Imperial College of Science,
Technology and Medicine

(University of London)

Department of Computing

"Database Schema Integration Tool"

by
Nikolaos Rizopoulos

Submitted in partial fulfilment
of the requirements for the MSc
Degree in Advanced Computing of the
University of London and for the
Diploma of Imperial College of
Science, Technology and Medicine.

September 2001

Abstract

Several database schema integration methodologies defining transformations between equivalent schemas have been proposed in the literature. This project adopts one of these methodologies and develops upon it to define a way of representing composite transformations between database schemas. A simple, high-level Application Program Interface (API) has been implemented that enables the definition of complex transformations and their execution on existing schemas. Finally, a graphical representation language is recommended that provides a data-model independent way of displaying database schemas.

This project succeeds in defining *templates* of composite transformations on database schemas, i.e. transformations irrespective of existing schemas. It employs a language that is model and schema independent and provides a way of expressing composite transformations in this language using the developed API. Essentially, this project extends the work described in [BT01] that enables the description of models, database schemas and primitive transformations.

A graphical tool has been built that demonstrates the utilization of the graphical representation language and allows the execution of template composite transformations on existing database schemas.

To my sister, my friends and Belle & Sebastian

Acknowledgements

I would like to thank my supervisor Dr Peter McBrien for his support and guidance throughout this project. I would also like to thank Mike Boyd and Nerissa Tong for their help and patience.

Finally, a special thanks goes to my friend and colleague Apostolos Oikonomou without whom this report wouldn't be ready on time.

Table Of Contents

Abstract I

Acknowledgements V

Table Of Contents VII

Table Of Figures X

Table Of Tables XII

Chapter 1 - Introduction 1

1.1 Project Outline 2

1.2 Report Structure 5

Chapter 2 - Background 7

2.1 Distributed Database Architecture 7

2.1.1 Distributed Databases Taxonomy 7

2.1.2 Five-Level Model for Federated Database Systems 8

2.2 Reasons for Schema Integration 9

2.3 Schema Equivalence 10

2.4 Schema Integration Process 11

2.5 Schema Integration Methodology adopted 12

2.5.1 Hypergraph Data Model 12

2.5.2 The Methodology 13

2.5.3 Mapping High-Level Data Models to HDM 14

2.6 Previous Work 15

2.6.1 Model Definitions Repository 15

2.6.2 Schemas and Transformations Repository 17

Chapter 3 - Architecture 20

3.1 Template Transformations 20

3.2 Template Transformations Framework 22
3.2.1 Template Transformation Language 22
3.2.2 Template Transformation Definitions 23
3.2.3 Template Transformation Execution 26
3.3 Graphical Representation Language 27
Chapter 4 - Template Transformation Language 31
4.1 Design 31
4.2 Statement Definition 31
4.3 General Considerations 35
Chapter 5 - Template Transformation Definition 37
5.1 Template Transformation Repository 37
5.2 Statement Invocation 38
5.3 A Simple Template Transformation Definition 44
5.4 A more complicated Template Transformation Definition 48
Chapter 6 - Template Transformation Execution 54
6.1 Template Execution Repository 54
6.2 Mapping Arguments 55
6.3 Statement Execution Trace 57
6.4 Template Transformation Execution Algorithm 62
Chapter 7 - Graphical Representation Language 65
7.1 Language Design 65
7.2 Language implementation 67
7.3 Graphical Representation's Statements 68
7.4 Statement Syntax 71
7.5 A Graphical Representation Definition Example 73
Chapter 8 - Conclusions and Future Work 76

References 78
Appendix A 80
Appendix B 84

Table Of Figures

Figure 1.1: Mapping ER to HDM 2
Figure 1.2: Schemas after composite transformation execution 3
Figure 1.3: Composite transformation in HDM notation 3
Figure 1.4: Schemas according to [BT01] 3
Figure 1.5: Fingerprints of statements used in the template transformation language 4
Figure 1.6: Fingerprint of a template composite transformation 4
Figure 1.7: Template Composite Transformation 5
Figure 2.1: Taxonomy of multidatabase systems 7
Figure 2.2: Five-Level Model for Federated Databases 9
Figure 2.3: MDR and STR repositories schema 19
Figure 3.1: Composite transformation example with two sub-entities 20
Figure 3.2: Composite transformation example with three sub-entities 21
Figure 3.3: Composite transformation with variables 21
Figure 3.4: The abstract form of a template transformation 24
Figure 3.5: TTSR, TTR and TER repositories schema 27
Figure 3.6: SMR repository schema 30
Figure 4.1: ForEach statement's abstract form 33
Figure 7.1: Different uses of the same primitive shape (a) rhombus can be used as an ER relationship (b) rhombus extended horizontally can be used as an ER generalization 65
Figure A.1: Load schema using the implemented Tool 80
Figure A.2: Choose a schema from the list to load 80
Figure A.3: Graphical Representation of chosen schema 81
Figure A.4: Choose transformation to execute 81
Figure A.5: Defining arguments for a template transformation execution 82

Figure A.6: Defining list arguments using the tool..... 83

Table Of Tables

Table 2.1: MDR model relation 16
Table 2.2: MDR construct relation 16
Table 2.3: MDR constructScheme relation 16
Table 2.4: ER model represented in the MDR 17
Table 2.5: STR schema relation 17
Table 2.6: STR object relation 18
Table 2.7: STR scheme relation 18
Table 2.8: STR transformation relation 18
Table 2.9: STR populated 19
Table 3.1: TTSR statements relation 23
Table 3.2: TTSR statement definition relation 23
Table 3.3: TTR composite transformations relation 24
Table 3.4: TTR composite transformation definition relation 25
Table 3.5: TTR arguments relation 25
Table 3.6: TTR composite transformation program relation 25
Table 3.7: TER arguments relation 26
Table 3.8: TER composite transformation invocation relation 26
Table 3.9: TER composite transformation execution relation 27
Table 3.10: SMR methods relation 29
Table 3.11: SMR method arguments relation 29
Table 3.12: SMR construct modeling relation 29
Table 4.1: Template Transformation Language's statements 31
Table 4.2: NewObject argument definition 32
Table 4.3: Primitive Transformation argument definition 33
Table 4.4: ForEach argument definition 34

Table 4.5: ExtendGeneralization Scheme argument definition..... 35
Table 4.6: Passing arguments by reference in the transformation language 36
Table 5.1: New Object statement invocation 40
Table 5.2: Primitive Transformation statement invocation 41
Table 5.3: ForEach statement invocation 43
Table 5.4: Extend Generalization's Scheme invocation..... 44
Table 5.5: Entity-Attribute Equivalence transformation 45
Table 5.6: Entity-Attribute Equivalence argument definition..... 45
Table 5.7: Entity-Attribute Equivalence 1st statement..... 46
Table 5.8: Entity-Attribute Equivalence 2nd statement..... 46
Table 5.9: Entity-Attribute Equivalence 3rd statement 47
Table 5.10: Entity-Attribute Equivalence 4th statement 47
Table 5.11: Entity-Attribute Equivalence 5th statement 48
Table 5.12: Mandatory Attribute and Total Generalization Equivalence..... 49
Table 5.13: Transformation executed in each iteration 50
Table 5.14: Mandatory Attribute-Generalization Equivalence 1st statement 51
Table 5.15: Mandatory Attribute-Generalization Equivalence 2nd statement 51
Table 5.16: Mandatory Attribute-Generalization Equivalence 3rd statement..... 52
Table 5.17: Mandatory Attribute-Generalization Equivalence 4th statement..... 52
Table 5.18: Mandatory Attribute-Generalization equivalence last two statements .. 53
Table 6.1: Template Transformation Invocation..... 54
Table 6.2: STR before template transformation invocation 55
Table 6.3: Transforming non-list compile-time arguments to run-time arguments.. 56
Table 6.4: Transforming compile-time list arguments to run-time arguments 57
Table 6.5: NewObject statement execution trace..... 58
Table 6.6: Primitive Transformation statement execution trace 59

Table 6.7: Extend Generalization Scheme statement execution trace: (a) STR and TTR before execution (b) STR and TER after execution 59

Table 6.8: ForEach statement. Execution initialisation. 60

Table 6.9: ForEach statement. Execution trace after creating object..... 61

Table 6.10: ForEach statement. Execution trace after add object..... 61

Table 6.11: ForEach statement. Execution trace after 2nd iteration. 62

Table 7.1: Standard properties of primitive shapes..... 66

Table 7.2: Shape-specific properties..... 66

Table 7.3: Methods supplied by classes representing the four primitive shapes 69

Table 7.4: SchemaManager supplied methods..... 69

Table 7.5: GraphicSchemaObject supplied methods 70

Table 7.6: Abstract Modelling example 73

Table 7.7: Setting a property to a shape..... 74

Table 7.8: ER entity graphical representation definition..... 75

Chapter 1 - Introduction

Large and distributed organizations require the use of data in a variety of formats. From flat files to object-oriented databases, organizations need to access these information sources for data-processing and data-storage. This diversity can become problematic when data have to be exchanged or shared between organizations or even databases inside the same organization. *Database schema integration* can solve this problem by unifying existing schemas and creating a single, *federated* schema [She90]. Schema integration provides interoperability between the various data sources and offers flexibility and efficiency when accessing heterogeneous databases.

Database schema integration is defined in [Bat86] as the activity of integrating the schemas of existing or proposed databases into a global, unified schema. This activity can be divided into two categories: *view integration* and *database integration*. View integration produces a conceptual description of a proposed database and database integration produces the global schema of a collection of databases. Both must produce a single schema that expresses all the data that existed in the original schemas. Different views of the same universe of discourse can be merged into a global view with view integration, while database integration can be exercised when there are various autonomous databases.

Several methodologies exist for the semantic integration of databases, most of them defining a set of transformations that map between schemas that are in some sense equivalent. This project adopts one of these methodologies and describes composite transformations on database schemas according to it. Parameterised composite transformations, named *templates*, are successfully defined. These template transformations are *schema-independent* in the sense that they can be applied to any schema. The work done in [BT01], which accomplishes the description of data models, schemas and primitive transformations according to the aforementioned methodology, is extended to support composite transformations and graphical representation definitions.

This report presents the work done in this project. It describes the template transformation language and the graphical representation language designed and implemented. Additionally, it presents the tool that has been developed to demonstrate the employment of this work.

1.1 Project Outline

The methodology, which is adopted in this project and analysed in [PM98], describes a general framework to support the schema transformation process. The methodology defines a low-level *hypergraph data model* (HDM) and explains how this model can be used to represent high-level models, like ER [Chen76] and UML [UML]. The HDM can describe a database schema using primitive constructs like *nodes*, *edges* and *constraints*. Figure 1.1 shows an example of how an ER schema can be represented in the HDM: an ER entity is mapped to an HDM node and an ER attribute to a node and an edge.

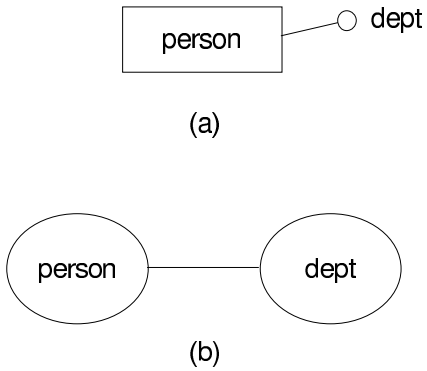

Figure 1.1: Mapping ER to HDM

A number of atomic, *primitive transformations* are defined on the HDM that can add/delete/rename nodes, edges and constraints. These primitive transformations can be used to construct *composite transformations* that preserve schema equivalence. An abstract description of a composite transformation is shown in Figure1.3: the composite transformation produces the schema of Figure 1.2(b) from the schema in Figure 1.1(b). Details of how the new constructs are derived are omitted.

In [MP99] the work of [PM98] is extended by providing a generic approach to defining the semantics of data models in terms of the HDM. Each construct in any model can be built using HDM's nodes, edges and constraints.

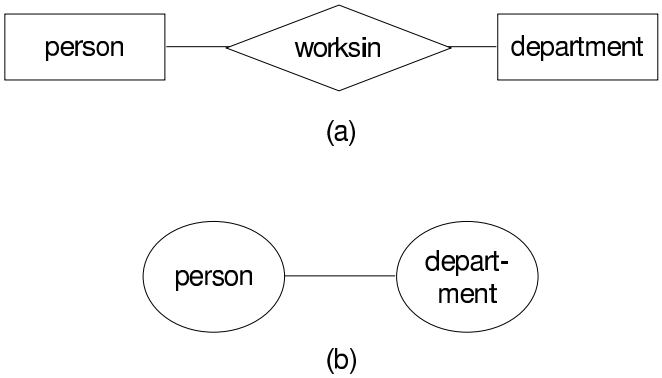

Figure 1.2: Schemas after composite transformation execution

```
addNode(department);
addEdge(worksin);
delEdge(personDeptEdge);
delNode(dept);
```

Figure 1.3: Composite transformation in HDM notation

[BT01] uses the work of [PM98], [MP99] and succeeds in mapping high-level data models to the low-level HDM and representing schemas in a model-independent fashion. This is accomplished by defining the constructs of each data-model along with their equivalent HDM type and representing instances of specific constructs as *objects* with an associated *scheme*. A database schema is represented as a list of these objects. Primitive transformations can be applied to specific schemas by adding/deleting/renaming pre-existing objects. Figure 1.4 shows in an abstract form how the schemas of Figures 1.1, 1.2 can be represented according to [BT01].

```
   schema1 = {objectPerson, objectDept}
schema2 = {objectPerson, objectDepartment,
              objectWorksIn}
```

Figure 1.4: Schemas according to [BT01]

The project presented in this report utilizes the work of [BT01] and develops a language that can be used to define template transformations. The designing of such a language resembles the designing of a programming language: expressing composite transformations is similar to coding programs and creating an algorithm to execute composite transformations is like building an interpreter that executes the written programs.

The developed language consists of *primitive statements* and a template transformation is represented as a sequence of these statements. Both statements and transformations can be considered as parameterised procedures. The rest of this

section presents statements and template transformation using an abstract *procedural* notation. It is important to mention that the examples shown below are merely a presentation of what has been *approximately* designed and implemented. The abstract procedural notation is used because it can be easily comprehended and gives an insight of what was really developed.

The implemented language consists of primitive statements, like `createNewObject` and `executePrimitiveTransformation`. Since statements are similar to procedures their fingerprints and their arguments must be defined. Figure 1.5 shows the fingerprints of the two aforementioned statements and Figure 1.6 shows the fingerprint of a template transformation.

```
Object createNewObject(schema, constructType, name, scheme);
 void executePrimitiveTransformation(action, schemaApplied,
                    schemaProduced, object);
```

Figure 1.5: Fingerprints of statements used in the template transformation language

```
schema attributeEntityEquivalence( initialSchema, entity,
       attribute, newEntityName, relationshipName);
```

Figure 1.6: Fingerprint of a template composite transformation

One of the arguments in every template transformation has to be the schema that this transformation is going to be operated on. The rest of the arguments depend on the specific transformation and on the statements it employs.

Figure 1.7 describes the template transformation that corresponds to the schema-specific transformation of Figure 1.3. The first three statements of Figure 1.7 perform the `addNode` statement of Figure 1.3, the next three statements perform an `addEdge` and the final statements correspond to the delete statements of Figure 1.3.

Invoking the template transformation on an existing schema, will execute a schema-specific composite transformation. The schema of Figure 1.2(a) can be produced from the schema of Figure 1.1(a) by invoking:

```
attributeEntityEquivalence(schema1,person,dept,department,worksin)
```

```
schema attributeEntityEquivalence( initialSchema, entity, attribute,
      newEntityName, relationshipName){

   intermediateSchema1 = new Schema();
   newEntity = createNewObject( intermediateSchema1, ERentity,
                                    newEntityName, null);
   executePrimitiveTransformation( add, initialSchema,
                                    intermediateSchema1, newEntity);

    intermediateSchema2 = new Schema();
    newRelationship = createNewObject( intermediateSchema2,
                            ERrelationship, relationshipName, scheme);
    executePrimitiveTransformation( add, intermediateSchema1,
                            intermediateSchema2, newRelationship);

    finalSchema = new Schema();
    executePrimitiveTransformation( del, intermediateSchema2,
                                    finalSchema, attribute);
    return finalSchema;
}
```

Figure 1.7: Template Composite Transformation

Additionally, in this project a graphical representation language has been developed that can be used to define how a schema can be presented graphically. The statements of this language are implementation-oriented since they rely on the invocation of methods that have been previously implemented. The language supports statements that create primitive shapes, like arcs, boxes, lines and rhombuses, and methods that can be used to manipulate these shapes and create more complex ones.

1.2 Report Structure

Chapter 2 provides with a background study on database schema integration. An overview of the architecture of distributed databases is given since that's where schema integration is more frequently used. This chapter also describes in detail the schema integration methodology of [PM98] and the previous work developed and explained in [BT01].

In Chapter 3, this project's work is justified and the project's architecture is presented.

Chapter 4 describes the template transformation language and Chapter 5 exhibits how the language can be employed using two suitable examples. Chapter 6

presents how template transformations are executed, i.e. how they operate on specific schemas. Chapter 7 explains in detail the graphical representation language developed.

Finally, chapter 8 gives our concluding remarks and some ideas for future work.

Chapter 2 - Background

This chapter is aiming to provide a background study of the field of database schema integration. The reasons that entail schema integration are identified and the integration process is described. Additionally, the term *schema equivalence* is explained. Finally, the methodology of [PM98] and the work of [BT01] that this project extends are presented.

Since schema integration is more frequently applied in distributed databases, an overview of their architecture is provided in the next section.

2.1 Distributed Database Architecture

2.1.1 Distributed Databases Taxonomy

Several alternative architectures for distributed databases have been proposed in [Ozsu99]. In this section we present the taxonomy described in [She90], illustrated in Figure 2.1.

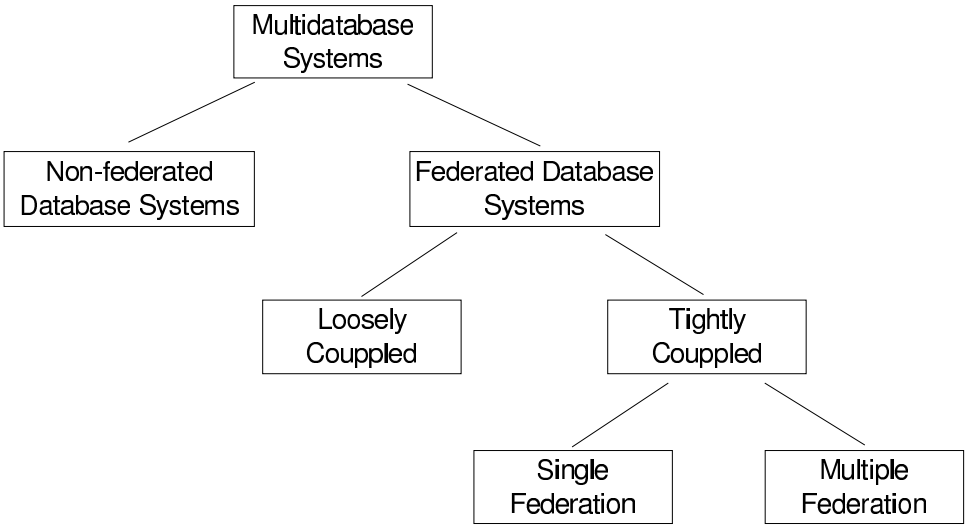

Figure 2.1: Taxonomy of multidatabase systems

A distributed database system, or *multidatabase* system [She90], consists of various databases distributed over several sites. These databases are called *component* databases. According to the autonomy of the component databases, multidatabase systems can be classified into two types: *non-federated* and *federated* systems. In non-federated database systems the component databases aren't autonomous. There is only one database administration authority that decides how

the component databases will be structured and accessed. In *federated* database systems the component databases are autonomous. The local database administrators have complete authority in the local database.

Federated database systems can be subdivided into *loosely coupled* or *tightly coupled* systems, based on who manages the federation and how the components are integrated. If the global database administrator maintains one or more global schemas that provide an integrated view of the whole database and these schemas can be used to access the data in the component databases using queries and updates, then the federated database is said to be tightly coupled. Otherwise, it is considered loosely coupled. In loosely coupled databases it's the user's responsibility to define the global schemas according to her application's needs.

Finally, tightly coupled database systems are said to have *single federation* if only one global schema is used, or *multiple federations* if more global schemas are used.

2.1.2 Five-Level Model for Federated Database Systems

In this section, we describe the architecture of tightly coupled federated database systems because in this category the notion of database integration exists.

The component databases provide a *local schema*, which is the conceptual schema for the particular database. This local schema has to be *translated* into the common data model of the federated database. The result of this translation is the *component schema*.

Each component database may not want to make available all the data represented in the component schema. Therefore, it *filters* the component schema into different partial views of the database, which are called *export schemas*. These export schemas are integrated into *global schemas*, which are again filtered to provide the *external schemas*. The external schemas are views of one global schema and are created according to the needs and rights of the user groups. The five-level model is illustrated in Figure 2.2.

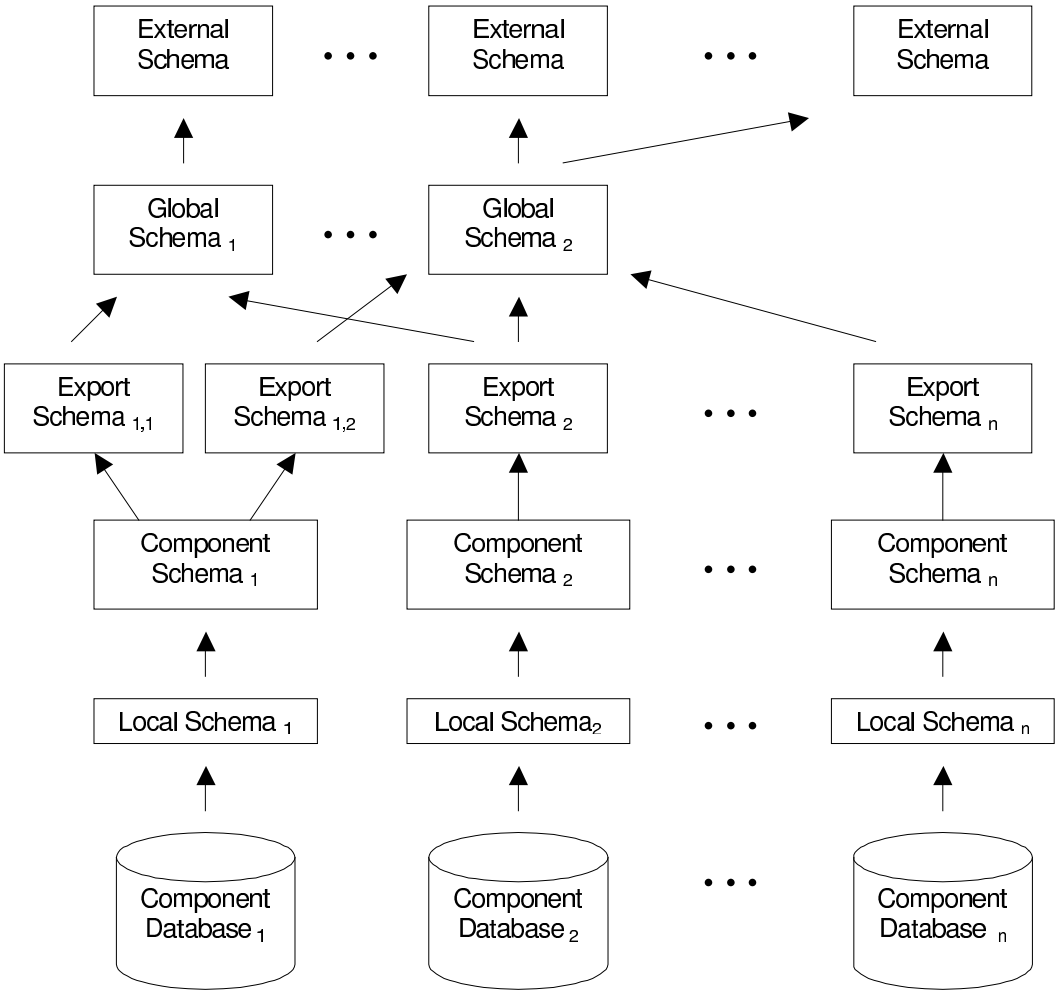

Figure 2.2: Five-Level Model for Federated Databases

2.2 Reasons for Schema Integration

In distributed databases the existence of various local databases and multiple schemas requires the use of schema integration. The use of different Database Management Systems creates *physical heterogeneity* and the numerous ways of representing data produces *semantic heterogeneity*. This section describes the causes of semantic heterogeneity since this is the primary problem schema integration is trying to solve.

According to [Bat86], the causes of semantic heterogeneity are:

1. Different perspectives. Designers have different information needs, thus they might describe the same concepts differently, e.g. defining different attributes for the same entity.

2. Equivalence among constructs. In data models several combinations of constructs can model the same concept equivalently. Models with greater descriptive abilities give rise to a larger variety of possibilities to represent the same situation.

The schema integration process tries to identify schema heterogeneity, which means that conflicts between schemas must be detected, and unify the underlying schemas by resolving these conflicts.

Conflicts can be classified as *semantic conflicts*, *descriptive conflicts* and *structural conflicts* [Spa92]:

1. Semantic conflicts arise when the designers have different perceptions on the real world representation of the same concepts (pluralism of perceptions), e.g. overlapping, included domains.
2. Descriptive conflicts arise when designers perceive different set of properties for the same concepts. Descriptive conflicts also include *homonyms* and *synonyms*. Two objects are homonyms when they have the same name but represent different concepts and synonyms when they represent the same concept but have different names.
3. Structural conflicts arise when different constructs are chosen to represent the same concept or when different integrity constraints are used to represent a relationship. These are divided into: *type conflicts* (different constructs), *dependency conflicts* (different cardinalities), *key conflicts* (different keys assigned to the same concept) and *behavioural conflicts* [Bat86].

2.3 Schema Equivalence

According to the previous section, equivalence between constructs is a basic reason for schema diversity. Schema integration methodologies identify these construct equivalences and extend them to schema equivalence. This section describes the different notions of schema equivalence found in the literature.

According to [Mil94], at the heart of schema integration lays the problem of detecting if two schemas or parts of schemas are equivalent. This rises from the fact that every time the schema is restructured, the original and restructured schemas must have some kind of equivalence relationship between them, in order for the restructuring phase to be successful. Schema equivalence is fundamental in methodologies that use primitive transformations to integrate schemas. For each transformation, the original and resulting schemas must obey one or more alternative notions of schema equivalence [Bat84].

In [Bat86], three categories of schema equivalence are identified:

1. *Transformational*: two schemas *S* and *S'* are equivalent if you can obtain *S'* by applying a set of atomic transformations on *S*.
2. *Mapping*: two schemas *S* and *S'* are equivalent if their instances have a one-to-one correspondence.
3. *Behavioural*: two schemas *S* and *S'* are equivalent if for any instance of *S*, there is a corresponding instance of *S'* that produces the same results with *S* for any query *Q*.

Other studies define schema equivalence based on *information capacity* [Mil93]. If one schema S_1 has more information capacity that another schema S_2 then every instance of the S_2 can be mapped to one of S_1 without any loss of information.

In [PM98] two notions of equivalence are defined, *unconditional equivalence* and *conditional equivalence*, which subsume transformational, mapping and behavioural equivalence mentioned above [MPinf98].

2.4 Schema Integration Process

Several approaches have been proposed in the literature for the schema integration process, all of them dividing the process into several steps/phases. In [Bat86], four steps are identified:

1. *Preintegration*. In this step an analysis on the schemas is carried out in order to decide which schemas to integrate and the order of their integration.
2. *Schema comparison*. The schemas are analysed and compared in order to determine the correspondences among the concepts and detect possible conflicts.
3. *Schema conforming*. The detected conflicts are resolved so that merging of various schemas is possible.
4. *Schema merging* and *restructuring*. Schemas are superimposed and an intermediate schema is produced. The intermediate schema is analysed and restructured in order to produce the final schema.

In [Yan97], 3 steps in the integration process are identified:

1. *Schema translation*. If the component databases are heterogeneous then the schemas are translated into a common intermediate representation. Then there is the task of mapping from one schema to another, which will identify equivalences between the concepts of the source and target model.
2. *Homogenisation*. Semantic heterogeneity and structural heterogeneity problems are resolved.

3. *Integration*. This step involves the merging and the restructuring of intermediate schemas. All schemas are merged into a single database schema, which is restructured to create the final schema.

It is obvious that these two approaches are very similar. Schema translation is similar to preintegration, and homogenisation corresponds to schema comparison and schema conforming. The processes of schema merging/restructuring and integration are exactly the same.

Other studies divide schema integration into even less steps, however all approaches are basically similar. Initially commonalities and discrepancies among input schemas have to be determined and then specific conflicts have to be detected and resolved. The role of the designer is very important in this process, because in order to resolve a conflict, the comprehension of the semantic relationships among the concepts involved in the conflict is necessary.

2.5 Schema Integration Methodology adopted

2.5.1 Hypergraph Data Model

In [PM98] a general notion of a schema as a hypergraph is defined. A schema in the hypergraph data model (HDM) is a triple $\langle Nodes, Edges, Constraints \rangle$. A *query q* over a schema $S = \langle Nodes, Edges, Constraints \rangle$ is an expression whose variables are members of $Nodes \cup Edges$.

Constraints is a set of boolean-valued queries over *S*. *Nodes* are uniquely identified by their names. *Edges* and *Constraints* have an optional name associated with them. Edges have a *scheme* that uniquely identifies them. This scheme can link any number of both nodes and other edges and therefore a labelled, directed, nested hypergraph is defined. For example, an edge can be described in HDM notation as: $<EdgeName, Construct_1, Construct_2, \ldots, Construct_n>$. Each $Construct_i$ can be either a node or an edge that might also link other nodes and edges. A node is described uniquely with its name: $<NodeName>$.

An *instance I* of a schema $S = \langle Nodes, Edges, Constraints \rangle$ is a set of sets satisfying the following:

1. each construct $c \in Nodes \cup Edges$ has an extent, denoted by $Ext_{S,I}(c)$, that can be derived from *I*,
2. conversely, each set in *I* can be derived from the set of extents $\{Ext_{S,I}(c) \mid c \in Nodes \cup Edges\}$,

3. for each $e \in Edge$, $Ext_{S,I}(e)$ contains only values that appear within the extents of the constructs linked by e (domain integrity),
4. the value of every constraint $c \in Constraints$ is true, the value of a query q being given by $q[c_1 / Ext_{S,I}(c_1), \ldots, c_n / Ext_{S,I}(c_n)]$ where $c_1,\ldots, c_n$ are the constructs in $Nodes \cup Edges$.

We call the function $Ext_{S,I}$ an *extension mapping*. A *model* is a triple $\langle S, I, Ext_{S,I}\rangle$. Two schemas are *equivalent* if they have the same set of instances. Given a condition f, a schema S *conditionally subsumes* a schema S' w.r.t. f if any instance of S' satisfying f is also an instance of S. Two schemas S and S' are conditionally equivalent w.r.t f if they each *conditionally subsume* each other w.r.t. f.

2.5.2 The Methodology

The schema integration process doesn't follow any steps except the preintegration stage, where the individual models are transformed into the hypergraph model described above. Essentially, a series of transformations are performed on the input schemas. Each transformation is a function that when applied to a model returns a new model. Each transformation has a *proviso* associated with it, which states when the transformation is successful. Unsuccessful transformations return an "*undefined*" model.

The transformations, as defined in [MPtec98], are:

1. *renameNode* $\langle fromName, toName\rangle$ renames a node. Proviso: *toName* is not already the name of some node.
2. *renameEdge* $\langle\langle fromName, c_1,\ldots, c_m\rangle, toName\rangle$ renames an edge. Proviso: *toName* is not already the name of some edge.
3. *addConstraint c* adds a new constraint *c*. Proviso: *c* evaluates to true.
4. *delConstraint c* deletes a constraint. Proviso: *c* exists.
5. *addNode* $\langle name, q\rangle$ adds a node named *name* whose extent is given by the value of the query *q*. Proviso: a node of that name does not already exist.
6. *delNode* $\langle name, q\rangle$ deletes a node. Here, *q* is a query that states how the extent of the deleted node could be recovered from the extents of the remaining schema constructs (thus, not violating property (ii) of an instance). Proviso: the node exists and participates in no edges.
7. *addEdge* $\langle\langle name, c_1,\ldots, c_m\rangle, q\rangle$ adds a new edge between a sequence of existing schema constructs $c_1,\ldots, c_m$. The extent of the edge is given by the

value of the query *q*. Proviso: the edge does not already exist, $c_1, \ldots, c_m$ exist, and *q* satisfies the appropriate domain constraints.

8. *delEdge* $\langle\langle name, c_1, \ldots, c_m \rangle, q \rangle$ deletes an edge. *q* states how the extent of the deleted edge could be recovered from the extents of the remaining schema constructs. Proviso: the edge exists and participates in no edges.

For each one of the primitive transformations a new transformation is defined that takes as an extra argument a condition which must be satisfied in order for the transformation to be successful. Such transformations are called *knowledge-based* transformations.

The execution of a primitive transformation *op* can be described as: *op arg c*, where *arg* are the primitive transformation's arguments and *c* is the condition that must be satisfied in a knowledge-based transformation.

A composite transformation is defined as a sequence $n \geq 1$ of primitive transformations: $op_1\ arg_1\ c_1; op_2\ arg_2\ c_2; \ldots; op_n\ arg_n\ c_n$ where the conditions c_i are optional.

2.5.3 Mapping High-Level Data Models to HDM

[MP99] explains how high-level data models can be defined in terms of the low-level HDM and how this mapping enables the automatic transformation of queries between schemas defined using different data models.

According to [MP99], the constructs of any semantic modelling language *M* may be classified as either *extensional constructs*, or *constraint constructs*, or both. Extensional constructs represent sets of data values from some domain. Each such construct in *M* must be built using the extensional constructs of the HDM, i.e. nodes and edges. There are three kinds of extensional constructs:

1. *nodal* constructs may be present in a model independent of any other constructs. For example, ER entities are nodal constructs. The scheme of each construct uniquely identifies the construct in *M*. A nodal construct maps into a node in the HDM.
2. *linking* constructs can only exist in a model when certain other nodal construct exists. ER relationships are linking constructs. The extent of a linking construct is a subset of the cartesian product of the extents of these nodal constructs. Linking constructs map into edges in the HDM.
3. *nodal-linking* constructs are nodal constructs that can only exist when certain other nodal constructs exist, and that are linked to these constructs. For

example, ER attributes are nodal-linking constructs. Nodal-linking constructs map in to a combination of a node and an edge in the HDM.

Constraint constructs represent restrictions on the extents of the extensional constructs of *M*. e.g. cardinality constraints in ER relationships and attributes. Constraints are directly supported by the HDM.

2.6 Previous Work

[BT01] describes how the methodology described in the previous section has been implemented. Mappings from high-level data models to the low-level HDM are successfully defined, and schemas are represented not as a triple of nodes, edges, constraints but as a list of *objects* with a specific construct type. An object can have any possible construct type since there is a mapping from the high-level constructs to the low-level ones. The notion of the construct *scheme*, defined in [PM98], still exists, in two different levels: (1) when a construct is defined, the types that can be included in its scheme must be defined, and (2) when an object of the specific construct is created, it must have an associated scheme containing objects that their types comply with the types of the object's construct scheme.

The work described in [BT01] is divided into two modules. One module is used to define semantic modelling languages, their constructs and their mapping to the HDM, and the other module is used to represent schemas corresponding to existing databases and primitive transformations on those schemas, i.e. one module is concerned with the metadata of modelling languages, and the other with modelling-language-specific descriptions of databases. All the model definitions and schema representations are stored in a relational database in order to provide a uniform and simple method of defining new models and new schemas or updating the existing ones. Model definitions are stored in the Model Definitions Repository and schema representations are stored in the Schemas and Transformations Repository.

2.6.1 Model Definitions Repository

The Model Definitions Repository (MDR) provides a way of defining terms of high-level data models (ER, UML, relational) in terms of the low-level HDM. This enables the automatic translation between the two levels and supplies interoperability between different models. The repository stores descriptions of data models, i.e. their constructs along with information needed to create instances of these constructs, and the mapping from each model to the HDM.

Next, the essential relations of the MDR are presented. Primary keys are underlined.

The `model` relation stores the existing modelling languages:

model	
MID	integer
name	varchar

Table 2.1: MDR model relation

The `construct` relation defines the constructs of each data model:

construct	
MID	integer
CID	integer
name	varchar
class	integer
isRoot	integer

Table 2.2: MDR construct relation

The `CID` field is the construct's identifier and the `MID` field defines the model this construct belongs to. The `class` field maps the construct to its corresponding one in the HDM.

The `scheme` relation describes what are the necessary arguments to create an instance of each construct, i.e. create the object's scheme:

constructScheme	
CID	integer
schemePos	integer
type	integer
argCID	integer
isKey	integer
HDMType	integer
lowerBound	integer
upperBound	integer

Table 2.3: MDR constructScheme relation

Table 2.4 shows how an elementary ER model can be defined in the MDR. This elementary ER model contains only entities and attributes. Since the ER attribute is a link-nodal object, its scheme consists of the construct the attribute is attached to and the attribute's name, while for an ER entity only the entity's name in necessary in its scheme.

model	
MID	name
1	ER

construct				
MID	CID	name	class	isRoot
1	1	entity	nodal	TRUE
1	2	attribute	link-nodal	TRUE

constructScheme							
CID	schemePos	type	argCid	isKey	HDMType	lowerBound	upperBound
1	1	NAME		TRUE	N1	1	1
2	1	NAME		TRUE	N2	1	1
2	2	CONSTRUCT	1	TRUE	P1	1	1

Table 2.4: ER model represented in the MDR

2.6.2 Schemas and Transformations Repository

The Schemas and Transformations Repository (STR) stores information about schemas and transformation pathways between them. The schema representation adheres to the MDR. Each schema is a list of objects and each object is an instance of a particular construct. Thus, each object must have an associated object scheme in accordance to the construct's scheme defined in the MDR. Additionally, the STR stores primitive transformations on schemas, as they have been defined in [PM98]. These transformations create paths between the existing schemas. Next, the more important relations of the STR are presented. For a more detailed description of the STR refer to [BT01].

The `schema` relation stores information on existing schemas: the identifier of each schema (`SID`) and details about the nature of the schema.

schema	
SID	integer
name	varchar
extSchema	integer
extDB	integer
dataSource	integer
isCached	integer

Table 2.5: STR schema relation

The `object` relation stores information about each object in a schema:

object	
SID	integer
CID	integer
OBID	integer
value	varchar
isTemp	integer

Table 2.6: STR object relation

The `OBID` field in the `object` relation is the object's identifier, and `SID`, `CID` specify the schema this object belongs to and its construct type.

The `scheme` relation describes the arguments of each construct instance:

scheme	
OBID	integer
schemePos	integer
SOBID	integer

Table 2.7: STR scheme relation

For example, an object that's an instance of an ER attribute would contain in its scheme the identifier (`SOBID`) of the object it is attached to.

Finally, the `transformation` relation represents primitive transformation on schemas. This relation also serves to identify pathways from one schema to another:

transformation	
action	varchar
fromObj	integer
toObj	integer
fromSchema	integer
toSchema	integer
function	varchar
constraint	varchar

Table 2.8: STR transformation relation

Table 2.9 shows how the ER schema of Figure 1.1(a) can be represented in the STR. The scheme of the ER attribute `dept` points to the ER entity `person` and itself. This table also shows how the primitive transformation of adding the ER entity `department` is represented in the STR.

schema					
SID	name	extSchema	extDB	dataSource	isCached
1	schema1	1	1	1	0
2	intermediate	0	0	0	0

object				
SID	CID	OBID	value	isTemp
1	1	1	person	0
1	2	2	dept	0
2	1	3	department	0

scheme		
OBID	schemePos	SOBID
1	1	1
2	1	2
2	2	1
3	1	3

transformation						
action	fromObj	toObj	fromSchema	toSchema	function	constraint
add	-1	3	1	2		

Table 2.9: STR populated

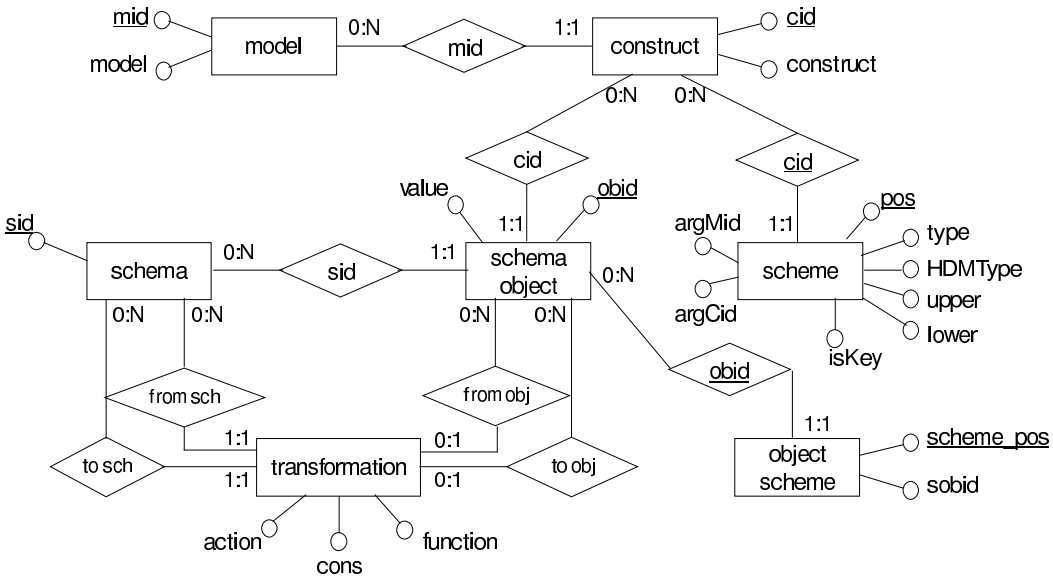

Figure 2.3: MDR and STR repositories schema

Chapter 3 - Architecture

The example described in Section 1.1 illustrates in an abstract form how template transformations are implemented but doesn't justify their existence. This chapter shows the applicability and necessity of template transformations and explains the framework developed in this project to support the template transformation definition process.

The final section presents the issues that arise when a graphical representation language is designed and gives an overview of the language that was developed in this project.

3.1 Template Transformations

Figure 3.1 shows two equivalent ER schemas. A composite transformation is performed on the schema in Figure 3.1(a) and produces the schema in Figure 3.1(b). The transformation removes the existing entity's attribute and creates a generalization with two sub-entities.

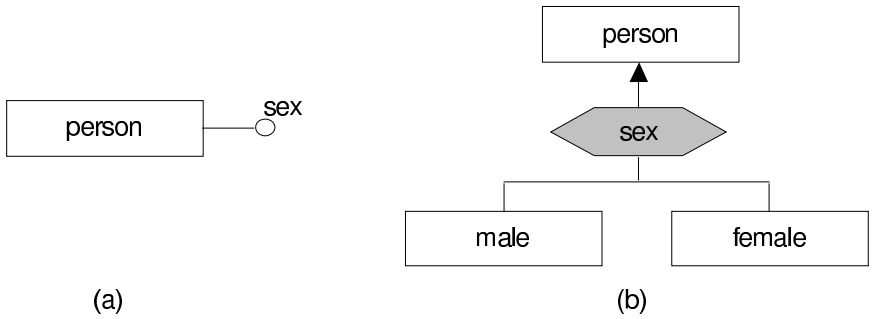

Figure 3.1: Composite transformation example with two sub-entities

Figure 3.2 shows two other equivalent schemas. The algorithm of the composite transformation that relates the two schemas is exactly the same as the one in Figure 3.1: again the attribute is removed and a generalization is added. The only difference between the two transformations is that the 2nd example creates three sub-entities therefore three `add` primitive transformations must be executed, instead of two in Figure 3.1.

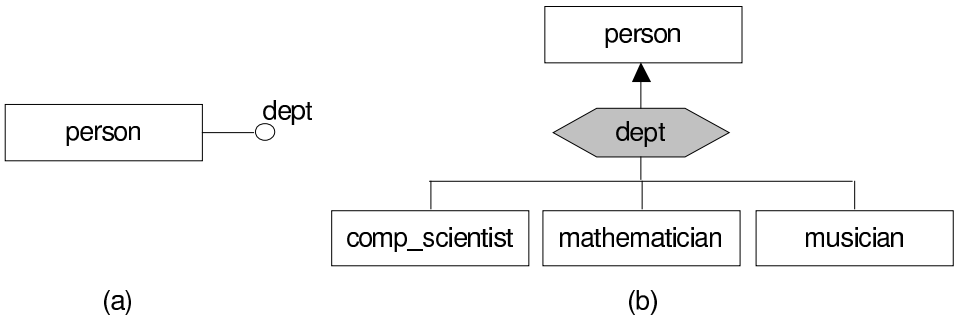

Figure 3.2: Composite transformation example with three sub-entities

This similarity between the two transformations exhibits the need for *schema-independent* definitions of composite transformations that will enable their execution on different schemas. In fact, it shows that composite transformations must describe algorithms and not executions on certain construct instances. This is identified in [PM98] and the above composite transformation is defined using *variables*:

```
add_E<e_1, {e(s) | s ∈ <Null, e, a> ∧ a(s) = v_1}>;
. . .
add_E<e_n, {e(s) | s ∈ <Null, e, a> ∧ a(s) = v_n}>;
add_G<total, e, e_1, . . ., e_n>;
del_E<e, a>  (a=v_1, . . ., v_n)
```

Figure 3.3: Composite transformation with variables

The notation used in Figure 3.3 is more high-level than the one described in Chapter 2, which uses nodes and edges. The mapping from the one language to the other is defined in [PM98]. An add_E transformation adds an ER entity, an add_G adds a generalization and a del_A deletes an attribute. The first add_E statements in Figure 3.3 add the entities e_1, …,e_N which are going to be the sub-entities of the generalization. The add_G statement creates a total generalization with scheme the sub-entities e_1, …,e_N and the pre-existing "general" entity `e`. Finally, del_A deletes the existing attribute `a`.

The use of variables enables the execution of the composite transformation of Figure 3.3 on both schemas presented in Figures 3.1(a) and 3.2(a). The transformation allows the creation of an undefined but finite number of sub-entities. In both figures, variable `e` maps to the entity `person`. In Figure 3.1(a), `N` is equal to 2: e_1 maps to `male` and e_2 maps to `female`. The attribute `a` is `sex`. In Figure 3.2(a),

N is equal to 3: e_1 maps to comp_scientist, e_2 maps to mathematician and e_3 maps to musician. The attribute a is dept.

As one understands, the definition of schema-independent composite transformations using variables is essential in order to extend the transformations' applicability and usefulness. In this project, such transformations are named *templates*. The next section describes the framework developed in this project that enables the definition of template transformations.

3.2 Template Transformations Framework

The aim of this project has been the definition of template transformations in a way that is completely independent of data models and programming languages. This is accomplished by extending the previous work of [BT01] and defining a template transformation language independent of any existing language implementation. The structure of relational databases is utilized to store template transformation definitions. In this way, template transformations can easily be created, updated and validated.

3.2.1 Template Transformation Language

The syntax of the template transformation language has been designed according to the descriptions of [PM98] and the implementation of [BT01]. The fact that a composite transformation is defined in [PM98] as a sequence of primitive transformations, suggests that the template transformation language should be composed of elementary statements and that a template definition should be a sequence of such statements.

According to the discussion in Section 3.1, a template transformation definition must support variables. This can be accomplished if the statements of the template transformation language are parameterised, i.e. arguments can be passed to them. Therefore, the abstract form of a template transformation statement is:

StatementName(arg_1, arg_2, …, arg_n)

The Template Transformation Syntax Repository (TTSR) defines the syntax that can be used in template transformation definitions. Essentially, this repository stores the template transformation statements and describes their semantics. In addition, the number and type of arguments that can be applied to each statement is defined.

The statements relation stores information about statements: the identifier (st_id) and name of each statement.

statements	
st_id	integer
name	varchar

Table 3.1: TTSR statements relation

The `statement definition` relation defines the arguments of each statement. It describes the type of each argument and determines its position in the statement's argument sequence. This relation can be considered to store the statements' fingerprints, e.g.:

`StatementName(ArgType`$_1$`, ArgType`$_2$`, …, ArgType`$_n$`)`

statement definition	
st_id	integer
arg_pos	integer
table_name	varchar
field_name	varchar
description	varchar

Table 3.2: TTSR statement definition relation

The `arg_pos` attribute in the `statement definition` relation defines the position of the argument in the argument sequence of the statement with identifier `st_id`. The type of the argument is defined by the `table_name`, `field_name` and `description` attributes. The `table_name` points to a relation and the `field_name` points to an attribute in the specified relation. The `description` attribute explains the semantics of the argument. In most cases, the specification of the `table_name` and `field_name` suffices for the description of an argument. In more complicated cases, the `description` attribute is also specified.

The statements' fingerprints are essential in a template transformation's definition and a statement's execution. When a statement is employed in a template definition, the correct number of arguments with the correct types must be passed to the statement. Statement argument definitions are also important in the developing of the module that will execute a template transformation, because they explain how a statement should be executed and where the data that are produced during the execution should be stored.

3.2.2 Template Transformation Definitions

Template transformation definitions are stored in the Template Transformation Repository (TTR). Each template transformation consists of a sequence of statements (defined in the TTSR) and has a sequence of arguments. This existence of arguments is something that is implied in the composite transformation

definition of Figure 3.3. There, although variables are manipulated, it is not clarified how these variables are instantiated when the composite transformation is performed on a specific schema.

```
/* description of template transformation */
TemplateTransformation(ArgType1 arg1, ArgType2 arg2, …,
ArgTypen argn){
      Statement1(arg11, arg12, …, arg1n);
      …
      Statementn(argn1, argn2, …, argnn);
}
```

Figure 3.4: The abstract form of a template transformation

A template transformation can be considered to be an implemented method in a procedural programming language, as shown in Figure 3.4. Figure 3.4 illustrates all the data that are stored in the TTR repository for each template transformation: the name, the transformation's description, its fingerprint and finally, the statements that constitute the transformation.

The TTR uses a database relation, named `composite transformations`, to store the identifier (`trans_id`), name and description of each template transformation.

composite transformations	
trans_id	integer
name	varchar
description	varchar
preserves_eq	integer

Table 3.3: TTR composite transformations relation

The `preserves_eq` attribute defines whether the template transformation preserves schema equivalence, i.e. if the initial and final schemas before and after the transformation's execution are equivalent. The `description` attribute stores a summary of what the transformation accomplishes.

The `composite transformation definition` relation, equivalent to TTSR's `statement definition` relation, defines the transformations' arguments, i.e. stores the template transformations' fingerprints.

composite transformation definition	
trans_id	integer
arg_pos	integer
table_name	varchar
field_name	varchar
description	varchar

Table 3.4: TTR composite transformation definition relation

The `arguments` relation (Table 3.5) defines argument sequences. Attribute `arg_id` identifies each sequence and attribute `arg_pos` identifies each argument's position in the sequence. This is a fundamental relation since its purpose is to store the arguments for each statement in a template transformation program; the transformation's statements are going to be executed according to the data stored in this relation. Therefore, these arguments determine the transformation's behaviour.

arguments	
arg_id	integer
arg_pos	integer
arg_id_ref	integer
arg_pos_ref	integer
value	varchar

Table 3.5: TTR arguments relation

The attributes `arg_id_ref`, `arg_pos_ref` define whether an argument refers to another one or has a specific value `value`. By having arguments referring to other ones, assignment of values is successfully achieved and variables are supported. The `arguments` relation can describe assignment between variables, e.g. $x = y$, where y has been previously assigned a value.

The `composite transformation program` relation stores the sequence of statements, i.e. the program, which implements a composite transformation's algorithm.

composite transformation program	
trans_id	integer
pc	integer
st_id	integer
arg_id	integer

Table 3.6: TTR composite transformation program relation

The program counter (`pc`) identifies when the statement with `st_id` should be executed and the `arg_id` attribute defines the statement's arguments.

3.2.3 Template Transformation Execution

This project discriminates between compile-time programs of template transformations and run-time executions of these transformations. The TTR stores the compile-time programs and the Template Execution Repository (TER) is used to execute template transformations on specific database schemas.

The TER behaves as the run-time memory in a program's execution, storing all the run-time values and arguments. A template transformation can be executed by invoking it on an existing database schema using the appropriate arguments. The execution of a template transformation will instantiate all the variables/arguments used in the transformation's program and a schema-specific composite transformation will be performed.

The `arguments` relation is used to store all the intermediate results during a template transformation execution. The run-time values that correspond to compile-time arguments/variables are stored in this relation.

arguments	
arg_id	integer
arg_pos	integer
value	varchar

Table 3.7: TER arguments relation

The `composite transformation invocation` relation is used to start the execution of a template transformation:

composite transformation invocation	
trans_id	integer
arg_id	integer

Table 3.8: TER composite transformation invocation relation

A template transformation invocation is similar to a procedure invocation:
`TemplateTransformation(`val_1`, `val_2`, …, `val_n`)`

Finally, the `composite transformation execution` relation stores a transformation's run-time execution. It's exactly the same as TTR's `composite transformation program` relation, but the arguments that are used in this case are all run-time arguments and all have specific values.

composite transformation execution	
trans_id	integer
pc	integer
st_id	integer
arg_id	integer

Table 3.9: TER composite transformation execution relation

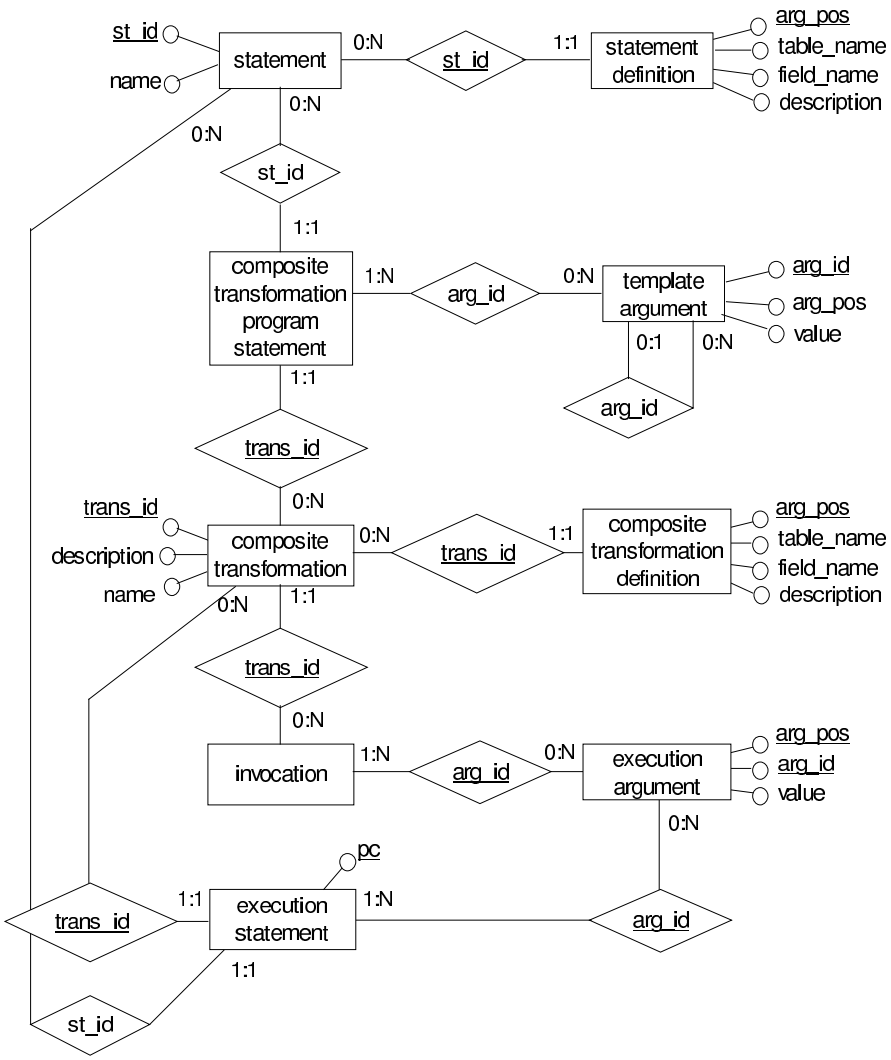

Figure 3.5: TTSR, TTR and TER repositories schema

3.3 Graphical Representation Language

This project has also been examining how constructs can be represented graphically. The project's objective has been the definition of a graphical

representation language that is data-model independent. This language should be able to represent any model's constructs therefore it should consist of elementary statements that create some primitive shapes and statements that can be used to manipulate these primitive shapes to construct more complex ones. The supported shapes should be parameterised in order to make them more versatile and simplify their manipulation.

One aspect in the designing of the graphical representation language has been the power of its statements. A low-level language that supports a single, primitive shape, e.g. point or line, can be defined and this would allow the creation of all other shapes by manipulating this single one, e.g. a rectangle can be represented as 4 lines. Although this low-level language has the ability to represent any possible shape, defining a construct's graphical representation using this language would be complicated since the definition would be extremely large. Thus, a more high-level language is needed.

The shapes that must be supported by this high-level language are the ones that are already being used in existing data models. Shapes that are widely used are rectangles, lines and circles. A rhombus is used in the ER model to represent relationships. In the extended ER model, the generalization construct is represented with a shape similar to a rhombus. Based to that, the language that has been designed supports four primitive shapes: a box, a line, an arc and a rhombus.

The basic characteristic of the designed language is that it is implementation-oriented. It uses as statements a set of methods that have been previously implemented in the Java programming language [JAVA]. Therefore, writing a graphical representation definition is similar to coding a program using an existing API. The reason why the developed language is implementation-oriented is because in order to represent constructs graphically, knowledge about these constructs and their relationship with other ones is necessary, e.g. drawing an ER attribute demands knowing the construct this attribute is attached to, in order to draw the line that will connect the two constructs. This knowledge can be obtained only by using implemented methods that supply it. Although the developed language depends on a specific implementation it can be easily extended and can support any type of modelling construct.

A construct's graphical representation definition is a sequence of statements/method calls. This definition must be applicable to any instance of the specific construct. For example, the graphical representation of an ER entity must be

independent of the entity's name. An abstract description of the ER attribute's graphical representation definition is the following:

drawLine;
drawArc;

The Schema Modelling Repository (SMR) stores the constructs' graphical representation definitions. Basically, SMR extends the MDR repository [BT01] by defining the graphical representation of each construct.

The SMR `methods` relation defines the Java implemented methods that can be used as statements in the definition of a construct's graphical representation. Each method has a name and an associated identifier.

methods	
meth_id	integer
class	varchar
name	varchar

Table 3.10: SMR methods relation

The `class` attribute in the `methods` relation defines the Java class file this method appears in.

The `method arguments` relation stores the class types of the methods' arguments.

method arguments	
meth_id	integer
pos	integer
type	varchar

Table 3.11: SMR method arguments relation

Each construct's modelling definition is stored in the `construct modeling` relation.

construct modeling	
CID	integer
pc	integer
method	varchar

Table 3.12: SMR construct modeling relation

The program counter (`pc`) defines when the specific method should be invoked.

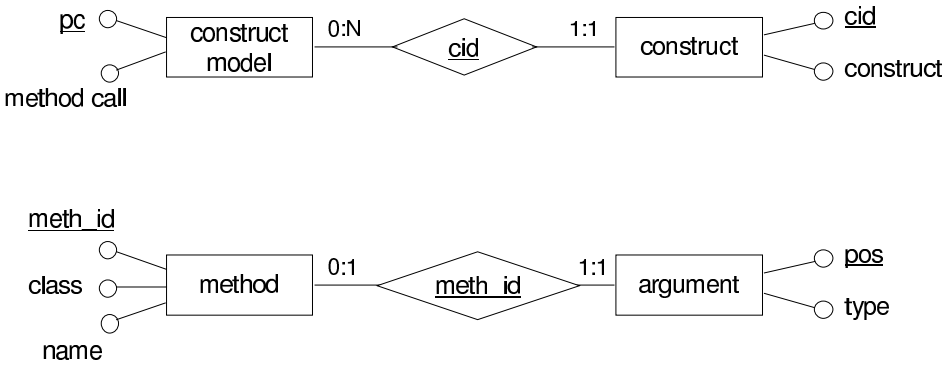

Figure 3.6: SMR repository schema

Chapter 4 - Template Transformation Language

This chapter explains in detail the language that can be used to define a template transformation. Section 3.2 has explained that the template transformation language must support elementary statements. This chapter identifies these elementary statements and explains how they are stored in the Template Transformation Syntax Repository.

4.1 Design

Since composite transformations are sequences of primitive ones [PM98], the developed template transformation language must support a statement that performs a primitive transformation. Since primitive transformations are represented in [BT01] as operations that manipulate existing objects, a statement that creates these objects is required. Additionally, an iteration construct is vital because iterations are extensively used in the composite transformations described in [PM98]. Such a statement will be able to iterate on a set of values and execute a block of statements for each value.

The next section describes in detail these statements. An abstract description of each statement as a parameterised procedure is provided, in addition to the statement's actual representation in the TTSR.

4.2 Statement Definition

The following table shows how the language's statements are defined in the TTSR. The `new_object` statement creates an object, the `primitive_transformation` statement executes a primitive transformation and the `foreach` statement is the iteration construct.

statements	
st_id	name
1	new_object
2	primitive_transformation
3	foreach
4	extend_generalization_scheme

Table 4.1: Template Transformation Language's statements

A statement that proved to be necessary during this project's implementation phase is the `extend_generalization_scheme` statement. This statement extends an ER generalization object's scheme and originated from the fact that an ER

generalization's scheme hasn't got a constant size but its size depends on the number of the generalization's sub-entities.

The fingerprints of the statements illustrated in Table 4.1 have to be defined. The arguments of the `new_object` statement must specify all the properties that characterize an object. The object's properties are defined in the STR `object` (Table 2.6) and `scheme` relations (Table 2.7). According to these relations, the properties of a schema object are: the schema this object belongs to, the object's construct type, the object's value and a property that defines whether this object is temporary or not. The object's scheme is also important and a generic way must be used to represent it, since its size isn't constant but depends on the object's construct type. Lists can be employed to represent schemes. Therefore, the `new_object` statement's fingerprint is:

```
void new_object(Schema belongsTo, ConstructType type,
                Object newObject, String value, boolean isTemp,
                List scheme);
```

A `newObject` argument must be passed by reference to the statement. The object created by the statement will be stored in this `newObject` argument.

The `new_object`'s fingerprint is stored in the TTSR in the following form:

statement definition				
st_id	arg_pos	table_name	field_name	description
1	1	object	SID	
1	2	object	CID	
1	3	object	OBID	
1	4	object	value	
1	5	object	isTemp	
1	6	arguments	arg_id	scheme

Table 4.2: NewObject argument definition

The schema that the new object belongs to is represented using a schema identifier (`SID`) and the object's construct type is defined by the construct identifier (`CID`). The created object is represented with an object identifier (`OBID`). The `value` and `isTemp` arguments have one-to-one correspondences. Finally, the scheme is represented using an argument sequence identifier (`arg_id`). Argument sequences can represent lists since, like lists, they don't have a specific size.

The arguments of the `primitive_transformation` statement are identified in the same way. The arguments of this statement are derived from the STR `transformation` relation (Table 2.8). First, the action that is going to be performed by the primitive transformation must be specified. This argument will define if the

primitive transformation will execute an add/delete/rename operation. Then, the object that the transformation is performed upon and the object that derives from this transformation, along with the schemas before and after the transformation must be specified. Finally, the function associated with this transformation and the constraint that might exist for this transformation must be identified. The function argument can be used to identify the extent of the added/deleted object and the constraint can make this primitive transformation a knowledge-based one [PM98]. The `primitive_transformation` statement's fingerprint is:

```
void primitive_transformation(String action, Object fromObj,
                Object toObj, Schema fromSchema, Schema toSchema,
                String function, String constrain);
```

This fingerprint is stored in the TTSR in the following form:

statement definition				
st_id	arg_pos	table_name	field_name	description
2	1	transformation	action	
2	2	transformation	fromObj	
2	3	transformation	toObj	
2	4	transformation	fromSchema	
2	5	transformation	toSchema	
2	6	transformation	function	
2	7	transformation	constraint	

Table 4.3: Primitive Transformation argument definition

A `foreach` statement is similar to a `for` loop construct existing in most programming languages. It iterates on a set of values and a block of statements is executed in each iteration. An abstract description of the `foreach` statement is the following one:

```
foreach e in Set{
        Statement1();
        …
        Statementn()
}
```

Figure 4.1: ForEach statement's abstract form

The block of statements can only contain one primitive transformation.

Based on this description, one of the `foreach` statement's arguments must be the set of values to iterate. Since the size of this set is undefined, this argument is represented using a list. Another essential argument is the block of statements that is going to be executed in each iteration. This block of statements is similar to a method call: the `foreach` statement can be considered to invoke a particular method

in each iteration instead of executing a series of statements. The initial and final schemas before and after the statement's execution must also be specified. The final schema is passed by reference to the statement and can be accessed after the statement has terminated to identify the resulting schema. Finally, an argument that will identify the objects that have been added/deleted/renamed during the `foreach` statement's execution must exist. This last argument derived in the language's implementation phase when specific composite transformations had to be developed. Finally, the `foreach` statement's fingerprint is:

```
void foreach(List values, Method blockOfStatements,
             Schema initial, Schema final, List objects);
```

The block of statements is represented using a `Method` type argument. Java [JAVA] provides such a type, while in other programming languages methods can be passed as arguments using pointers.

The `foreach` statement's fingerprint is stored in the TTSR in the following form:

statement definition				
st_id	arg_pos	table_name	field_name	description
3	1	arguments	arg_id	values iterated
3	2	composite transformation program	trans_id	block of statements
3	3	object	value	current value
3	4	schema	SID	initial schema
3	5	schema	SID	final schema
3	6	arguments	arg_id	created objects

Table 4.4: ForEach argument definition

The list that is passed to the `foreach` statement as the first argument is represented using an argument sequence identifier. The argument defining the block of statements to be executed at each iteration is specified using a composite transformation identifier, which is basically the same as saying that the block of statements is a `Method`. The initial and final schemas are represented using schema identifiers and the list of objects that are added/deleted/renamed during the statement's execution are mapped to an argument sequence, since the number of these objects is not known at compile time. Table 4.4 shows that the `foreach` statement has another argument which specifies the current value that is being iterated. This argument is used internally by the statement and is necessary in the statement's execution.

The `extend_generalization_scheme` statement extends a generalization's object scheme by adding to it the generalization's sub-entities. The arguments necessary for the execution of this statement is the list of sub-entities to be added and the generalization object's scheme. Also, another argument is needed that will specify from which point the scheme should be extended. This argument is redundant in the case of generalization objects but its existence widens the use of this statement and makes it applicable to various other objects, like relationships. The statement's fingerprint is:

```
void extend_generalization_scheme(List values,
                                  List scheme, int start);
```

In the TTSR this statement is stored as follows:

statement definition				
st_id	arg_pos	table_name	field_name	description
4	1	arguments	arg_id	sub-entities
4	2	arguments	arg_id	scheme id
4	3	arguments	arg_pos	starting pos

Table 4.5: ExtendGeneralization Scheme argument definition

4.3 General Considerations

The previous section mentioned that arguments can be passed by reference to statements. This is the case when abstract descriptions of statements are considered, i.e. when statements are considered to be procedures. In the template transformation language this characteristic of passing arguments by reference is implemented by omitting positions from argument sequences. The missing positions will be restored during the statement's execution.

For example, a `new_object` statement can be invoked by passing the `newObject` argument by reference:

```
Object newObject;
new_object(schema,"ERentity",newObject&,"person",false,scheme);
```

In the template transformation language, the same invocation can be performed by omitting the `OBID` argument from the argument sequence. Table 4.6 shows the argument sequence that can be applied to the `new_object` statement. The `OBID` argument (`arg_pos=3`) is omitted. A value for this argument will be created at run-time and placed in the missing position.

arguments		
arg_id	arg_pos	value
10	1	schema
10	2	ERentity
10	4	person
10	5	FALSE
10	6	scheme

Table 4.6: Passing arguments by reference in the transformation language

Chapter 5 - Template Transformation Definition

This chapter explains how template transformations are constructed. First, the relations of the Template Transformation Repository presented in Section 3.2.2, are extensively described. Then, the statement invocation process is illustrated since a template transformation is essentially a series of statement invocations. Finally, the template transformation definition process is explained with the use of two examples.

5.1 Template Transformation Repository

The Template Transformation Repository divides template transformations into two categories based on their characteristic to preserve or not schema equivalence. The `preserves_eq` attribute in the `composite transformations` relation (Table 3.3), determines whether one transformation preserves schema equivalence, or not; a value of zero (false) means that the transformation doesn't preserve schema equivalence, and a value of one (true) means it preserves equivalence. Transformations that don't preserve schema equivalence are basically the ones that are used in `foreach` statements as blocks to be executed in each iteration.

Template transformations that don't preserve schema equivalence don't have arguments and therefore they can't be invoked directly. Template transformations that preserve schema equivalence have arguments and their argument definitions are stored in the `composite transformation definition` relation (Table 3.4) in exactly the same way with statements. The identification of these arguments is a more complicated process than in the case of statements, since each template transformation represents a different algorithm and for each algorithm there are different information needs. This argument identification process is exhibited in Sections 5.3 and 5.4 where two template transformations examples are illustrated.

The TTR `arguments` relation (Table 3.5) stores the compile-time arguments applied to each template transformation's statement. Arguments cannot exist autonomously but must be part of an argument sequence. Even if a statement requires only one argument, an argument sequence has to be defined, which in that case will simply contain only one argument. Positions in an argument sequence might be omitted if the values for the corresponding arguments are created at run-time as it was explained in the previous chapter.

If the value of an argument is known at compile-time, then it can be directly stored in the relation's `value` attribute. The `value` attribute is of type `varchar`, since values of all other types can be represented by character strings. An argument whose value isn't known at compile-time can refer to the argument that will hold the desired value when the statement is executed. Attributes `arg_id_ref` and `arg_pos_ref` define which argument in which argument sequence is referred. Missing positions in argument sequences can also be referred, since the arguments for these positions will be created at run-time and a value will be assigned to them. Finally, arguments can also refer to a template transformation's arguments. The only difference is that in this case the `arg_id_ref` attribute should be less or equal to zero. The referred argument is at the `arg_pos_ref` position in the template transformation, which is currently being executed, argument sequence.

The `argument` relation can also be used when a list of values has to be represented. The `arg_id` uniquely identifies each list, which is sorted according to `arg_pos`. For example, an object's scheme can be stored in the `arguments` relation as a list, with elements the identifiers of the objects referred by the scheme. The set of values iterated by a `foreach` statement can also be stored in this way.

A template transformation's statement sequence is stored in the `composite transformation program` relation (Table 3.6). The `trans_id` attribute identifies the template transformation, the `pc` attribute defines the transformation's program counter and the `st_id` identifies the statement that is going to be invoked at the specific program counter. Finally, the `arg_id` attribute shows the argument sequence that is going to be applied to the aforementioned statement. Before giving an example of a template transformation's program definition, the following section presents how individual statements can be invoked.

5.2 Statement Invocation

This section explains the behaviour of primitive statements. This behaviour depends on the statements' arguments because arguments are the only input to statements. The detailed examination of how each statement's arguments must be instantiated is fundamental, since this instantiation process must be performed before the statement is invoked.

In the case of the `new_object` statement (Section 4.2 and Table 4.2 define the statement's fingerprint), the arguments that are mandatory and have to be passed by value to the statement are: the object's construct type, its value, an argument specifying whether the object is temporary or not and the object's scheme. The

`newObject` argument has to be passed by reference and the object's schema can either be passed by reference or by value, depending if a new schema should be created or not, respectively.

Using the template transformation language, the above description indicates that the construct type `CID`/2nd argument, the `value`/4th argument, the `isTemp`/5th argument and `scheme`/6th argument arguments must be included in the argument sequence that is going to be applied to the `new_object` statement. The `OBID`/3rd argument must be omitted from the sequence and the `SID`/1st argument can either be omitted or not, depending if a new schema should be created or not. During the statement's execution, the created object's OBID is going to be placed in the argument sequence's 3rd position, and the created schema's SID, if a schema is created, is going to be placed in the sequence's 1st position. The position of each argument has been defined in the statement's fingerprint (Table 4.2).

Since the `scheme` argument is a list, it must be defined in another argument sequence and the `new_object`'s 6th argument, i.e. the `scheme`, must point to the `arg_id` of this sequence. Since an object's scheme always contains the object's OBID and this isn't known until the object has been created, i.e. at run-time, the argument that will store the created object's identifier is referred, i.e. `new_object`'s `OBID` argument.

Using a procedural notation, a `new_object` statement that creates an ER entity in a new schema can be invoked as follows:

```
scheme = new List(newObject&);
new_object(newSchema&,"ERentity",newObject&,"department",false,
            scheme);
```

Table 5.1 shows how the same statement can be invoked using the template transformation language. The examples that will follow in this section assume the existence of a template transformation with `trans_id=10`.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
1	2			1
1	4			department
1	5			0
1	6			2
2	1	1	3	

composite transformation program			
trans_id	pc	st_id	arg_id
10	1	1	1

Table 5.1: New Object statement invocation

The statement with `st_id=1` is invoked, i.e. the `new_object` statement according to Table 4.1. The argument sequence applied to this statement has `arg_id=1`. The first argument is omitted and a new schema will be created. The 2nd argument has `value=1`, which means that the object that is going to be created is going to be an ER entity (according to Table 2.4 the construct with `CID=1` is an ER entity). The name of the new entity is `department`, i.e. the new object's `value`/4th argument argument will be `department`. The `isTemp`/5th argument is 0, i.e. false. The argument sequence with `arg_id=2` stores the new object's scheme. There is only one element in the scheme list, because according to Table 2.4 ER entities have one-sized schemes. In the first position of the ER entity's scheme, the entity's OBID must be stored. The OBID is not known until run-time but the argument that will store it is the `new_object`'s `OBID`/3rd argument. Therefore the argument in the first position of the object's scheme, refers to the `new_object` argument sequence's 3rd position.

The `primitive_transformation` invocation is less complicated because there are no list-type arguments. The statement's arguments are presented in Table 4.3. All the arguments of the statement are mandatory and are passed by value to the statement, except from the `toSchema` argument which is optional. This argument can either be passed by reference to the statement, and then a new schema will be created, or passed by value. This means that all the arguments of the `primitive_transformation` statement must be present in the argument sequence that is going to be applied to the statement, except from the 5th one, which can be omitted.

Using a procedural notation, a `primitive_transformation` statement, which adds the object created by the previous `new_object` invocation to an existing schema with `SID=10`, can be invoked as follows:

```
primitive_transformation( add, null, newObject, existingSchema,
                          newSchema, function, null);
```

The `fromObj` argument is `null` since the operation isn't performed on an existing object. The `toSchema` argument isn't passed by reference but the object's `newSchema` is supplied. The `function` argument is specified to define the extent of the added object. The transformation isn't a knowledge-based therefore the `constraint` argument is `null`. Table 5.2 shows how the same statement can be invoked using the template transformation language.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
3	1			add
3	2			-1
3	3	1	3	
3	4			10
3	5	1	1	
3	6			function
3	7			

composite transformation execution			
trans_id	pc	st_id	arg_id
10	2	2	3

Table 5.2: Primitive Transformation statement invocation

The primitive statement of Table 5.2, performs an add operation since the statement's 1st argument has `value=add`. According to [BT01], in every add transformation the `fromObj` attribute in the transformation relation (Table 2.8) has to be equal to `-1`. Therefore, the `fromObj`/2nd argument in the statement's argument sequence has `value=-1`. The `toObj`/3rd argument refers to another argument at position `arg_pos_ref=3` of argument sequence with `arg_id=1`. The referred argument sequence has been defined in Table 5.1, and it's the argument sequence applied to the `new_object` statement described previously. The 3rd argument of this sequence is the OBID of the created object. Therefore, this primitive transformation is going to add the object that was created by the `new_object` invocation. The `fromSchema`/5th argument has `value=10` which the SID of the schema it was assumed to exists. The `toSchema`/6th argument refers to the argument at position `arg_pos_ref=1` of the argument sequence with `arg_id=1`. The referred argument

stores the schema of the object that was created by the previous `new_object` invocation. Thus, the primitive transformation is going to add the created object to schema with `SID=10` and the object's schema is going to be produced. The 7th argument stores the query that defines the added object's extent. The `constraint`/8th argument is `null`.

In the case of the `foreach` statement, the only arguments that are mandatory are: the set of values to be iterated, the method that is going to be executed at each iteration and the initial schema. These arguments are passed by value to the statement. The rest of the arguments (Table 4.4), which are the statement's final schema and the list of objects that have been added/deleted/renamed during the statement's execution, are passed by reference. The argument that defines the current value iterated by the statement must also be passed by reference.

Using the template transformation language, the above description indicates that the `iterated values`/1st argument, the `transformation`/2nd argument, and the `initial SID`/4th argument must be included in the argument sequence that is going to be applied to the `foreach` statement. The `final SID`/5th argument, the `list of objects`/6th argument and the `current value`/3rd argument must be omitted from the sequence. During the statement's execution the positions in the argument sequence corresponding to these last three arguments will be restored.

Since the `iterated values` argument is a list, it must be defined in another argument sequence and the statement's 1st argument must point to the `arg_id` of this sequence.

Using the procedural notation, a `foreach` statement that iterates on two values and executes an existing method with `trans_id=20` at each iteration can be invoked as follows:

```
values = new List("male", "female");
foreach( values, existingMethod, currentValue&, existingSchema,
            finalSchema&, objects&);
```

Table 5.3 shows how the same statement can be invoked using the template transformation language.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
4	1			5
4	2			20
4	4			10
5	1			male
5	2			female

composite transformation execution			
trans_id	pc	st_id	arg_id
10	3	3	4

Table 5.3: ForEach statement invocation

The statement with `st_id=3` is invoked, i.e. the `foreach` statement according to Table 4.1. The argument sequence applied to this statement has `arg_id=4`. The 1st argument points to the argument sequence with `arg_id=5`. In this sequence the set of values to be iterated is stored. In this example, the values are two: `male` and `female`. The 2nd argument points to the block of statements that is going to be executed at each iteration. This block of statement has `trans_id=20`. Finally, the initial schema that the `foreach` statement is going to be operated on has `SID=10`. The 3rd, 5th, 6th arguments are going to be restored at run-time.

Finally, in an `extend_generalization_scheme` statement all the arguments are mandatory and therefore all the arguments must be present in the argument sequence that is going to be applied to the statement. The statement's arguments have been defined in Table 4.5. The `values`/1st argument is a list and therefore it must be defined in another argument sequence, the `scheme id`/2nd argument points to an argument identifier and the `starting pos`/3rd argument is an integer. The value of this last argument is equal to 4 because the sub-entities in the generalization's scheme start from position 4 and onwards.

Using the procedural notation, an `extend_generalization_scheme` statement that extends the scheme of an existing generalization object is presented below. It is assumed that two objects with `OBID=40` and `OBID=50` exist:

```
subentities = new List(existingObject1, existringObject2);
extend_generalization_scheme(subentites,
                    existingGeneralization.getScheme(), 4);
```

Table 5.4 shows how the same statement can be invoked using the template transformation language. It is assumed that an argument sequence with `arg_id=30` exists, representing the argument sequence applied to a `new_object` statement that created the existing generalization object.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
6	1			7
6	2	30	6	
6	3			4
7	1			40
7	2			50

composite transformation execution			
trans_id	pc	st_id	arg_id
10	3	4	6

Table 5.4: Extend Generalization's Scheme invocation

The extend_generalization_scheme statement is invoked by applying the argument sequence with `arg_id=6`. The first argument of the sequence is the sub-entities list and points to another argument sequence where this list is stored. The sub-entities in this specific example are the objects with OBID=40 and `OBID=50`. The second argument refers to an argument in an argument sequence that has been pre-defined. Since this argument sequence corresponds to a `new_object` statement, the 6th argument in that sequence points to the new object's scheme, according to Table 4.2. The 3rd argument has `value=4` specifying that the generalization's scheme should be extended from position 4 and onwards.

The statements' executions initiated from the invocations described in this section are explained thoroughly in Chapter 6.

5.3 A Simple Template Transformation Definition

This section describes how the composite transformation that was used as an example in Chapter 1 can be defined in the template transformation language, according to what has been discussed in the previous section.

The template transformation is described in Section 1.1 and is presented in Figure 1.7 using a procedural notation. It transforms an ER attribute into an ER entity, which is connected with an ER relationship to the entity the attribute was previously attached to. The transformation's name is *entity-attribute equivalence* and its arguments are: (1) the initial schema, i.e. the schema the transformation is going to be operated on, (2) the existing entity, (3) the existing attribute, (4) the new entity's name and (5) the new relationship's name.

composite transformations			
trans_id	name	description	preserves_eq
1	entity-attribute equivalence	transforms attribute to entity	1

Table 5.5: Entity-Attribute Equivalence transformation

Table 5.5 shows how the template transformation is defined in the Template Transformation Syntax Repository. The `description` attribute gives an overview of the transformation and the `preserves_eq` attribute is equal to 1, i.e. true, since this transformation if it's performed on a specific schema applying the correct arguments will produce a new schema equivalent to the original one.

The transformation's five arguments must be defined in the `template transformation definition` relation. The initial schema is represented using a schema identifier, the entity and the attribute are objects, thus they are represented using object identifiers and the last two arguments are the values for the two objects that are going to be created by the transformation.

composite transformation definition				
trans_id	arg_pos	relation_name	attribute_name	description
1	1	schema	SID	initial schema
1	2	object	OBID	entity
1	3	object	OBID	attribute
1	4	object	value	new entity name
1	5	object	value	relationship name

Table 5.6: Entity-Attribute Equivalence argument definition

Figure 1.7 shows the statement sequence implementing the template transformation. First, a `new_object` statement is required that will create the new entity. This statement's invocation is presented in Table 5.7. Omitting the first argument in the argument sequence denotes that a new schema should be created when the statement is executed. The new entity's name is the transformation's fourth argument therefore this argument must be referred by `new_object`'s `value` argument. The rest of the arguments are derived in the same way as it was explained in previous section's `new_object` invocation.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
1	2			1
1	4		4	
1	5			0
1	6			2
2	1	1	3	

composite transformation execution			
trans_id	pc	st_id	arg_id
1	1	1	1

Table 5.7: Entity-Attribute Equivalence 1st statement

The second statement in the template transformation's program adds the created object to the transformation's initial schema. This invocation is similar to the one in Table 5.2. In this case, the `toObj`/3rd argument and the `from`/`toSchema` arguments (4th-5th arguments) refer to other arguments. The `toObj` argument refers to the `new_object`'s `OBID` argument. The `fromSchema` argument refers to the transformation's 1st argument, which specifies the transformation's initial schema, and the `toSchema` argument refers to the `new_object`'s `SID` argument.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
3	1			add
3	2			-1
3	3	1	3	
3	4		1	
3	5	1	1	
3	6			query
3	7			constraint

composite transformation execution			
trans_id	pc	st_id	arg_id
1	2	2	3

Table 5.8: Entity-Attribute Equivalence 2nd statement

The third statement creates a new ER relationship. The `SID`/1st argument is omitted. The 2nd argument stores the object's CID, which is `CID=3` for and ER relationship according to the MDR. The relationship's name is the transformation's 5th argument therefore this argument must be referred by `new_object`'s `value` argument. The scheme of an ER relationship's object stores the OBID of the relationship object and the OBIDs of the objects that are connected with this relationship. In this case, the relationship connects the pre-existing entity with the

one that was created in this template transformation (1st statement). Therefore, the scheme refers to the pre-existing entity, passed as the 2nd argument to the template transformation, and to the created entity.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
4	2			3
4	4		5	
4	5			0
4	6			5
5	1	4	3	
5	2		2	
5	3	1	3	

composite transformation execution			
trans_id	pc	st_id	arg_id
1	3	1	4

Table 5.9: Entity-Attribute Equivalence 3rd statement

The fourth statement, which adds the created relationship, is practically the same as the 2nd statement of this template. Now, the `fromSchema` argument is derived from the previously created entity and the `toSchema` argument from the created relationship.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
6	1			add
6	2			-1
6	3	4	3	
6	4	1	1	
6	5	4	1	
6	6			query
6	7			constraint

composite transformation execution			
trans_id	pc	st_id	arg_id
1	4	2	6

Table 5.10: Entity-Attribute Equivalence 4th statement

The final statement in the template transformation program deletes the pre-existing ER attribute. Omitting the `toSchema` argument in the primitive transformation statement indicates that a new schema must be created during the statement's execution. The new schema's SID will be stored at the 5th position in the `primitive_transformation`'s sequence, i.e. the `toSchema` argument.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
7	1			del
7	2		3	
7	3			-1
7	4	4	1	
7	6			query
7	7			constraint

composite transformation execution			
trans_id	pc	st_id	arg_id
1	5	2	7

Table 5.11: Entity-Attribute Equivalence 5th statement

5.4 A more complicated Template Transformation Definition

The template transformation presented in the previous section explains the key points in a template transformation's program definition, but doesn't use all the statements that have been defined, like `foreach` and `extend_generalization_scheme`. Therefore, in this section a more complicated transformation is presented that employs these statements. Only these two statements are explained thoroughly since the remaining statements have been already described in detail in the previous example.

The template transformation that is going to be examined is called *Mandatory Attribute and Total Generalization Equivalence*. This transformation has been used as an example in Section 3.1 and its definition has been presented in Figure 3.3 using the HDM notation. The transformation removes a pre-existing entity's attribute and creates a generalization. The generalization's sub-entities together with the generalization object are added to the initial schema. The transformation can be performed on a schema like the one in Figure 3.1(a), and produce the schema in Figure 3.1(b).

composite transformations			
trans_id	name	description	preserves_eq
2	mandatory attribute and total generalization equivalence	removes attribute and creates generalization	1

composite transformation definition				
trans_id	arg_pos	table_name	field_name	description
2	1	schema	SID	initial schema
2	2	object	OBID	entity
1	3	object	OBID	attribute
2	4	argument	arg_id	sub-entity names
2	5	object	value	genelization name

Table 5.12: Mandatory Attribute and Total Generalization Equivalence

The arguments for the template transformation are derived from Figure 3.3. The names of the sub-entities are necessary since one object for each one of the sub-entities must be created. The pre-existing entity is also necessary since the created generalization object's scheme must refer to it. Additionally, the created generalization's name and the attribute that is going to be deleted must be passed as parameters.

The template transformation's program can also be identified from Figure3.3. First, a `foreach` statement is necessary that will iterate on the set of sub-entities' names, will create an object for each name and add it to the underlying schema. Then the generalization object must be created and added. Since the generalization object's scheme contains a constraint object identifying whether this generalization is total or partial, the constraint object has to be created before the generalization. Finally, a `primitive_transformation` must be invoked that will delete the pre-existing attribute.

Before defining the template transformation's program, the transformation that is going to be executed inside the `foreach` statement has to be defined. This transformation, as it was mentioned earlier, must create a new object and add it to the latest created schema. Therefore, this transformation consists of two statements, a `new_object` and a `primitive_transformation`. The definition of this transformation is shown in Table 5.13. The existence of an argument sequence with `arg_id=8` is assumed. This argument sequence is going to be applied to the `foreach` statement of the template transformation.

composite transformations			
trans_id	name	description	preserves_eq
2	foreach block for trans_id=2		0

composite transformation execution			
trans_id	pc	st_id	arg_id
2	1	1	9
2	2	2	11

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
9	2			1
9	4	8	3	
9	5			0
9	6			10
10	1	9	3	
11	1			add
11	2			-1
11	3	9	3	
11	4	8	5	
11	5	9	1	
11	6			
11	7			

Table 5.13: Transformation executed in each iteration

Argument sequence with `arg_id=9` defines the `new_object`'s arguments and `arg_id=10` is the new object's scheme. Sequence with `arg_id=11` stores the `primitive_transformation`'s arguments.

The most important feature in this transformation definition is the references to the `arg_id=8` sequence. The `new_object`'s arguments refer to the 3rd argument of the sequence and the `primitive_transformation`'s arguments refer to the 5th argument of the sequence. The argument sequence with `arg_id=8` is going to be applied to the template transformation's `foreach` statement, therefore, according to Table 4.4, the 3rd and 5th argument of the sequence store the current value that is being iterated and the last schema that has been created during the `foreach` execution. In this example, the current value iterated is the name of the sub-entity that is going to be created. This name is the value of the new object that is being created inside the `foreach` statement, therefore the `new_object` statement should refer to this value. The final schema points to the schema that was created in the last iteration therefore it's the `fromSchema` argument in the `primitive_transformation`'s argument sequence.

The full template transformation's program can now be specified. First, the `foreach` statement has to be invoked. According to Section 5.2, a `foreach` statement's mandatory arguments are the 1st, 2nd and 4th arguments. The 1st argument of this statement is the list of the sub-entity names. This list is passed as the fourth argument to the template transformation (Table 5.12). The 2nd argument of the `foreach` statement is the transformation that is going to be executed in each iteration. This transformation has been defined in Table 5.13 and has `trans_id=2`. The 4th argument of the `foreach` statement is the template transformation's initial schema.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
8	1		4	
8	2			2
8	4		1	

composite transformation execution			
trans_id	pc	st_id	arg_id
3	1	3	8

Table 5.14: Mandatory Attribute-Generalization Equivalence 1st statement

The second statement in the template transformation is a `new_object` statement, which will create the constraint object necessary for the generalization's scheme that is going to be created afterwards. Constraint objects have `CID=0` according to the MDR.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
12	2			0
12	4			total
12	5			0
12	6			13
13	1	12	3	

composite transformation execution			
trans_id	pc	st_id	arg_id
3	2	1	12

Table 5.15: Mandatory Attribute-Generalization Equivalence 2nd statement

The third statement must create the generalization object. The schema of the new object must be the same with the schema of the constraint object previously created. Therefore, the 1st argument refers to the constraint object's `SID` argument.

The generalization's scheme isn't complete since only the first three positions have been defined. These positions refer to the object itself, to the constraint object and to the "general" entity in the generalization.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
14	1	12	1	
14	2			4
14	4		5	
14	5			0
14	6			15
15	1	14	3	
15	2	12	3	
15	2		2	

composite transformation execution			
trans_id	pc	st_id	arg_id
3	3	1	14

Table 5.16: Mandatory Attribute-Generalization Equivalence 3rd statement

In order to create the full generalization's scheme, the `extend_generalization_scheme` statement must be invoked. This will add to the generalization's scheme the sub-entities. These sub-entities have been previously created by the `foreach` statement and their object identifiers are stored in the list defined by the 6th argument of the `foreach` statement (Table 4.4). The 1st argument for the `extend_generalization_scheme` statement is this list of identifiers. The 2nd argument refers to the argument that stores the generalization's run-time scheme. The 3rd argument is equal to 4 because the sub-entities in a generalization's scheme start from the fourth position and onwards.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
16	1	8	6	
16	2	14	6	
16	3			4

composite transformation execution			
trans_id	pc	st_id	arg_id
3	4	4	16

Table 5.17: Mandatory Attribute-Generalization Equivalence 4th statement

The final two statements add the generalization to the more recent schema and delete the attribute passed as parameter to the template transformation.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
18	1			add
18	2			-1
18	3	15	3	
18	4	8	5	
18	5	15	1	
18	6			query
18	7			
19	1			del
19	2		3	
19	3			-1
19	4		15	1
19	6			query
19	7			contraint

composite transformation execution			
trans_id	pc	st_id	arg_id
3	5	2	18
3	6	4	19

Table 5.18: Mandatory Attribute-Generalization equivalence last two statements

Chapter 6 - Template Transformation Execution

This chapter's objective is to explain how template transformations are executed. First, the relations of the Template Execution Repository, mentioned in Section 3.2.3, are thoroughly described. Then, each statement's execution trace is exhibited. Finally, a template transformation's execution trace is illustrated and the algorithm use to execute template transformations is supplied in pseudocode.

6.1 Template Execution Repository

The Template Execution Repository can be considered as the run-time stack of a template transformation's execution.

The `arguments` relation (Table 3.7) stores all the run-time arguments created. It's practically the same as the `arguments` relation in the TTR, but in this case arguments don't have to refer to other arguments to define their values. Even if a compile-time argument refers to another argument, at run-time the referred argument is going to have a specific value and the referrer can extract that value.

The `composite transformation invocation` relation (Table 3.8) stores the invocations of template transformations. The `trans_id` attribute identifies which transformation is invoked and the `arg_id` attribute shows which argument sequence is applied to the transformation. The arguments that are used in a template transformation's invocation must be run-time arguments, defined in the TER's `arguments` relation. All the arguments defined for a template transformation are mandatory. An example invocation is shown in Table 6.1. Table 6.2 presents the state of the Schemas and Transformations Repository (STR) before the template transformation's invocation.

composite transformation invocation	
trans_id	arg_id
2	2

arguments		
arg_id	arg_pos	value
1	1	male
1	2	female
2	1	1
2	2	1
2	3	2
2	4	1
2	5	sex

Table 6.1: Template Transformation Invocation

schema					
SID	name	extSchema	extDB	dataSource	isCached
1	s1	1	1	1	0

object					
SID	MID	CID	OBID	value	isTemp
1	1	1	1	person	0
1	1	2	2	sex	0

scheme		
OBID	schemePos	SOBID
1	1	1
2	1	2
2	2	1

Table 6.2: STR before template transformation invocation

The template transformation in Table 6.1 has been defined in Section 5.4. The STR in Table 6.2 represents the schema in Figure 3.1(a) and the invocation of the above transformation will produce the schema shown in Figure 3.1(b).

When a transformation's invocation is defined in the `composite transformation invocation` relation, its execution begins. This means that the `composite transformation execution` relation (Table 3.9) will be populated. This relation is exactly the same as TTR's `composite transformation program` relation: the executed template transformation is identified and the invoked statement, along with the run-time arguments, is defined for each program counter.

6.2 Mapping Arguments

Before executing a statement, its compile-time arguments must be transformed to run-time arguments. The arguments are divided into two categories based on how their value can be extracted: *direct* arguments and *indirect* arguments. Direct arguments have a compile-time specific value (the `value` attribute is assigned) and indirect arguments refer to other arguments to define their value (`arg_pos_ref` and/or `arg_id_ref` are assigned). Also, arguments can be categorized depending on their values' type into *list* arguments and *non-list* arguments. An argument is said to be a list argument if its value is an argument sequence identifier, otherwise it is a non-list argument.

A direct, non-list argument's value can be directly copied to its run-time equivalent argument. An indirect, non-list argument's value must be extracted from the referred argument. The referred argument must have been already defined and it

must have been already mapped to a run-time argument. Therefore, the referred argument's value can be extracted from its run-time equivalent argument.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
9	1			value1
10	1	9	1	

(a)

arguments		
arg_id	arg_pos	value
19	1	value1
20	1	value1

(b)

Table 6.3: Transforming non-list compile-time arguments to run-time arguments

Table 6.3 exhibits the argument transformation process when the arguments belong to the non-list category. The argument at `arg_id=9`, `arg_pos=1` is mapped to the run-time argument at `arg_id=19`, `arg_pos=1`. Since it is a direct argument it's value can be directly copied to the run-time equivalent argument. The argument at `arg_id=10`, `arg_pos=1` is a non-direct argument and its value must be extracted from the referred argument. Before obtaining the value, the referrer argument must be mapped to a run-time equivalent, in this case: `arg_id=20`, `arg_pos=1`. In order to extract the value, the run-time equivalent of the referred argument must be identified. This is accomplished by examining previous mappings. In this case, it's the argument stored at `arg_id=19`, `arg_pos=1`. The value of this argument is copied to the argument at `arg_id=20`, `arg_pos=1`.

Extracting the value of a direct, list argument is a more complicated process. In this case, the value of the argument is a compile-time argument sequence identifier. The argument sequence defined by this identifier must be transformed to a run-time argument sequence and the `arg_id` of the run-time sequence is going to serve as the run-time value of the direct, list argument. The process of transforming the compile-time argument sequence to a run-time one is the following: all the arguments in the sequence must be mapped to run-time equivalents and should be treated as non-list arguments, i.e. their values (direct or indirect) should be copied to their run-time equivalents.

The case of a non-direct, list argument is exactly the same as for a non-direct, non-list argument.

arguments				
arg_id	arg_pos	arg_id_ref	arg_pos_ref	value
9	1			10
10	1			value1
10	2			value2
11	1	9	1	

(a)

arguments		
arg_id	arg_pos	value
19	1	20
20	1	value1
20	2	value2
21	1	20

(b)

Table 6.4: Transforming compile-time list arguments to run-time arguments

Table 6.4 exhibits the argument transformation process when the arguments belong to the list category. The direct, compile-time, list argument at `arg_id=9`, `arg_pos=1` is mapped to the run-time argument `arg_id=19`, `arg_pos=1`. Since the argument points to a compile-time `arg_id`, the argument sequence defined by this `arg_id` must be transformed to a run-time sequence. In this case, the argument sequence with `arg_id=10` is mapped to the run-time argument sequence with `arg_id=20`, and the value of the direct argument is going to be this run-time `arg_id`, i.e. 20. The indirect argument at `arg_id=11`, `arg_pos=1` mapped to the run-time argument at `arg_id=21`, `arg_pos=1`, refers to a previously defined argument and copies the value from the referred run-time equivalent argument.

6.3 Statement Execution Trace

This section examines the execution of each statement.

When a `new_object` statement is executed, a new object is created in the STR. This object's identifier is stored in the statement's `OBID`/4th argument. If the schema argument is omitted then a new schema is also created in the STR and the created schema's SID is stored in the statement's `SID`/1st argument. Finally, the compile-time argument sequence storing the new object's scheme is transformed to a run-time argument sequence and the object's scheme in the STR is updated. The execution trace of the invocation shown in Table 5.1 is exhibited in the following table.

schema					
SID	name	extSchema	extDB	dataSource	isCached
100	temp1	0	0	1	0

object				
SID	CID	OBID	value	isTemp
100	1	20	department	0

scheme		
OBID	schemePos	SOBID
20	1	20

composite transformation execution			
trans_id	pc	st_id	arg_id
10	1	1	1

arguments		
arg_id	arg_pos	value
3	1	100
3	2	1
3	3	20
3	4	department
3	5	0
3	6	4
4	1	20

Table 6.5: NewObject statement execution trace

The execution of a `primitive_transformation` statement is simple. The compile-time argument sequence applied to the statement must be transformed to a run-time argument sequence according to the algorithm described in the previous section. If the statement's `toSchema`/5th argument is missing then a new schema is created in the STR and its identifier is stored in the `primitive_transformation`'s argument. This is the case when a delete operation is performed by the primitive transformation. The execution trace shown in Table 6.6 is produced by the `primitive_transformation` statement's invocation in Table 5.2. Note that the statement as it has been defined in Table 5.2, adds the object created in the previous execution to an already existing schema with `SID=10`. Therefore, the `toObj`/3rd and `toSchema`/5th `primitive_transformation`'s arguments derive from the `new_object`'s execution shown in Table 6.5.

transformation						
action	fromObj	toObj	fromSchema	toSchema	function	constraint
add	-1	20	10	100	function	

composite transformation execution			
trans_id	pc	st_id	arg_id
10	2	2	5

arguments		
arg_id	arg_pos	value
5	1	add
5	2	-1
5	3	20
5	4	10
5	5	100
5	6	function
5	7	

Table 6.6: Primitive Transformation statement execution trace

The execution of the `extend_generalization_scheme` statement only affects STR's scheme relation. Before the execution of this statement a `new_object` statement must have been executed, which would have created a generalization object. The execution trace shown in Table 6.7 is produced by the statement's invocation shown in Table 5.4.

schema					
SID	name	extSchema	extDB	dataSource	isCached
101	temp2	0	0	0	0

object				
SID	CID	OBID	value	isTemp
101	1	21	person	0
101	1	40	male	0
101	1	50	female	0
101	0	24	total	0
101	4	25	sex	0

scheme		
OBID	schemePos	SOBID
25	1	25
25	2	24
25	3	21

arguments		
arg_id	arg_pos	value
300	6	301
301	1	25
301	2	24
301	3	21

(a)

scheme		
OBID	schemePos	SOBID
25	4	40
25	5	50

composite transformation execution			
trans_id	pc	st_id	arg_id
10	3	4	7

arguments		
arg_id	arg_pos	value
7	1	8
7	2	301
7	3	4
8	4	40
8	4	50
301	4	40
301	5	50

(b)

Table 6.7: Extend Generalization Scheme statement execution trace: (a) STR and TTR before execution (b) STR and TER after execution

The `arg_id=7` sequence in Table 5.4 is mapped to the run-time argument sequence with `arg_id=8`. In Table 6.7 it has been assumed that the `arg_id=30` that is employed in Table 5.4 has been previously mapped to the run-time `arg_id=300`. The `extend_generalization_scheme` statement hasn't got any particular difficulties in its execution. The only complicated part is the mapping of its compile-time arguments to run-time ones, which can be easily accomplished if the algorithm presented in Section 6.2 is carefully performed.

The most complicated statement is the `foreach` statement. In order to explain the statement's execution the example shown in Section 5.4 is used. Section 5.4 shows the definition of a template transformation that employs a `foreach` statement. Only the execution of the template transformation's `foreach` statement is presented here. Table 6.1 shows the invocation of the template transformation and Table 6.2 shows the state of the STR before the invocation and therefore before the `foreach` statement's execution, which is the transformation's first statement.

First, the `foreach` statement's arguments have to be transformed to run-time arguments. The 1st argument's value is extracted from the transformation's 4th argument. The 2nd argument is a direct, non-list argument and therefore copied to the TER's arguments table, and finally, the 4th argument refers to the initial schema, which according to the transformation's argument has `SID=1`. Therefore, the run-time argument sequence for the `foreach` statement execution is the following one:

composite transformation execution			
trans_id	pc	st_id	arg_id
2	1	3	3

arguments		
arg_id	arg_pos	value
1	1	male
1	2	female
3	1	1
3	2	2
3	4	1
3	5	1

Table 6.8: ForEach statement. Execution initialisation.

When the statement is executed, the initial schema is copied to the statement's 5th argument (Table 6.8), which stores the final schema after the statement's execution. In this way, if the iterated value list is empty the final schema will be the same as the initial schema.

Then the values that are elements of the list defined by the statement's 1st argument are iterated. The first value in the specific example is `male`. This value is copied to the statement's 3rd argument, which stores the current value iterated, and

the block of statements is executed. This block of statements will first create a new object. As it was mentioned in the template transformation's program definition (Section 5.4), the value of the new object will be extracted from the `foreach` statement's current value argument, i.e. `male`. Table 6.9 shows the state of the STR and TER after the `male` object has been created.

object				
SID	CID	OBID	value	isTemp
10	1	50	male	0

scheme		
OBID	schemePos	SOBID
50	1	50

arguments		
arg_id	arg_pos	value
3	1	1
3	2	2
3	3	male
3	4	1
3	5	1
3	6	4
4	1	50

Table 6.9: ForEach statement. Execution trace after creating object.

Even though during the `new_object` statement a new schema is created, this schema's SID isn't stored at the `foreach` statement's `final SID` argument. The `final SID` argument is updated after an iteration has completely finished.

Next, the block of statements adds the created object to the last created schema. The schema SID of the last created schema is going to be extracted from the `foreach` statement's `final SID`/5th argument.

transformation						
action	fromObj	toObj	fromSchema	toSchema	function	constraint
add	-1	50	1	10		

arguments		
arg_id	arg_pos	value
3	5	10

Table 6.10: ForEach statement. Execution trace after add object.

When the execution of the first iteration terminates, the schema SID of the schema that was produced in the iteration is stored in the `final SID` argument and the OBID of the object that was created is stored in the program's memory.

object				
SID	CID	OBID	value	isTemp
11	1	51	female	0

scheme		
OBID	schemePos	SOBID
51	1	51

transformation						
action	fromObj	toObj	fromSchema	toSchema	function	constraint
add	-1	51	10	11		

arguments		
arg_id	arg_pos	value
3	1	1
3	2	2
3	3	female
3	4	1
3	5	11
3	6	4
4	1	50
4	2	51

Table 6.11: ForEach statement. Execution trace after 2nd iteration.

In the same way the next iteration is executed. The new iterated value is `female`. An object will be created and added to the final schema in the same way as described in the previous iteration. The `foreach` statement's `final SID` argument is going to be updated after the iteration finishes. Also, the identifiers of the objects that were added during the statement's execution will be inserted into a new argument sequence and the statement's 6th argument will store the `arg_id` of this sequence. In this example, this new argument sequence has `arg_id=4`.

6.4 Template Transformation Execution Algorithm

```
int executeTransformation(transID, runtimePC, argMapping, resultingSchema,
newObject)
templatePc = 1;
runPC = runtimePC;
while( statementExists(transID, templatePC) ){
  statement = getStatement(transID, templatePC);
  arguments = getArguments(statement);
  if( statement_is("new_object", statement) ){
     thisMapping = map(arguments, argMapping);
     updateMapping(argMapping, thisMapping);
     copy = makeCopy(arguments, argMapping);
     setOBID(copy, getNewOBID());
     if( SIDmissing(copy) ){
        setSID(copy, getNewSID())
     }
```

```
      execute(runPC, getStID(statement), copy);
      updateSTR(statement, copy);
      resultingSchema = getSID(copy);
      newObject = getOBID(copy);
      templatePC++;
      runPC++;
   }
   if( statement_is("primitive_transformation", statement) ){
      thisMapping = map(arguments, argMapping);
      updateMapping(argMapping, thisMapping);
      copy = makeCopy(arguments, argMapping);
      setOBID(copy, getNewOBID());
      if( toSchemaMissing(copy) ){
         setToSchema(copy, getNewSID())
      }
      execute(runPC, getStID(statement), copy);
      updateSTR(statement, copy);
      resultingSchema = getToSchema(copy);
      newObject = getOBID(copy);
      templatePC++;
      runPC++;
   }
   if( statement_is("foreach", statement) ){
      thisMapping = map(arguments, argMapping);
      updateMapping(argMapping, thisMapping);
      copy = makeCopy(arguments, argMapping);
      initialSID = getInitialSID(copy);
      setFinalSID (copy, initialSID);
      newObjects = null;
      while( valuesExist(statement) ){
         val = getNextValue(statement);
         setCurrentValue(copy, val);
         runPC = executeTransformation(getBlockID(statement);
                   runPC, argMapping, resultingSchema, newObject);
         add(newObject, newObject);
         setFinalSchema(copy, resultingSchema);
      }
      setObjects(copy, newObjects);
      templatePC++;
   }
   if( statement_is("extend_generalization_scheme", statement) ){
      thisMapping = map(arguments, argMapping);
      updateMapping(argMapping, thisMapping);
      copy = makeCopy(arguments, argMapping);
      subEntities = getSubEntities(statement, copy);
      scheme = getScheme(copy)
      while( subentitiesExist(subEntities){
         subEntity = getNextSubEntity(subEntities);
         extendScheme(scheme, subEntity);
      }
      templatePC++;
      runPC++;
   }
   return runPC;
}
```

Chapter 7 - Graphical Representation Language

This chapter explains in detail the language that can be used to define the graphical representation of data-model constructs. The required statements are identified and how these statements can be stored in the Schema Modelling Repository is exhibited. Finally, it is explained how the designed language can be employed and extended.

7.1 Language Design

In Section 3.3 it is explained that the graphical representation language should be data-model independent and that it should consist of elementary statements that support primitive shapes and enable the manipulation of these shapes. The primitive shapes should be parameterised in order to be more useful and flexible. The language that was implemented in this project supports four primitive shapes: box, line, arc and rhombus.

All of these shapes have properties that in some extend define how the specific shape is going to be represented graphically. In this way shapes are parameterised and can be applied to different situations by changing their properties. For example, a rhombus can be used to represent both ER relationships and ER generalizations by changing its property that defines whether the shape should be stretched or not.

Figure 7.1: Different uses of the same primitive shape (a) rhombus can be used as an ER relationship (b) rhombus extended horizontally can be used as an ER generalization

There are some *standard* properties that apply to all shapes and some *shape-specific* properties. The valid values of these properties depend on each shape; values that don't have a valid meaning for a specific shape are ignored. Table 7.1 shows the standard properties that apply to all four shapes.

The `text` property is associated with a `fontName`, `fontSize`, `fontStyle`, `fontColour` and `textLocation` property. The asterisk in Table 7.1 means that each shape has a number of text labels associated with. For example, a box with one text label can represent an ER entity, while a line with two text labels can represent the

line that connects an ER entity and an ER attribute, with the labels defining the cardinalities in the relation.

properties			
name	type	example	default value
style	string	dashed	solid
background	string	grey	white
foreground	string	purple	black
location	(int, int)	(100, 200)	(0, 0)
width	int	10	0
height	int	10	0
text*	string	person	
fontName*	string	sans serif	dialog
fontSize*	int	10	12
fontStyle*	string	italic	bold
fontColour*	string	blue	black
textLocation*	(int, int)	shape-specific	

textLocation values	
shape	values
box	west, east, north, south, southeast, ..., in, out
rhombus	
arc	
line	start, end, center, up, down, over

Table 7.1: Standard properties of primitive shapes

Only the `textLocation` property has shape-specific valid values. The value `west`, which has a valid meaning for a box shape, it will be ignored by a line shape. In general, shapes can ignore the re-setting of their properties and use only the properties' default values.

Except from the standard properties displayed in Table 7.1, shapes also have shape-specific properties, i.e. properties that don't apply to all shapes. Table 7.2 shows these properties.

shape specific properties		
shape	property	type
box	rounded	boolean
	arcWidth	int
	arcHeight	int
	layout	string
	layoutAxis	string
rhombus	extend	int
arc	startAngle	int
	endAngle	int
line	point1	(int, int)
	point2	(int, int)
	startArrow	boolean
	endArrow	boolean
	startAggregation	boolean
	endAggregation	boolean

Table 7.2: Shape-specific properties

A box can be rounded if its `rounded` property is true. In that case the `arcWidth` and `arcHeight` properties can also be specified to define the arc's horizontal and vertical diameter. A rhombus can be extended horizontally or vertically if the `extend` property's value is 1 or 2 respectively. An arc can represent a circle if its properties are `startAngle=0`, `endAngle=360`. A line shape's starting and ending points must be specified.

7.2 Language implementation

As it was already mentioned in Section 3.3, the graphical representation definitions must be independent of construct instances. Therefore, the modelling language must provide a way to parameterise these definitions in order to make them applicable to any construct instance. This requirement together with the fact that each construct instance must be able to access its scheme and identify associated construct instances, leads to the use of Java as the basis of the graphical representation language. Another reason for using Java has been the simplicity of drawing shapes using the `java.awt` API.

The language's statements are method invocations of previously implemented methods in Java. Therefore, the definition of a construct's graphical representation is similar to using an existing API. This API can be used to manipulate shapes and draw them. Methods to change the primitive shapes' properties are also provided.

Even though the language is implementation-oriented it can be easily extended. This is due to the fact that the execution of a construct's graphical representation definition utilizes a Java's characteristic: Java reflection [JREF]. Java reflection allows the invocation of methods that are not known at compile-time. This means that as soon as a method is implemented in Java it can be employed by the graphical representation language and update the representation of a construct or add new modelling definitions while the database schema tool is running. Therefore, new shapes can be added to the four primitive ones and new properties can be added to each shape at run-time.

Another characteristic of the developed language is that a run-time stack is assumed every time a graphical representation definition is executed. It is the existence of this run-time stack that makes the definitions applicable to any construct instance, since it is similar to supporting variables. A description of a graphical representation definition's execution is presented in Section 7.5. Before examining

how definitions are executed, the statements that can be used in these definitions must be specified.

7.3 Graphical Representation's Statements

An overview of the class hierarchy is necessary before specifying the language's statements, since the language's statements are practically the methods that appear in the classes that are present in this hierarchy.

The language uses two classes that are provided by [BT01]: `Schema` and `SchemaObject`. `Schema` represents a database schema. It consists of `SchemaObjects` that represent specific construct instances along with their schemes. Except from those two classes, for each primitive shape a separate class has been developed: `Box`, `Line`, `Arc` and `Rhombus` are all classes that implement the `SchemaShape` interface. A `GraphicSchemaObject` is an implementation of the `GraphicObject` interface and contains one `SchemaObject` and one `SchemaShape`. The `SchemaObject` supplies with information about the object's scheme, and the `SchemaShape` supplies with information about the graphical representation of this object. Finally, a `SchemaManager` exists that manipulates `GraphicObjects`. The `SchemaManager` iterates on its list of `GraphicObjects` and depending on the `GraphicObject`'s construct type the corresponding graphical representation definition is executed. This execution will draw the specific construct instance.

The language's statements are the methods supplied by the aforementioned classes. The rest of this section identifies and explains these methods.

The classes representing the four primitive shapes are the only ones with knowledge of how specific shapes are drawn. They are the only classes that supply a `draw()` method. They store all the standard and shape-specific properties and by changing the values of these properties using the supplied *set* methods, the representation of the primitive shapes can be altered. `Boxes`, `Lines`, `Arcs` and `Rhombuses` operate on a `java.awt.Graphics` object to represent themselves graphically, which is provided by the `SchemaManager`.

Primitive Shapes	
void draw( )	all
SchemaShape setGraphics(Graphics)	all
Point getEdgePosition(Point)	all
SchemaShape setShapeLocation(Point)	all
SchemaShape setText(String)	all
Point getEntityEdgePosition(Point)	rhombus
Point getSpecEdgePosition(Point)	rhombus
Point getGenEdgePosition(Point)	rhombus
SchemaShape setExtend(int)	rhombus
SchemaShape setPoint1(Point)	line
SchemaShape setPoint2(Point)	line

Table 7.3: Methods supplied by classes representing the four primitive shapes

Table 7.3 shows the methods that are supplied by the primitive shape classes. All classes have a draw method, but before invoking it, the `setGraphics` method must be called, which will define the `java.awt.Graphics` object the shape should be drawn upon. The `getEdgePosition` method returns a `Point` that can be used to start/end a line from/to this shape. The `Point` returned is the shape's closest one to the `Point` passed as parameter. All shapes must supply this method since any shape can have lines starting/ending from/to it. The `getEntityEdgePosition` method also returns a shape's `Point` but in this case it is known that the line that will start/end from/to the returned `Point` will end/start to/from an entity. The `getSpecEdgePosition` and `getGenEdgePosition` methods are similar to the previous one with the only difference that the related construct will either be a specialized entity or a generalized one. The last three methods are only supplied by a `Rhombus` object but this should probably be reconsidered. The `setExtend`, `setPoint1`, `setPoint2` methods are methods for setting the corresponding specific-shape properties, while the `setShapeLocation` and `setText` methods correspond to the standard properties `location` and `text`.

SchemaManager
static Arc createArc( )
static Line createLine( )
static Box createBox( )
static Rhombus createRhombus( )
Graphics getGraphics( )
GraphicObject getGraphicObject(SchemaObject)
int getGeneralisationExtend( )

Table 7.4: SchemaManager supplied methods

Table 7.4 shows the methods that `SchemaManager` provides. Except from the methods that create the four primitive shapes, a `getGraphics` method is supplied.

This method returns the `java.awt.Graphics` object that the constructs should be graphically represented upon. `SchemaManager` also provides a method that returns a `GraphicObject` from the list of `GraphicObjects` that manages, corresponding to the `SchemaObject` passed as parameter. Finally, a method that returns an integer representing the value of a `Rhombus`'s `extend` property extending the rhombus horizontally, is supplied.

A `GraphicSchemaObject` contains information for its graphical representation and for its scheme.

GraphicSchemaObject
String getValue( )
Point getLocation( )
GraphicManager getManager( )
SchemaShape setShape(SchemaShape)
SchemaShape getShape( )
SchemaObject getReferencedConstruct( )
SchemaObject[] getReferencedConstructs( )
SchemaObject getGeneralConstruct( )
SchemaObject[] getSpecialConstructs( )

Table 7.5: GraphicSchemaObject supplied methods

The methods: `getLocation`, `getManager`, `setShape` and `getShape` are associated with the graphical representation of this object. The first method returns a `Point` determining where this object is located, the second method returns the `SchemaManager` this object belongs to and `setShape`, `getShape` can be used to define the object's primitive shape representation. The remaining methods (Table7.5) are concerned with the object's relationship with other objects. Except from the `getValue` method which returns the value/name of this object, the `getReferencedConstructs` method returns the objects this object is attached to, (`getReferencedConstruct` can be used when it's already known that this construct is attached to only one object), `getGeneralConstruct` can be operated on an object that corresponds to a generalization and returns the generalized object in the generalization and `getSpecialConstructs` returns the specialized objects, i.e. the generalization's sub-entities.

The next section explains the syntax that can be used to invoke the methods presented in this section.

7.4 Statement Syntax

The API described in Tables 7.3, 7.4, 7.5 can be easily used to define constructs' graphical representation definitions. For example, suppose that a construct exists in some data model, represented graphically using a rectangle of width=50 and height=10. This construct can be modelled in the developed language using a `Box`. A `Box` can be created easily using the `SchemaManager`'s API (Table 7.4). The following definition could represent this construct graphically:

```
ConstructRepresentation(GraphicSchemaObject instance){
    Box shape = SchemaManager.createBox();
    Point location = instance.getLocation();
    shape.setWidth(50);
    shape.setHeight(10);
    shape.setShapeLocation(location);
    shape.setGraphics(instance.getManager().getGraphics());
    instance.setShape(shape);
    shape.draw()
}
```

First, the primitive shape is created. The obtained object cannot be drawn directly since some of its basic properties haven't been initialized yet. The location of the shape and its size must be defined. Additionally, the `java.awt.Graphics` object where the construct is going to be drawn upon must be defined. The default values are used for the remaining properties.

The above graphical representation definition uses variables, which haven't been defined in the developed graphical representation language. Instead, a run-time stack is provided that can store intermediate results and serve as a collection of variables. The way this stack is accessed depends on the statements' syntax and the Java method this statement corresponds to.

A statement in the developed language has the following syntax:

`method arg`.

The first argument, `method`, defines the method that is going to be invoked and the second argument, `arg`, further explains how the invocation should be performed. This second argument is optional. The `method` argument can have one of the following forms:

1. `this.methodName`: the method with the specific name is going to be invoked on the object `this`, which is defined to be the object that is currently being drawn and is an instance of the `GraphicSchemaObject` class.

2. `ClassName.methodName`: this is a static method invocation, i.e. the method with the specific name is `static`. `ClassName` must be the full package name of the class. The name of the class must not be `this`.

3. `methodName`: the method with the specific name is going to be invoked on the object lying on top of the run-time stack.

The second argument, `arg`, in a statement can either be "-n" or it can be specific value, like a number or a string.

If the `method`'s syntax defines (1) an invocation on the current `GraphicSchemaObject`, or (2) a static method invocation the following cases are identified:

a. *the method hasn't got any parameters and* `arg` *is undefined*: the method is directly invoked and the returned value, if there is one, is pushed onto the stack.
b. *the method hasn't got any parameters and* `arg=-n`: the `-n` option means that the object on top of the stack is an array. This array is popped out of the stack and the specific method is invoked as many times as the length of the array. The returned results, if there are any, are stored in an array, which is pushed on top of the stack after all the invocations are performed.
c. *the method has one parameter and* `arg` *is undefined*: the object on top of the stack is popped out and used as the parameter in the method's invocation. The result is pushed onto the stack.
d. *the method has one parameter* `and arg=-n`: the object on top of the stack is an array, which is popped out. This array is iterated and for each object in the array the method is invoked using as parameter the specific object. The results are stored into an array which is pushed onto the stack.
e. *the method has one parameter and* `arg` *is a value*: the method is invoked with parameter the value of the `arg`.

If the `method`'s syntax defines that the method must be invoked on the object on top of the stack, and the object on top of the stack isn't an array then the method invocation is exactly the same as for cases (1), (2). If the object on top of the stack is an array the following cases are identified:

a. *the method hasn't got any parameters and* `arg` *is undefined*: the array on top of the stack is popped out and the method is invoked on each object of the array. The results, if any, are pushed on top of the stack as an array object.

b. *the method hasn't got any parameters and* `arg=-n`: this is the same case as the previous one, only now since the `arg` is set to –n a pop operation on the stack is executed before the method invocations.
c. *the method has one parameter and* `arg` *is undefined*: first the array is popped out and then another pop operation is executed. The second pop operation returns the value that is going to be used as the method's parameter. Finally, the method is invoked on each object of the array. Results are pushed onto the stack.
d. *the method has one parameter and* `arg=-n`: the array is popped out and another pop operation is performed. The second pop operation returns an array whose elements are going to be used as parameters in the method's invocation. The method is invoked on each object of the first array and the corresponding object in the second array is used as parameter. The two arrays must have the same number of elements. The array of the returned results is pushed on top of the stack.
e. *the method has one parameter and* `arg` *has a specific value*: again the array is popped out and now the value specified by `arg` is used as paraneter in the method's invocation.

Some cases that are missing from the above statement execution description are invalid cases.

The methods that have been defined in Tables 7.3, 7.4, 7.5 and are supported by the graphical representation language can have at most one parameter. This isn't restricting since similarly to programming languages, methods can be chained and the same result as having multiple parameters is achieved.

7.5 A Graphical Representation Definition Example

Each construct's graphical representation definition is stored in the Schema Modelling Repository's `construct modeling` relation (Table 3.12). The definition is a sequence of method calls using the syntax described in the previous section, e.g.:

construct modeling		
CID	pc	method
1	1	methodCall1
1	2	methodCall2

Table 7.6: Abstract Modelling example

This section shows how an ER entity's graphic representation can be defined using the developed modelling language.

An ER entity is represented graphically as a rectangle with the entity's name centered inside the rectangle. A rectangle can be created in the developed language using a `Box` object. Since all shapes have associated text labels, the ER entity's name can be represented as a `Box`'s text label.

According to the example illustrated in Section 7.4, the ER entity could have been described as follows:

```
ErEntityRepresentation(GraphicSchemaObject entity){
     Box shape = SchemaManager.createBox();
     String name = entity.getValue();
     shape.setText(name);
     Point location = entity.getLocation();
     shape.setShapeLocation(location);
     shape.setGraphics(entity.getManager().getGraphics());
     entity.setShape(shape);
     shape.draw()
}
```

This description doesn't use the developed language's syntax but provides with an idea of the methods that are needed. The variable `entity` that is used in this description is the keyword `this` used in the developed language's syntax. The static method invocations are practically the same in both descriptions.

Table 7.7 shows how the first three statements in the above description can be represented using the developed language.

construct modeling		
CID	pc	method
1	1	this.getValue
1	1	SchemaManager.createBox
1	2	setText

Table 7.7: Setting a property to a shape

Table 7.7 creates a `Box` object and sets its `text` property. The existence of the stack is very important. When the first statement is executed the entity's name/value will be pushed onto the stack. Then a static method is invoked that has no parameters according to Table 7.4 and returns a `Box` object, which is pushed onto the stack. Then the `setText` method (Table 7.3) is invoked on the object on top of the stack, i.e. the `Box` object. This method will pop the `Box` out of the stack, then it will execute another pop operation to get its parameter (the entity's name), and finally it will push the returned `Box` onto the stack. This Box is the same as the previous one but its text property has been updated.

Using the same rationale, the full definition of an ER entity's graphic representation is shown in Table 7.8.

construct modeling		
CID	pc	method
1	1	this.getValue
1	2	this.getLocation
1	1	this.getManager
1	2	getGraphics
1	1	SchemaManager.createBox
1	2	setGraphics
1	1	setShapeLocation
1	2	setText
1	1	this.setShape
1	2	draw

Table 7.8: ER entity graphical representation definition

Chapter 8 - Conclusions and Future Work

This project developed upon the work described in [BT01] and based on the framework presented in [PM98] it designed and implemented a template transformation language. This report has illustrated the importance of template transformations and explained in detail the syntax of the template transformation language designed. The statements that constitute the language have been examined in detail, their semantics and their run-time behaviour has been presented. Also, it has been described how this language can be employed to define template transformations and how these transformations can be executed on specific schemas.

The work done in this project can initiate numerous research studies. Since a framework has been proposed and a language has been implemented, there are many issues that can be investigated:

1. The language can be extended and more statements can be defined. For example, the function and constraint properties of primitive transformations can be parameterized by defining new statements. These statements can be used to identify the objects' extents. Except from the iteration construct that has been implemented, a conditional construct might be useful.

2. Instead of extending the template transformation language, other tools can be built upon it. A "typing" language can be implemented, as opposed to the "structural" language developed in this project, which will enable the users to type their template transformation definitions. These typed definitions can then be transformed to template transformation definitions as they have defined in this report.

3. Constraints on template transformations can be specified. The implemented language doesn't support constraints on template transformations. A template transformation will always be executed independent of its correctness on the specific schema, or the correctness of its arguments. It's the user's responsibility to perform the template transformation on the right schemas with the correct arguments. Specifying constraints on template transformations can be further extended to automate the process of identifying when a template transformation is applicable.

4. The process of identifying the reverse transformations of defined template transformations can be automated. When a new template transformation is defined, its reverse transformation should be automatically created.

5. Extend the language so that template transformations defined to create constructs of a specific data-model, can be parameterized to create the corresponding constructs of a user defined data model.

This project has also been examining the definition of a graphical representation language that can be used to define how constructs' instances can be displayed. The language that has been developed uses specific method implementations, but this doesn't impose any restrictions. The language supports an extended ER model and more models can be supported like UML and relational model.

Finally, the tool that has been implemented in this project to demonstrate the employment of the graphical representation language can be extended to supply a graphical user interface enabling the definition of schemas and data-models.

References

[Bat84] C. Batini, M. Lenzerini, *A methodology for data schema integration in the entity relationship model*, IEEE TSE10(6), 650-664, 1984.

[Bat86] C. Batini, M. Lenzerini & S. B. Navathe, "*A comparative analysis of methodologies for database schema integration*," ACM Computing Surveys 18 (December 1986).

[BT01] M. Boyd and N. Tong. The automed repositories, model definition repository & schema transformation repository. Technical Report, The AutoMed Project, Imperial College, University of London, UK, June 2001.
http://www.doc.ic.ac.uk/~pjm/automed/techreports/automed_repository.ps

[Chen76] P.P-S. Chen, '*The Entity Relationship Model - Towards a Unified View of Data*', ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976, pp. 9-36.

[JAVA] http://www.java.sun.com

[JREF] http://java.sun.com/products/jdk/1.1/docs/guide/reflection/

[MPinf98] P.J. McBrien and A. Poulovassilis. *A formalisation of semantic schema integration*. Information Systems, 23(5):307--334, 1998.

[MPtec98] P.J. McBrien and A. Poulovassilis. Automatic migration and wrapping of database applications--- a schema transformation approach. Technical Report TR98-10, King's College London, 1998.

[MP99] P.J. McBrien and A. Poulovassilis. *A uniform approach to inter-model transformations.* In Advanced Information Systems Engineering, 11th International Conference CAiSE'99, volume 1626 of LNCS, pages 333-348. Springer-Verlag, 1999.

[Mil93] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. *The Use of Information Capacity in Schema Integration and Translation*. In Int. Conference on Very Large Data Bases, pages 120--133, 1993.

[Mil94] R.J. Miller, Y.E. Ioannidis, and R. Ramakrishnan. *Schema equivalence in heterogeneous systems: Bridging theory and practice*. Information Systems, 19(1):3--31, 1994.

[Ozsu99] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[PM98] A. Poulovassilis and P.J. McBrien. *A general formal framework for schema transformation*. Data and Knowledge Engineering, 28(1):47--71, 1998.

[She90] A.P. Sheth & J. A. Larson, "*Federated database systems for managing distributed, heterogeneous and autonomous databases*," ACM Computing Surveys 22 (September 1990), 183--236.

[Spa92] S. Spaccapietra, C. Parent, and Y. Dupont. *Model independent assertions for integration of heterogeneous schemes*. VLDB Journal, 1(1):81--126, July 1992.

[UML] http://www.uml.org/

[Yan97] L.L. Yan, M.T. Ozsu, L. Liu. Accessing Heterogeneous Data Through Homogenization and Integration Mediators. *In Proc. 2nd Int. Conf. On Cooperative Information Systems* (CoopIS '97), June 1997, pp. 130--139

Appendix A

This appendix presents the Database Schema Integration Tool that has been developed in this project. The tool demonstrates the employment of the Graphical Representation Language and enables the execution of template transformations on specific schemas.

The user is able to access the list of schemas that have been defined in the Schemas and Transformations Repository and choose to load one of them using the tool.

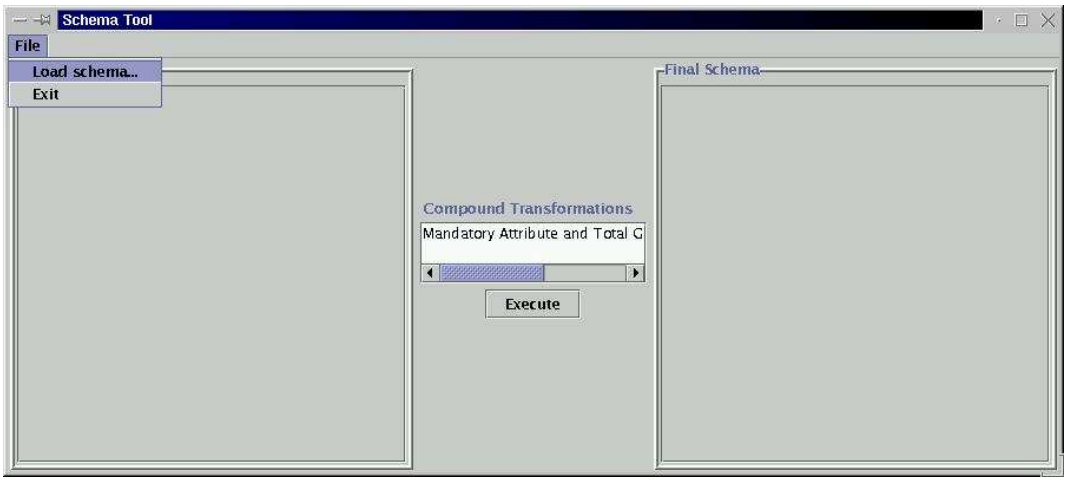

Figure A.1: Load schema using the implemented Tool

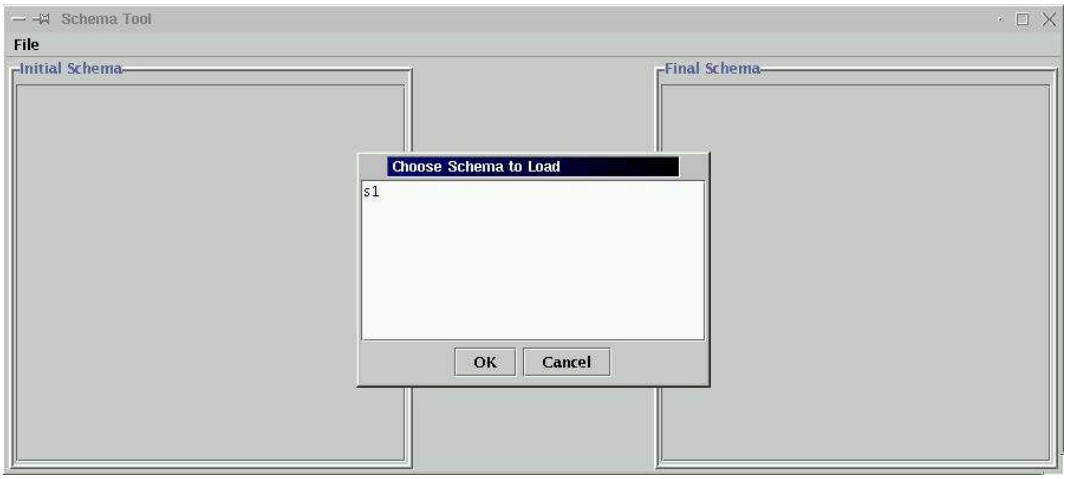

Figure A.2: Choose a schema from the list to load

The chosen schema will be displayed by the tool. This is accomplished by iterating on the schema's constructs and interpreting their graphical representation definition. The tool can support schemas defined in any data model.

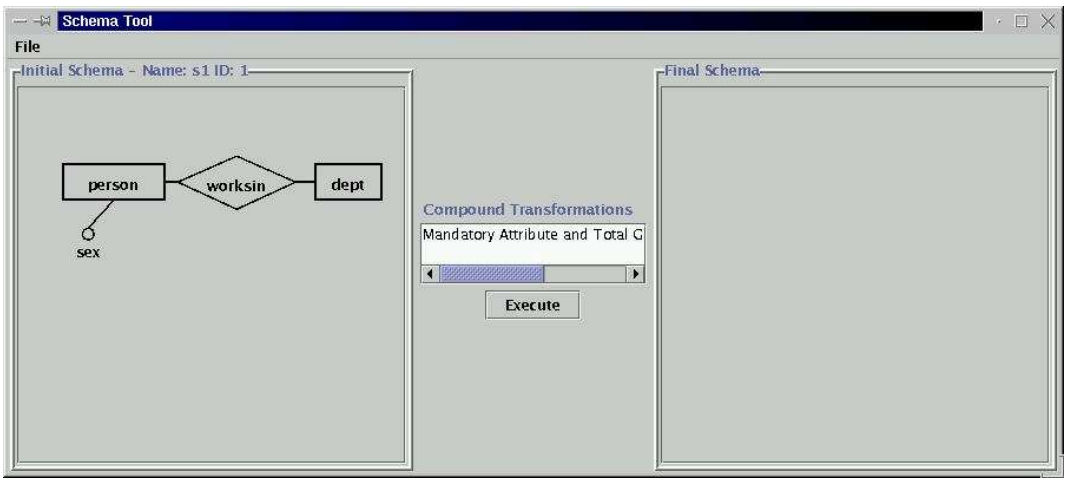

Figure A.3: Graphical Representation of chosen schema

Except from loading schemas and representing them graphically, the tool enables the execution of template transformations on specific schemas. The graphical user interface displays the list of template transformations that have been defined in the Template Transformations Repository. This list is presented in the center of the tool's main window. (Figure A.3). The user can choose a transformation from the list to execute. When a template transformation is chosen another window pops up displaying the chosen transformation's summary (FigureA.4).

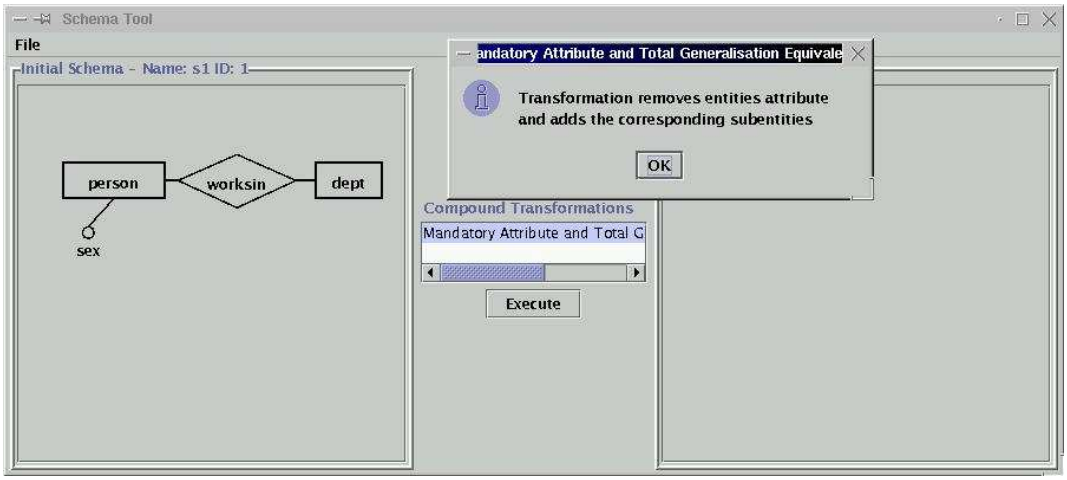

Figure A.4: Choose transformation to execute

In order for the transformation to be executed, the user has to click the *Execute* button. This action will pop up a window enabling the specification of the chosen transformation's arguments (Figure A.5). Each argument's name is displayed above the text-box that can be used to define its value and identifies the type and . which Using the arrows (left, right) the user can iterate on the transformation's arguments and change their values by updating the value appearing in the text-box and clinking on the *Set* button.

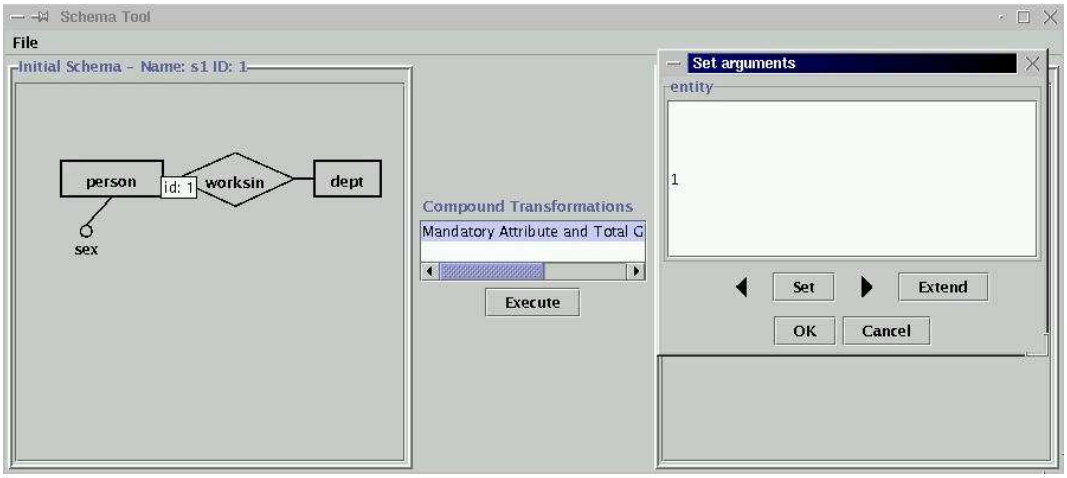

Figure A.5: Defining arguments for a template transformation execution

Specifying the arguments' values may demand knowing object and schema identifiers, as they have been stored in the STR. The schema's name and identifier is displayed above the schema's graphical representation and each object's identifier can be obtained by clicking the mouse's right button above the object. For example Figure A.5 shows that the identifier of the ER entity *person* is 1.

The tool also supports the definition of list-type arguments. In fact it supports the definition of fully nested arguments, i.e. arguments that are lists and may contain other arguments that may also be lists, etc. In order to specify a list argument, the *Extend* button must be clicked. This action will transform the argument that is currently being displayed into a list argument. Then the user can define the elements of this list using the arrows to iterate inside the list and using the *Set* button to define their values (Figure A.6).

By clicking the OK button in the arguments' window will execute the transformation that has been previously chosen by applying to it the arguments defined in this window. The transformation will be executed and a new schema will be created in the STR. The final schema is displayed by the tool (Figure A.7).

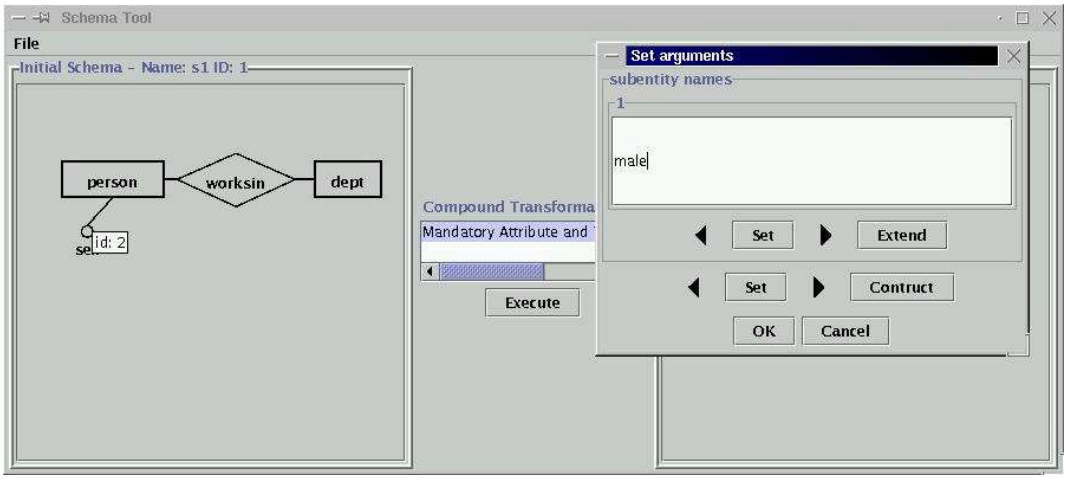

Figure A.6: Defining list arguments using the tool

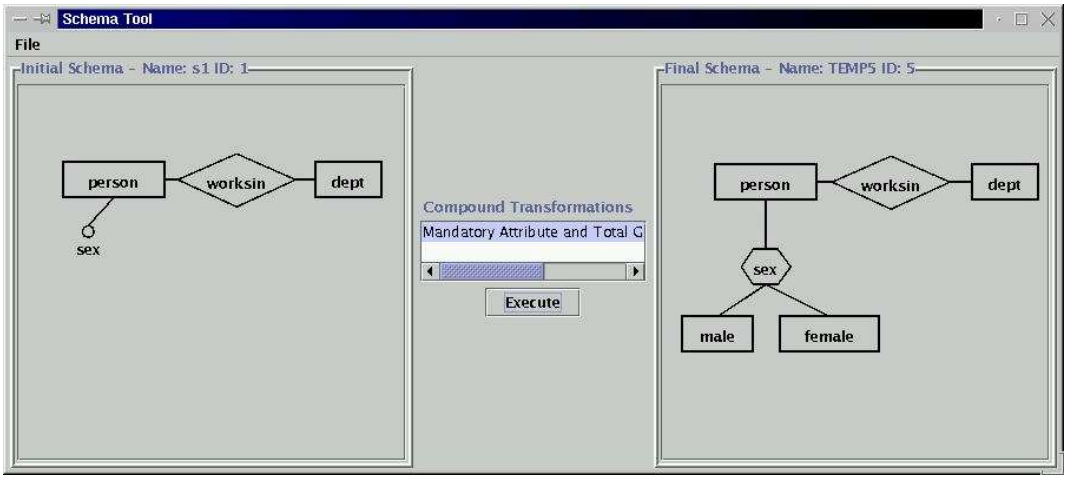

Figure A.7: Graphical representation of the schema produced after the template transformation's execution

Appendix B

This project's implementation, including this report, is available to download from http://www.doc.ic.ac.uk/~nr600/automed.tar.gz. All the necessary Java source files needed to populate this project can be obtained from this location. Instructions on how to built the repositories and initialise them are also given. Additionally, a manual is supplied that explains how the implemented API can be employed to define new template transformations.