design revisions, database migration and such scenarios. Especially in a multi-user environment, schemas may be used by different groups or people and are modified to different versions. At some point in time, it is necessary to reintegrate the different versions and have a final unique version of the schema. Basically, the problem of creating the unique version is to merge the modified schemas. Previous works on schema merging describe how to merge two schemas given the mapping between them. In those works, the two schemas need not come from a common source schema. In fact, the original schema is often unavailable. However, in our scenario, we have the original schema, and we use it in decision-making. This simplifies what might otherwise be a complex matching procedure. We attempt to find a generic solution to the schema reintegration problem (i.e., when the original schema is present).

In this thesis, we created a framework that implemented the schema reintegration algorithms using generic model management operators. These generic operators have been widely mentioned and explained abstractly in many previous papers. Here we have implemented each operator required, which necessitated formally defining and creating the algorithm for each operator used for this specific schema re-integration purpose. Our contributions are: (a) determining that the generic operators can be used for schema reintegration, and (b) designing, implementing, and analyzing the model management operators used in details.

Table of Contents

Abstract ii

Table of Contents iv

List of Tables viii

List of Figures ix

Acknowledgements xiii

1 Introduction 1

1.1 Motivation 1

1.2 Problem Statement 5

1.3 Contributions 5

1.4 Thesis Outline 6

2 Background 7

2.1 Generic Model Management 7

2.1.1 Model 8

2.1.2 Mapping 11

2.1.3 The Inclusion Rules 13

2.1.4 Generic Model Management Operators 16

2.2 Generic Merge 18

2.2.1 Rondo 19

2.3 Three-Way-Merge 20

2.4 Summary 28

3 Operators 30

3.1 Operator *Merge* 30

3.1.1 Input 31

3.1.2 Output 32

3.1.3 The Merge Algorithm 32

3.2 Operator *Diff* 36

3.2.1 Input 37

3.2.2 Output 39

3.2.3 Algorithm 41

3.2.4 Related Work 43

3.3 Operator *range* 46

3.3.1 Input 46

3.3.2 Output 46

3.3.3 Algorithm 47

3.3.4 Related Work 48

3.4 Operator *Apply* 49

3.4.1 *Apply* 50

3.4.2 Function Γ in Reintegration 51

3.4.3 Related Work 53

3.5 Operator *Match* 54
3.5.1 Input 54
3.5.2 Output 55
3.5.3 Algorithm 55
3.5.4 Related Work 59
3.6 Summary 59

4 Support Elements 60
4.1 Definition 62
4.2 Usability of Support Elements 63
4.3 Support Elements Adding Algorithms 65
4.3.1 Algorithm to Find Which Elements Need Support . . 66
4.3.2 ENS Representatives 68
4.3.3 Support Elements Adding Engine and Algorithms . . 69
4.4 Discussion 78
4.4.1 The Root Element 78
4.4.2 Support Elements in Input 79

5 Reintegration System 80
5.1 Reintegration (Three-Way-Merge) Algorithm 80
5.2 Discussion of Problems in Reintegration System 82
5.2.1 Null Model 82
5.2.2 Support Elements in Input 84
5.2.3 Model Validation 86
5.3 Summary 87

6 Experiment 89

6.1 Operator Experiments 91

6.2 Support Element Engine Experiments 92

6.3 Reintegration System Experiments 94

7 Conclusion and Future Work 98

7.1 Conclusion 98

7.2 Future Work 99

Bibliography 101

A Reintegration System Example 104

List of Tables

2.1 Three-Way-Merge Rules 22

2.2 Three-Way-Merge Example 27

3.1 Element Correspondence for Each Operator 57

6.1 Experiments Performed for Each Operator 92

List of Figures

2.1 Different relationship kinds 9
2.2 Model *Product* 10
2.3 ModelMapping *Product* 13
2.4 Illegal Model 13
2.5 Morphism in Rondo 20
2.6 Three-Way-Merge Example [14] 21
2.7 Three-Way-Merge Result 21

3.1 Mapping between Model A and Model B 31
3.2 Merge result of Model A, Model B and Mapping 31
3.3 A Mapping between Model *A* and Model *B*, Model *A* is the *Preferred* Model. 34
3.4 Mapping between Model *O* and Model *B*. Element *c* is not mapped 37
3.5 *Diff* result from Figure 3.4, assuming that *O* is the base model. Element *a* is shaded as support element 38

4.1 Support Elements Mapping Example 60

4.2 The *Diff* result in Figure 4.1 with the support elements needed to make it a valid model. Support elements are visualized as shaded nodes throughout this thesis. 63

4.3 Mapping between a Model with a Support Element and a Normal Element 65

4.4 Elements Need Support & Representatives 68

4.5 Greedy Algorithm Example 76

4.6 Redundancy in Greedy Algorithm: $B - E$ are candidate support elements. The Greedy Algorithm will add all of them as support elements, but D is not necessary. 76

5.1 Support element c in Input model, which is not needed in Merge result. 85

6.1 Model Reintegration System 90

A.1 The Original Model MO 105

A.2 The First Modified Model MA 106

A.3 The Second Modified Model MB 107

A.4 The Mapping Map_{OA} between Model MO and Model MA . 108

A.5 The Mapping Map_{OB} between Model MO and Model MB . 109

A.6 The Model Map'_{OA} from *Apply*(Map_{OA}) 110

A.7 The Model Map'_{OB} from *Apply*(Map_{OB}) 111

A.8 The Model $Changed_A$ 112

A.9 The Model $Changed_B$ (A NULL model) 113

A.10 The Mapping $Map_{ChangedBChangedA}$ between Models $ChangedB$ and $ChangedA$ 114

A.11 The Mapping $Map_{ChangedAChangedB}$ between Models $ChangedA$ and $ChangedB$ 115

A.12 The Difference Model A' between Models $ChangedA$ and $ChangedB$ 116

A.13 The Difference Model B' between Models $ChangedB$ and $ChangedA$ 117

A.14 The Mapping Map_{AB} between Models MA and MB 118

A.15 The Merged Model G from Models MA and MB 119

A.16 The Mapping $Map_{GA'}$ between Models G and A' 120

A.17 The Merged Model GA' from Models G and A' 121

A.18 The Mapping $Map_{GA'B'}$ between Models GA' and B' 122

A.19 The Merged Model GAB from Models GA' and B' 123

A.20 The Model $Deleted_A$ Showing The Difference between Models MO and MA 124

A.21 The Model $Deleted_B$ Showing The Difference between Models MO and MB 125

A.22 The Mapping Model Map_{DACB} between Models $Deleted_A$ and $Changed_B$ 126

A.23 The Mapping Model Map_{DBCA} between Models $Deleted_B$ and $Changed_A$ 127

A.24 The Difference Model $ShouldDelete_A$ between $Deleted_A$ and $Changed_B$ 128

A.25 The Difference Model $ShouldDelete_B$ between $Deleted_B$ and $Changed_A$ 129
A.26 The Mapping Model between Models GAB and $ShouldDelete_A$ 130
A.27 The Difference Model $GABSDA$ between Models GAB and $ShouldDelete_A$ 131
A.28 The Mapping Model between Model $GABSDA$ and Model $ShouldDelete_B$ 132
A.29 The Final Result Model $Result$, Difference between Models $GABSDA$ and $ShouldDelete_B$ 133

Acknowledgements

I would like to thank Rachel A. Pottinger for being my supervisor and supporting me on finishing this thesis. She provided great guidance and help on my work. I would like to thank Dr. Edwin M. Knorr for being my second reader and giving me great comments on this thesis.

I would like also to thank all the people at database management lab and the University of British Columbia. I have really enjoyed the time working with them. Finally, I would like to thank my wife Huifen Li, my daughter Cecilia Sun and all my family members for their supports. I couldn't have accomplished this thesis without them.

Chapter 1

Introduction

1.1 Motivation

Database schemas are frequently modified and updated because of requirement changes, design revision, environment changes, customers' requirements changes, project re-engineering, etc. Additionally, in a teamwork environment, different teams may work remotely from each other and would also potentially make changes on database schema structures. Such changes can cause problems for all users of the database. Let's consider the following scenario:

Company O has designed a product management system that includes a schema named "product" with the following data definition:

```
CREATE TABLE PRODUCTS (
          ProductID int PRIMARY KEY,
          ProductName string,
          Brand string,
          Quantity real,
          UnitPrice int
);
```

This schema is used by two groups, A and B, separately to develop the system. $Product_O$ is the original schema and $Product_A$ and $Product_B$ are the schemas as modified by groups A and B respectively. During the development, each group makes a lot of changes, both on the design of the system and the database. Comparing with $Product_O$,the main differences in schema $Product_A$ is that the property *Brand* has been deleted:

```
CREATE TABLE PRODUCTS (
        ProductID int PRIMARY KEY,
        ProductName string,
        Quantity real,
        UnitPrice int
);
```

While, in schema $Product_B$, a new property *Discount* has been added.

```
CREATE TABLE PRODUCTS (
        ProductID int PRIMARY KEY,
        ProductName string,
        Brand string,
        Quantity real,
        UnitPrice int,
        Discount real
);
```

The changes made by groups A and B fit their local needs. After some time, however, the two parts need to be integrated. While the schema and data from each group are very similar, they cannot be reintegrated easily. Company O has the original schema $Product_O$, which is a baseline to compare the changes that each subsidiary has made and match the components that are from the original one. Based on this initial schema, we expect to generate the following schema as the integration result:

```
CREATE TABLE PRODUCTS (
        ProductID int PRIMARY KEY,
        ProductName string,
        Quantity real,
        UnitPrice int,
        Discount real
);
```

Previous research shows that in reintegration scenarios the goal is to keep *all changes* [14]. Here the attribute *Brand* has been deleted as in schema A; it should not appear in the final version schema. Because the new attribute "Discount" appears in the version B, it is believed that the new property is valuable and should be added in the final version.

Reintegration has been researched in various scenarios before. In [12], the authors show a flexible object merging framework that defines the merging policy targeting to different applications and the context of the collaborative activities, so that the reintegration process can be done in automatic, semi-automatic, and interactive ways. It also tries to be generic to suit objects with arbitrary structure and semantics. In [2], the authors give a framework describing how to synchronize file systems. It focuses on how to resolve update conflicts.

Reintegrating a small number of small schemas can be done manually. However, if the schemas are more complicated and there are many schemas, it is not efficient and productive to do it manually. It is necessary to have a way to solve it programmatically. Some previous works, such as [11], can do reintegrations programmatically. However, these solutions have to be reprogrammed for each data model, and the operations that are done are not always generalizable to other schema operations.

Schema management can usually be abstracted as Model Management [4][1]. A *Model* is not bounded to any kind of specific schema, but is considered to be a general form for schema integration. In Model Management, we use generic model management operators to manipulate the models. It will be more valuable to use a generic way to achieve the reintegration goal, so that it can also be used in other scenarios that can also be represented by model management, such as reintegrating different versions of UML's or ER diagrams. Frameworks and algorithms developed for Model Management are generic and can be applied to a variety of data models.

1.2 Problem Statement

In this thesis we focus on analyzing and presenting the problem of reintegration of models. Given an original model MO and two modification versions MA and MB, where MA and MB are generated from MO by using model management operators which provide *History* properties showing the mappings from MA and MB to MO, we generate an integrated model MG that takes into consideration the changes made in both MA and MB. In [14], the authors have given an algorithm for the steps necessary to do with such reintegration. Here we solve the problem programmatically by analyzing and implementing the details of each operator and solving the conflicts that arise in each operator and each step.

1.3 Contributions

We have the following contributions in this thesis:

- We analyzed the generic model management operators that are used in the schema reintegration system and gave details on the algorithms for implementation.
- We discussed the details of "Support Element" that are used to help the integrity of models. We give three algorithms to select support

elements based on different expectations. The "Support Element Engine" is generic and can be used by any model management operator.

- We fully designed and implemented the Three-Way-Merge system that can be used for Schema Reintegration purposes.

1.4 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 gives background knowledge on generic model management, generic merge and the three-way-merge algorithm. In Chapter 4, we discuss the support elements and the algorithms used to add support elements to models. In Chapter 3, the operators used in the reintegration system are formally defined and analyzed, including *Diff*, *Range*, *Apply* and *Match*. In Chapter 5, the Three-Way-Merge algorithm and the Schema Reintegration system are explained. In Chapter 7, we talk about the experiments and give conclusions and suggestions for future work.

Chapter 2

Background

In this chapter, we describe previous works on generic model management, model manipulation operators and merging methods.

2.1 Generic Model Management

There are many different kinds of schemas: relational, XML, etc. Schema manipulations to schemas share many commonalities to problems related to objects and relationships operations outside of databases. For example, UML, ER diagrams, file systems and ontologies also present a kind of schema structure that can be used to generate different kinds of data models and designs. When there are different modified versions of these kinds of objects, the reintegration process is very similar to the problem of database schema reintegration. The issues all relate to the objects and the mappings between them, and how to resolve conflicts and keep updates from different versions.

If the research of schema manipulation is only restricted to database schema, it would lose the potential to solve similar problems, such as those of ER and the other models mentioned above. Therefore, previous research of Schema Manipulation has been abstracted to a higher level: the *Model* level [1] [5] [4]. A *Model* is defined as a complex application artifact. It is usually represented as a directed graph and it can abstract and represent many applications, including relational schemas, XML DTDs, ontologies, UMLs, file systems, and network flows. This abstraction generates an OO like style of data model and platform. Therefore, it can utilize some benefits of OO design and it is generic enough to be applied to any specific data model, including schema management.

2.1.1 Model

Models [5][14][11] can be represented with graphs. The graph is composed of elements and relationships between these elements.

Each element contains a set of properties, which describe the detail of the element, such as the name, constraints, or any needed information of these properties. Each element has a required property *Name*, *ID* and *History*. Property *Name* is used to represent the name of the element. *ID* is used to uniquely identify the element. The *History* property records the last operations to the element. In other words, it shows where and how this element is generated. For example, one element may have a history property of: *"diff(300012)"*, meaning this element is generated by applying the *diff* operator to the first element with ID: 300012. Note that the element must have another unique ID different from 300012.

A relationship is a binary link between two elements and must be one of following five types: *Associates, Contains, Has-A, Is-A*, and *Type-Of.*

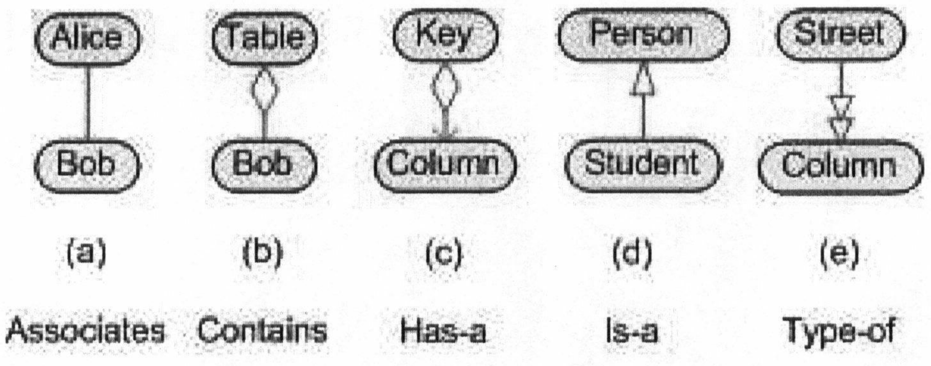


Figure 2.1: Different relationship kinds

These five relationship types are illustrated in Figure 2.1. The relationship types and Figure 2.1 come from [14]. More specifically, the semantics of the relationship types are as follows:

1. *Associate*, $A(x, y)$, is a weak relationship. It simply expresses that if x *Associate* y, then they have a very weak relationship; it implies no restrictions to the other as shown in Figure 2.1(a).

2. *Contains*, $C(x, y)$, means x *Contains* y. It shows a *container* - x and *containee* - y relationship. The existence of the *containee* relies on the existence of the *container*. For example, in Figure 2.1(b), if *table* is deleted, then attribute *Bob* cannot be kept either. *Contains* is transitive and acyclic.

3. *Has-a*, $H(x, y)$, means x *Has-a* sub-component y. It is similar to *Contains* in the sense that it also expresses the hierarchy of component vs. sub-component. The difference is that it can be cyclic and the sub-component does not need to be deleted when the component

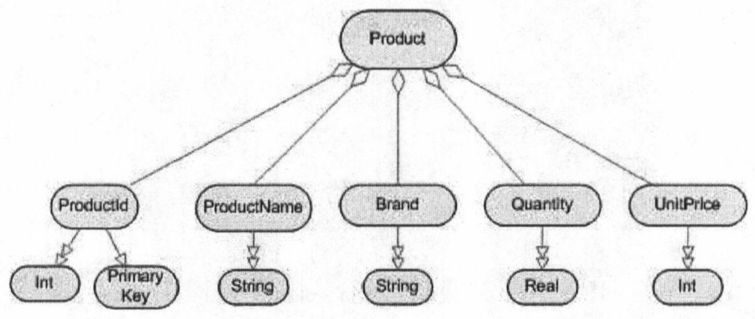


Figure 2.2: Model *Product*

is deleted. For example, in Figure 2.1(c), if the relation deletes the *key*, the *column* can still exist in the relation.

4. *Is-a*, $I(x, y)$, means x *Is-a* specialization of y. It is very like the idea of inheritance in Object Oriented Design, which expresses a specialization relationship. In Figure 2.1(d), *Student* is a special kind of *Person*. $Is - a$ is transitive and acyclic.

5. *Type-of*, $T(x, y)$ means x's type is y. It expresses the type of an element. In Figure 2.1(e), it means the *Street* is the type of *Column*. It is required that one element can only have one *Type-of* relationship. In [14], there are more details of the *one-type* restriction.

We can represent the *product* schema in our previous schema examples using a graph like Figure 2.2. For example, in Figure 2.2, element *product* has five *Contains* relationships and each subelement has a *Type_of* relationship. The *Product_id* element also has an *Is_a* relationship showing that it is a primary key.

2.1.2 Mapping

One important representation in model management is the *mapping*, which shows the relationship between two models. Without the *mapping*, it is hard do anything to manipulate models. As in [3], we assume that a *mapping* is also a model, which means it also includes elements E_{model} and relationships R_{model} that any model has. Moreover, it also contains:

- Two models, $Model_{domain}$ and $Model_{range}$. The *mapping* defines the relationship between them.
- Mapping relationships R_{map}. A mapping relationship r is between an element $e_m \in E_{model}$ and an element $e \in E_{domain} \cup E_{range}$. Elements in *domain* and *range* that have $ModelMappingRelationships$ from the same element in *map* are related to each other. The type of *mapping* relationship can be either "Equality" or "Similarity". Here the expression of "Similarity" is different from the way in [14]. In [14], "Similarity" is presented as a "how related" property in the element. If the element is marked as a "Similarity" type, all the elements in the domain or range that the element connects to are treated as similar to each other. Here we use the idea used in [13], such that the elements are the same, but the mapping *relationships* they connected to could have different types, "Equality" or "Similarity". In this way, whether the elements are equal or similar does not rely on the mapping element, but on each element in the domain and range separately. The same mapping element can be "similar" to one element and equal to

another. This also frees the mapping elements from showing "how related". The mapping elements are the same as normal elements in a model and can exist in a mapping model without any mapping relationships linked. It makes more sense to the OO design. In this way, "similarity" mapping relationships can be easily identified from "equality" mapping relationships at the relationship level.

For example, Figure 2.3 shows a mapping between the original *product* schema (partial) and the version of group A. The domain model is the original *product* schema and the range model is the *product* schema of version A. The map includes ModelMapping elements that are used to match the elements in model domain and range. Each ModelMapping element has a *ModelMapping-Relationship* to one of elements in the *domain* and *range* models. The relation may be equality, meaning they are exactly the same; or similarity, meaning that they are similar but have differences. There may also be existing elements in the mapping model that are not model mapping elements, but normal model elements to express some structure of the mapping.

In summary, a *mapping* defines how the *domain* and *range* models are related to each other. The ability to name both mapping relationships and all other relationships in the mapping means that each element in the mapping is not required to be the origin of a mapping relationship. However, the mapping must be a model. To do this, it must be adhere to the inclusion rules that follow in section 2.1.3.

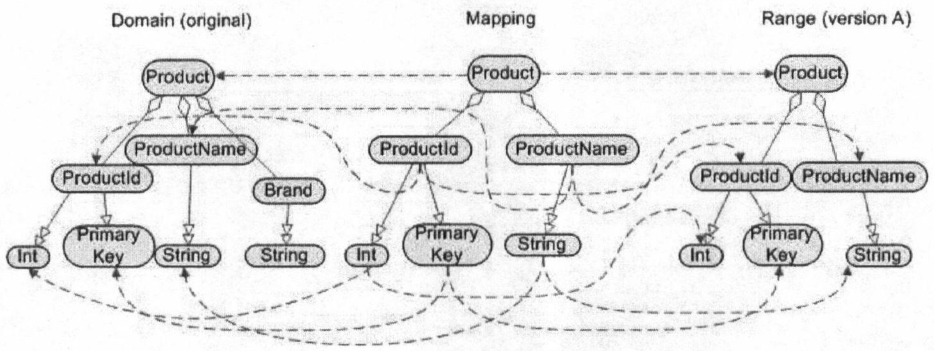


Figure 2.3: ModelMapping *Product*

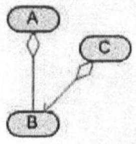


Figure 2.4: Illegal Model

2.1.3 The Inclusion Rules

In order for the operators to be composable, the result of any operator must be a model. To be a model, all elements must be connected by relationships showing that the elements are members of the model. It is not always true that all relationships among a number of elements result in a valid model. For example, Figure 2.4 shows an example that even though all the three elements are connected by relationships, they do not construct a legal model.

If an invalid model is used as the input of some operators, the result may be unpredictable and error-prone. In model management, the completeness can be verified using *inclusion rules* [14]. By saying legal, it means that all the elements and relationships can be covered using the *inclusion rules* and there is no dangling element or relationship.

The *inclusion rules* are based on the relationship types in the model. Each model has a unique root element, e_R, and all the other elements, E_L, have direct or indirect relationships, R_e, to the root element. The rules include:

- The root element, e_R, is always in the model.
- $I(p, q), p \in E_L \rightarrow q \in E_L$: If p has a *Is-a* relationship to q, and p is an element of Model, then q is also in the Model.
- $H(p, q), p \in E_L \rightarrow q \in E_L$: If p has a *Contains* relationship to q, and p is an element of Model, then q is also in the Model.
- $T(p, q), p \in E_L \rightarrow q \in E_L$: If p has a *Type-of* relationship to q, and p is an element of Model, then q is also in the Model.
- $p \in E_L, q \in E_L \rightarrow R(p, q) \in R_{e_L}$: If p, q are all elements of Model, then all the relationships between p and q are in the Model.
- $M(p, q), p \in E_L \rightarrow M(p, q) \in R_{e_L}$: If p is an element of Model, then all the model mapping relationships that have p as the origin are in the model.

Because the *inclusion rules* determine when an element is in a model by the relationships we need find all the relationships in a given model, including those relationships that can be deducted implicitly. The implications include those relationship types that are transitive and the $cross-kind-relationship$ implications.

Transitive relationship types include $Contains$, $Has-a$ and $Is-a$. The $cross-kind-relationship$ implications are:

- If p (*Type-of*) q and q (*Is-a*) $r \Longrightarrow p$ (*Type-of*) r
- If p (*Is-a*) q and q (*Has-a*) $r \Longrightarrow p$ (*Has-a*) r
- If p (*Is-a*) q and q ($Contains$) $r \Longrightarrow p$ ($Contains$) r
- If p ($Contains$) q and q (*Is-a*) $r \Longrightarrow p$ ($Contains$) r
- If p (*Has-a*) q and q (*Is-a*) $r \Longrightarrow p$ (*Has-a*) r

Using these rules, we can find all the explicit or implicit relationships in the model and then we can determine if a given model is legal or not. The *inclusion rules* can also be used to remove the redundant relationships that can be implied using existing relationships.

We say element X is $inclusion-implied$ by a relationship R (denoted $II(R,X)$ if: (1) $R(X,Y)$ and the inclusion rules state that if $R(X,Y)$ and Y is in a model, then X is in the model, or, (2) $R(Z,X)$ and the inclusion rules state that if $R(Z,X)$ and Z is in a model then X is in the model.

2.1.4 Generic Model Management Operators

There are several operators in model management, such as Diff, Delete, Match, Extract, Domain, Range, Compose, Invert, Apply, Copy, and ModelGen [11][14][3]. These operators give a programmatic way to transform models in abstract levels. Here we briefly introduce the functionality of these operators, which will be used to implement our generic schema reintegration.

- *Diff* - $Diff(x, y) = z$: Takes two models, x and y, as input and returns model z, which contains the elements that are in x but not in y. *Diff* will be discussed in more detail in Section 3.2.
- *Delete* - $Delete(x, y) = z$: Takes one model x and a set of elements y, and deletes elements y from x and returns the result model z.
- *Match* - $Match(x, y) = Map_z$: Takes two models x and y and finds the mapping relationships between two given models and returns the mapping model Map_z. *Match* will be discussed in more detail in Section 3.5.
- *Extract* - $Extract(x, y) = z$: Takes one model x and a set of elements y, which contains partial elements of x, and selects those elements in y from x and relationships related to y and returns the result model z.
- *Domain* - $Domain(x, y_{map}) = z$: Takes a model x and a mapping model y that has x as the domain of the mapping, and returns a

partial model z of x that contains only the elements having mapping relationships in y.

- *Range* - $Range(x, y_{map}) = z$: Takes a model x and a mapping model y that has x as the range of the mapping, and returns a partial model z of x that contains only the elements having mapping relationships in y. *Range* will be discussed in more detail in Section 3.3.

- *Compose* - $Compose(M_1(x, y), M_2(y, z)) = M_3(x, z)$: Takes two mapping models M_1 and M_2, which are mappings between model x and y, and between y and z respectively. The operator *Compose* generates a new mapping model M_3 between x and z. While quite complicated in some cases, it is not complicated in our schema reintegration. More information on composition can be found in [3].

- *Invert* - $Invert(Map(M_1, M_2)) = Map'(M_2, M_1)$: Takes a mapping model *Map* between domain model M_1 and range model M_2, swaps the domain and range roles in the mapping and returns a new mapping model Map' that has M_2 as domain and M_1 as range.

- *Apply* - $Apply(x) = y$: Takes a model x and *applies* a function to each element in x to make changes according to the function and returns the modified model M'. *Apply* will be discussed in more detail in Section 3.4.

- *Copy* - $Copy(x) = y$: Takes a model x and return a new model y that has the same elements and relationships as x.

- *ModelGen* - $ModelGen(x) = y$: Takes a type of model x and generates a new type of model y from x. For example, generate a relational schema from an XML schema.

2.2 Generic Merge

Given two models and the mapping between them, the generic merge generates the model that includes all the information from the two input models.

There are several algorithms and implementations [14][11] describing generic merge. The general steps are:

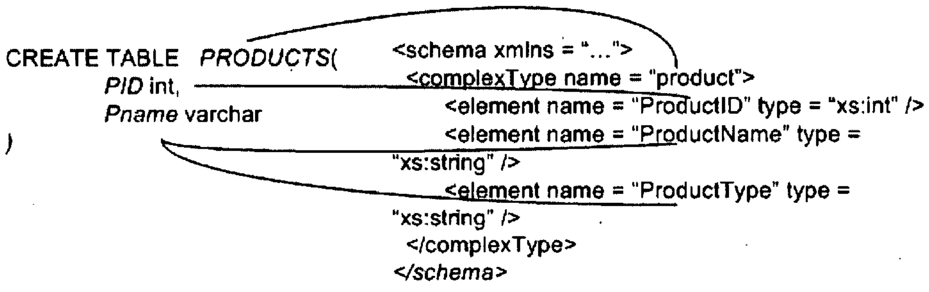
First, generate the necessary elements. Basically, the elements are the set union of the two input models, where the mapped elements are treated as the same principle and are only represented using one element in the result. Each such representative element includes all the properties of the elements that it represents. Then, the relationships are merged. One key factor is that any relationships in the domain/range and the mapping should not be lost. All the elements, whether they are in the domain/range or the mapping, have representatives in the result model. All the relationships are between elements in the domain/range and mapping, so all the relationships can be preserved in the mapping result. However, if all the relationships are simply added to the final result, there may be conflicts existing, such as cycles or multiple types for an element. There may also be some existing redundant relationships that can be implied and induced from other relationships. So the final step is to check and fix these kinds of conflicts.

2.2.1 Rondo

Rondo [11] is the first complete prototype of generic model management system. Rondo systematically defines the key model management operators, such as *Domain*, *Invert*, *Compose*, etc, and suggests several new generic operators, such as *Extract*, *Delete*, etc. It also shows a schema integration system that can be used for relational database Schemas or XML Schemas.

The differences of our approach to Rondo's lie on the data structure. Rondo uses a new data structure to represent the mapping. It is called a *morphism*. A morphism is a binary relationship between two models. Figure 2.5 is an example from Rondo that shows a morphism between a relational schema and an XML schema. In implementations, a morphism is a pair of elements. This is a convenient way to represent mapping relationships.

However, in our framework, we have everything existing as first-class objects. The idea of a morphism violates the OO principle. The existing of mapping relies on the models and cannot be presented by itself. Therefore, we use a mapping model to present the mapping relationships. Furthermore, in our system, every step gives an output of a complete model, which is easy to maintain and extend. All of our operators have first-class objects as input and output. These operators can be easily used in any other generic model



```
CREATE TABLE  PRODUCTS(
        PID int,
        Pname varchar
)
```

```
<schema xmlns = "...">
  <complexType name = "product">
    <element name = "ProductID" type = "xs:int" />
    <element name = "ProductName" type =
"xs:string" />
    <element name = "ProductType" type =
"xs:string" />
  </complexType>
</schema>
```

Figure 2.5: Morphism in Rondo

management tasks.

2.3 Three-Way-Merge

In this section, we review the Three-Way-Merge algorithm discussed in [14], which is the basic algorithm used in this thesis.

The Three-Way-Merge algorithm merges a given model and two different modified versions of it into one model. For example, Figure 2.6 shows the example used in [14]. There is the original model O and two different modified versions, model A and model B. The goal is to create a final model, L, that captures all of the changes in A and B. The modification of model A is that the element d changed its parent from the root to element b, while in Model B, the element c was deleted.

When these models are to be merged, because element c is deleted in Model B, the final version should capture the deletion and exclude c. Element d is modified in Model A; therefore, the final version should also

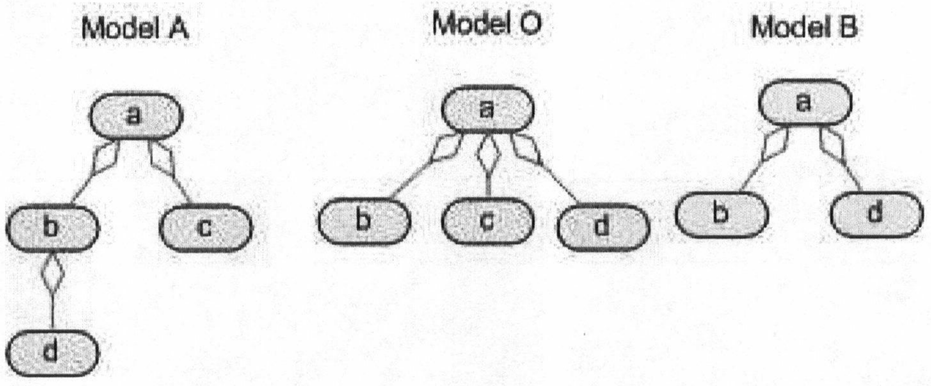


Figure 2.6: Three-Way-Merge Example [14]

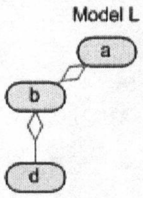


Figure 2.7: Three-Way-Merge Result

capture the modification and change the parent element of d to b from a. The merged result model is shown in Figure 2.7.

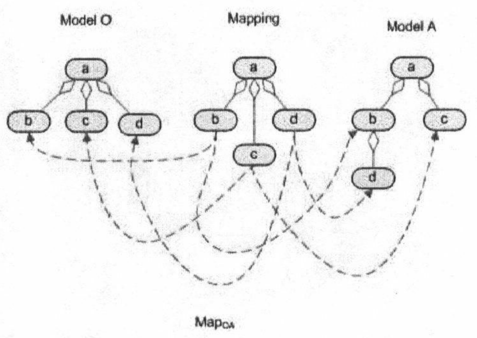
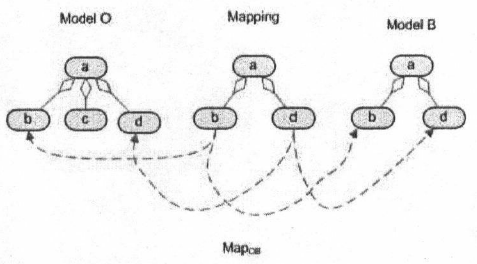
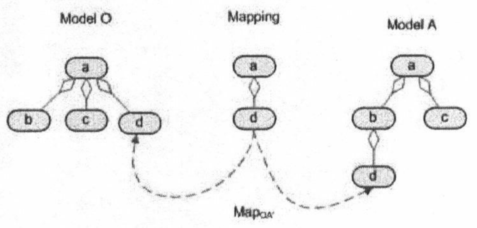
There are several rules targeting the different changes made to each modified version. Let A, B denote the two modified versions, O denote the original version, and L be the final merged version. The rules are shown in Table 2.1. It defines different situations that different versions may have and how the final version should be with regard to the element. For example, in row 4, it means if the element is in O, it is deleted in one model (A here) and modified in the other model (B here), then in the final result L, the modified version should be kept, it is *in* the final model.

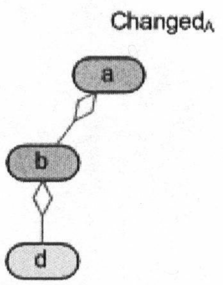
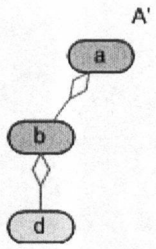
Based on these rules, the three-way merge algorithm in [14] will do the

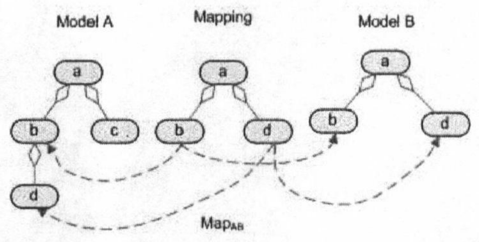
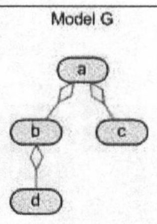
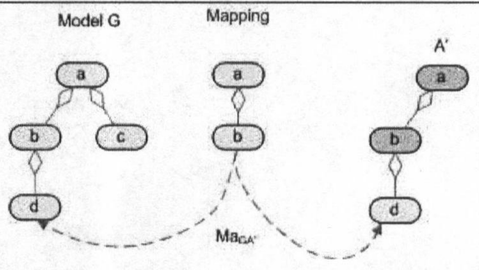
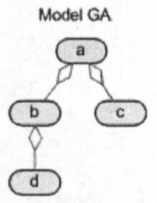
#	O	A	B	L
1	$\otimes$	add	add	$\odot$
2	$\odot$	unmodified	unmodified	$\odot$
3	$\odot$	deleted	unmodified	$\otimes$
4	$\odot$	deleted	modified	$\odot$
5	$\odot$	modified	unmodified	$\odot$
6	$\odot$	modified	modified	$\odot$

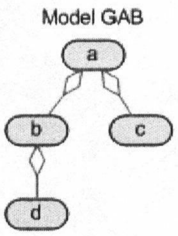
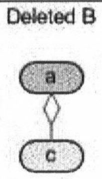
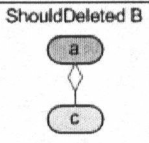
Table 2.1: Three-Way-Merge Rules
$\odot$: element in model
$\otimes$: element *not* in model

following steps to merge models A, B and O. The result is the same as shown in Figure 2.7. Note that because we are only demonstrating a simple example, some steps may generate empty models.

Step	Operation	Figure
1	$Map_{OA} = Match(O, A)$ By History Property	
2	$Map_{OB} = Match(O, B)$ By History Property	
3	$Map_{OA'} = Apply(Map_{OA})$ if $e \in Map_{OA}$ and $domain(e)$ is identical to $range(e)$, then delete e	
4	$Map_{OB'} = Apply(Map_{OB})$ if $e \in Map_{OB}$ and $domain(e)$ is identical to $range(e)$, then delete e	$\emptyset$

5	$Changed_A = range(Map_{OA'})$ the things changed in A	
6	$Changed_B = range(Map_{OB'})$ the things changed in B	∅
7	$Map_{ChA\_ChB} = Match(Changed_A, Changed_B)$	∅
8	$Map_{ChB\_ChA} = Match(Changed_B, Changed_A)$	∅
9	$A' = Diff(Changed_A, Changed_B, Map_{ChA\_ChB})$	
10	$B' = Diff(Changed_B, Changed_A, Map_{ChB\_ChA})$	∅

11	$Map_{A\_B} = Match(A, B)$ According to OIDs	
12	$G = Merge(A, Map_{A\_B}, B)$	
13	$Map_{G\_A'} = Match(G, A')$	
14	$GA = Merge(G, Map_{G\_A'}, A')$ with preference for A'	
15	$Map_{GA'\_B'} = Match(GA', B')$	∅

16	$GAB = Merge(GA', Map_{GA'\_B'}, B')$ with preference for B'	
17	$Deleted_A = Diff(O, A, Map_{OA})$	$\emptyset$
18	$Deleted_B = Diff(O, B, Map_{OB})$	
19	$Map_{DeletedA\_ChangedB} = Match(Deleted_A, Changed_B)$	$\emptyset$
20	$Map_{DeletedB\_ChangedA} = Match(Deleted_B, Changed_A)$	$\emptyset$
21	$ShouldDelete_A = Diff(Deleted_A, Changed_B, Map_{DeletedA\_ChangedB})$	$\emptyset$
22	$ShouldDelete_B = Diff(Deleted_B, Changed_A, Map_{DeletedB\_ChangedA})$	
23	$Map_{GAB\_SDA} = Match(GAB, ShouldDelete_A)$	$\emptyset$

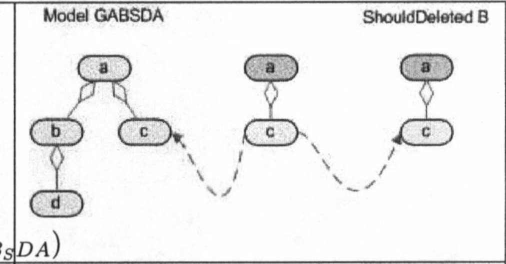
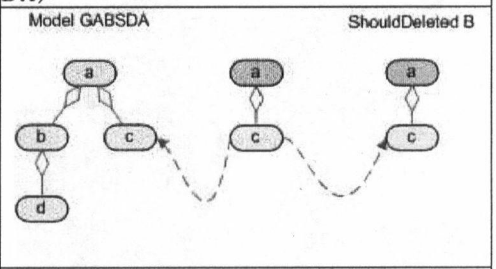
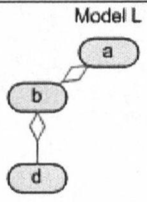
24	$GABSDA = Diff(GAB, ShouldDelete_A, Map_{GAB_SDA})$	 Model GABSDA ShouldDeleted B a b c d a c a c
25	$Map_{GABSDA\_SDB} = Match(GABSDA, ShouldDelete_B)$	 Model GABSDA ShouldDeleted B a b c d a c a c
26	Final *result* $= Diff(GABSDA, ShouldDelete_B, Map_{GABSDA\_SDB})$	

Table 2.2: Three-Way-Merge Example

In the original algorithm, it does not mention how to deal with the completeness problem of models. For example, in step 5, the *range* operator will only keep the element d in the result. It is desirable to keep the result not being a simple element, but still a model. Here the elements a and b are added to keep the structure of the model. In order to distinguish them from the result elements, they are marked as *support elements*. Not only could the operator *range* generate an incomplete model, other operators could also have the same problem and need *support elements* to help in the result model. A key factor is that because Rondo considers only morphisms, it does *not* use support elements. Support elements are discussed briefly in [3], but one contribution of this thesis is a fuller understanding of support elements. We discuss them more fully and describe our conclusions in Chapter 4.

2.4 Summary

Model Management gives an abstract way to solve meta-data management problems. By using graph theories, models can be programmatically manipulated by the model management operators.

The mapping between two models is important and used very often in most operators and applications. However, the presentation of mapping varies a lot in different systems. It can be a first-class object - a model; it can be a morphism that uses direct links between model elements; or it can be just some pairs without data structure support. Some have been researched deeply, such as *match* and *merge*. The other operators have been introduced and discussed, but few have details on the algorithm and few have been researched for the behavior of the operator and how they interact with each other. These operators are still in an abstract level. Even if the idea is clear, the actual implementation varies depending on different kind of applications that the operators use. In this thesis, we only use first-class objects (models) and the operators are implemented based on the requirement of schema reintegration and functionality.

Chapter 3

Operators

In this chapter, we discuss the generic model management operators that are used in our reintegration system. As described Section 2.3, there are five operators used: *Merge*, *Diff*, *Range*, *Apply*, and *Match*.

In Section 3.1, we briefly review *Merge*. We have fully analyzed and implemented the other four operators, which are discussed in this chapter. We discuss *Diff* in Section 3.2, *Range* in Section 3.3, *Apply* in Section 3.4 and *Match* in Section 3.5.

3.1 Operator *Merge*

Merge has been fully described and implemented in [14] as discussed in Section 2.2. Because it is a generic model management operator, it can directly fit into our schema reintegration system as one of the operators.

Merge works as follows: Given two models, M_1 and M_2, and the mapping, *map*, showing their relationships, *Merge* generates a new model, M_G, that unions the two models without any duplicates and conflicts. For example, Figure 3.1 shows an input of *Merge* and Figure 3.2 shows the result

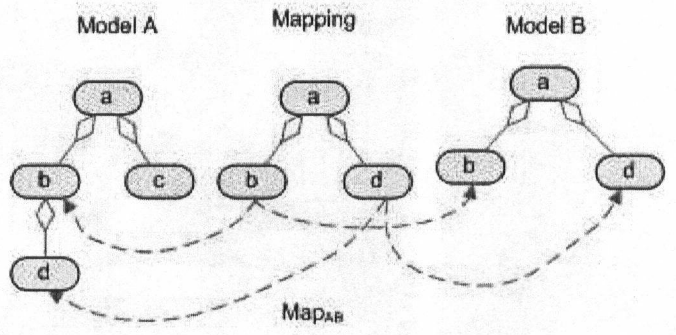


Figure 3.1: Mapping between Model A and Model B

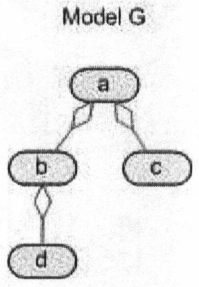


Figure 3.2: Merge result of Model A, Model B and Mapping

model of $Merge$ operation.

3.1.1 Input

The $Merge$ operator has three input models, M_1, M_2 and the mapping between them, Map. Map defines how M_1 and M_2 are related as described in section 2.1.2. One of M_1 or M_2 can be designated as the *preferred model.* When there are conflicts between M_1 and M_2, $Merge$ will follow the behavior of the *preferred model* to break the conflicts.

3.1.2 Output

The *Merge* operator generates a model as output, which contains all the information in M_1, M_2 and *Map*, but has duplicates eliminated and conflicts resolved. The duplicates are from the *Map*, which shows the relationship of elements in M_1 and M_2.

3.1.3 The Merge Algorithm

The Merge algorithm [14] includes the following steps:

1. Initialize: Create a new empty model G as the result.

2. Elements: Group the elements from M_1, M_2 and *Map* based on the mapping relationships. If there is a mapping relationship $R(e_i, e_m)$, where $e_i \in M_1 \cup M_2$ and $e_m \in Map$, then e_i and e_m belong to the same group. In the original paper, e_m must be an *Equality* mapping element for e_i and e_m to be grouped together. The difference between *Equality* and *Similarity* is in the mapping elements. In this thesis, mapping elements are all the same; the relationship R differentiates between *Equality* and *Similarity*. Therefore, rather than requiring e_m to be an *Equality* element, *Merge* requires that R is an Equality type mapping relationship. All the elements in M_1, M_2 and *Map* are divided into each group. Then for each group, create a corresponding element in G.

3. Element Properties: Intuitively, each element e created in G has a cor-

responding group of element(s) E_m from M_1, M_2 and Map. Element e has the union of the properties that all the elements in E_m have, excluding the $History$, ID and $HowRelated$ properties. When the same property p is contained in multiple elements in E_m, its value in e is determined following the order of Map, *Preferred Model*, *Any Model*. Formally, for a group of elements E_m, a property p, and some elements $e_m \in E_m$ have the property p, if there is an element $e_{map} \in Map \cap E_m$ that has property p, then the value of p in e is same as the value of p in e_{map}. If e_{map} does not exist, it tries the same rule but from the *preferred model*, then the other model. Whenever p is included in more than one element in the same model, its value is chosen arbitrarily.

For example, Figure 3.3 shows a simple mapping between model A and B, where model A is designated as the *preferred* model. Elements E_A and E' of model A both map to the same mapping element E_M of Map, which also maps to element E_B in model B. Suppose elements E_A, E' and E_B contains the same property p, but different values, v_A, v' and v_b respectively. When the two models are merged, elements E_A, E', E_M and E_B belong to the same group. The corresponding element $E_G \in G$ also contains property p. To decide the value of p, the algorithm first checks the *mapping*. Since the *mapping* does not have the property p, the algorithm will further check the elements in model A, which is the *preferred* model. Because both element E and E' are in model A and contain property p, the algorithm will arbitrarily choose one from them. Therefore, the resulting element E_g will have a property p, whose value is either v_A or v'.

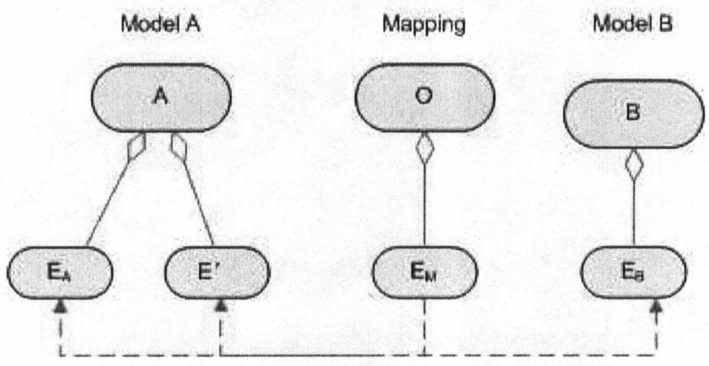


Figure 3.3: A Mapping between Model A and Model B, Model A is the *Preferred* Model.

The ID property is set to a new number assigned by the system. The $History$ property is set to the value of "*Merge(IDS)*", where IDS includes all the IDs of the elements in the group E_m. This gives the traceability of the elements in G to the original elements that it merges. If M_1 or M_2 is a mapping, the element e is a mapping element given the corresponding elements in group E_m, M_1 or M_2, are also mapping elements. If e is a mapping element, its $How - Related$ property is determined following the same selection order as the other properties.

4. Relationships: Each element in G has a corresponding group of elements in M_1, M_2 and Map. If there exists a relationship $R(e, f)$, where e and f are from different groups, a same type of relationship $R'(e', f')$ is created in G, where e' and f' are elements in G and correspond to e and f. Mapping relationships will not be created in G since if element e_1 and e_2 have a mapping relationship, they would have been in the same group and the mapping relationship between

them would not be copied to G. Since there is no "similarity mapping element" any more in the system, the steps to process the similarity mapping elements are ignored. Finally, the relationships that can be applied from other relationships in G are removed.

5. Fundamental conflict resolution: The previous steps generate a model G that includes the duplicate-free union of the input models. However, there may be some conflicts existing in G at the meta-meta-model level. This step resolves the conflicts according to the rules and strategies that have been specified.

For example: in Figure 3.1, *Mapping* shows how Model A and Model B are related. Basically, the elements are matched by names shown in the figure. Figure 3.2 shows the result after the algorithm is executed upon the mapping. Note that there was a has-a relationship between element a and d after all relationships are added based on the input models. However, since the has-a relationship from a to d can be implied from a to b and b to d, this relationship is deleted in the final result.

To build the reintegration system, we must extend the algorithm to take into consideration the use of support elements mentioned in Chapter 4. The *Merge* operation usually does not need extra support elements for the merged result elements if the input models have no support elements. However, if the input models have support elements, they need to be taken care of in the result model.

According to the merge algorithm, each element e in the result model G has a corresponding group of elements E_m which are from the input models. If all the elements in E_m are support element, the element e may act as a support element in G as well. It may not be necessary and may need to be deleted. Therefore, it will be marked as "delete" by creating a property of "*delete=true*" and then when the model is passed to the "*Support Elements Engine*" mentioned in Chapter 4, the engine will decide if it will be a support element or be deleted.

3.2 Operator *Diff*

The *Diff* operator is used to compute the difference between two models, given the mapping between the two models. Intuitively, it will find all the objects in the first model, that we refer to as the base model, that do not have corresponding matching in the second model. As in *Merge*, the mapping should have been created by some other operators, such as match, and is assumed to be given.

In previous works, such as [11] [3] [5], the *Diff* operator is either mentioned as a combination of other operators, or using extra mapping to show which elements are the *Diff* result and which are not. In this thesis, we give a detailed definition of *Diff* and the algorithm to compute it. We also use the support elements to make the result a self-contained model.

Diff works as follows: suppose there are two models M_1 and M_2. The

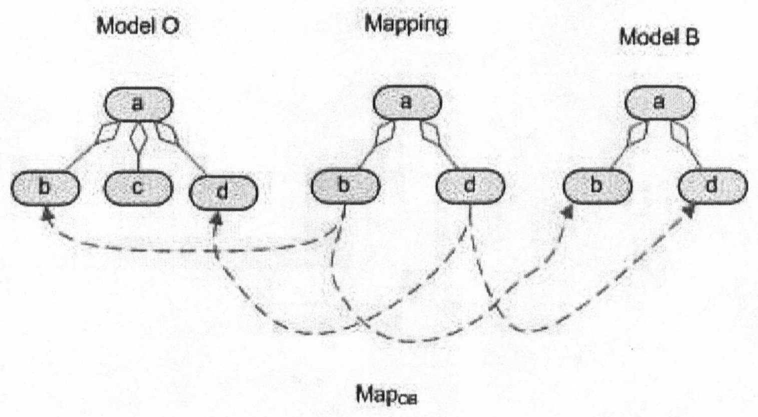


Figure 3.4: Mapping between Model O and Model B. Element c is not mapped

relation between them is also provided, which is presented as a mapping *map*. Formally, $Diff(M_1, Map, M_2) = M_d$ finds the elements that appear in the base model M_1 but not in model M_2 based on *map*. The result will be a new model named M_d.

3.2.1 Input

The *Diff* operator has two input models, M_1, and M_2, one of which has been denoted as the "base" model (say, M_1), i.e., the model which the result model is based on, and the Mapping *map* .

The mapping *map* tells how the model M_1 is related to M_2. Without knowing *map*, the computation will be meaningless. The format of the mapping is covered in depth in section 2.1.2

To see how *Diff* works, we present an example in Figure 3.4. Figure 3.4

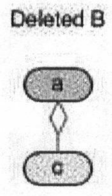


Figure 3.5: *Diff* result from Figure 3.4, assuming that O is the base model. Element a is shaded as support element

shows a mapping between two models, Model O and Model A, where we assume that Model O has been designated as the base model. Note that elements b and d of Model O are matched to corresponding elements in Model A with the "Equality" mapping relationship. Therefore, these two elements should not be part of the result model, since both elements appear both in the base model and in the other model. The element c has no mapping relationship, and it will be part of the result model. Sometimes, elements may have a mapping relationship with some elements in the mapping, but do not match to any other elements in the other model. These elements should also be included in the result model. The result is shown in Figure 3.5.

A "Similarity" Model Mapping Relationship means that the Model Element e_1 is similar to the corresponding element based on the mapping. Such elements need to be kept in the result model to show the difference.

3.2.2 Output

The expected output result will be a model M_d , which includes all the elements that are in the model M_1, but not involved in any "Equality" type Model Mapping Relationships with any Model Mapping Element in the mapping *map* that has Model Mapping Relationship with elements in the other model.

Therefore, intuitively, the result model M_d is a sub-model of the input base model M_1. Each Model Element E_d in M_d is generated from some element E_1 in M_1. Therefore, E_d has the same properties of E_1 and those relationships of E_1 such that the other elements of the relationships are also in M_d.

Furthermore, because the result model M_d contains Model Elements and Relationships that correspond to only some elements and relationships in M_1, the initial result may not be a complete model. As described in the [14], all the elements and relationships in a model must conform to the inclusion constraint: an element or relationship is included in the base model if and only if there is a path of containment from the root to that element or relationship. Therefore, some elements may not be included in M_d after applying all the inclusion rules (i.e., searching for a containment path). Clearly, we would like our result, M_d, to be a valid model, both from the perspective of wanting to make sure that the operators are composable and from the perspective of wanting to make sure that there is enough information to solve future problems using M_d. To solve the integrity problem, the model is processed to "Support Element Engine" as described in chapter 4. Algorithmically, this means that we have several phases to the algorithm. To compute the different elements in M_d, we first delete all the elements that have matching element in *map*; this corresponds to the elements (and relationships) that are truly in the difference of the models, but this result is not a model. Then we add back some elements and relationships as the support elements. However, because our model is not a tree (i.e., there can be more than one containment path from the root to a given node), we have to decide which deleted elements should be added to model M_d as the support elements. This can be done using the algorithms described in Chapter 4.

To summarize, the result model M_d includes the following:

1. Elements: For each element $e_1 \in M_1$ s.t. there is no $e_2 \in M_2$ and $e_M \in map$ s.t. $M_e(e_M, e_1)$, $M_e(e_M, e_2)$, $\exists$ a new element $e_d \in M_d$, s.t. e_d has a new ID, and inherits the name and all the properties except the $History$ property of e_1 . Give the new $History$ property of e_d as $Diff(e_1)$.

   - History property. Each result element will have a new History property, with the value $Diff(ID(e_1))$ Because the new History has the original element ID, we can easily trace back to the original and find the history of this element.

2. Relationships: For each relationship $R_d(e_1, e_2) \in M_1$, $e'_1, e'_2 \in M_d$, s.t. $ID(History(e'_1)) = ID(e_1)$ and $ID(History(e'_2)) = ID(e_2)$, $\exists R'_d \in M_d$ and R'_d conform to the *inclusion rules*. The relationships of the result model M_d are all inherited from M_1, which means each relationship $R'_d \in M_d$ must have a corresponding relationship $R_d \in M_1$ and the origin/destination element of R'_d corresponds to the origin/destionation element of R_d. However, not all the relationships of M_1 have corresponding relationships in M_d. Only those corresponding relationships of M_1 that have both its origin elements and destination elements (including different elements and support elements) included in M_d and conform to the *inclusion rules* are included in the result model M_d.

3. Supporting elements: As described above, supporting elements are those nodes added to support the model integrity of the result elements. These elements will include an extra property, named "Support" with value *true*, to indicate that they are support elements, but not part of the real result elements. They are added as described in Chapter 4.

3.2.3 Algorithm

To compute the *Diff* result, the algorithm includes the following steps:

1. *Duplicate*: Make a full copy of model M_1, including all its elements and all relationships in M_1 (i.e., we do not include relationships that

are incident on elements in M_1 but are not in M_1 according to the inclusion rules). Note that the mapping relationships in Map are not part of M_1, and therefore they are not copied. However, it is still necessary to keep the information about which elements correspond to those elements $e \in M_1$ that have mapping relationships $M_e(x, e)$, where $x \in Map$ and there exists some element $e_2 \in M_2$ s.t.$M_e(x, e_2)$. Let the set of such e elements $= E_{remove}$, which are the elements set to be deleted. To show which elements have mapping relationships, a list, L, is built to include all the elements in E_{remove}. The new model is called M_c.

2. *Marking*: For each element $e \in E_{remove}$, e is marked as "delete". These elements are not in the result elements set, but may appear in the model as support elements.

3. *Support Elements*: Giving the model M_c to the support element engine to check which elements should be added as support elements. Those support elements that are decided by the support element engine are marked "support" and do not have the "delete" mark any more.

4. *Delete*: For each element $e \in M_c$ that has the "delete" mark, e is deleted from M_c and all the relationships that have e as either an origin or a destination are also deleted.

5. Finally, return M_c.

3.2.4 Related Work

There are several papers that mention the diff operation. They are summarized as follows:

- Rondo[11]: There is no explicit definition of *diff* , but it uses $All(s1) - Domain(s1\_s2)$ to do the same job as diff if the two models are $s1$ and $s2$. This is described as: all elements of s_1 without the matched (and thus not deleted) element. The result is a new schema and a mapping between the result schema and the original schema, which describes how these two are related. In Rondo, modules are mapped using Morphisms, which are just binary relations.
  For example, the following shows two schemas:
  s1.Orders(Oid, OrderDate, Employee, Customer, PONum, SalesTaxRate)
  s2.Orders(Oid, OrderDate, Customer, PONum, SalesTaxRate, ShipDate FreightCharge, Rebate)
  In Rondo, the mapping is represented using the following morphism:

s1	s2
Oid	Oid
OrderDate	OrderDate
Customer	Customer
PONum	PONum
SalesTaxRate	SalesTaxRate

  Then the operation combination *All(s1) - Domain(s1_s2)* would be the whole elements set in s1, which include all the attributes of $s1$, minus the left part of the morphism, and the result is simply computed by set operations. This result elements set itself cannot represent the data structure. The result elements set has to be combined with the original schema to show a meaningful result, so that it can be used in other operators.

Rondo does not mention how the "-" operator is implemented. It does not mention support nodes either. It uses mapping (morphisms) indicating how the deleted result module is related to the original module. Our method is more self-contained.

- Vision[5]: Difference is believed to be basically the same as matching, except that the answer needs to highlight the differences. The diff operation is said to be a Full OuterMatch. In *Vision*, it treats diff as a contrast of match. It does not explicitly describe the input and output of the diff. However, the desired result can be computed from some hints: In Vision, schemas are represented as first class objects; Mapping is also a first class object. In the description of Match, it tries to find how the domain and the range objects are related. Therefore, in the Differencing, it should be assumed that the main task is to find how the domain and the range objects differ. It is different from what we have described about diff, where we assume the mapping is in hand, but Vision somehow tries to find the mapping (and difference).

- Applying Model Management to Classical Meta Data Problems [3]:

  In this paper, the input is the model and the mapping. (M_1, map_1) and the result also include two parts, the objects that are not referenced in the mapping and a new mapping between the result model and the original model: (M_1', map_2).

Three problems are considered in this paper. 1. Root is always included in the result object. 2. To ensure the result model is a well-formed model, all the objects on the path of has-a relationships from the root to the result objects are also included. These added objects are support objects. 3. Use a new mapping between the original model and the diff result to mark the support objects.

Marking support objects in the result is treated as introducing another structure (the marked model) and is avoided. Their paper defines the desired input and output of diff in detail. However, in our diff, we will have different methods to add support elements, which will ensure least elements are added. Moreover, we will consider all the inclusion constraints rather than only the "*hasA*" relationships. Because in our system, we have created the structure of "property", we can utilize this structure to mark the support elements. Therefore, we do not need to add another mapping, which may otherwise make the system more complex.

In the *Diff* operator of this thesis, we keep all the input models and output models as first-class models and we use "Support Elements" to keep the integrity of these models. Our *Diff* allows "Support Elements" appearing in both input models and output models.

3.3 Operator *range*

Given two models, one being the *range* model and the other being the *domain* model, and the mapping between them, the *Range* operator finds the elements that appear in the *range* model of a mapping that have corresponding mapped elements in the *domain* model. The mapping itself is presented as a model, which contains elements that correspond to the domain model (M_d), the range model (M_r) and other mapping elements (E_m). The range model here is not the same as the result model (M_{range}) by the Range operator. The Range result model M_{range} only contains elements that refer to part of the range model M_r. All the elements in M_r that are referred by elements in M_{range} must be associated in a mapping relationship with some elements in the domain model. We will give more details when discussing the output part.

3.3.1 Input

The *map* model includes the domain model, the range model and the mapping elements, which describe the relationships between elements in the domain and the range. Therefore, the *map* model itself provides enough information for the *range* operator. It is the only input for the operator *range*.

3.3.2 Output

The output result would be a new model that copies the range model in the input *map* model, but only those elements that have been "mapped" to some elements in the domain model are the result elements and the result elements would have new ID's. In order to keep the result as a complete model, some unnecessary elements are also needed to act as support elements.

Formally, for each element $e \in M_r$, there is a corresponding element $e_{range} \in M_{range}$ such that $\exists e_o \in M_{map}$ (the Mapping), and $\exists e_d \in M_{domain}$ (the domain model), and there exists model mapping relationships r_1 and r_2 such that r_1 is between e_o and e_{range} and r_2 is between e_o and e_{domain}.

The relationships and support elements would be added in the same way as for operator diff.

In each element, $range(id(e_{range}))$ will be added to the history property.

3.3.3 Algorithm

The algorithm is very similar to the algorithm in the operator *diff* in that it also selects part of the model in *map*, but it selects the contrary part to what *Diff* selects.

The *range* operator includes the following steps:

1. Duplicate: Extract the *range* model from the input mapping. Make a new model, M_r, which copies all the elements (with new ID's) and

relationships of model *range*.

2. Mark: Firstly, iterate all the elements in M_r and mark them as "delete". For each element $e \in M_r$, find the corresponding element $e_{range} \in range$, such that e is generated by copying e_{range}. If $\exists e_m \in M_{map}$ and $e_d \in M_{domain}$ such that there exists Model Mapping relationship $r_1(e_m, e_{domain}$ and $r_2(e_m, e_{range})$, remove the "delete" mark of e. In this step, all the elements that should be in the result model are selected and all the other elements are marked as "delete".

3. Add Support Elements: Give the model M_r to the "Support Element Engine" to check which elements should be added as support elements. Those support elements that are decided by the "Support Element Engine" are marked "support" and do not have the "delete" mark any more.

4. Delete: For each element $e \in M_r$ that has the "delete" mark, e is deleted from M_r and all the relationships that have e as either an origin or a destination are also deleted.

5. Return: Finally return model M_r.

3.3.4 Related Work

Operator *range* was only previously discussed in Rondo[11]. In Rondo, *range* is defined as *Domain(Invert(map))*. It firstly reverse the role of *domain* and *range* in the input *map* and then return the *domain* model. It is because the operator *Domain* is defined as a primitive operator and the operator *range* can be generated using existing primitive operators.

Because the *map* in Rondo uses morphisms, in other words, the mappings are element pairs between *domain* and *range*, the *domain* or *range* operator can then simply extract the needed half in the morphism. It is a straight-forward operation.

The *range* in this thesis uses first-class models, so it needs to extract some elements that are really acting in the mapping. Here we also show the detailed algorithm and use "Support Elements" to make the result self-contained. The *domain* operator will be exactly the same and only care about the schema in the other side of the mapping.

3.4 Operator *Apply*

The operator *Apply* is different than the other operators. It is a generic method that takes effect on each element in the input Model.

There is no specific target on how *Apply* affects each element. Instead, *Apply* provides a generic template for a function which affects the elements of some model. When *Apply* is used, it must bind to some real function, Γ, to perform the expected functionality. Therefore, operator *Apply* is like a delegate function that has Γ as a internal function pointer.

In this section, we discuss both *Apply* and function Γ used in the reintegration system.

3.4.1 *Apply*

3.4.1.1 Input

The input of *Apply* includes the following:

- A Model, M, which contains the elements that are to be affected by the operator.
- An internal function, Γ, which is the real function that will take effect on the elements in M.

3.4.1.2 Output

The output of *Apply* is a Model, which is generated from model M, but all the elements have been processed with function Γ.

Formally, $M_A = \mathit{Apply}(M)$, and for each element $e_A \in M_A$, $\exists e_M \in M$ such that $e_A = \Gamma(e_M)$.

3.4.1.3 Algorithm

The algorithm for *Apply* includes the following steps:

1. Duplicate: Create a new model, M_A, which copies all the elements, E_o (with new ID's) and relationships of model M. Update the *History* property of element $e \in M_A$ as $\mathit{Apply}(e_o)$.

2. Execute: For each element $e \in M_A$, execute function $\Gamma(e)$. Note that because the functionality of Γ is not unique, the result element may be changed in one of several ways. For example, it may be deleted. However, because the system always guarantees the integrity of the model, if the element is to be deleted by function Γ, it is marked as "delete" and will be processed with the Support Element Engine to decide if it should be deleted or act as a support element.

3. Add Support Elements: process the model M_A with the Support Element Engine. The elements that are marked as "delete" but need to exist as support elements will be marked as "support".

4. Delete: For each element $e \in M_A$ that has a "delete" mark, e is deleted from M_A and all the relationships that have e as either an origin or a destination are also deleted. If M_A is a mapping, the model mapping relationships of "support" elements are also deleted.

5. Return model M_A.

3.4.2 Function Γ in Reintegration

The function Γ that is created to be used by *Apply* in our Reintegration system only has one target: it checks the input mapping element e to see if the elements that e links to in the domain and range are identical. If they are the same, element e is deleted.

Two elements, $a \in domain$ and $b \in range$, are believed to be identical if:

- They are mapped by a mapping element and "equality" mapping relationships (i.e., $\exists$ an element $m \in Map$, s.t. $M_e(m, a)$ and $M_e(m, b)$).

- They have all the same properties (both the name and the value), except for the "history" property. If each property $p(X)$ of a and X is not ID or $History$, there is a property $p'(X)$ of b and $p(X) = p'(X)$.

- Element a and b should have same kind of relationships, of which a and b are the destinations, and the origins should match each other. For each non-mapping relationship $r(x, a)$ in the domain, there exists a relationship $r'(y, b)$ in the range, where r' has the same type as r, and there exists a mapping element e in map, such that there exists mapping relationships $M_e(e, x)$ and $M_e(e, y)$. Then a and b are believed to be identical. Here it only considers the relationships from the "parent", and if a child changed, it will only be considered when the child element is inspected. Otherwise, if both the origins and destinations are checked, one change will be caught in two places and be redundant.

3.4.2.1 Input

The input is an element e of a Mapping Model M. It can be a normal element or a mapping element.

3.4.2.2 Output

Strictly, the function does not have Output. Instead, it modifies the input element e and updates e with the proper action. If the function needs to delete e, it only mark it as "delete".

3.4.2.3 Algorithm

The algorithm includes the following steps:

1. From the given model mapping model M, extract the *domain* and *range* in M - M_d and M_r respectively. For the given mapping element e, find the elements $e_d \in M_d$ and $e_r \in M_r$.

2. Comparison:

   - Firstly, for each property $p_d \in e_d$, such that $Name(p_d) \neq$ "*History*", if $\exists$ property $p_r \in e_r$ such that $Name(p_r) = Name(p_d)$ and $Value(p_r) = Value(p_d)$, continue; otherwise, mark e as "delete" and return.

   - Secondly, for each non-mapping relationship $r_d \in M_d$, such that $destination(r_d) = e_d$ and $origin(r_d) = e_{od}$ ($e_{od} \in M_d$), check that $\exists r_r \in M_r$ and $destination(r_r) = e_r$, $origin(r_r) = e_{or}$ ($e_{or} \in M_r$), such that $\exists e_m \in M$ and $\exists$ model mapping relationship $r_1, r_2 \in M$, and $destination(r_1) = e_{od}$, $destionation(r_2) = e_{or}$. If so, continue; otherwise, mark e as "delete" and return.

3.4.3 Related Work

The operator *Apply* was first discussed in [3], where it gives the formal definition. The purpose of this operator is to reduce the need for application programs to navigate a model. The input function can define a purpose and the *Apply* operator defines the rules and algorithms to traverse the model. However, it does not have any implementation details. Here we have discussed the input, output and algorithms in detail, and the need to involve support elements.

3.5 Operator *Match*

Match finds the mapping relationships between the elements in two input models. In this thesis, we do not try to solve the general matching problem, which is the subject of many other papers such as [15] [11] [3] [5]. The mapped elements are based on *History* properties of the elements in the input models. It is assumed that only the model management operators appear in the History properties. If two elements in two different models can be tracked to the same element from the *History* properties, the two elements match to each other. The details of tracking from the *History* properties based on different operators are discussed in the algorithm part.

3.5.1 Input

The *Match* operator takes two models as input. The ID of each element in the models will be used to track which elements are derived from the same element and can be matched. Here, it is assumed that the History properties of all the elements are available, so that each element can be tracked based on the full history information.

3.5.2 Output

The Match operator returns a mapping model. This mapping model contains the two input models as *domain* and *range*, the model mapping elements E, and the model mapping relationships R_{map} that present the matching relationships. It also contains relationships R_{model} that connect these model mapping elements E. These relationships are based on one of the two input models. The purpose of these relationships is to link the mapping model elements and they should not create any additional information for the input models. Because the output model would not include more elements than any of the input models, the model relationships of the output model can be created based on any of them. Sometimes, only part of the input elements could have been matched. In this case, some mapping elements might have been created with no mapping relationships to any other model elements. They are only used to connect the mapped mapping elements to the root based on the relationship given in the input models. These elements would act as support elements.

3.5.3 Algorithm

The mapping model includes elements and relationships.

1. Elements:

   - Create a new Mapping Model M, and make the input *domain* and *range* the internal domain and range respectively. Create a model mapping element e_m for each element $e_d \in domain$.
   - Build the element correspondences, Cor_u. The matched elements in *domain* and *range* may not be generated from an element directly. They may have been manipulated by several operators in multiple steps. Therefore, the only available source of all the elements is the whole space, which is called the *universe*, U, here. In order to create the element correspondences, for each element $e \in U$, extract its *History* property P_h. P_h usually has the format of *OPERATOR(IDs)*, where *OPERATOR* is one of the operators: *Match, Diff, Merge, Compose, Apply, ModelGen, Select, Range, Domain* and *Delete.* The *history* with any of these operators except $Match$ provide tracing information. The *ID* of e and IDs can be put into one correspondence. For example, if $ID(e) = 300008$ and the $History$ property is $Merge(300001, 300002)$, the created correspondence will be {300008, 300001, 300002}. The number of IDs in $OPERATOR(IDs)$ depends on different operators, which is summarized in Table 3.1:

     If either $ID(e) \in$ correspondence $cor1$ or $IDs \in cor2$ before, then the corresponding IDs in $cor1$ and $cor2$ are combined and we generate a new correspondence which includes all the related IDs.

Operator	Created Correspondence
$x = Match(y, z)$	N/A
$x = Diff(y, z)$	{x, y}
$x = Merge(y, z, w)$	{x,y,z,w}
$x = Compose(y, z)$	{x,y,z}
$x = Apply(y)$	{x, y}
$x = Copy(y)$	{x, y}
$x = ModelGen(y)$	{x, y}
$x = Select(y)$	{x, y}
$x = Range(y, z)$	{x, y}
$x = Domain(y, z)$	{x, y}
$x = Delete(y, z)$	{x, y}

Table 3.1: Element Correspondence for Each Operator

- Locate the Matched Elements: M has the same number of elements as *domain*. Therefore, for each element $e_d \in domain$, $\exists e_m \in M$, such that e_m is generated from e_d. If $\exists e_r \in range$, $core_e \in Cor_u$, such that $ID(e_d) \in core_e$ and $ID(e_r) \in core_e$, then e_m is marked with a new *History* property as $Match(ID(e_d), ID(e_r))$; otherwise, e_m is marked as "delete".

2. Create Relationships: M has the same corresponding relationships to the ones in *domain*. For each non-mapping relationship $r_d \in domain$, let $Origin(r_d) = e_{od}$, $Destination(r_d) = e_{dd}$; Find the elements e_{om}, $e_{dm} \in M$, which are corresponding elements of e_{od} and e_{dd} , create relationship $r_m \in M$, such that $Origin(r_m) = e_{om}, Destination(r_m) = e_{dm}$ and the relationship type is same as r_d.

3. Put M into the Support Element Engine to find the necessary support elements.

4. Remove elements in M that are marked as *delete* and return M.

In the above algorithm, it is assumed that the matched elements in *domain* and *range* are one-to-one. However, it may happen that the matching is one-to-many or many-to-many. In this case, the above algorithm only need a small modification: limit the usage of correspondence to one time. The "Locate the Matched Elements" step will be changed to:

- Each element $e_d \in domain$, $\exists e_m \in M$, such that e_m is generated from e_d. Find the correspondence $cor_e \in Cor_u$, such that $e_d \in cor_e$; Find all the elements $E_c = \{e_c | e_c \in cor_e$ such that $e_c \in domain$ or $e_c \in range$; Mark the *History* property of e_m as $Match(ID(E_c))$; **Remove** cor_e **from** Cor_u. If the correspondence cor_e cannot be found, mark the element e_d as "delete".

In this way, if elements $e_1, e_2 \in domain$ match to the same group of elements $E_r \in range$, when e_1 is processed, the *history* property of the corresponding element e_m will include all the elements,e_1, e_2 and all E_r. Then, this corresponding tuple will be deleted. When e_2 is processed, it cannot find the corresponding elements. Therefore, the element $e_{m2} \in M$ that is generated from e_2 will be marked as "delete" and will probably be deleted or act as a "support element", but not have mapping relationships to any elements in *domain* or *range*. This guarantees that each group of matched elements only have one mapping element in M.

3.5.4 Related Work

The *Match* operator has been widely discussed,(e.g., [15] [11] [3] [5]). Generic *Match* is not discussed in this thesis. The *Match* operator in this thesis is used for the Reintegration system environment and the matching process is done with the help of the *History* properties.

3.6 Summary

In this chapter, we have discussed the five operators that are used in the Reintegration system. Four of them have not been discussed in such detailed level before. We have formally defined the input, output and algorithms to be used for each operator. Moreover, we have used the idea of Support Elements on all the operators, which features heavily in each operator, and has not previously been described in detail. To handle support elements, we designed and implemented the Support Elements Engine, which will be discussed in Chapter 4. It is a generic engine that can be used by any operator. This method makes the input and output of each operator more self-contained and more generic.

Chapter 4

Support Elements

In Model Management, most operators generate new models. It is valuable to ensure that all outputs of operators are composable For this to be true, the output of each operator must be a *valid model.* Here the term *valid model* means that all the elements and relationships in the output must conform to the inclusion rules mentioned in chapter 2.1.3. However, some operations will delete some elements or relationships from a model and this may make some elements "unconnected", and thus not included in the model. Consider the example shown in Figure 4.1. It is a mapping between *Product*1 and *Product*2. If we apply the *Diff* operation on it to compute the difference between them, the result only includes the *Product* element (the root) and the *Pid* element. They are not connected and it is hard to say what relationships they should have between them.

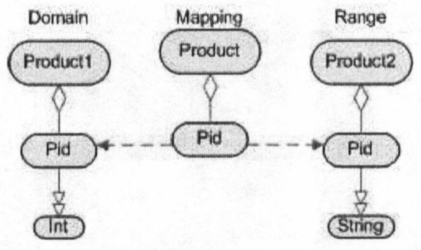


Figure 4.1: Support Elements Mapping Example

In order to keep the result still as a complete model, some elements are added to the result model to support the structural integrity of the model. They are originally in the model, but may have been deleted. However, because these elements should not be included in the result, they must be identified from the other elements in the model. Here they are called *support elements*.

The *support elements* idea was first proposed in [3]. To identify the support elements, it uses a *mapping* between the result model and the original model. Only the result elements have mapping relationships to the original model elements, and those support elements are not mapped. This way has its benefits. It does not require a new data structure to *mark* the support element and it only uses available structures, model and mapping, to identify the result. However, the result of the operators that need support elements will then become a pair $< M, mapping >$. When the results are further used as input for other operators, it will make the process complex. The operators cannot always expect a simple model as input, but sometimes the input is a pair, which is actually a new data structure. If there are a number of such steps, the result model will become more complex and hard to follow or trace.

Another method to mark the support elements is also mentioned in [3], which is to simply mark the elements as *support*. The disadvantage of this method is that it introduces another structure, the *marked model*. This method is not recommended. In our framework, we have the *property* attributes in each element. We can simply use the *property* as a marker to identify if the element is a *support* element.

4.1 Definition

In model management, *Support Elements* are those elements that are included in the model only for model completeness and integrity purposes. They are included in the model, but except for linking other elements together, they do not have any other functionalities. If any of the support elements are not included, the rest of the elements would not be a complete model anymore. If elements that need support are deleted, those support elements that only support the deleted elements should also be deleted.

Formally, for a model M with elements E_M, we divide E_M into the sets E_M^s, where E_M^s contains all support elements, and E_M^e, where E_M^e contains all elements in the model that appear as normal, non-support elements. These sets are disjoint and completely partition E_M. That is $E_M^s \cap E_M^e = \emptyset$ and $E_M^s \cup E_M^e = E_M$.

Additionally, let M' be the attempt at inducing a model from the elements of E_M^e. That is, M' is built by (1)adding E_M^e to M', and (2) adding to M' each relationship r, s.t. $origin(r) \in M'$ and $dest(r) \in M'$. Then based on the inclusion rules M' is not a model. Similarly, *all* elements in E_M^s are required for the result to be a model. That is, let M'' be any model $M - E_s'$, where E_s' is a non-empty subset of E_M^s. Then by the inclusion

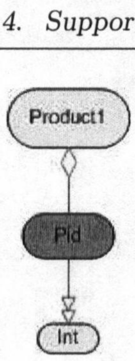


Figure 4.2: The *Diff* result in Figure 4.1 with the support elements needed to make it a valid model. Support elements are visualized as shaded nodes throughout this thesis.

rules, M'' is not a model.

To visualize the difference, we shade the elements to identify the support elements. For example, the *Diff* result from the example in Figure 4.1 would be the model shown in Figure 4.2. Note that the element *Pid* is acting as a support element and is shaded. In the underlying representation, this is represented by adding a property *support* = *true* to the *Pid* element to show that it is now a support element.

4.2 Usability of Support Elements

Though originally motivated as a requirement for *Diff* [3], we show that almost all the other operators may also need support elements. For example, in the *range* operator, only the elements in the range model that have mapping relationships will be in the result. This means that only some elements of the original model will be in the final result. In the operator *match*, two models may have a leaf element matched, but not on their ancestors. To make the mapping be a first-class object, we need make their ancestors support elements. In order to keep the result as a model, we have to use support elements to help. If we have support elements existing in the result model, the result may be further used in other operators. This means that support elements may be quite common in both input and output models.

It is valuable to keep all our input and output as full models. In this way, we can make our operators more generic, composable, and easy to maintain. The output of one operator can be used as the input of another operator. If we are expecting a kind of model, we can always assume that it is a full completed model that conforms to the inclusion rules. It will be troublesome if the *Diff* operation only generated a set of discrete elements without structure and cannot be used in those operators that need models as input.

Our purpose is to keep the result model complete. However, we should not add any extra information to the result and we should not lose any information either. For example, in Figure 4.2, we cannot just add in a type-of relationship, or any relationship between the element *product* and *int* that would make it a model. This addition may make the operation not accurate. By adding the support element *pid*, as shown in Figure 4.2, we can keep the original structure and mark those elements that do not belong to the result model. We can achieve the goal of not adding any extra information or losing anything essential. Moreover, the support elements also help in improving the traceability of operations. If a model is manipulated

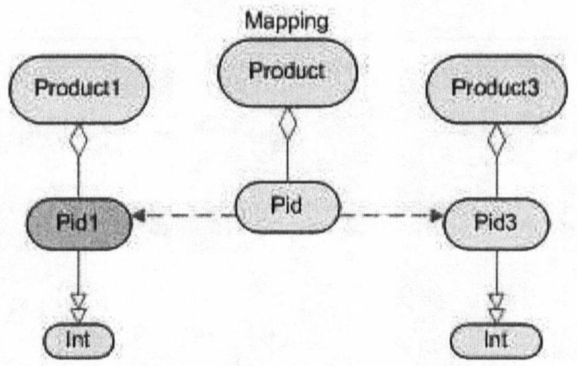


Figure 4.3: Mapping between a Model with a Support Element and a Normal Element

by a series of operations, some support elements may become not supported anymore in the final result; but, we can still trace the generation of the elements from the *history* property. Suppose the model in Figure 4.2 is used in a *merge* operator with another model as shown in Figure 4.3. In the result model, *Pid* will be a normal element (non-support) with a history property of $merge(ID(pid_1), ID(pid_3))$.

4.3 Support Elements Adding Algorithms

In this section, we discuss the engine for adding support elements that the algorithms use to determine which support elements to add.

We consider as input to the problem of adding support elements a model M, and a set of elements $E^e_M \subseteq E_M$ that are required elements in our result: the model M'. Our goal is to find the support elements set E_s to add to M'. By adding the set E_s, the partial graph M' of M includes only elements in

E_s and E^e_M, and the relationships whose origin and destination are all in $E_s \bigcup E^e_M$. The partial graph M' is then our return result.

Because our model is a graph, not a tree, for a given element that needs support elements, there may exist several paths to the root. We have to select a proper path based on some expectations or rules and add the elements in the path but not in E^e_M.

First we determine which elements need support, E_{ENS}, in Section 4.3.1. Then we determine which elements E_{ENSR}, $E_{ENSR} \subseteq E_{ENS}$, can represent all the elements in E_{ENS} in Section 4.3.2. We only need consider E_{ENSR} when we look for support elements. Finally, we discuss the different algorithms used to find the support elements for E_{ENSR} in Section 4.3.3.

4.3.1 Algorithm to Find Which Elements Need Support

Firstly, we need find our target elements set that need support, which is called Elements Need-Support (E_{ENS}). Basically, any graph search algorithm can be used to recursively include the elements by the inclusion rules starting from the root, which is always in the model by definition. Here a breadth-first search method is used. A queue can be used to keep track of which elements to explore next. It is also necessary to avoid cycles by remembering which elements have been used in the queue.

The algorithm takes a model M' as input. Model M' can be any valid model, including a mapping. There are two categories of elements of M'.

One is the normal elements, E^e_M, which are the elements that the result model must contain and some of which need support elements. The other category is the elements, E^d_M, that can be deleted and which may need to be support elements. These elements are different from E^e_M in that they have a property *delete* with value *true*, meaning that they are to be deleted if they are not support elements. Note that this input is also the input of the whole "Support Element Engine".

The expected result of this part of algorithm is a set of elements, $E_{ENS} \cup E^e_M$, that cannot be included in the model without the support of some elements, $E_s \cup E^d_M$.

The algorithm details are as follows:

```
1. Define element set $E_{covered} = \emptyset$, $E_{queue} = \emptyset$;

2. Put the root element of $M'$ into $E_{covered}$ and $E_{queue}$;

3. While( $E_{queue}$ IS not Empty) Do:
        LET $e = Dequeue(E_{queue})$;
        IF $e$ has been visited, NEXT;
        FIND the corresponding element $e' \in M$ that corresponds to $e$;
        FOR EACH inclusionrule relationship $R$
            IF $e'$ is the origin of a relationship R;
            LET $e_d$ be the corresponding destination;
            IF $e_d \in E^e_M$
                PUT $e_d$ into $E_{covered}$;
```

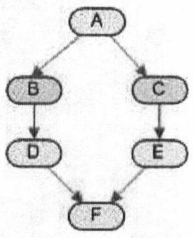


Figure 4.4: Elements Need Support & Representatives

```
        PUT $e_d$ into $E_{queue}$;
    END WHILE

4. RETURN $E_{ENS} = E - E_{covered}$;
```

4.3.2 ENS Representatives

Not all the elements in E_{ENS} need to be considered when adding support elements. In the example shown in Figure 4.4, elements B and C are not in E^{e}_{M}. E_{ENS} includes elements D, E, F. However, it is not necessary to consider element F when adding support elements, because if either D or E is included in the model by adding proper support elements (B, C here), F will automatically be included in the model. Although D, E, F are connected together, it is still necessary to consider both D and E as the ENSR, because they do not have ancestor- descendant relationships, and adding D will not ensure that e is in the model, and vice-versa.

Therefore, if relationship r is between elements A and B, $A \in E_{ENS}, B \in E_{ENS}$ and $II(r, B)$ (i.e., r implies inclusion of B, see Section 2.1.3), by which it can induced that if A is in the model then B is also in the model, then it is only necessary to consider adding A as a support element, and not B, since adding A would also include B . Therefore, in this step, only some elements from the E_{ENS} set are selected as representatives. Only these ENS Representatives (ENSR) are considered to add support elements. Each ENSR element may represent a set of elements in E_{ENS}. This may reduce the number of elements to be considered and speed up the overall process.

This step takes the following as input: the model M' and the output elements set E_{ENS} from previous step in section 4.3.1. The algorithm is: Iterate each element $e \in E_{ENS}$ and test if $\exists e_p \in E_{ENS}$, such that if $e_p \in M$ then $e \in M$ by inclusion rules. If so, we delete e from E_{ENS}. Finally, we get the set E_{ENSR}, the representative elements set.

4.3.3 Support Elements Adding Engine and Algorithms

Because support elements are used in almost all the operators in the reintegration system, the adding process will be a separate, reusable engine, so that all the operators can share the process and algorithms.

The Engine has the following input and output:

1. Input: The input includes the original model M, the current uncompleted result model M', whose elements set as E^e_M. The whole el-

ements set of M as E. Here M' is a copy of M, except that the elements which are not in the result set E_M^e are marked as *delete* by adding the property *delete* = *true*. If these elements are marked as *support* later by adding the property *support* = *true*, they will be kept as *support* elements. For those elements in E_M that are not marked as *support*, they will be deleted in the final step. In this way, the algorithm can operate on one model and avoid the frequent steps to locate the corresponding element in the original model.

2. Output: Updated model M' such that all the elements of ENS are included by inclusion rules.

To find the support elements set, there are many different ways. Here we list three methods. Each of them has different advantages and disadvantages. Section 4.3.3.1 discusses the "Shortest Path Algorithm", which is a fast algorithm and the most straightforward way: we consider each element in $ENSR$ separately. Section 4.3.3.2 discusses the "Least Support Elements" algorithm, which is valuable when the number of support elements is desired to be as small as possible. For example, when support elements are costly for storage reasons, it will be better to add as few support elements as possible. Section 4.3.3.3 discusses the "Greedy Algorithm", which is useful when the model structures are complicated and the elements interact with each other a lot.

4.3.3.1 Shortest Path Algorithm

One straightforward method is to find the shortest path for each ENSR element. For each ENSR element, the algorithm uses a reverse breadth-first search to track its ancestors. The ending condition is the tracing process reaches to either the root element, or any element that is already included in the model, or an element that has been marked as a support element by using the inclusion rules. Then all the elements in the found path that did not belong to the result elements set are included in the result model as support elements. Furthermore, because the path is found with breadth-first search, it is guaranteed that it is the shortest path. The detailed algorithm is as follows:

1. For each element $e_m \in ENSR$, put e_m into the processing queue Q.

2. While the queue is not empty and there is no path found, let e_q be the first element in Q , use reverse breadth-first search method. For each relationship r that is in the inclusion rules and for which e_q is the destination, find the elements E_o, such that for each element $e_o \in E_o$ there exists a relationship r between e_o and e_q and $II(r, e_q)$.

3. For each element $e_o \in E_o$, if $e_o \in E_{ENS}$, put it into Q and record the corresponding path; otherwise, if $e_o \in E^{e}_{M}$, or $e_o \in E_{ENS}$ and e_o has a property of "*support* = *true*", then the shortest path for element e has been found.

4. Find out the whole path from e to e_o. In this path, if any element

is marked as *delete*, change its property to *support*. For each such element, recursively apply the *inclusion rules* again to find its descendant. If any of its descendant is in the E_{ENSR}, remove it from E_{ENSR}.

5. When all the elements in E_{ENSR} have been covered, the algorithm is done. Finally remove those elements from M' if they still still marked as *delete* by having the property of "*delete=true*".

The advantage of this algorithm is the speed. Even to reach the root, the average case complexity for this algorithm is $O(\log(n))$. However, since we need to keep track of each level, the space usage is very high.

4.3.3.2 Least Support Elements Algorithm (LSE)

In some applications, it may be valuable to use as few support elements as possible. This algorithm can minimize the number of support elements to be added overall. This means if n support elements are added to the model and make up all the elements in ENSR included in the model by the inclusion rules, it would be impossible to find any other better combination with fewer than n support elements that could also include all the elements of ENSR in the model. To determine the least support elements, it is necessary to find all the possible paths that all the ENSR elements may use to be included in the model. The steps are as follows:

1. The first step is very similar to the Shortest Path Algorithm in that it also uses the reverse breadth-first search for each element $e \in E_{ENSR}$. It also keeps an elements tracing queue Q for the breadth-first search

similar to that for the Shortest Path Algorithm. The difference is that it does not stop when it finds the first element that is already in the model. Instead, it will only record this path (but not trace this path), and it will continue with other possible paths. When all the possible paths have been found, which means Q is empty, we terminate the tracing process. We record all the possible paths, but do not delete elements from ENSR when we find all the paths, because we are not sure yet which path will be used in the final result.

2. Repeat the above step for each element in ENSR. Then we have all the paths set for $e_i \in E_{ENSR}$. For example, the following may be the final result of the paths to be considered:

   $path(e_0)[0] = \{e_{11}, e_{12}, e_{13}\}$

   $path(e_0)[1] = \{e_{11}, e_{15}\}$

   ...

   $path(e_1)[0] = \{e_{11}, e_{12}, e_{17}, e_{18}\}$

   $path(e_1)[1] = \{e_{19}, e_{20}, e_{21}\}$

   ...

   $path(e_2)[0] = \{e_{22}\}$

   ...

   $path(e_n)[k] = \{e_{21}, e_{26}, e_{27}\}$

   The paths may have intersections. Each element can choose one of the available paths to add support elements. Different combinations of paths chosen by each element in ENSR have different number of elements by union the elements of selected paths.

3. Compute the total number of elements of each path combination by selecting a path from each element's path set. For the above example, we have:

   $count(path(e_0)[0] \cup path(e_1)[0] \cup path(e_2)[0]) = 6$, (counting elements e_{11}, e_{12}, e_{13}, e_{17}, e_{18}, e_{20})

   $count(path(e_0)[0] \cup Path(e_1)[1] \cup path(e_2)[0]) = 7$, (counting elements e_{11}, e_{12}, e_{13}, e_{19}, e_{20}, $e_{21,22}$)

   ...

4. Pick up a combination of paths with the minimal *count* and mark all the elements in the selected paths that are not in the model (marked as *delete*) and mark them as *support*.

LSE guarantees that only minimum support elements are selected. However, the complexity of this algorithm is relatively high.

4.3.3.3 Greedy Algorithm

The idea of the Greedy Algorithm is to select the element that can cover most of the ENSR elements to be a support element in each step. The detailed algorithm is as follows:

1. Find all the elements E_{dp}, such that for each element $e_{dp} \in E_{dp}$, $\exists e \in ENSR$ and a relationship r between e_{dp} and e, and $II(r, e)$.

2. For each element in E_{dp}, compute the out degree: the relationship r

that is included in the $inclusion - rule$ and $II(r, e)$ for some element $e \in ENSR$.

3. Select the one with highest out degree, e_h, and mark it as a support element by adding property "*Support = true*".

4. Put e_h into ENSR.

5. Remove those elements in ENSR that can be covered by e_h.

6. Find the direct parents of e_h and put them into E_{dp}.

7. Repeat from step 2, until ENSR is empty.

The Greedy Algorithm cannot guarantee an optimal result (i.e., the least support elements). For example, Figure 4.5 shows an example of using the Greedy Algorithm. The algorithm may choose $D_0 ... D_n$ as the support elements, because D_n has the highest out degree. However, we know that the least support elements should be only B and C. This example demonstrates that in some cases, the Greedy Algorithm may generate a worse selection of support elements. However, it can generate a "better than naive" result and has an acceptable complexity. In practice, since database schemas are mostly well structured and the depth of the model graph is small, the Greedy Algorithm can often find the "important" elements to be added as support elements first. These elements can play a good role on supporting elements in $ENSR$.

Redundancy in the Greedy Algorithm

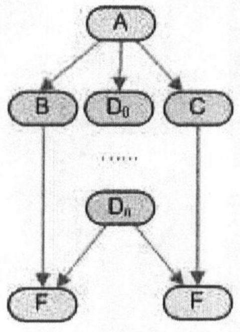


Figure 4.5: Greedy Algorithm Example

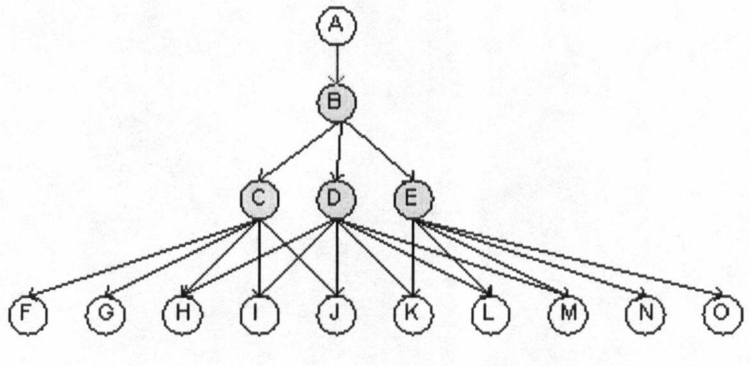


Figure 4.6: Redundancy in Greedy Algorithm: $B-E$ are candidate support elements. The Greedy Algorithm will add all of them as support elements, but D is not necessary.

The Greedy Algorithm that it cannot always produce the optimum result (the least support elements). Moreover, sometimes elements are included as support elements, but the model could still be completed even if those elements are not support elements. For example, Figure 4.6 shows an example of adding support elements using the Greedy Algorithm.

The model shows that elements F to O are result elements and elements B,C,D,E are not in the model and are candidates for support elements.

Following the Greedy Algorithm, the steps are:

- ENSRs are elements F to O, and the out-degree D for the direct parents are: $D(C) = 5$, $D(D) = 6$, and $D(E) = 5$
- Element D is firstly selected as the support element, and it covers elements H to M.
- The updated ENSRs are elements F, G, D, N, O, and the out-degrees are: $D(C) = 2$, $D(B) = 1$ and $D(E) = 2$
- Then, the element with highest degree, C, is selected as the support element, then E is selected.
- Finally, B is selected as the support element and we are done.
- The support elements are B, C, D, E.

However, notice that element D is redundant. If D is not a support element, all the ENSRs can still be covered by C and E.

In our Schema Reintegration system, all three algorithms have been implemented and tested. In practice, we have used the Shortest Path Algorithm the most, because most models that represent database schemas are trees. In such cases, the Shortest Path Algorithm can correctly, efficiently, and optimally solve the problem. However, if the model structure is a complex structured graph, and it is necessary to consider the cost of the total number of added support elements, the other two algorithms can help. When the number of added support elements is really critical, for example, adding a support element will cost several gigabytes of disk space, then the "Least Support Elements" gives the optimal solution. When the model structure is very broad (high out-degree on most elements) and has complex relationships between elements, the "Least Support Elements" is very costly. It needs to find all the paths to some elements already included in the model. In such cases, the "Greedy Algorithm" gives a good solution to quickly find the support elements.

4.4 Discussion

4.4.1 The Root Element

In Model Management, each Model is required to have a unique root element. In a mapping, it is also assumed that the root elements of the domain and range match each other. When the models are manipulated by operators the root element may not be deleted from the model. For example, in the *Diff* operator, if the *root* elements of the domain and range are matched in the mapping, the *root* of the result model will be a support element. However, in the support elements adding algorithms, one of the conditions to stop tracking each path is when the path reaches the root. Therefore, the root element must be processed separately. In any operator, if support elements are to be added, it is necessary to guarantee that the root element is included in the model first. If not, it should be marked as a support element at the beginning of any operations.

4.4.2 Support Elements in Input

Most operators need support elements in their result model. These results can be further used as input to other operators. Therefore, each operator needs to consider how to deal with the support elements in their input. Moreover, it should also be considered when such models (which already have support elements) need to add more support elements. These issues will be addressed in Section 5.2.2.

In this chapter, we discussed in detail the support elements used in model management. Support elements are helpful to maintain the integrity of models during model management operations without adding extra information or losing information. We gave an algorithm to locate the support elements and we gave three support elements adding algorithms that could be used in different situations.

Chapter 5

Reintegration System

Having developed all the operators needed for the reintegration algorithm, in this chapter, we discuss the details of the reintegration system (Section 5.1) and some complicated problems that have to be solved (Section 5.2).

5.1 Reintegration (Three-Way-Merge) Algorithm

As shown in the example in Table 2.2 in Chapter 2, the reintegration algorithm can be done in 26 steps using the operators described in Chapter 3. The input of the algorithm is one original model O and two different modified versions O, Model A and Model B. In model A and B, each element e has a *History* property indicating which element $e_o \in O$ e is derived from. Again, since each Model Management operator provides the history property, this information would be easy to derive even if there were more intervening operators.

The output of the algorithm is a single model M_{result}, which is generated conforming to the rules mentioned in Table 2.1.

At the highest level, the reintegration steps include the following:

1. Merge the two modified versions, Model A and Model B, into Model M_G. This includes all the existing elements in both versions (steps 10-11). When two elements from two models are merged, sometimes, there are conflicts. For example, one element may have the property "*minOccur = 3*" and in the other model, the matched corresponding element may have the same property with a different value, such as "*minOccur = 5*". In the merge theory, there is usually a preferred model to break the conflict. If the preferred model does not include the latest modification, it may override the latest version. Therefore, the merged version need check whether the modified part in Model A and the modified part in Model B are included. This is done by using merge again between Model M_G and the changed parts of A and B. Steps 1 to 9 find the elements that are changed in A or B but not both. Steps 12 to 16 merge the changes with M_G. However, M_G may also include those elements that should be deleted.

2. Find the deleted elements in Model A and Model B separately, namely $M_{DeletedA}$ and $M_{DeletedB}$, which may need to be deleted from the merged version M_G. Steps 17 to 18 find these deleted elements. However, these deleted elements may have been modified in the other version. Therefore, not all the elements in $M_{DeletedA}$ and $M_{DeletedB}$ can be deleted from M_G.

3. Find the those elements that are deleted in one model but not modified in the other. These elements are those that can be really removed from the wholly merged version. Steps 19 to 26 find those elements

and delete them from the merged model M_G.

5.2 Discussion of Problems in Reintegration System

We have implemented the reintegration system, including all the operators discussed in Chapter 3 and the reintegration algorithm [14] shown as an example in Chapter 2 and summarized in Section 5.1. The system was tested with several test samples, from very simple ones to very complicated ones. We discovered some interesting problems that have not been previously studied.

5.2.1 Null Model

Our experiments started with the simple example shown in Chapter 2. Though this example is simple, it can raise some unexpected results. For example, many operations generate a resulting model that is *NULL*, i.e., the result model includes no element or relationships. Previous work had not identified this as a possibility, but is clearly a special case that must be handled properly. We treat the NULL model as a special type of model.

The *Null* model is necessary. Many operators take a pair of models as input, such as *Diff*, *match*, *range*, and *match*. Also, the *map* can be an input. Without the *Null* model, these operators would not be able to handle the cases where input models have no elements. But intuitively, the existence of the null model makes sense to real problems. For example, in the example in Table 2.2, Step 7, models $changed_A$ and $changed_B$ have no common elements. Their mapping will result in a null model. We need to define the behavior of each operators when one or more input models are null.

- In *Diff*, if the *domain* model is a null model, then the result model is null too. If the *range* model or the *map* is a null model, then the result model is exactly the same as the *domain* model.
- In *merge* if any of *domain* or *range* is a null model, the result will be the same as the other model (*range* or *domain*). In *merge* it is assumed that the two *root* elements of the *domain* and *range* are mapped to each other if neither of them is null. Therefore, the *map* cannot be null.
- In *range*, if the *domain* model is null, the *map* will be null too. Therefore, the result model will be same as the *range* model. If the *range* itself is null, the result will be null too.
- In *match*, if any of the *domain* or *range* is null, the result mapping will be null too.
- In *Apply*, if the input model is null, obviously a null model will be returned.

Furthermore, if all the elements in a result model are support elements, this model will be a null model, instead of a model that includes all support elements.

5.2.2 Support Elements in Input

As we have mentioned in Section 4.4.2, support elements may appear in the input models of each operator. While the Reintegration system is composed of the operators and the result of one operator will be used in future operators as input, the support elements will be more likely to appear in input models. They need to be handled with clear rules. Here is how they are handled in the system.

The first step is that all the support elements in input models are marked as "delete" candidates. More specifically, if an element in the input models has a property of "*p(support) = true*", this property is deleted and we add a new property as "*p(delete) = true*". This means that these elements may be deleted after the operator finishes. Because the "support elements" may not useful in the result model, i.e., it is not necessary to "support" any elements. For example, Figure 5.1 (a) shows a mapping between model A and model B, which includes a support element c to support the element d to be included in the model. The merge result is shown in (b). Obviously, element d is now included in the model because of the existence of element b, and support element c is not necessary any more.

The second step is that the model with elements marked as "delete" is processed by each operator normally as input. The elements marked as

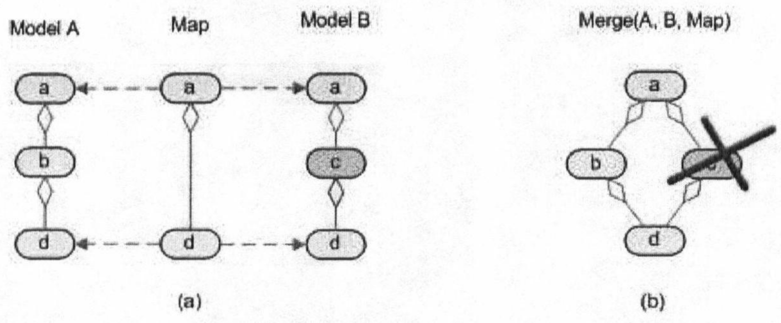


Figure 5.1: Support element *c* in Input model, which is not needed in Merge result.

"delete" are treated as normal elements in the algorithms. However, each operator needs to handle the *delete* property in the input element differently.

- In *Merge*, each element in the result model represents a group of corresponding elements. If any element in the group does not have the *"delete" property*, the corresponding result element will not have the *"delete" property.*

- In *Diff*, if the domain model contains elements that have the *"delete" property* , the corresponding elements in the result model also have the *"delete" property.*

- In *Range*, if the range model contains elements that have the *"delete" property*, the corresponding elements in the result model also contain the *"delete" property.*

- In *Apply*, the *"delete" property* of the elements in input model will be kept in the corresponding elements in the result model, unless the *Apply* function modifies this property on purpose.

- In *Match*, each model mapping element in the result map represents a group of elements. Only when all the elements in the group have the *"delete" property*, the corresponding element in the result map also has the *"delete" property.* Otherwise, the elements in the result map do not have the *"delete" property.*

In the third step, the result model will be processed by the "Support Elements Engine" as usual and if the elements that are marked as "*delete*" are still essential to support some other elements, they will be marked as "*support*" by the engine. If the elements are still marked as "*delete*", they will be deleted from the result.

The specific handling of "Support Elements" enables the Reintegration system to work for any cases with full "Support Elements" functionality support. The "Support Elements" can appear in any place. The Support Element Engine has been fully discussed in Chapter 4.

5.2.3 Model Validation

All the operators in the Reintegration system assume that the input models are valid. All the algorithms in each operator also rely on valid models. However, this assumption may be challenged when the input models are complex.

During system testing, we tried some complex examples that include hundreds of elements. Some human errors may make the input models invalid. For example, some elements are supposed to be in a model, but cannot be inclusion-implied, or the relationships have cycles. If the input models have such kinds of invalid problems, the performance and result of each operation can hardly be guaranteed. This may lead to system instability. Therefore, all the input models need to be validated before being used. The model validation procedure includes validating the elements and relationships. The rules used are the "Inclusion Rules" shown in Section 2.1.3. Starting from the root, if all the elements and relationship can be included by using the inclusion rules, then the model is valid. If the model is mapping, we also check if the two roots of the *domain* and *range* are mapped. The correctness of logic of the input models is not validated in this system. It depends on the user to make sure the logic in the model is correct. Invalid input models are rejected from proceeding further during Reintegration.

In this way, the system becomes more stable and could be used on complex models, which are more general, and hence the system is more valuable.

5.3 Summary

The Reintegration system integrates all the operators that are required for Reintegration. Because all the operators are designed in a way that they can consume the output models from other operators, they can be easily incorporated into the system to accomplish the integration task. The "Support Element Engine" is designed to support all the operators, so that each operator can focus on its functional algorithm.

This kind of design makes the system more flexible and extensible. It is suitable for future model management operators development and other relevant model management topics. The system also includes a graphic presentation that enables the models and algorithm procedures to be visualized. This makes the system easier to be validated and observed for discovering interesting issues.

We have made the system stable in a sense that it always checks the validation of input models, and during each algorithm. From our experiments, we have also discovered the corner cases in the system that need to be considered to make the system more general and compatible. These corner cases are not usually discussed but they are still technically essential to make everything stable in the system.

All these efforts make the system ready to accomplish the schema reintegration. Now that the system is ready, we describe the experiments on the system in Chapter 6.

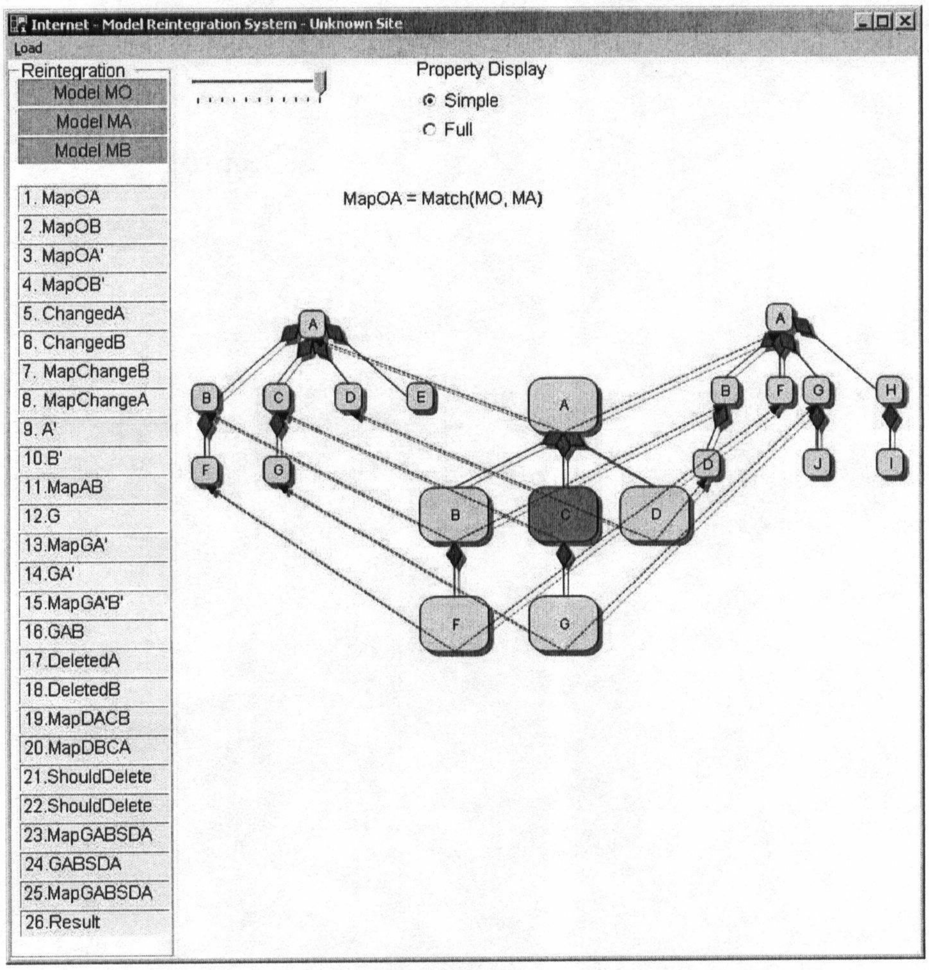
Chapter 6

Experiment

The implementation of the Model Reintegration System is based on the previous work of Merge in [14]. The system is written in C# and includes the classes for the basic data structures, such as Element, Model, Relationship, Mapping, and operators. We built each operator as a class, so that the class can be decoupled from other operators, and is easy to extend. We also built a class that integrates all the operators and follows the Three-Way-Merge algorithm. In order to make the system more transparent and clear, the result models in the steps are all visualized using Visual Studio .NET 2003 and the FlowChart .NET component [8]. All the elements are displayed using an automatic tree-structure arrangement plan. This is a good way for observing the elements and relationships for demonstration purpose. Figure 6.1 shows a mapping in the Model Reintegration System interface that we used for demoing the reintegration procedure.

The system includes the following main parts:

- Model Display
- Model Selection
- Support tools.



Property Display
[✓ Simple]
Full

MapOA = Match(MO, MA)

Figure 6.1: Model Reintegration System

After a demo is loaded, the user can select any model in the steps by clicking the related item in the left bar, and the selected model will be displayed in the main area. There are some functionalities to make the system more convenient for the users. For example, there are two modes in which to view the model: the simple mode only shows the name of each element, and the full mode shows all the details of each element, including all the properties. There is also a zoom bar to view the model in different sizes.

6.1 Operator Experiments

While we developed each operator, we performed numerous experiments to ensure the correct behavior of each operator. These experiments are designed to cover different situations that may happen in the input. For example, the following are part of the experiment cases that were used for testing *Diff*:

- Element in *domain* and has a valid match to some elements in the *range*
- Element in *domain* but has no mapping relationship
- Element in *domain* and has a mapping relationship with some element in the map, which has no mapping relationships with elements in *range*
- Multiple elements in *domain* match to the same element in *range*
- One element in *domain* matches multiple elements in *range*

#	Operator	# of experiment cases
1	*Merge*	25
2	*Diff*	6
3	*Range*	6
4	*Apply*	2
5	*Match*	4

Table 6.1: Experiments Performed for Each Operator

- All the elements in *domain* have valid matched elements in *range*

We have designed different test cases for different operators. We have covered different rules in each operator, including some special scenarios that have been mentioned. In each operator and some basic modules, there are also some functionalities that accomplish a single task. For example, we have a function to verify if a model is valid. Table 6.1 shows the experiments that we have performed for the operators, not including the test on basic functions, support elements and the whole system.

From these experiments, we learned a lot of corner cases that we must cover for a complete system, such as how to deal with the root element in each operator. These observations are helpful when using these operators in the reintegration system.

6.2 Support Element Engine Experiments

The "Support Element Engine" described in Chapter 4 is an assistant module that all the other operators rely on. Because its input is only a model and its output is the same model with support elements found and marked, the experiments on this module were performed in both single module scenarios and integration scenarios with other operators. We have designed 18 test cases to covered different scenarios. We also tested the engine with mappings models. The engine is expected to be able to support any kind of model.

Moreover, because we have three different support elements adding algorithms (See Sections 4.3.3.1 4.3.3.2 and 4.3.3.3), all of the experiments were performed using all the algorithms separately. We have also designed cases to show that different algorithms would generate different results on the selection of support elements.

We also tested the performance of the engine. We have designed test cases that include elements arranged in a wide structure, for example, one level contains more than 10 elements, and in a deep structure, for example, the model contains more than 5 levels. These models have very complex relationships between elements in each level and in different levels. The result shows that all three algorithms can finish in less than one second, even for large models, which is acceptable speed. Among the three algorithms, the shortest path algorithm is a little bit faster than the other two. This is because this algorithm searches support elements in a direct way, which is less than the height of the graph from the root. The support elements adding step for each element is independent from each element. However, in the other two algorithms, the correlation of each element that needs support has to be considered. This increased the complexity of the algorithms. Therefore the Shortest Path Algorithm is the one that we recommend if the number of support elements added is not very small.

6.3 Reintegration System Experiments

We designed four experiments to test the reintegration system. Because the system is composed of only the five operators mentioned in this thesis, these tests are also a way to further test each operator. In the experiments, we primarily used the shortest path algorithm in the Support Element Engine. However, in the last two more complex experiments, we have randomly chosen one of the three support element adding algorithms in each step of the reintegration system. This ensures that all three algorithms work properly.

In order to quickly create models for testing, we also designed functions to automatically add elements to models and find and add history properties to match to corresponding elements in another model (usually the original model). This gives us a quick way of adding elements and building the structure of models and the relationships between elements in different models, so that we can put more of our focus on the reintegration system itself.

Our first experiment is a simple one mentioned in [14] (Merging Models Based on Given Correspondences). The original model includes four elements. The derived models contain two scenarios: one element is modified in one model and not changed in the other; one element is deleted in one model and not changed in the other. Even from this simple experiment, we found many improvements that the system needed in order to be more stable. For example, the simpler the models are, the more likely there are null models appearing in the result of operators, which may be used by other operators as input. From Table 2.2, we can see that there are 11 results among the total 26 steps are that null models. This brought our attention to handling null models for each operator and it is necessary to define the correct behavior when the input has null models, as discussed in Section 5.2.1. We have attached screen shots of all the steps of this experiment in Appendix A.

The rules mentioned in Table 2.1 are the principle cases that our system should cover. Therefore, our second experiment was designed to include elements that can cover all the scenarios in the rules. The original model includes eight elements which are arranged in three levels. One derived model contains eight elements and the other contains seven elements. These elements are arranged in a designed hierarchy to cover each scenario that we are going to test.

Furthermore, we also performed experiments on the cases that Rondo [11] uses. This includes a relational schema case and an XML schema case. Both schemas are translated into models that can be used in our system and run the Reintegration algorithm on them.

In the first of these two experiments, we created a four-level parent-child hierarchy data model. The first level is the root for the whole database; the second level includes one element for each schema in the database; the third level includes one element for each attribute; and the last level includes the type of each attribute and if it is a primary key, there is also a child element of *primary key*. The translated models contain the following information:

- Model O: 37 elements, in 4 levels
- Model A: 49 elements, in 4 levels
- Model B: 45 elements, in 4 levels
- Mapping relationships between O and A: 33
- Mapping relationships between O and B: 29
- Mapping relationships between A and B: 31
- Final result model contains 57 elements

In the second example, we translate each XML node of the example as an element in the each model. The translated models contain the following information:

- Model O: 27 elements in 5 levels
- Model A: 32 elements in 5 levels
- Model B: 26 elements in 5 levels
- Mapping relationships between O and A: 19
- Mapping relationships between O and B: 20

- Mapping relationships between A and B: 16
- Final result model contains 37 elements

The overall test results showed that our system achieves the correct output. The first two experiments resulted in the results dictated by three-way merge. The last two cases got the same results as Rondo. These were the results that were dictated by the rules of three-way merge, and is the behavior that make sense to most people.

From these two more complex examples, we observed that it is very important to always keep the input model and output model valid. Sometimes, a dangling relationship or element is a human mistake while the model is created. Therefore, we added a validation step to make sure that all the input models are valid, that is, they are compatible with the inclusion rules. This extra step makes the system more reliable.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we discussed the details of the Model Management operators: *Diff* (Section 3.2), *Range*(Section 3.3), *Apply*(Section 3.4) and *Match*(Section 3.5) . We formally defined each operator and gave algorithms after analyzing them in detail. We showed that these operators always need support elements to maintain integrity.

We developed a model reintegration framework that integrates all the operators and accomplishes the Reintegration task, which is the essential algorithm for schema reintegration purposes. The model reintegration system demonstrated how the original schema can help when merging different versions of modified schemas correctly.

The schema reintegration system can be used in many areas, such as for relational schemas, XML schemas, UML, and ontologies. When the circumstance happens that there are one original schema and two or more different modified versions of the schema, the schema reintegration can facilitate the automation of the reintegration steps.

We discovered the need for, and also discussed the "Support Element Engine" in detail. This engine is designed and developed in a generic way, so that all other operators can simply rely on it to select the appropriate elements as the support elements. This method does not add any extra information to the result of each operator and does not lose any existing information either. It is also a way to keep tracking the history of the evolvement of each element. With this engine assisting of model management, all the other operators can now focus on their own algorithms and leave the structure problem of their result models to the engine in the final step.

Our experiments have shown that these generic model management operators indeed can be used for schema reintegration purposes. This system can be used for different kind of schemas and generate good result. The rules implemented in each operator can be customized to be compatible with different utilizations in the real world. We always keep "generic" in mind when we design our system. This system can be easily extended to other relevant applications.

7.2 Future Work

In this thesis, we focused on analyzing and implementing the operators *Diff*, *Range*, *Apply*, and *Match* (based on History property), which are for reintegration. There are still many other operators in model management to be explored, such as *compose* and *modelGen*. These operators can be useful for other model management purposes.

In our reintegration system, models are created manually. This is only for demo purposes. The transformation from a schema to our model is slow if it is done manually. It will be very helpful if there are interpreters that can transfer each kind of schemas to our models.

The *Match* operator in the system is based on the "History" property. This works fine and is reliable in the middle of the algorithm after the models are built. However, in the first step, when we create the input models, we have used the element names to automatically build the corresponding "History" properties. This is because we do not discuss other automatic matching methods. The automatic matching in model management has been discussed widely, for example, [11][15][6][7][10][9]. If a reliable *matcher* can be integrated into the system and build the initial relationships between the input models and elements, the system will be more complete and have better usability.

Bibliography

[1] Suad Alagic and Philip A. Bernstein, *A model theory for generic schema management*, DBPL, 2001, pp. 228–246.

[2] S. Balasubramaniam and Benjamin C. Pierce, *What is a file synchronizer?*, ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM), 1998, pp. 98–108.

[3] P. A. Bernstein, *Applying model management to classical meta data problems*, Conference on Innovative Data Systems Research (CIDR), 2003, pp. 209–220.

[4] Philip A. Bernstein, *Generic model management: A database infrastructure for schema manipulation.*, CoopIS, 2001, pp. 1–6.

[5] Philip A. Bernstein, Alon Y. Halevy, and Rachel A. Pottinger, *A vision of management of complex models*, SIGMOD Record **29** (2000), no. 4, 55–63.

[6] David W. Embley, Li Xu, and Yihong Ding, *Automatic direct and indirect schema mapping: experiences and lessons learned*, SIGMOD Record **33** (2004), no. 4, 14–19.

[7] Fausto Giunchiglia and Mikalai Yatskevich, *Semantic matching*, Knowledge Engineering Review **18** (2004), no. 3, 265–280.

[8] MindFusion Limited, *Flowchart.net component*, Version 3.2.2, 2006.

[9] Jayant Madhavan, Philip A. Bernstein, AnHai Doan, and Alon Halevy, *Corpus-based schema matching*, ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05) (Washington, DC, USA), IEEE Computer Society, 2005, pp. 57–68.

[10] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm, *Generic schema matching with cupid*, VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 2001, pp. 49–58.

[11] Sergey Melnik, Erhard Rahm, and P. A. Bernstein, *Rondo: A programming platform for generic model management*, ACM SIGMOD International Conference on Management of Data (SIGMOD), 2003, pp. 193–204.

[12] Jonathon P. Munson and Prasun Dewan, *A flexible object merging framework*, Conference on Computer Supported Cooperative Work (CSCW), 1994, pp. 231–242.

[13] Rachel A Pottinger, *Processing queries and merging schemas in support of data integration*, Ph.D. thesis, University of Washington, 2004.

[14] Rachel A. Pottinger and P. A. Bernstein, *Merging models based on given correspondences*, Technical Report UW-CSE-03-02-03, University of Washington, 2003.

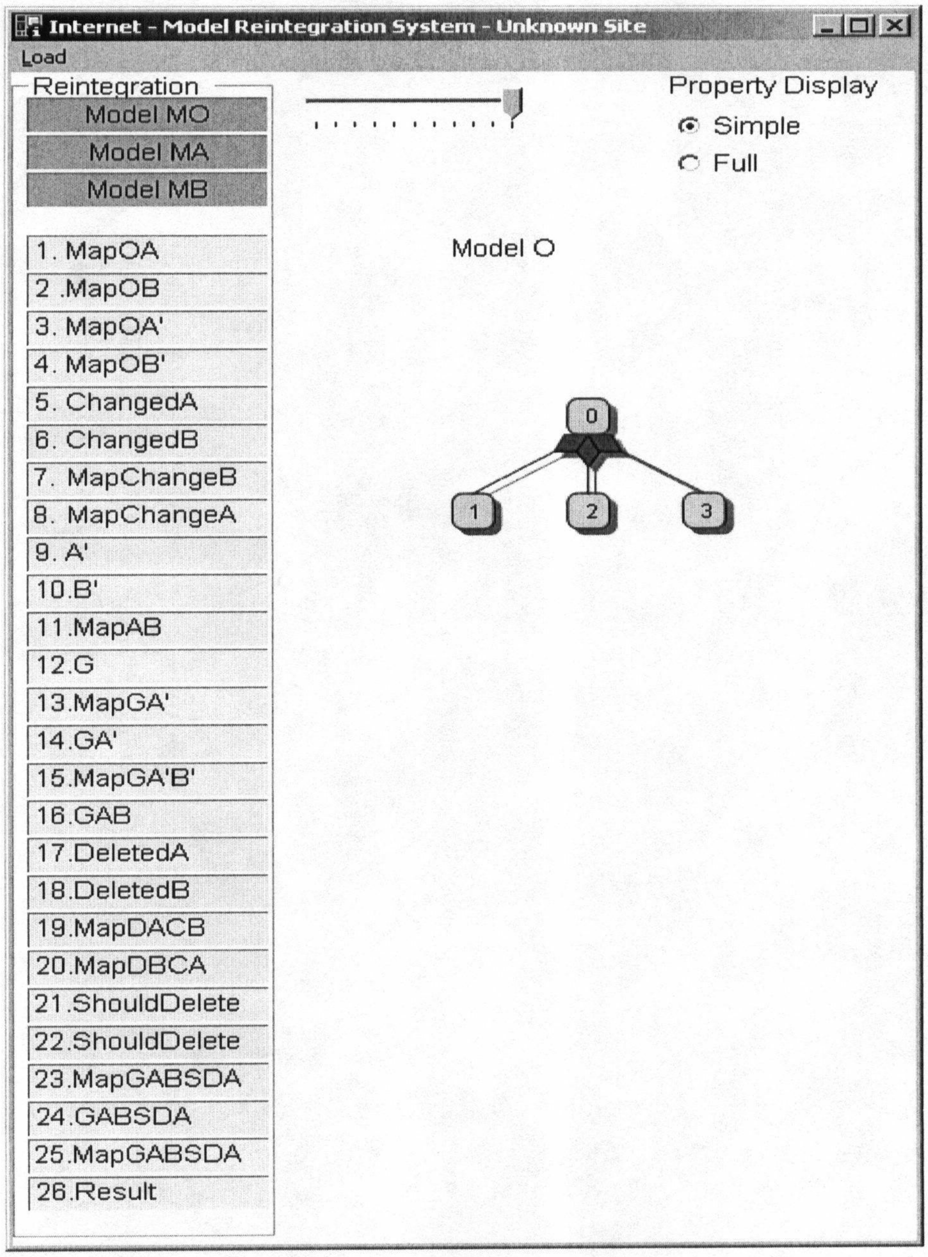
[15] Erhard Rahm and Philip A. Bernstein, *A survey of approaches to automatic schema matching.*, VLDB J. **10** (2001), no. 4, 334–350.

Appendix A

Reintegration System Example

Here we list the screen shots of the first experiment that we did on the Reintegration System.

Figure A.1 shows the original model and Figures A.2 and A.3 are the two different modified versions. Figures A.4 through A.29 list all the steps that correspond to the procedures shown in Table 2.2.



Property Display: [✓ Simple] Full

Figure A.1: The Original Model *MO*

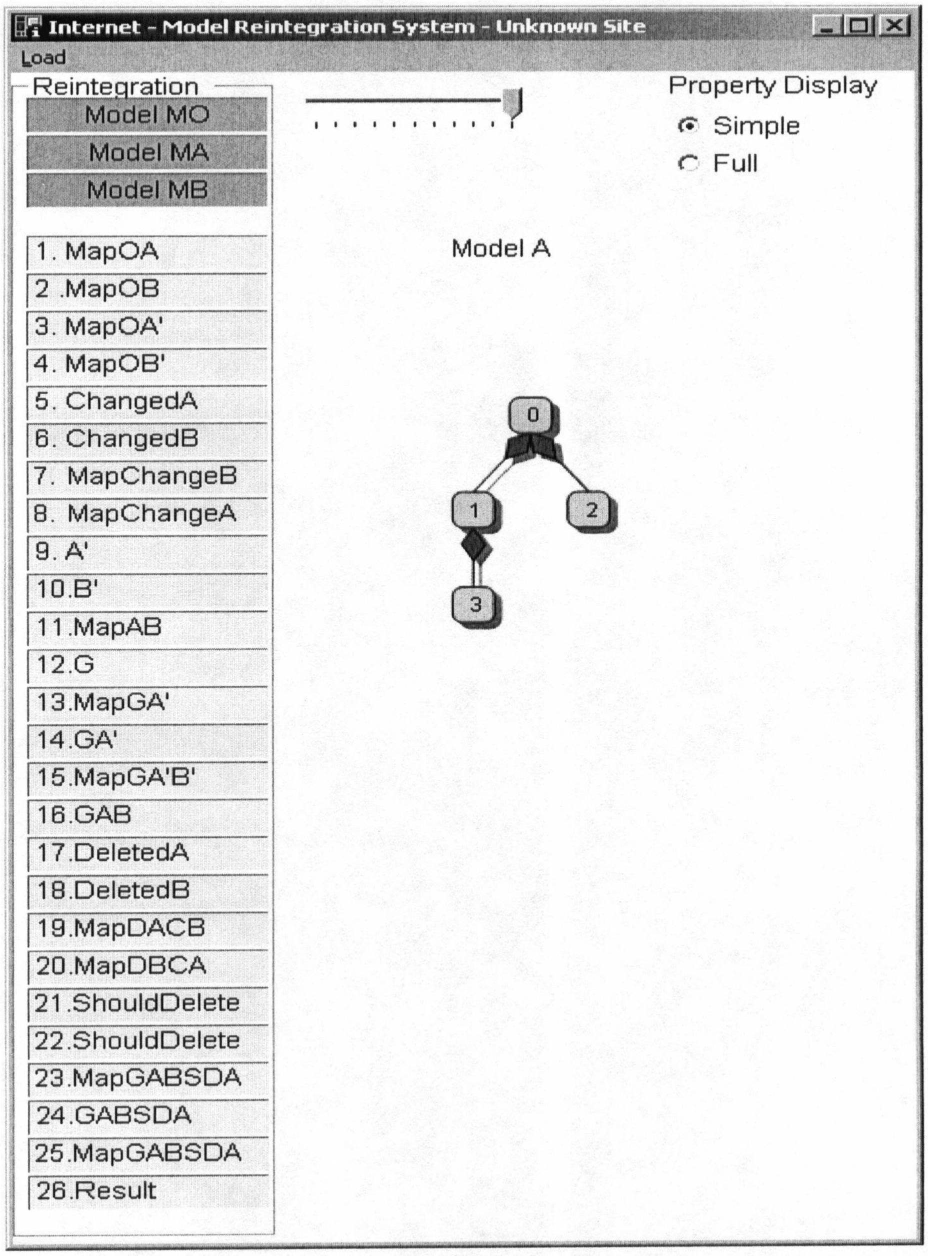
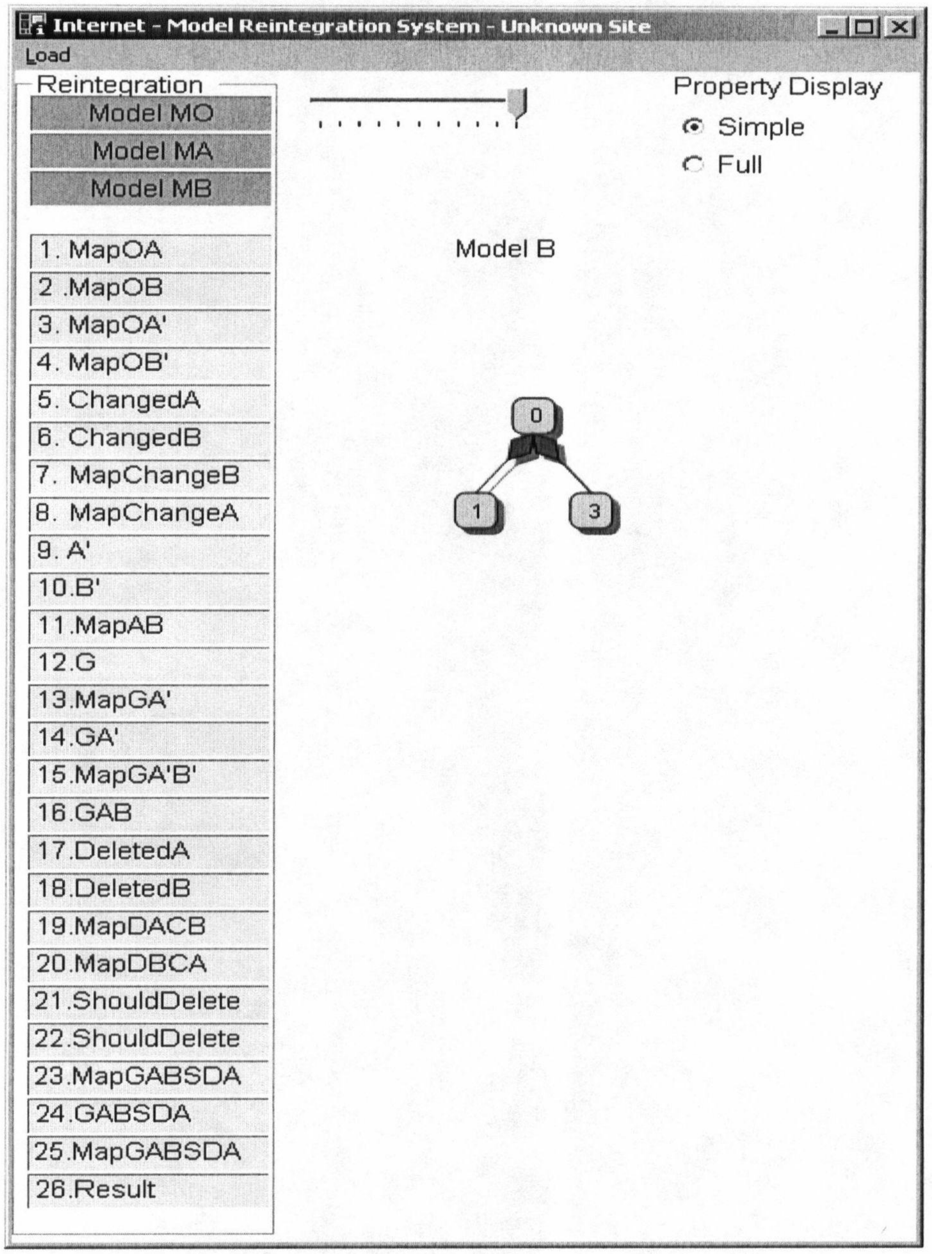
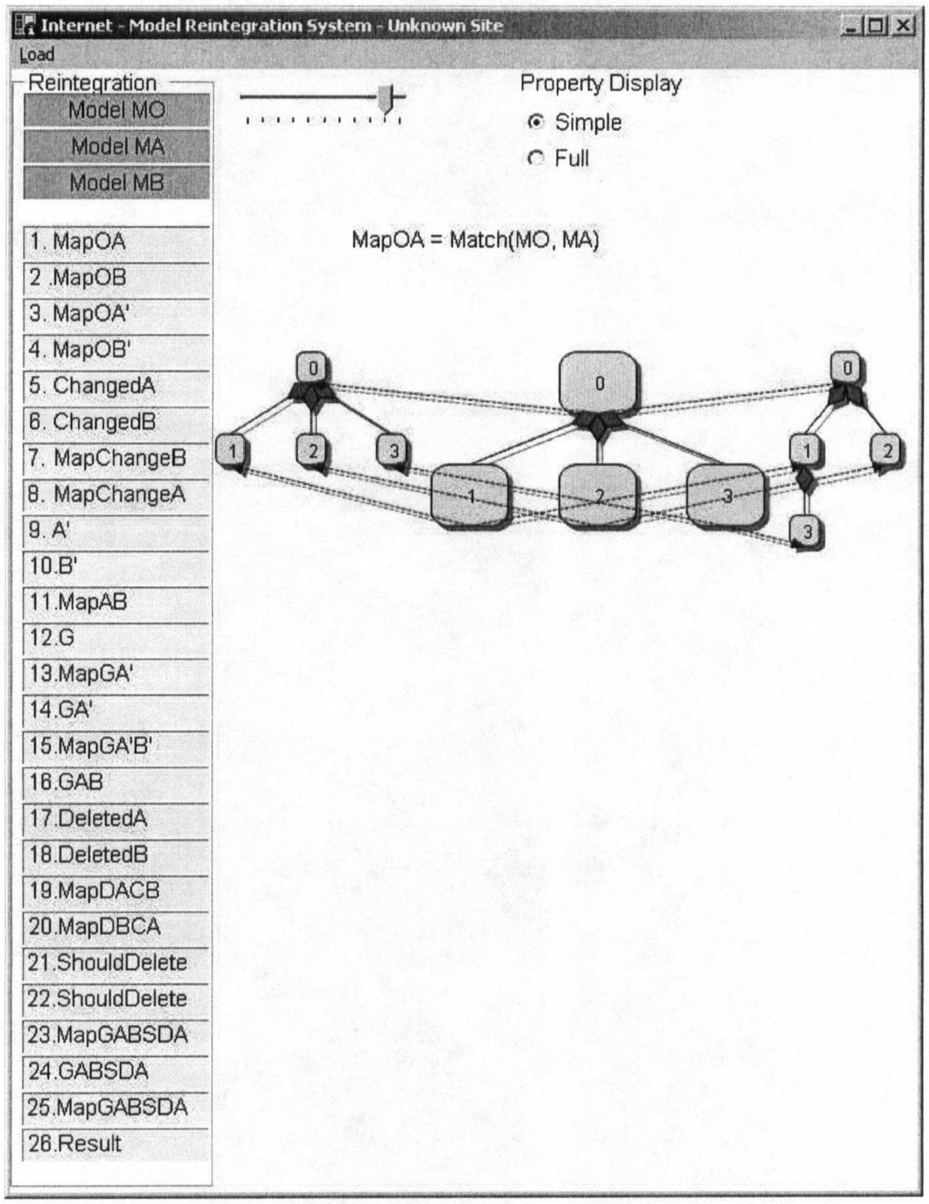


Figure A.2: The First Modified Model MA



Property Display: [✓ Simple] Full

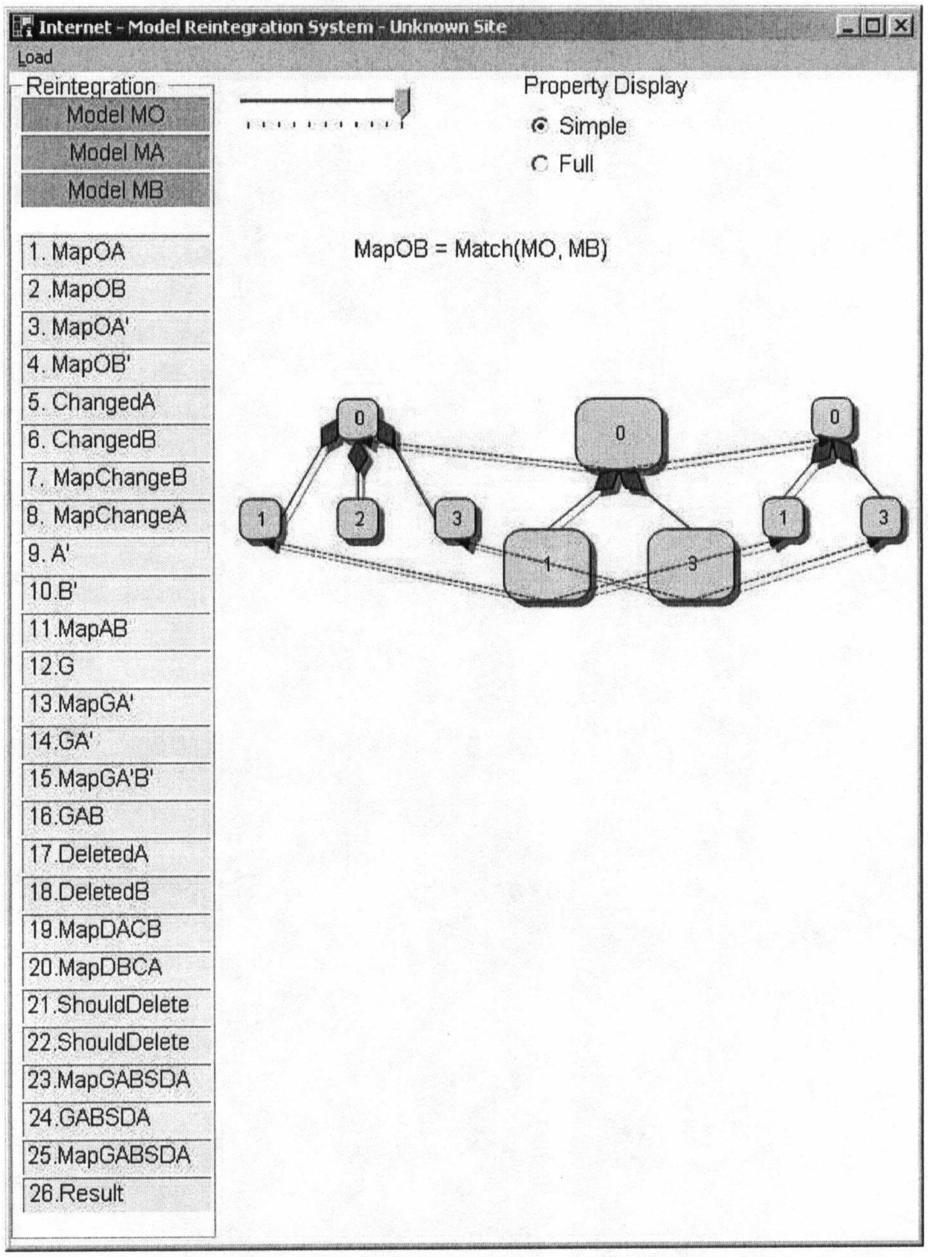
Figure A.3: The Second Modified Model MB



Property Display
[✓ Simple]
Full

MapOA = Match(MO, MA)

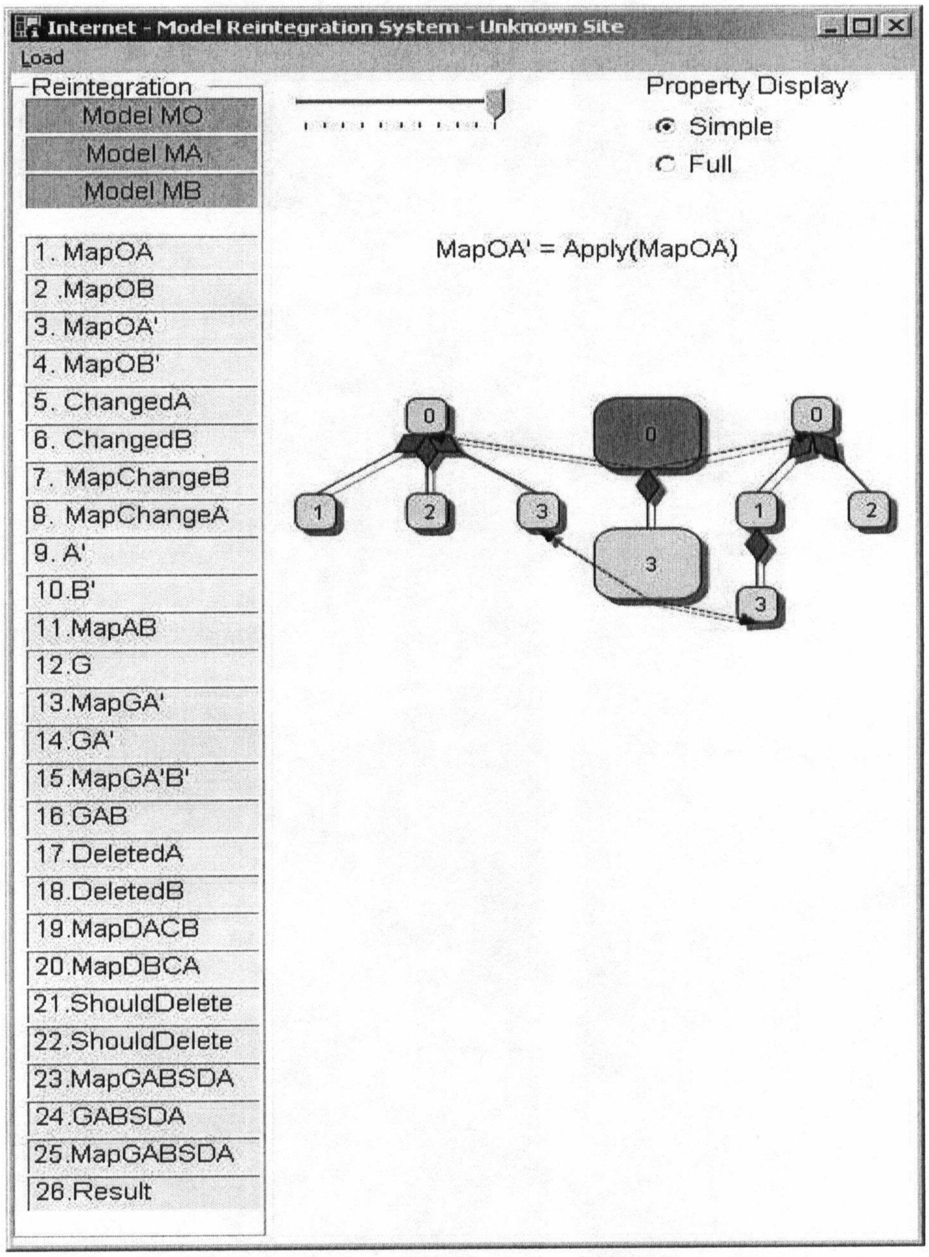
Figure A.4: The Mapping Map_{OA} between Model MO and Model MA



MapOB = Match(MO, MB)

Property Display: [✓ Simple] Full

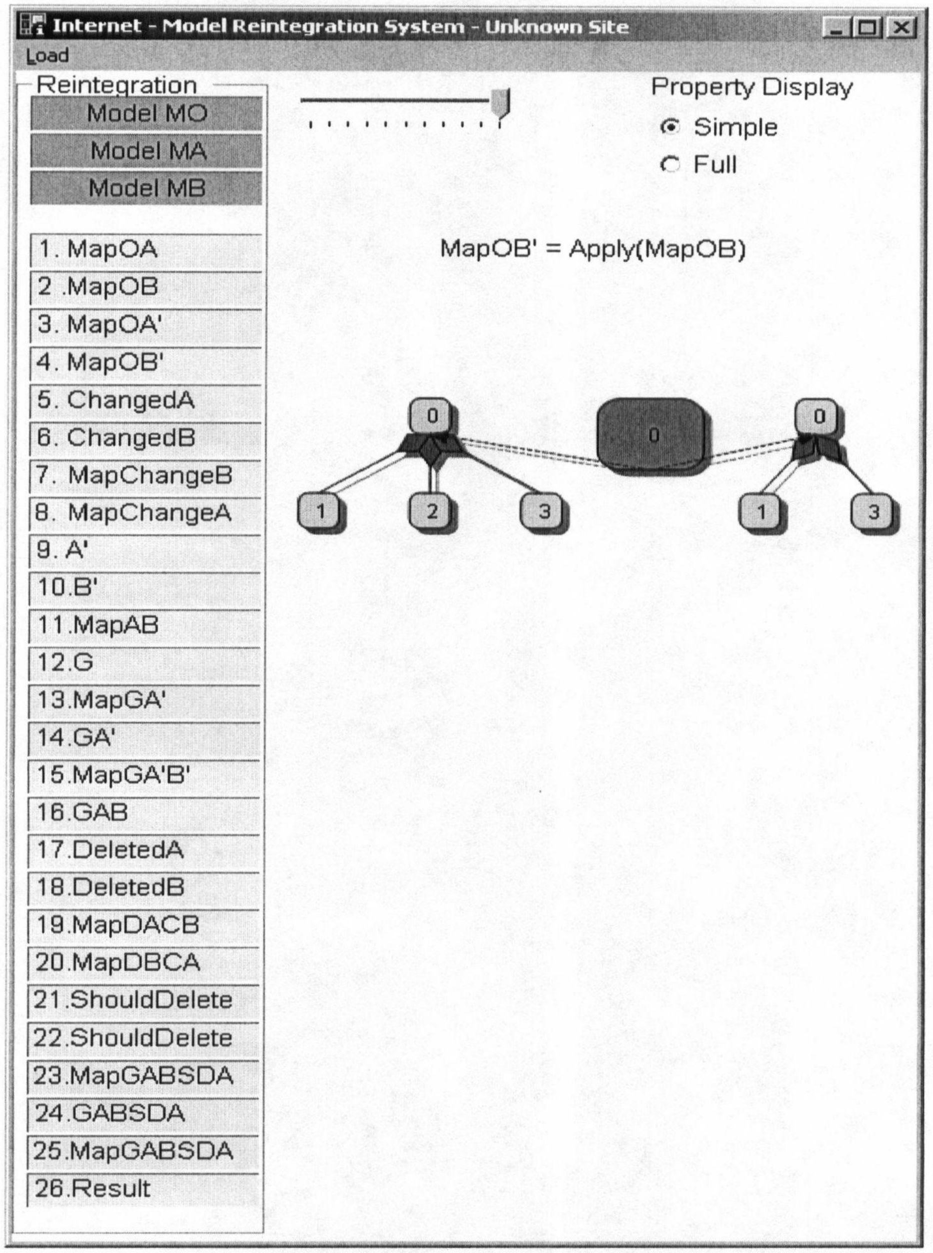
Figure A.5: The Mapping Map_{OB} between Model MO and Model MB



Property Display: [✓ Simple] Full

MapOA' = Apply(MapOA)

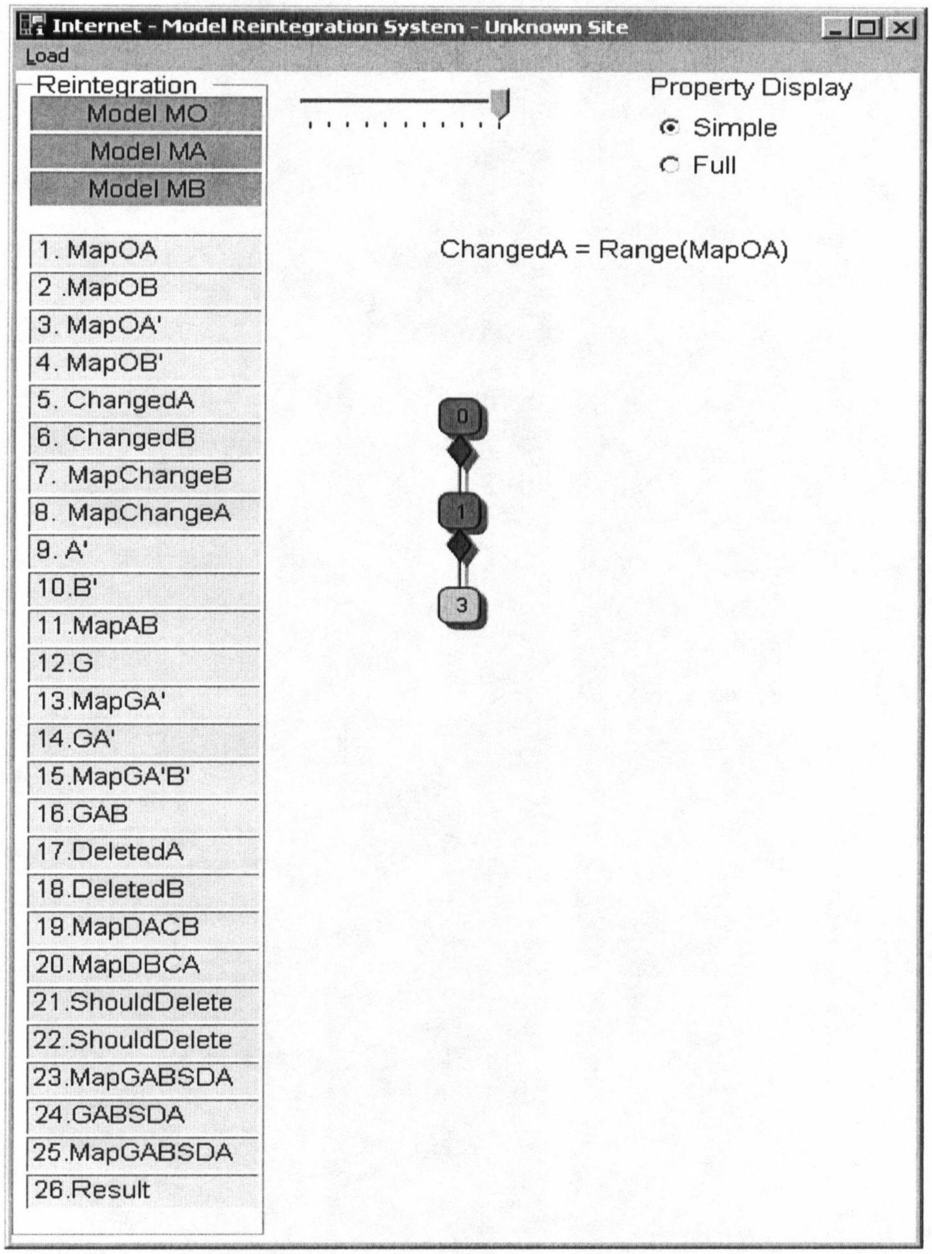
Figure A.6: The Model Map'_{OA} from $Apply(Map_{OA})$



Property Display: [✓ Simple] Full

MapOB' = Apply(MapOB)

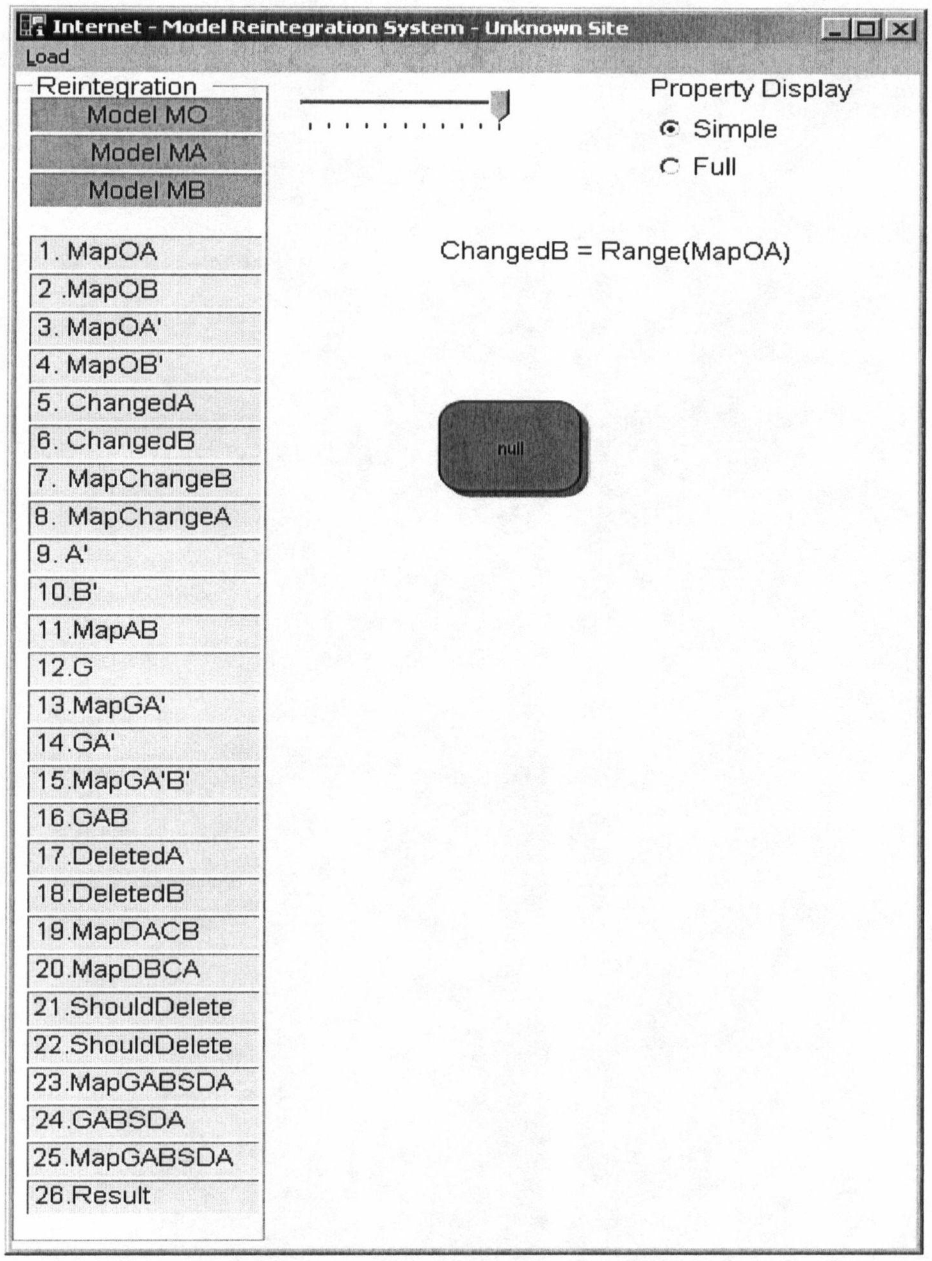
Figure A.7: The Model Map'_{OB} from $Apply(Map_{OB})$



Property Display: [✓ Simple] Full

ChangedA = Range(MapOA)

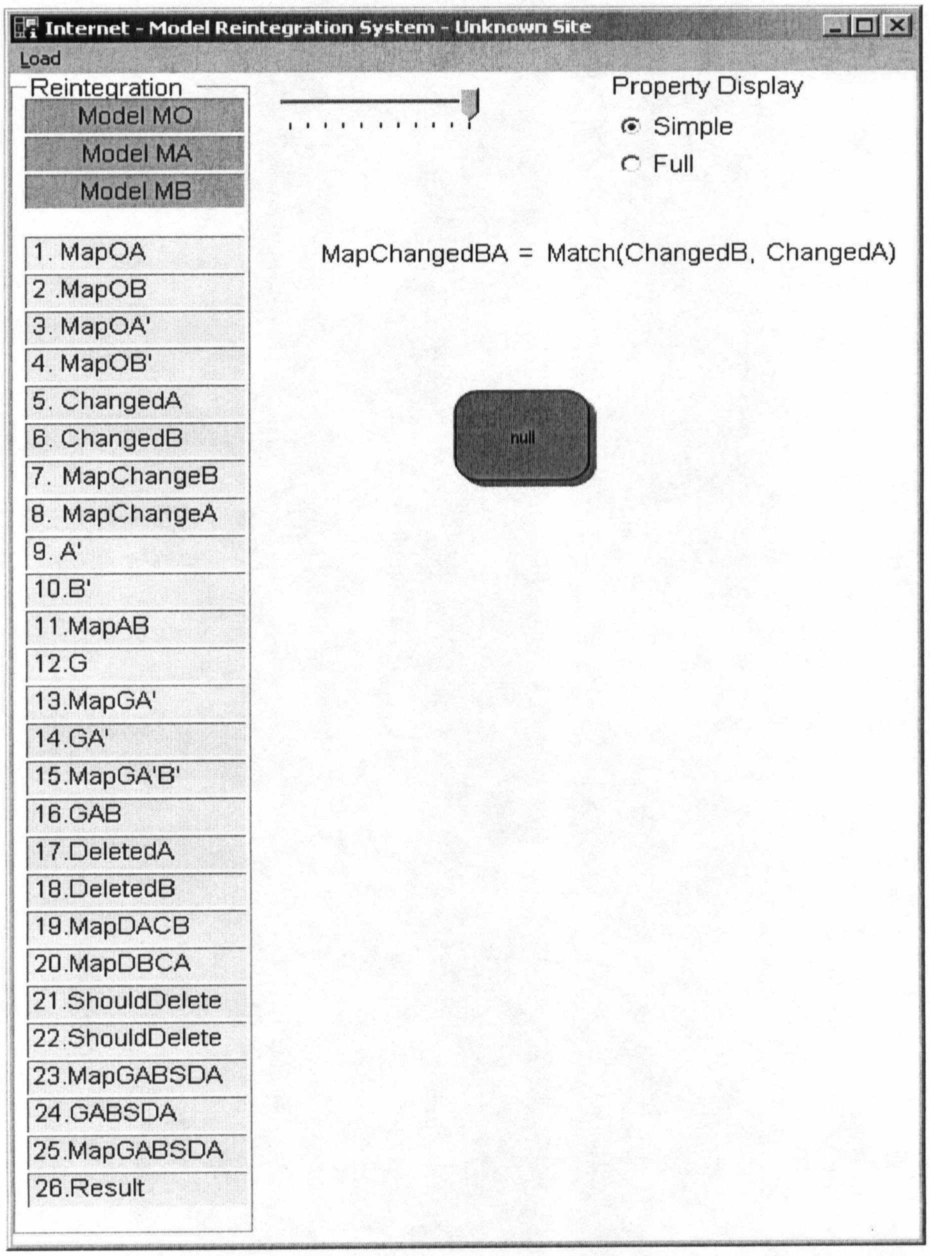
Figure A.8: The Model $Changed_A$



Property Display: [✓ Simple] Full

ChangedB = Range(MapOA)

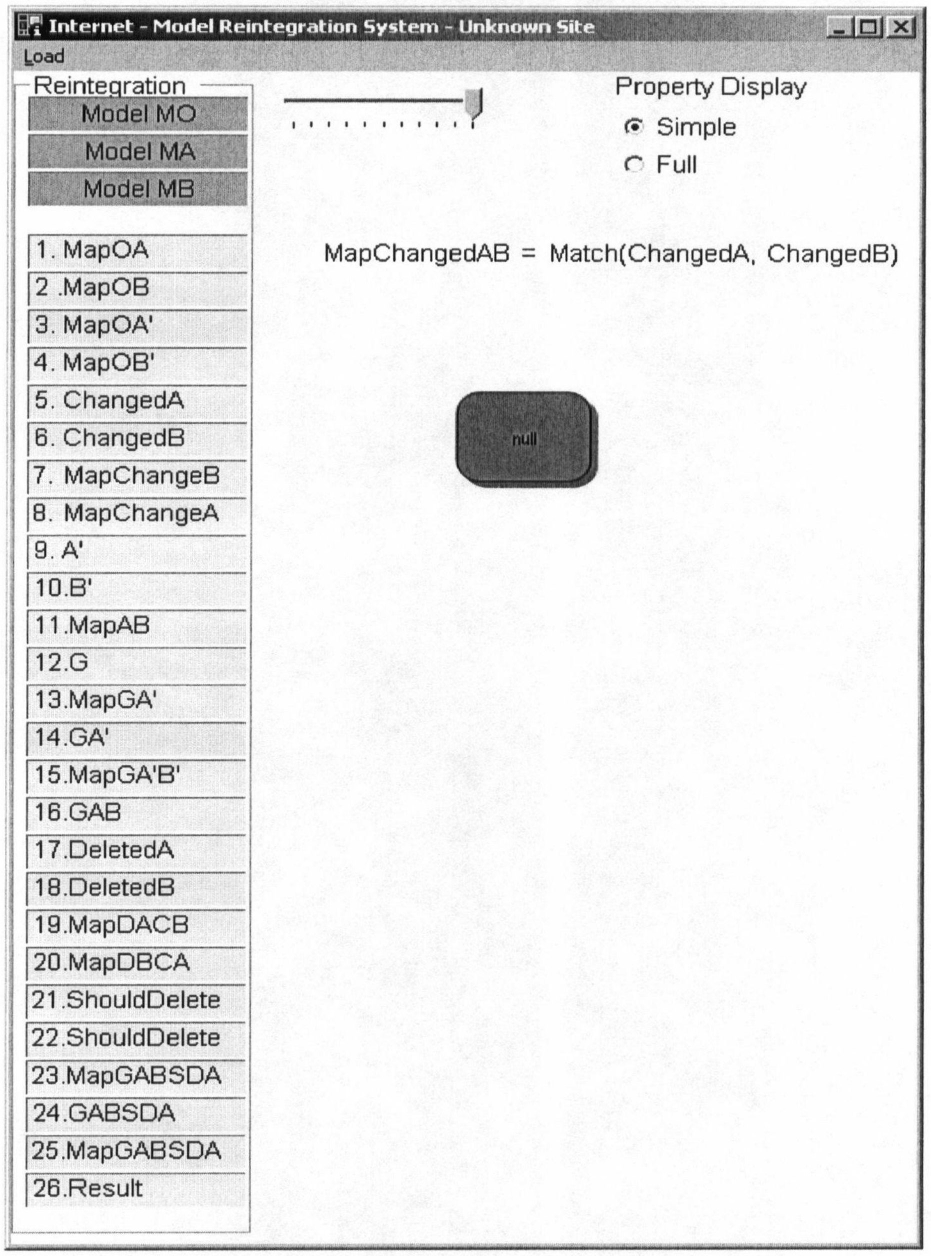
Figure A.9: The Model $Changed_B$ (A NULL model)



Property Display
[✓ Simple] Full

MapChangedBA = Match(ChangedB, ChangedA)

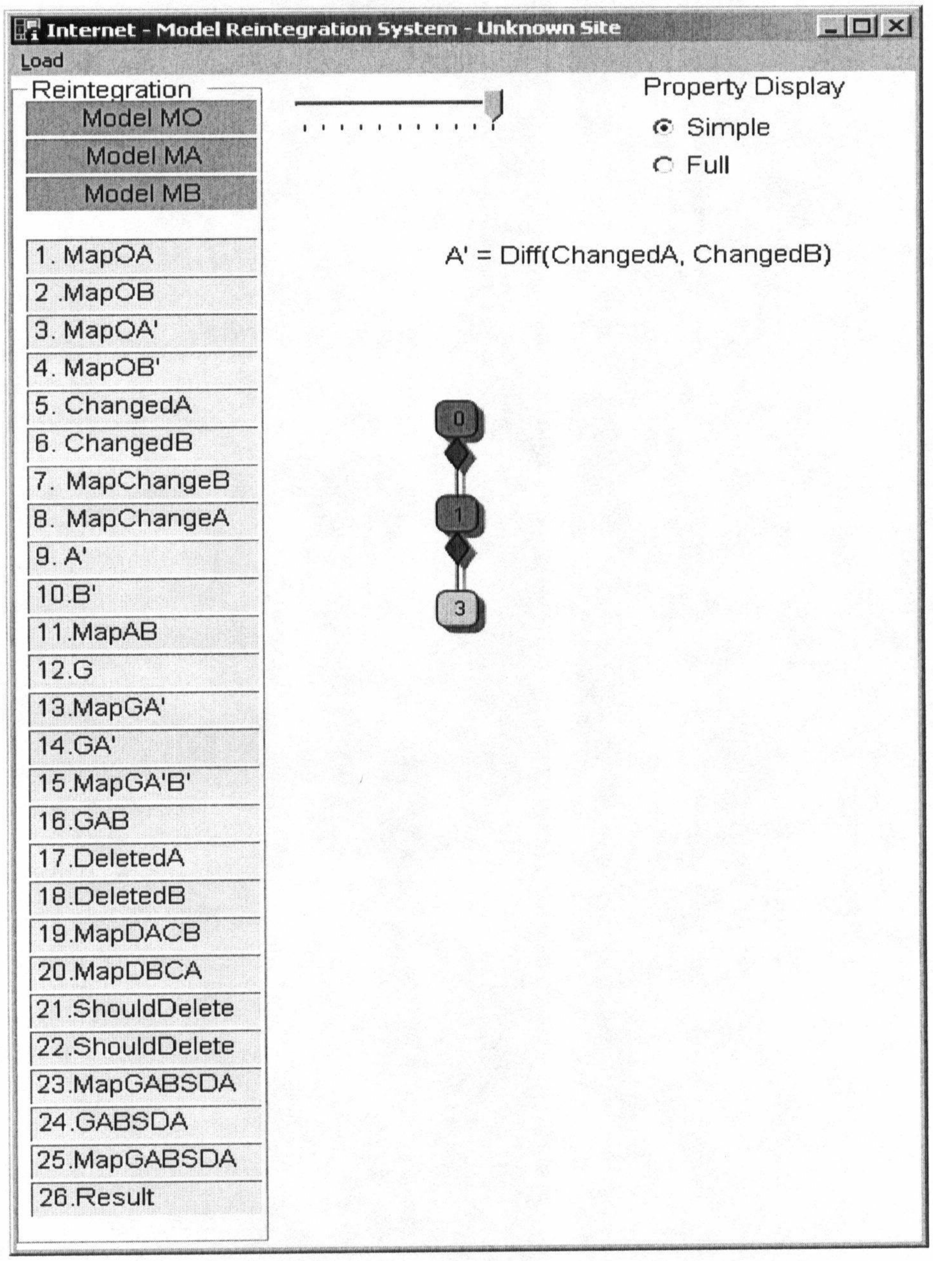
Figure A.10: The Mapping $Map_{ChangedBChangedA}$ between Models $ChangedB$ and $ChangedA$



Property Display: [✓ Simple] Full

MapChangedAB = Match(ChangedA, ChangedB)

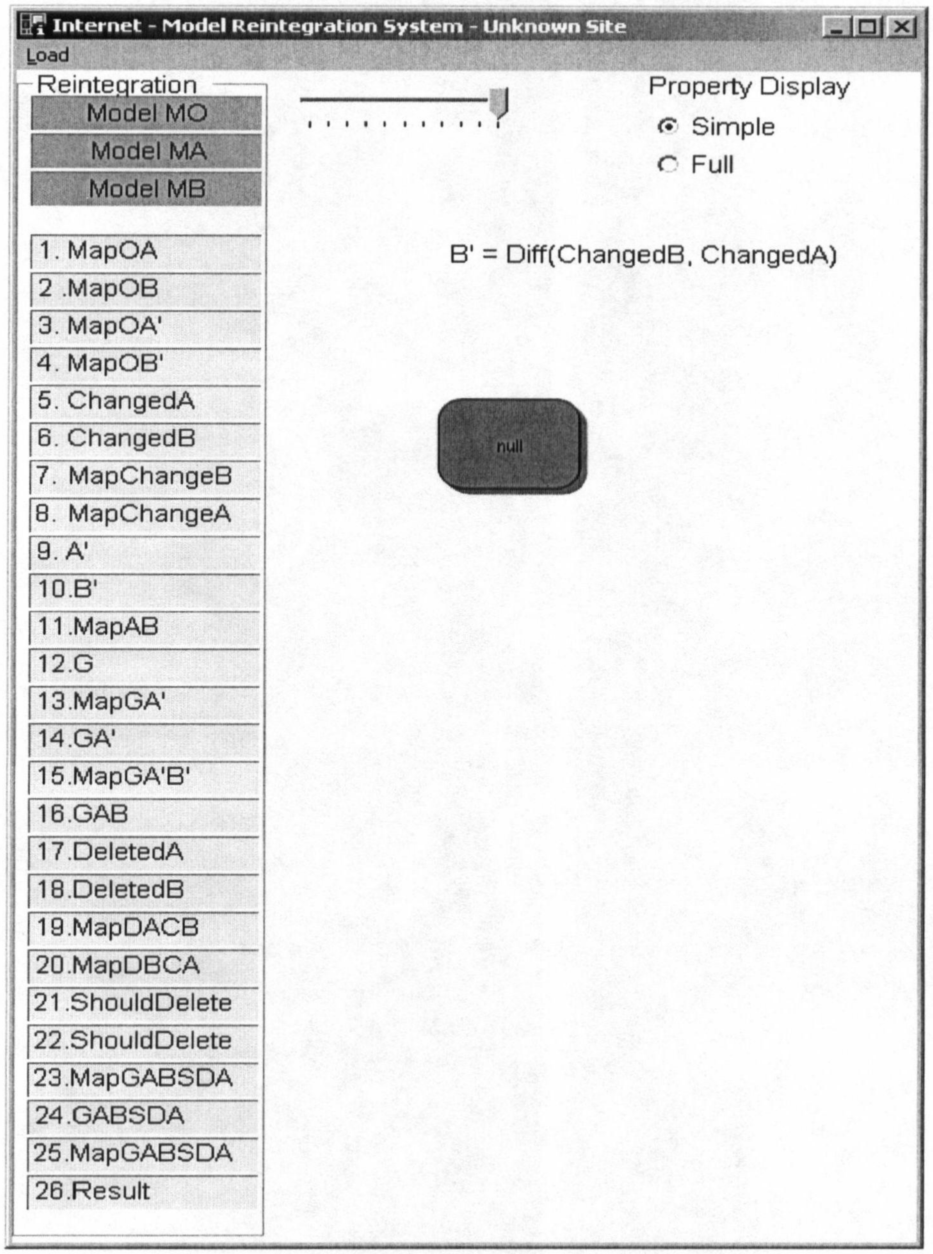
Figure A.11: The Mapping $Map_{ChangedAChangedB}$ between Models $ChangedA$ and $ChangedB$



Property Display: [✓ Simple] Full

A' = Diff(ChangedA, ChangedB)

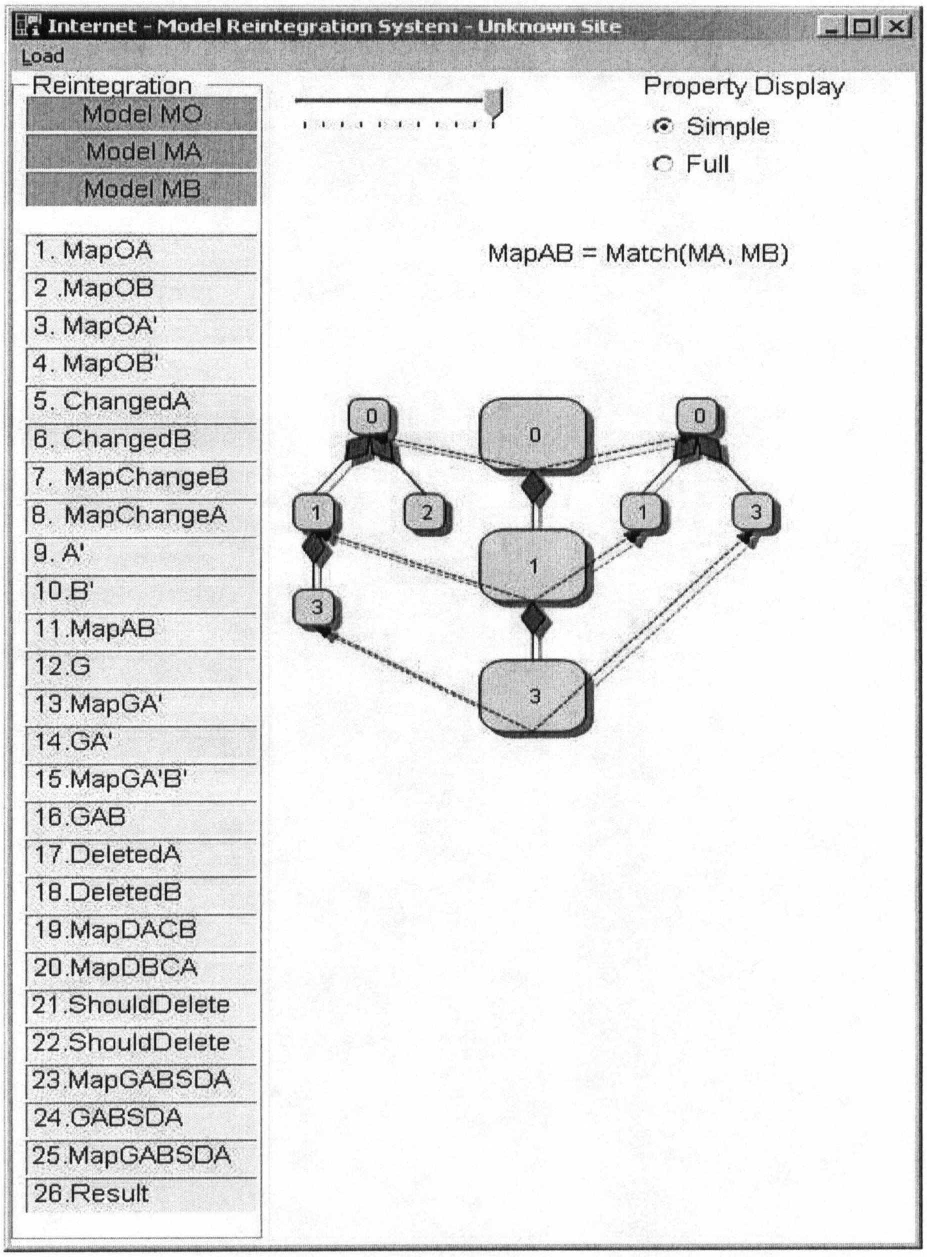
Figure A.12: The Difference Model A' between Models *ChangedA* and *ChangedB*



Property Display: [✓ Simple] Full

B' = Diff(ChangedB, ChangedA)

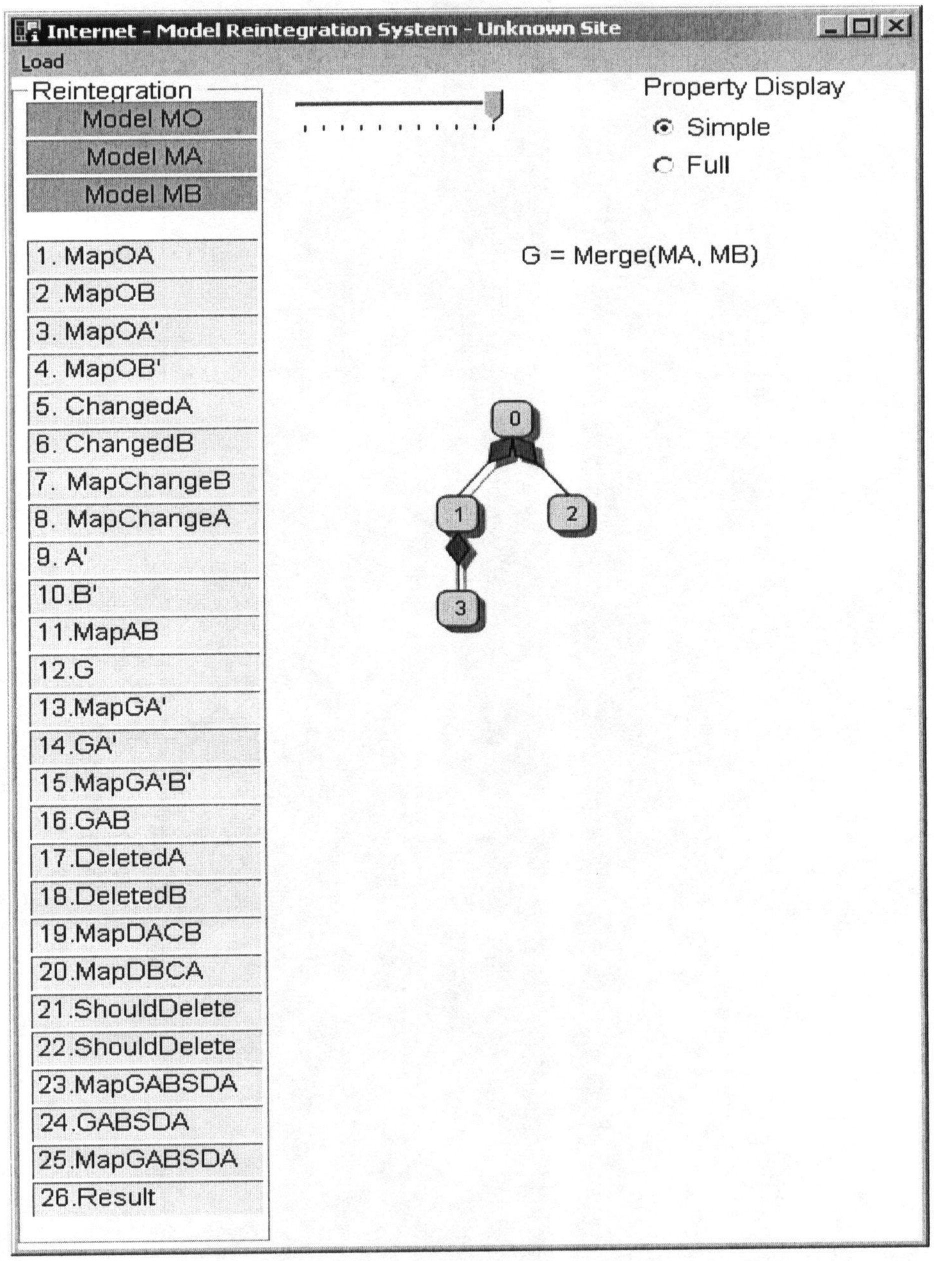
Figure A.13: The Difference Model B' between Models $ChangedB$ and $ChangedA$



Property Display
[✓ Simple] Full

MapAB = Match(MA, MB)

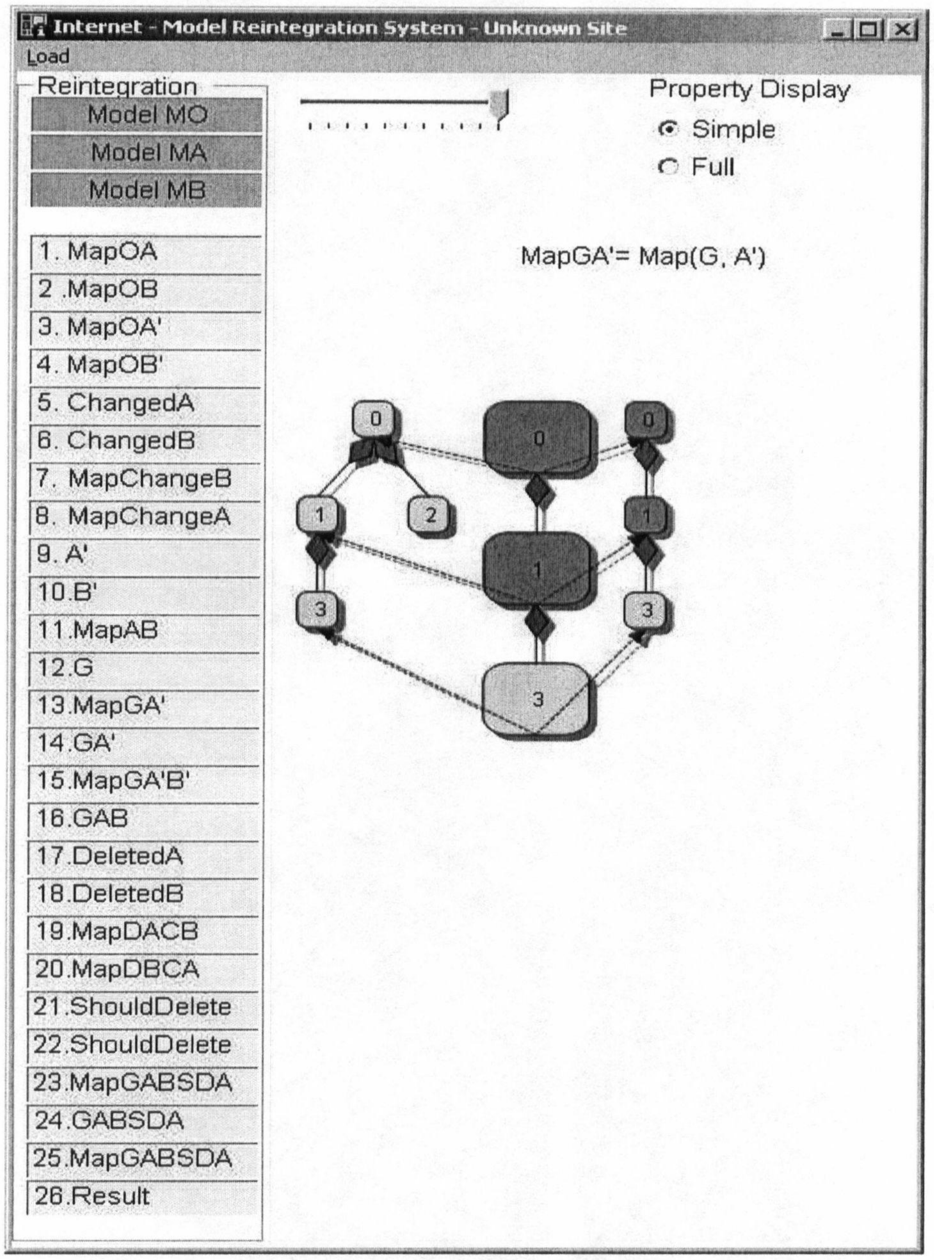
Figure A.14: The Mapping Map_{AB} between Models MA and MB



Property Display: [✓ Simple] Full

G = Merge(MA, MB)

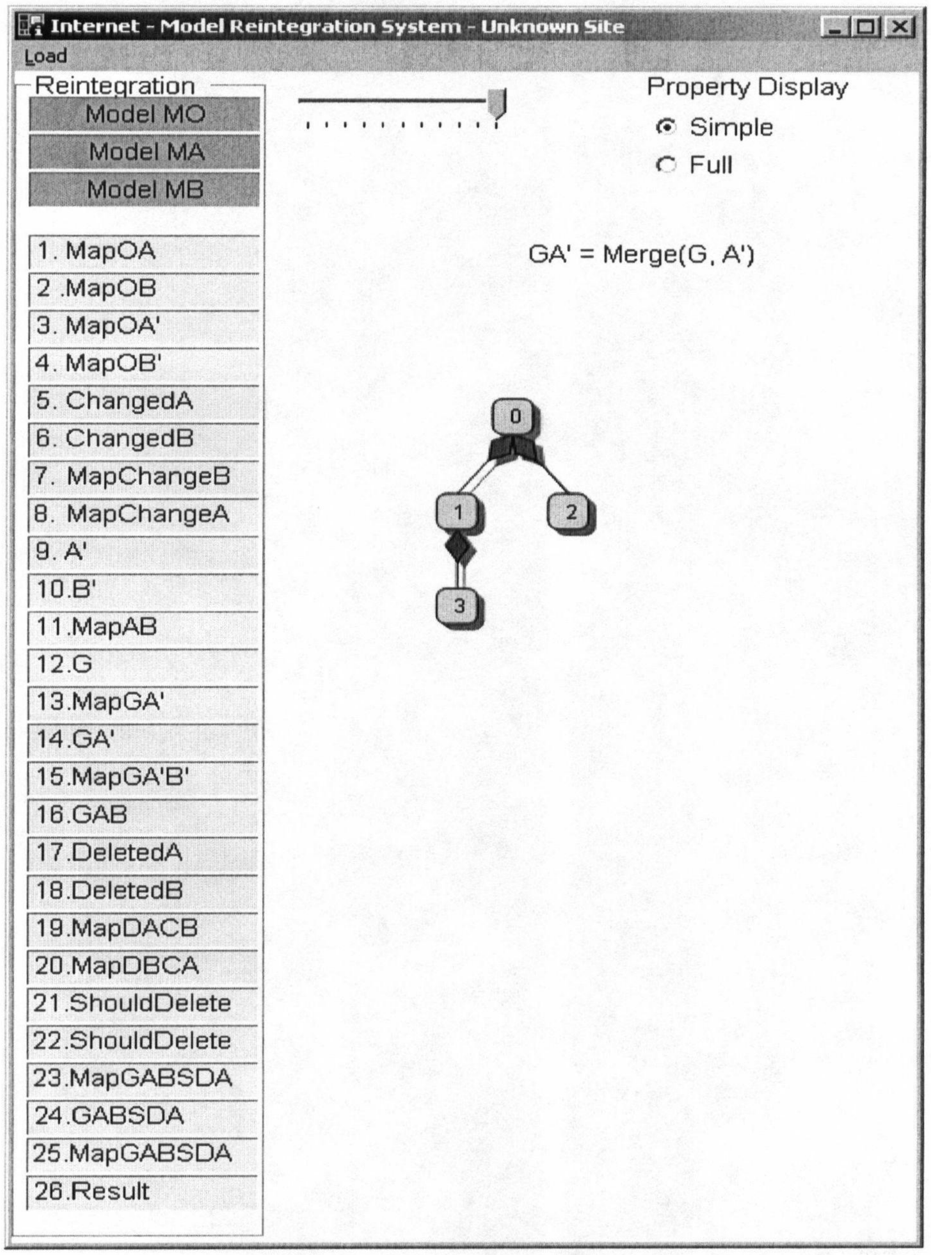
Figure A.15: The Merged Model G from Models MA and MB



Property Display: [✓ Simple] Full

MapGA'= Map(G, A')

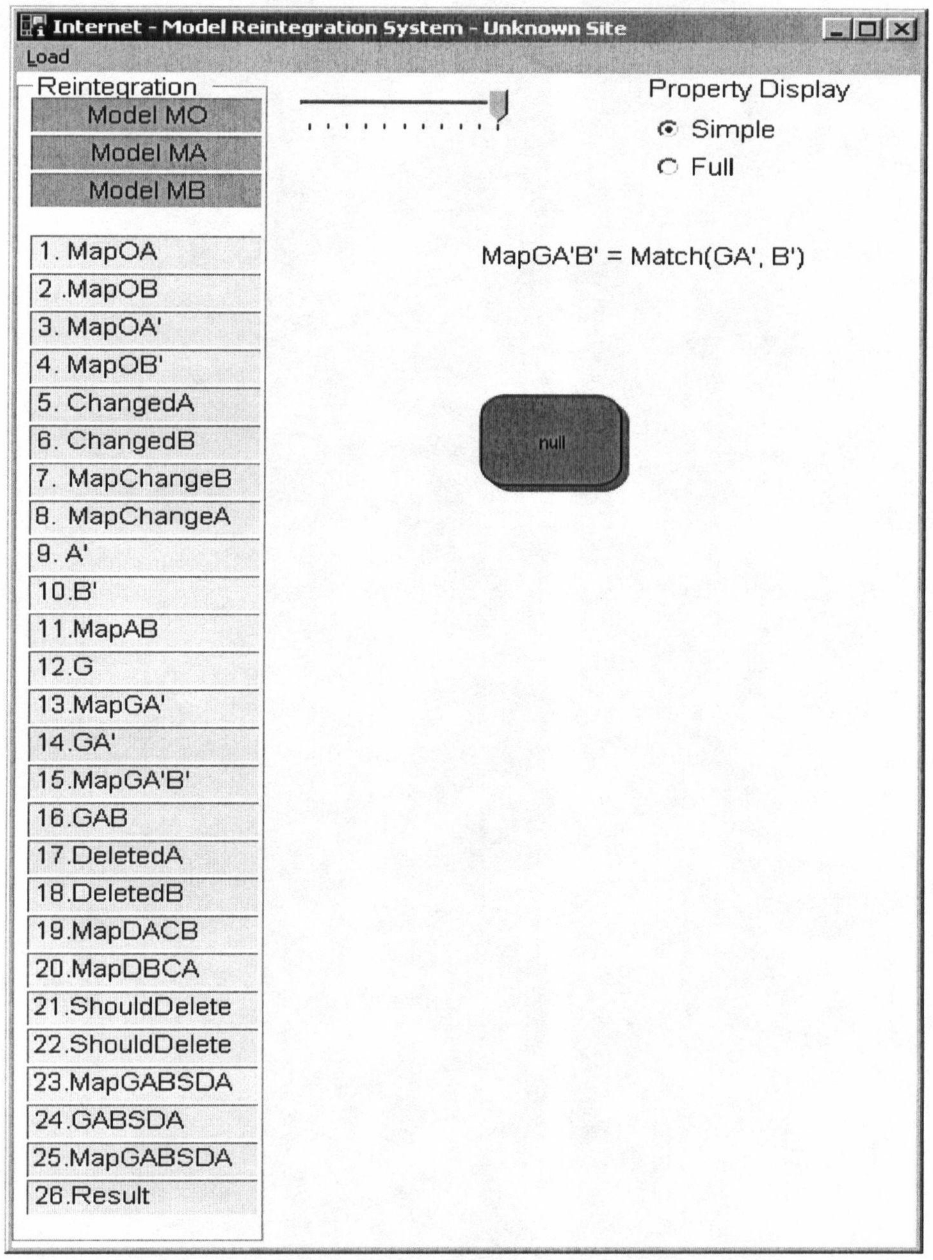
Figure A.16: The Mapping $Map_{GA'}$ between Models G and A'



Property Display: [✓ Simple] Full

GA' = Merge(G, A')

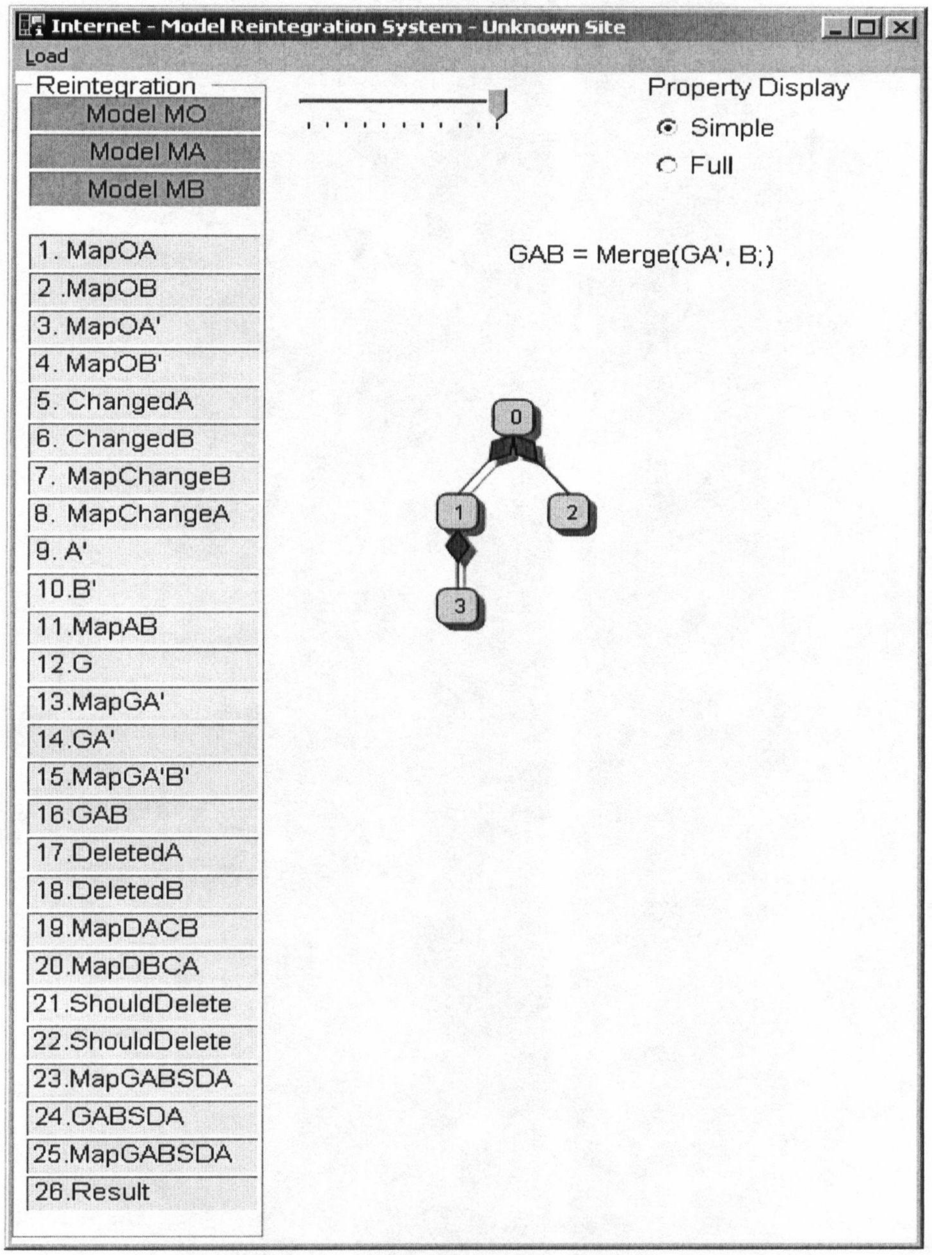
Figure A.17: The Merged Model GA' from Models G and A'



Property Display: [✓ Simple] Full

MapGA'B' = Match(GA', B')

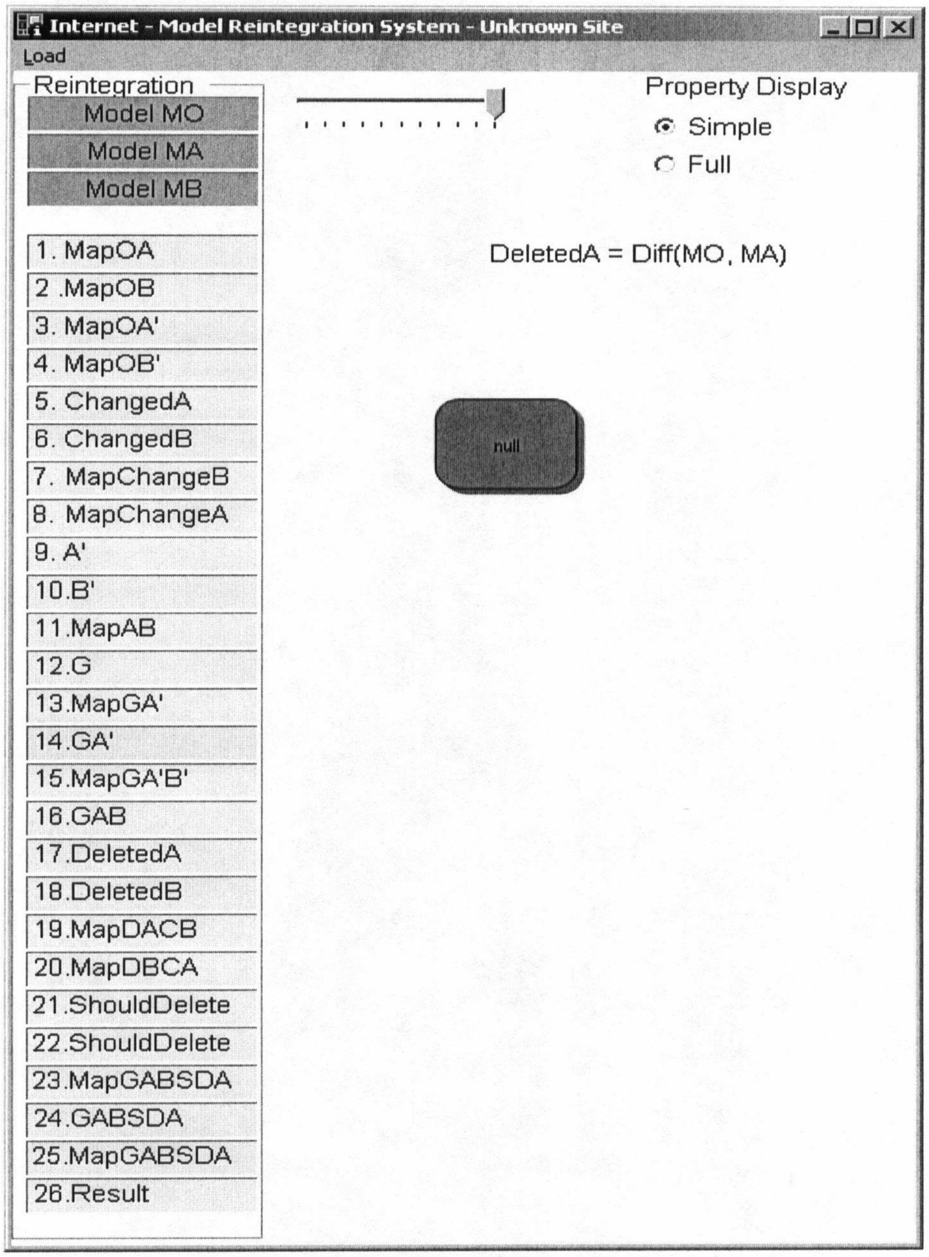
Figure A.18: The Mapping $Map_{GA'B'}$ between Models GA' and B'



Property Display: [✓ Simple] Full

GAB = Merge(GA', B;)

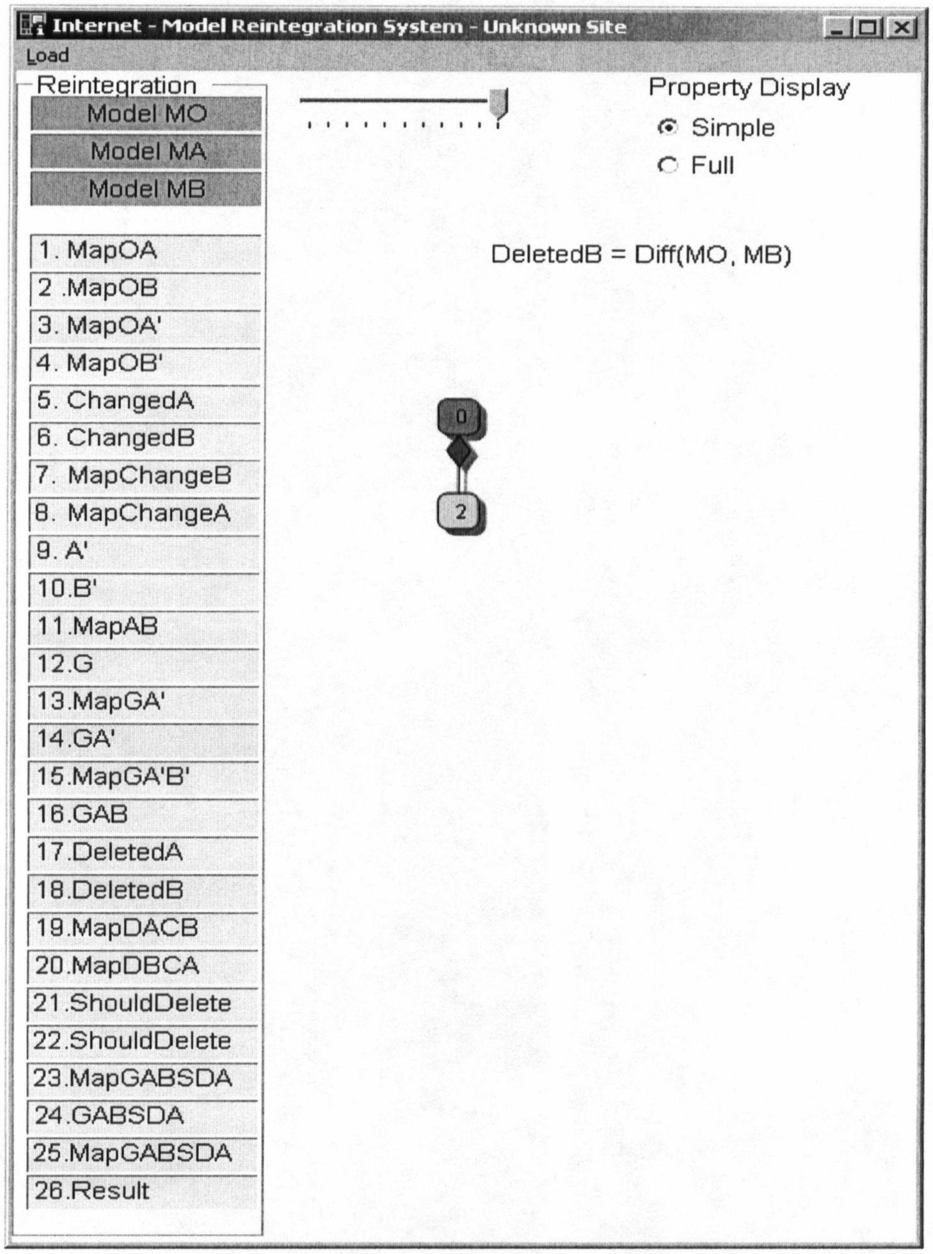
Figure A.19: The Merged Model GAB from Models GA' and B'



Property Display: [✓ Simple] Full

DeletedA = Diff(MO, MA)

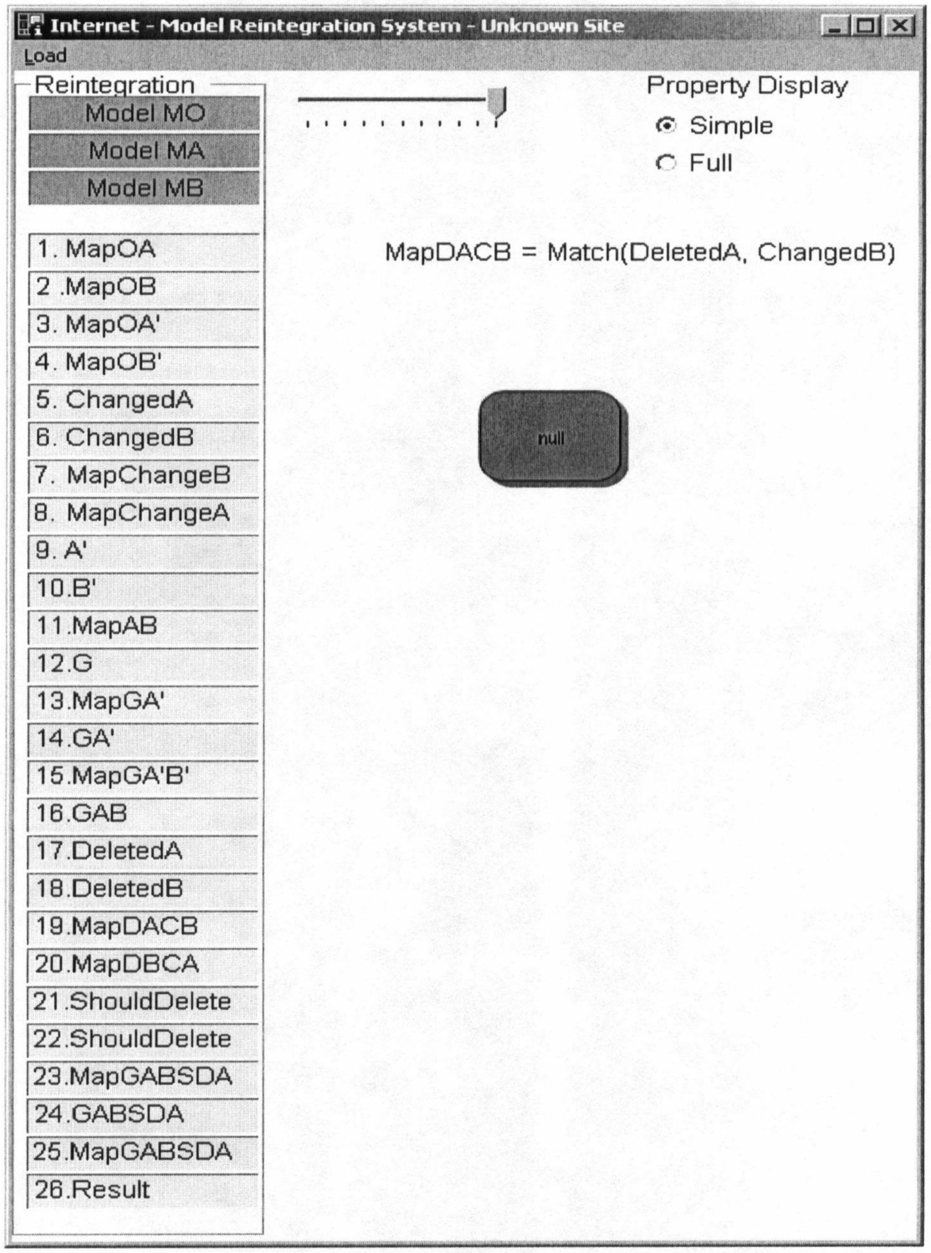
Figure A.20: The Model $Deleted_A$ Showing The Difference between Models MO and MA



Property Display: [✓ Simple] Full

DeletedB = Diff(MO, MB)

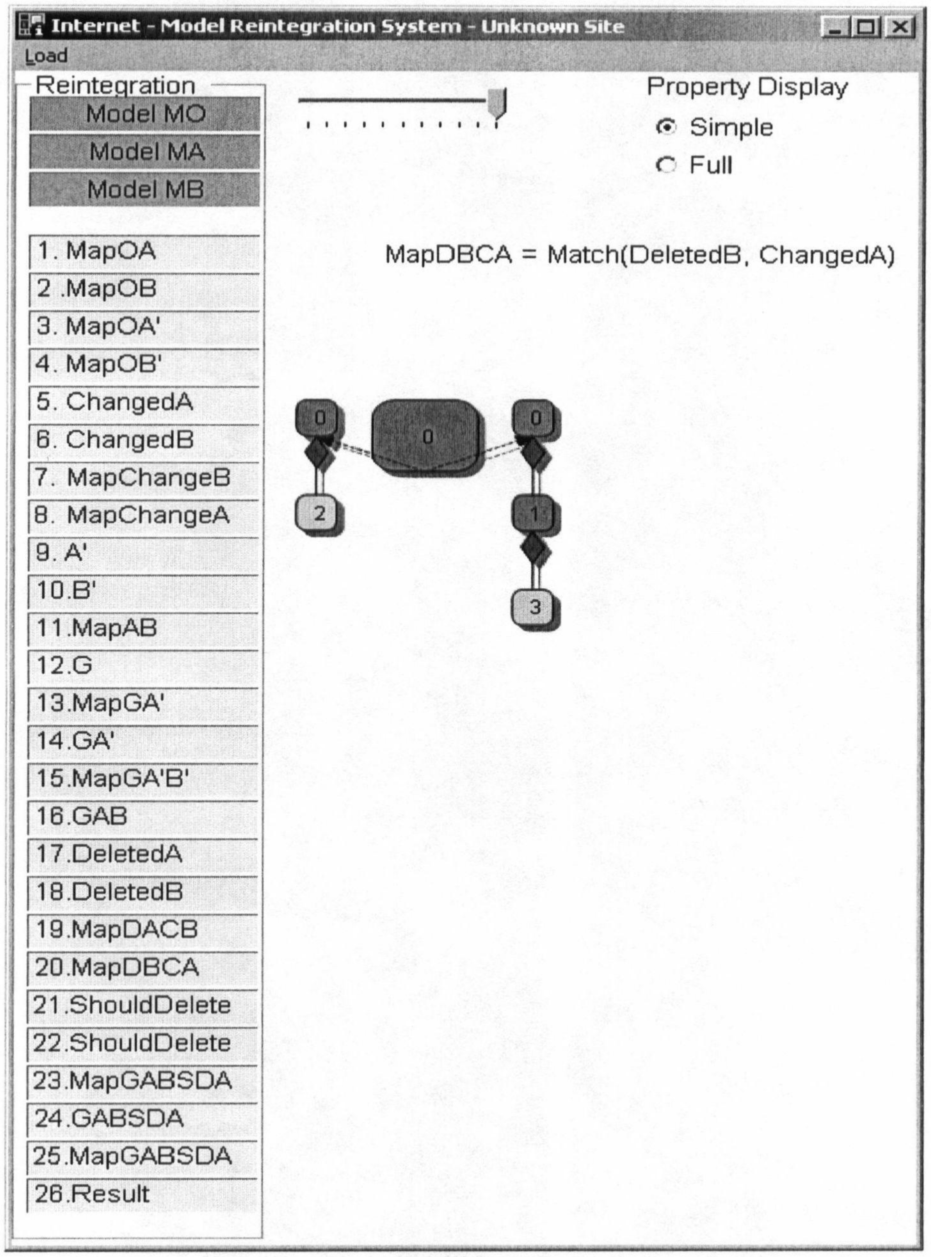
Figure A.21: The Model $Deleted_B$ Showing The Difference between Models MO and MB



Property Display: [✓ Simple] Full

MapDACB = Match(DeletedA, ChangedB)

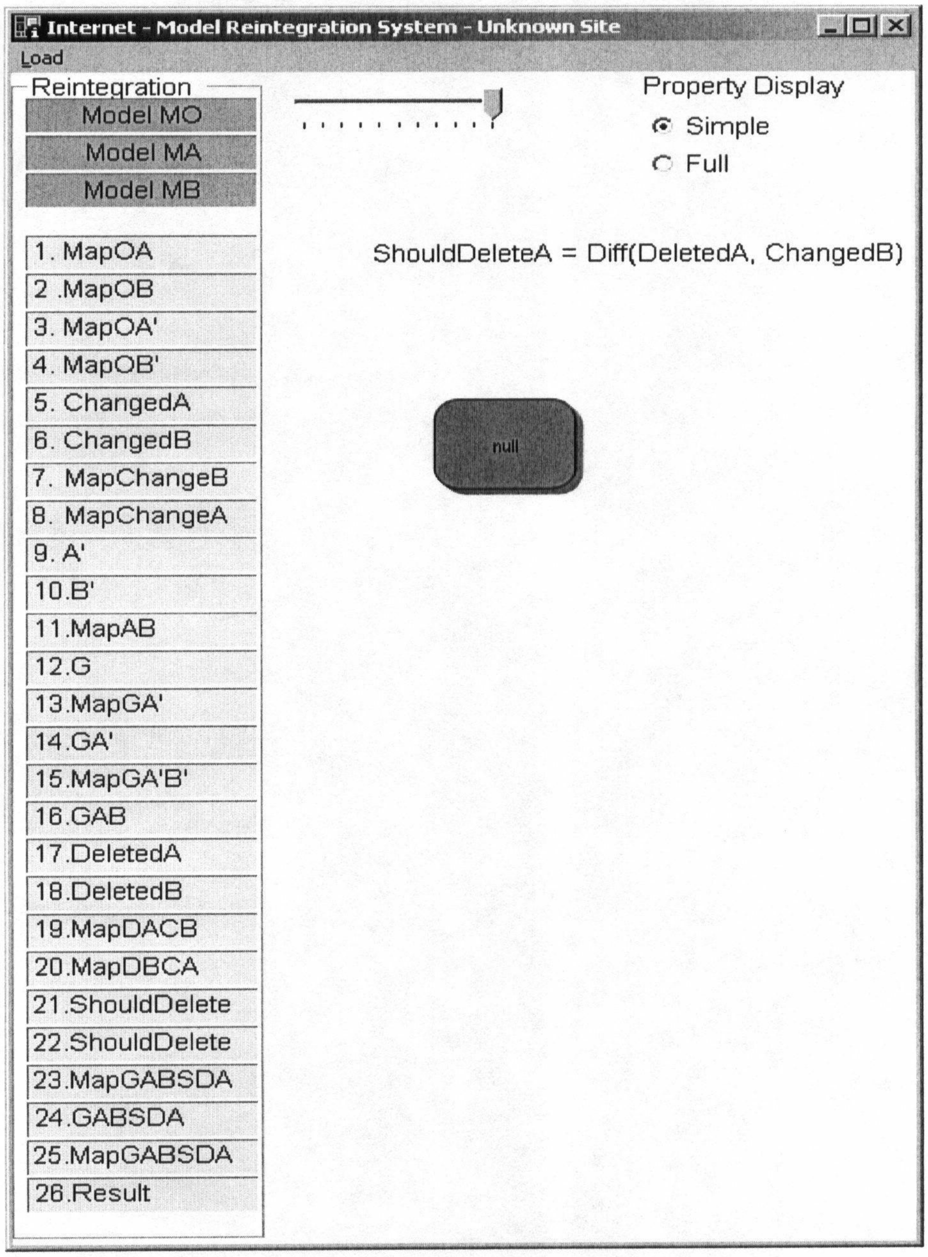
Figure A.22: The Mapping Model Map_{DACB} between Models $Deleted_A$ and $Changed_B$



Property Display: [✓ Simple] Full

MapDBCA = Match(DeletedB, ChangedA)

Figure A.23: The Mapping Model Map_{DBCA} between Models $Deleted_B$ and $Changed_A$

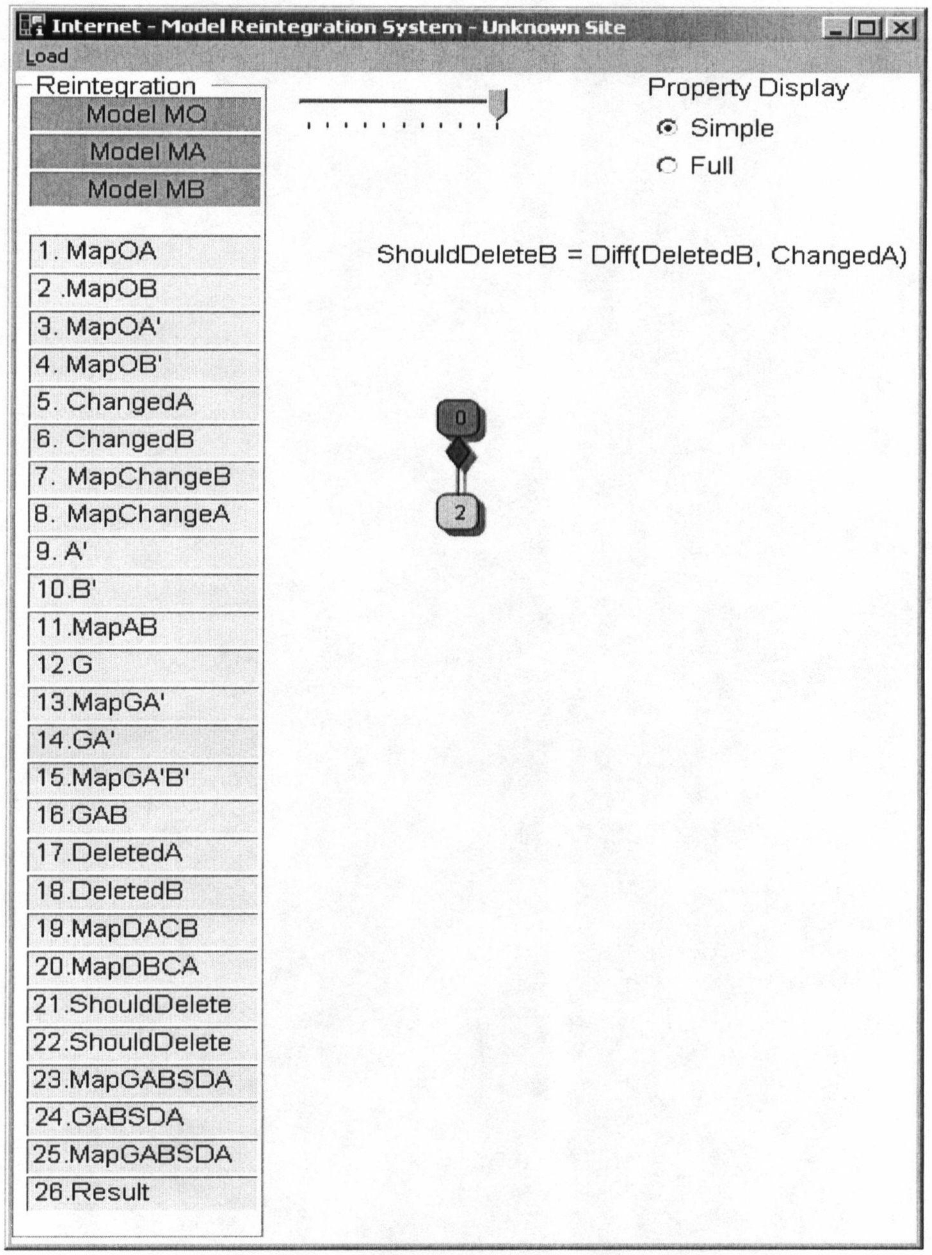


Property Display

[✓ Simple] Full

ShouldDeleteA = Diff(DeletedA, ChangedB)

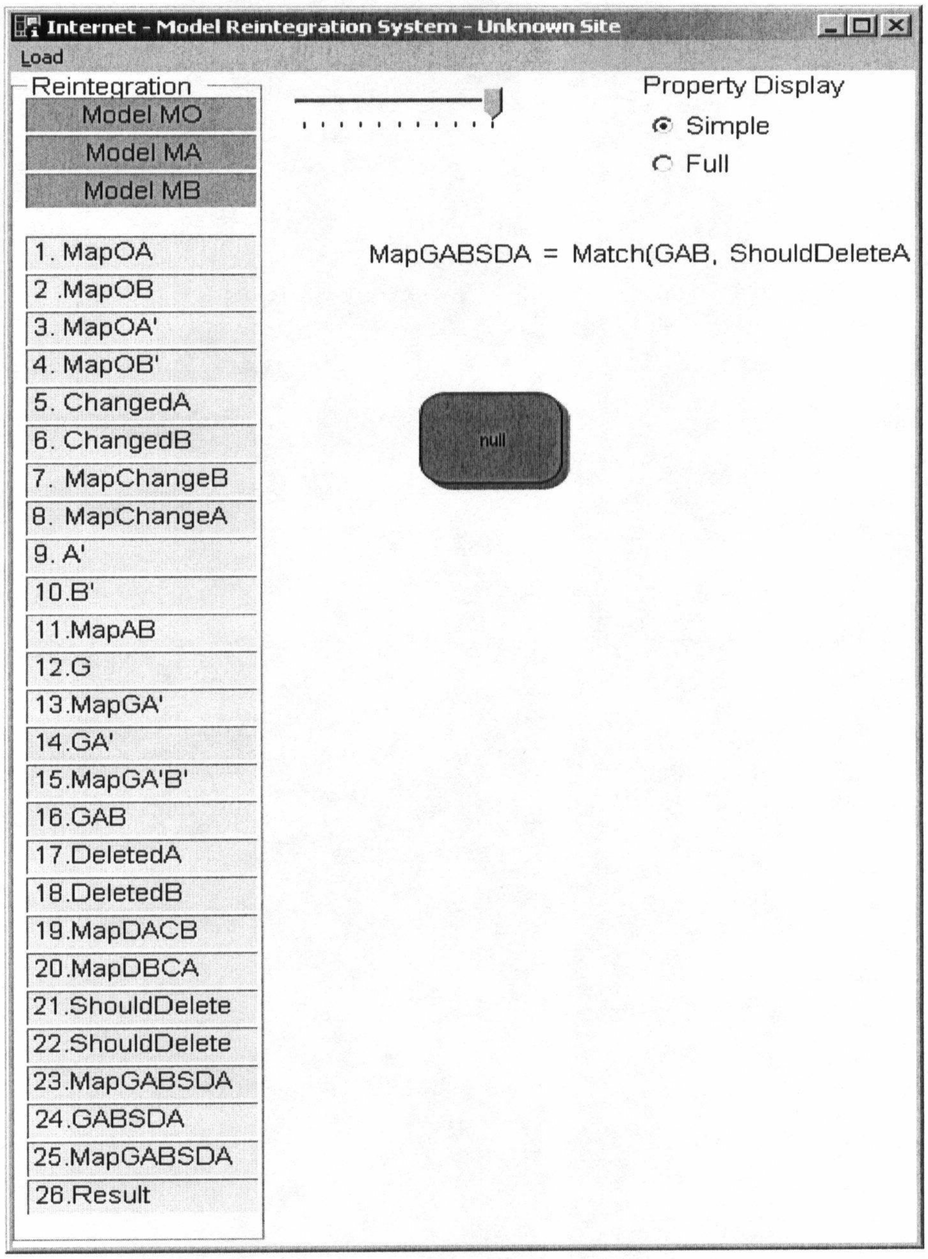
Figure A.24: The Difference Model $ShouldDelete_A$ between $Deleted_A$ and $Changed_B$



Property Display: [✓ Simple] Full

ShouldDeleteB = Diff(DeletedB, ChangedA)

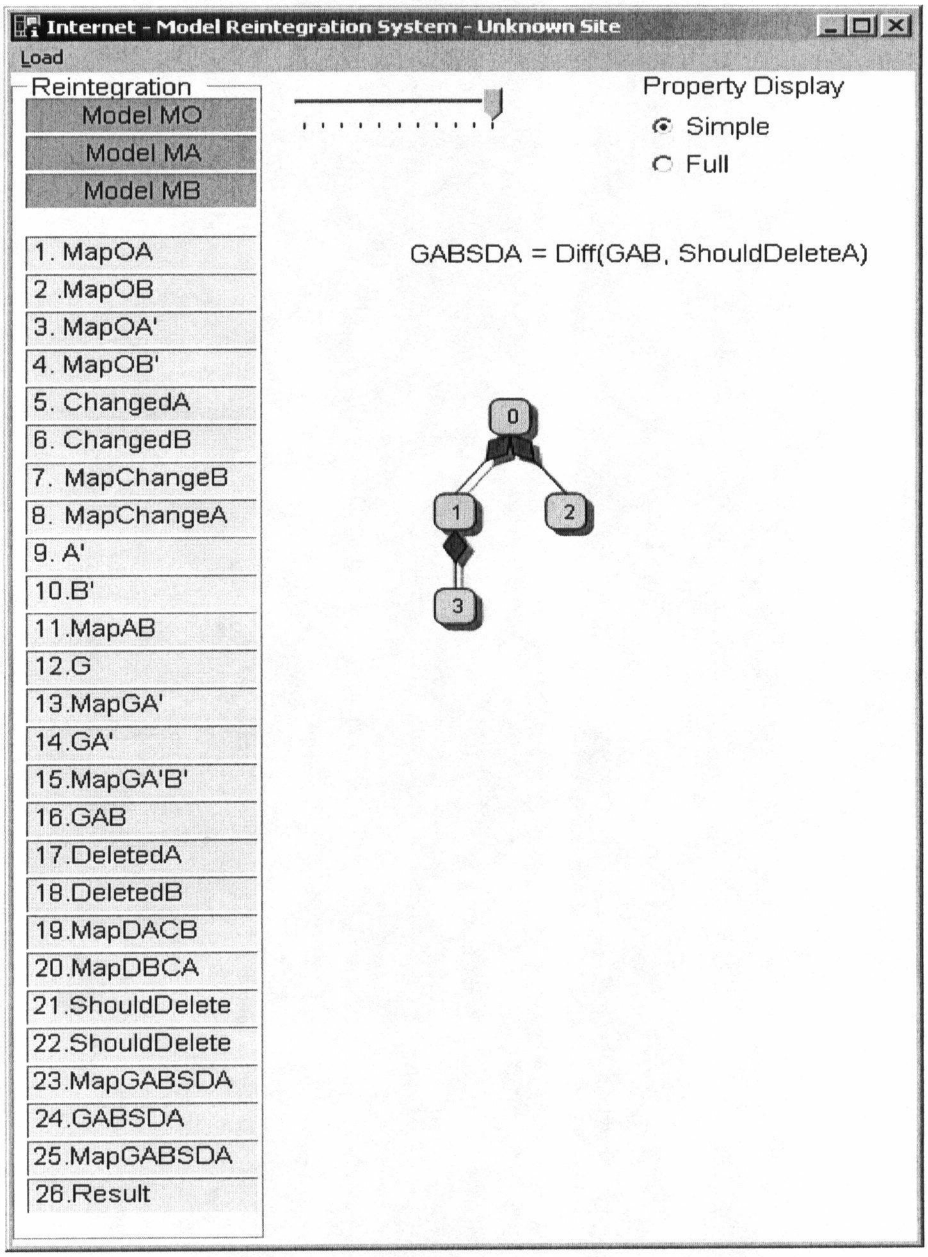
Figure A.25: The Difference Model *ShouldDelete*$_B$ between *Deleted*$_B$ and *Changed*$_A$



Property Display: [✓ Simple] Full

MapGABSDA = Match(GAB, ShouldDeleteA

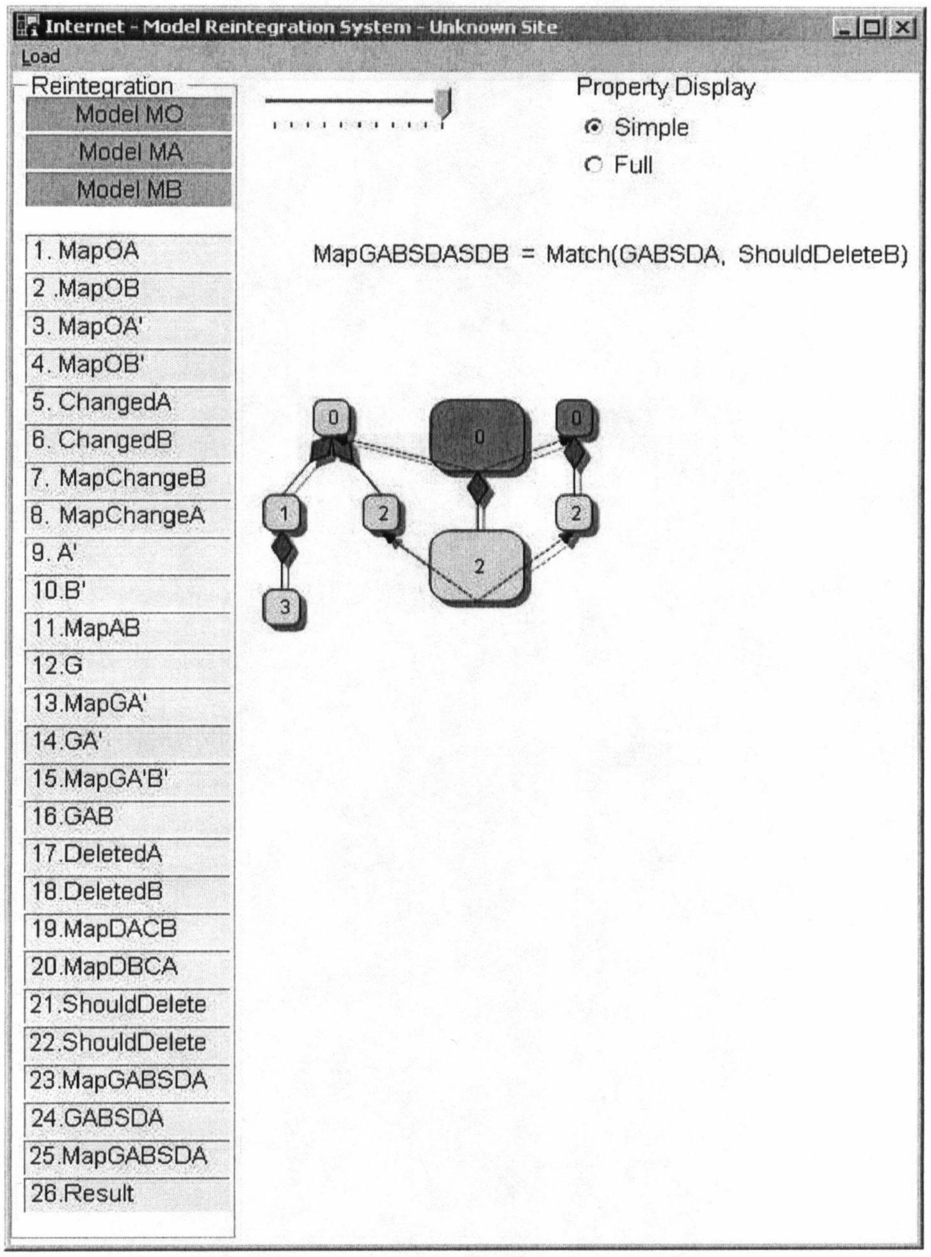
Figure A.26: The Mapping Model between Models *GAB* and $ShouldDelete_A$



Property Display: [✓ Simple] Full

GABSDA = Diff(GAB, ShouldDeleteA)

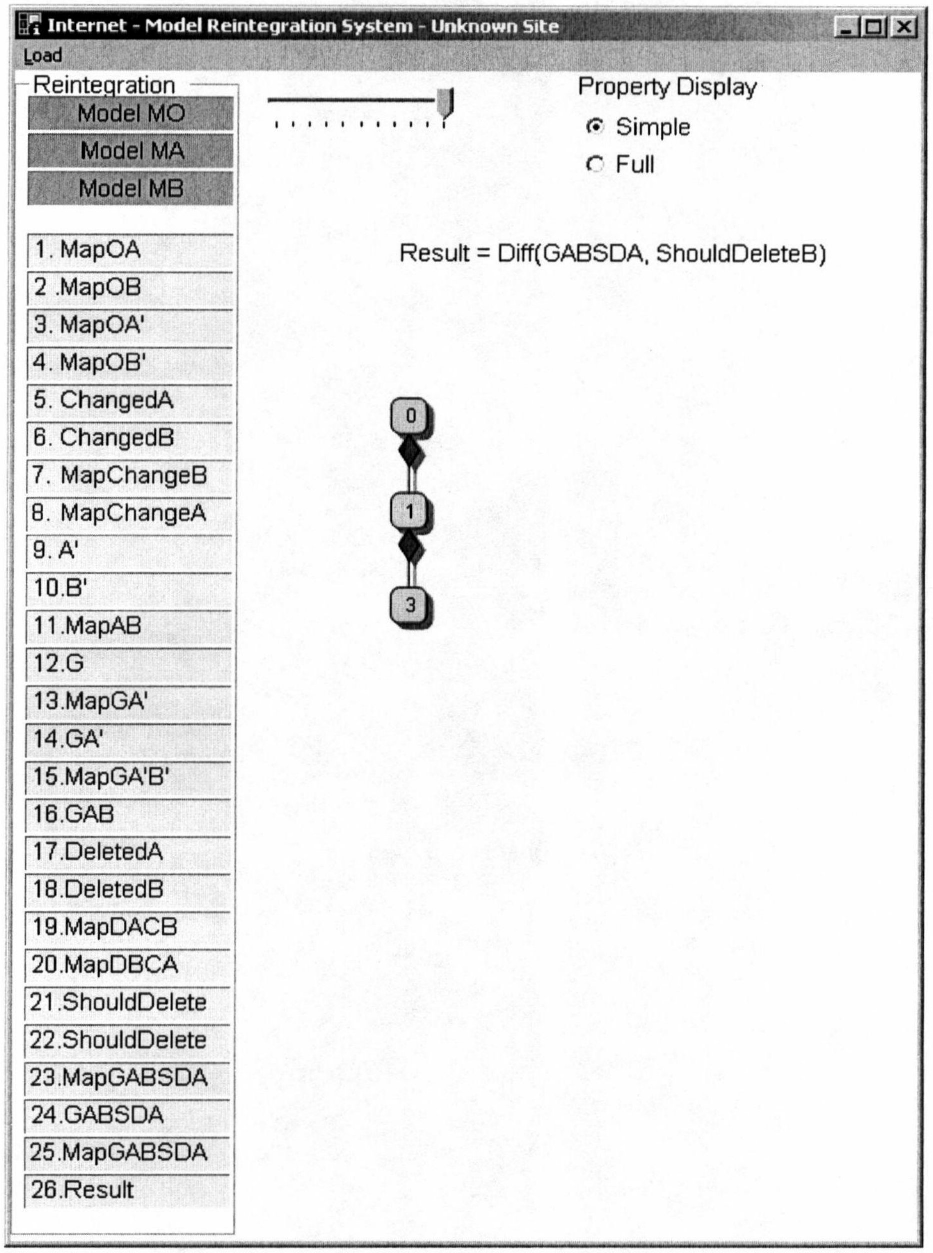
Figure A.27: The Difference Model $GABSDA$ between Models GAB and $ShouldDelete_A$



Property Display
[✓ Simple]
Full

MapGABSDASDB = Match(GABSDA, ShouldDeleteB)

Figure A.28: The Mapping Model between Model $GABSDA$ and Model $ShouldDelete_B$



Property Display: [✓ Simple] Full

Result = Diff(GABSDA, ShouldDeleteB)

Figure A.29: The Final Result Model *Result*, Difference between Models *GABSDA* and $ShouldDelete_B$