Staffordshire University
School of Computing
SOCTR/97/03

Data Transformation Based On Data Model Derivation

Dr. Albert Alderson
School of Computing, Staffordshire University,
PO Box 334, Beaconside, Stafford, ST18 0DG
Email: cmtaa@soc.staffs.ac.uk
Dr. John W. Cartmell, Anthony Elliott
Lincoln Software Ltd, Marlborough Court, Pickford Street, Macclesfield, Cheshire, SK11 6JD

Abstract

Where data needs to be transferred between databases, data from the host (source) may need to be transformed so that it may be loaded into the target (recipient). This paper describes an approach to data transformation based on the idea of making the data model of host appear to be the same as the data model of the target, enabling the simplest transfer of data. This is achieved by expressing the target data model as a derivation of the host data model.

This approach was developed in the context of Lincoln Software's ToolBuilder meta-CASE product - a tool for creating CASE tools. ToolBuilder's database technology supports derived attributes, relationships and entity types. This derivation capability is the basis of the data transformation facilities.

1. Introduction

It is a common requirement to take data from a host database and load it into a target database. Where the host and target are of exactly the same form (have the same data model), each element of data taken from the host can be directly entered into the target without any modification. When the data models differ, data values for the target must be computed from data values of the host - a data transformation is required. If the target already contains data, the data transformation also involves merging of data. This suggests an approach to data transformation in which the host is made to appear to be of the same form as the target.

This approach requires that for each entity type, attribute and relationship in the target we must identify an equivalent entity type, attribute and relationship in the host. Where a direct mapping is not available, we must logically extend the host data model to allow such a mapping. This logical extension can be achieved by defining how to derive each element of the target data model from the host data model. Once a mapping has been made between the host and target data models, it is possible to generate programs to undertake the data transformation. This approach was developed and implemented in the context of ToolBuilder.

ToolBuilder is a meta-CASE product - from a specification of a Computer Aided Software Engineering (CASE) method it generates a product quality CASE tool supporting that method [1]. The generated CASE tool operates on an underlying database (database) of a form developed by the Eclipse project [2, 3] whose design has been greatly influenced by the Functional Model of Data [4, 5].

ToolBuilder implements data transformations in four stages:

1. The relationship between the data models is specified by a transform between them.
2. Programs are generated, using the transform which will dump the data from any database having the host data model and load it into any database having the target data model.
3. Data is dumped from the host database, using the generated dump program, into an intermediate file.
4. The data is loaded into the target database from the intermediate file using the generated load program.

The implementation described here encompasses the cases in which:

- not all elements of the target data model are to be populated with data from the host,

- there are elements of the host data model that have no counterpart in the target,
- only some part of the data in the host is to be transferred to the target, that there is data in the host database that is not to be transported into the target database, even though a mapping could exist
- the target contains data with which the incoming data must be merged.

2. Overview of the Database's Functional Approach

The functional approach is characterised by:

- the way it models data in terms of types and functions. Data is represented as typed entities and the type of an entity determines the attributes it has. Attributes have types, such as String or Integer, or they may be entity-valued (references to other entities) in which case they are typed by an entity type. So a string valued attribute such as the name of a man is a function with the signature:

  name: man -> String

  and an entity-valued attribute such as the husband of a woman is a function with the signature:

  husband: woman -> man

- providing attributes which may be single-valued or many-valued. So the children of a woman is a function with a signature:

  children: woman -> Sequence Of person

- the manner in which data may be accessed through the use of functional expressions representing derived data, which may be typed with a derived type. So, if we have a an entity type person with an attribute birth_date of type date, we may define the derived attribute age of type integer, which is computed from today's date and the value of birth_date. Further, if we have an entity type person with an attribute gender, which may be female or male, then we may derive the entity types man and woman depending upon the value of the attribute gender. Equally we may access data through functional composition, so that the name of a woman's husband has the signature:

  husband/name: woman -> String

  Note that function composition can be interpreted as navigation along a path through the data.

ToolBuilder's database technology views the data of the CASE tool as a collection of entities, each entity being a uniquely identifiable instance of some entity type. Create and delete functions may be defined for each entity type, with default implementations being available. Each entity has attributes and relationships (entity-valued attributes) according to its entity type.

Access to attributes and relationships is functional, with the possibility of each having its own read and update functions. For economy, the functions are defined for each attribute and relationship type. In all, four functions can be defined:

- The show function takes the place of the normal database read function.
- The edit function takes the place of the show function when a value is being extracted for editing.
- The validate function checks a value before it is stored in the database.
- The update function takes the place of the normal database update function.

Composition relationships and reference relationships are distinguished. Composition relationships are used to structure data. A composition relationship is one whose destination entity depends on the relationship for its existence. When a composition relationship is deleted, its referenced entity and substructure are deleted. The substructure consists of all those entities that can be reached from

an entity by traversing composition relationships. Reference relationships are used to cross reference data. Deleting a reference relationship has no effect on the existence of entities.

ToolBuilder allows a wide range of derived attributes and relationships to be constructed. Their values are not stored but are computed from the values of other relationships and attributes.

Derived attributes are defined in terms of some other attribute and ultimately on a stored value. A common case is where the attribute takes the value of an attribute of the destination of some relationship. The value of a derived attribute is calculated from the value of the attribute on which it is based. The functions for the attribute type of the derived attribute determine how this takes place and how updates are treated.

Derived relationships are defined in terms of other relationships, and they are derived ultimately from stored relationships. Derived relationships may be defined to be:

- paths in which component relationships are composed together end to end,
- aggregations in which derived relationships are put together side by side,
- user-defined in which a user supplied function provides the derivation. Paths and aggregations are implemented by in-built functions.

Paths are like relational joins. Each component relationship is evaluated in the context of the results of evaluating previous component relationships. They may be defined to be followed recursively. They are not updatable. A single-valued path is composed from single-valued or many-valued component relationships, and has a single value which is reached by traversing each component relationship in turn. A many-valued path may have single-valued or many-valued component relationships, is evaluated as all entities which are reachable by evaluating each component in turn, and may be set-valued in which case resulting entities are distinct.

Aggregations are like relational unions. A single-valued aggregation may have single-valued or many-valued component relationships, is evaluated as the first non-UNDEFINED result obtained by evaluating the components in turn, and is updatable if all its component relationships are updatable. A many-valued aggregation may have single or many-valued component relationships, is evaluated as the union of the results of evaluating the component relationships, may be set-valued in which case resulting entities are distinct, and is not updatable.

A single-valued user-defined derived relationship is defined by providing the implementation of the read_relation function of that derived relationship. A many-valued relationship is defined by providing the implementation of the read_relation, next and release functions of that derived relationship.

3. Transforms

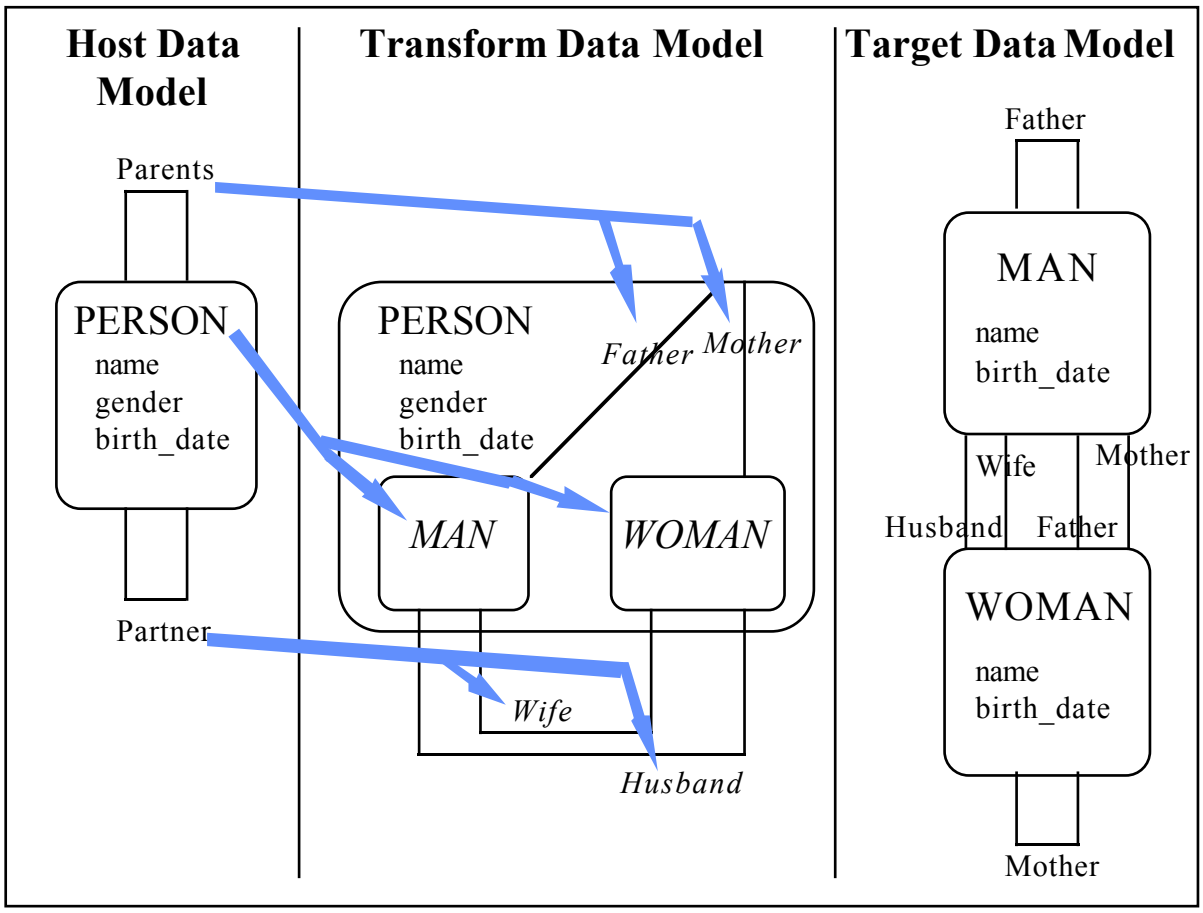
The purpose of a transform (data transformation) is to populate the target database, so the specification of the transform defines where the data is to come from. This is done by specifying how the entity types, attributes and relationships of the target data model may be derived from those of the host data model.

The transform specification is captured in a transform data model as an association between a view of a host data model and an view of a target data model. The target view describes that part of the target data model that will be populated. This corresponds via the transform specification to the host view from which it is to be populated. A view of a data model is a collection of stored or derived entity types and attributes and of relationships defined among the entity types. The entity types, attributes and relationships of a view are defined in the data model. A good way to think of

the action of a transform is that data is pulled out of the host database through the host view into the target database through the target view.

The transform data model associates entity types, attributes and relationships in the host view with those in the target view. Each entity type, attribute and relationship in the transform data model is associated (automatically) with the entity type, attribute and relationship of the same name in the target data model, and by default in the host data model. However, the association between the transform data model and the host data model may be given explicitly.

Transform entity types define the structure of the entities as they are represented during transformation. A transform entity type may be explicitly associated with a particular host entity or (to enable entities of different host entity types to be transformed to entities of a single target entity type) with many host entity types. So, for example, host entity types *female* and *male* could be associated with the transform entity type *human*, causing entities of both host entity types to create entities of type *human* in the target database.



Entities of a single host entity type can be transformed to entities of different target entity types, by defining a function of the transform entity type to derive the entity type in the target data model. So, for example, target entity types *male* and *female* could be derived from a host entity type *human* having an attribute *gender*. In the transform data model, *male* and *female* are defined as subtypes of *human*. The derived type function defined on *human* determines if an entity is of derived type *male* or *female* using *gender*. This derived type is used to create the entity in the target database.

A transform attribute or relationship may be explicitly associated with a particular attribute or relationship in the host data model.

Relationships in the transform model can be either composition or reference. Entities for transformation are reached along the host data model relationships (which may be either composition or reference) associated with transform composition relationships. They are transformed into destinations of the target data model relationships associated with the same transform composition relationships

4. Executing a Transform

When executing a transform it is necessary to indicate from where in the host database the data is to be dumped and to where in the target database it is to be loaded. The data to be dumped is located by a *root relationship*, as is the place to which it is to be loaded. A root relationship is a single-valued relationship from the root of a database to a distinguished root entity type. An entity of this entity type, and all of its substructure, is known as a *region*. The host root relationship specifies the root of the region of the host database to be transformed. Corresponding target root relationship specifies the entity in the target database where the transformed data is to be loaded. A transform can deal with many corresponding pairs of host and target regions during one execution.

The extent of the substructure of a region is primarily determined by the transform data model, particularly the transform composition relationships, but can be further limited by preconditions which may prevent dumping or loading of entities or their substructure.

Data transformations are implemented so that all composition references for all regions are dealt with prior to reference relationships and attributes. This ensures that when reference relationships are loaded all entities in the regions will have been created. No further entities are created as a result of loading a reference relationship.

4.1 Resolution

At the start of the load phase an intermediate file exists, which corresponds in structure to the transform data model and has sections of data, each with a transform entity type, corresponding to entities. The key process during the load phase is to locate the entity in the target database which is to be loaded with each section of data from the intermediate file. This location process is termed *resolution*. If an existing entity cannot be located one is created. The located or created entity is the resolved entity. If the resolved entity is newly created then the data can be loaded directly into it. If the resolved entity already exists then the data must be merged with that already recorded for it.

The entity to receive data is located with the respect to the currently resolved entity. To initialise this process the first resolved entity is that identified by the target root relationship for the current region. Subsequent entities are resolved along transform composition relationships, from the currently resolved entity and this proceeds recursively, visiting all of the entities in each region in turn.

If the parent entity is itself newly created, then no resolution will be attempted for any of its children. At the time a child is encountered in the intermediate file, created entities will not have any of their attributes or reference relationships set, so there will be no useful values on which resolution can take place.

Where the target composition relationship, corresponding to the transform composition relationship, is single valued, the existence or otherwise of the sought entity is simply determined. If

the composition relationship is single-valued and is already defined, then its destination is the resolved entity.

Where the relationship is multi-valued determining the target entity requires further information. As part of its definition a transform relationship may have an identifying attribute. Resolution seeks amongst the referenced entities of the relationship for one with a value of the identifying attribute matching that in the intermediate file. If found, this becomes the resolved entity. If not, then an entity is created and this becomes the resolved entity.

4.2 Reference Relationship Resolution

There is a similar resolution process required when reference relationships are being applied to a resolved entity. The source entity is known, it is the resolved entity. The destination entity must be located through a resolution process.

The simplest case is that in which the reference relationship is required between two entities in the regions being transformed. Such relationships are termed internal relationships. In this case, the path to the destination entity will have been recorded in the intermediate file during the dump phase.

The complex case concerns a reference relationship between an entity within the regions being transformed and one outside. Such relationships are termed external relationships. In this case the path to the destination entity will not have been recorded in the intermediate file during the dump phase and it is necessary to locate destination entity.

The scope is defined by a constraint determined by two further single-valued relationships - the source and destination base relationships. These are used to locate a triangle of entities - the source and destination of the reference relationship and common base entity. The source entity is the currently resolved entity. Potential common base entities are those from which the currently resolved entity can be reached via the source base relationship, and from which potential destination entities can be reached via the destination base relationship. The scope is the set of potential destination entities. If the scope contains more than one entity, the identifying attribute is used to select one. If resolution fails to locate one destination entity, the relationship is not transformed. If a destination entity is resolved, the reference relationship is created if it does not exist.

4.3 Different Merging Strategies

A merge function can be defined for any transform composition relationship to be called after a destination entity is resolved to an existing entity. This function can be used to manipulate the resolved entity.

A merge function can be defined for any transform reference relationship and is called :

- whenever it is determined that an internal relationship already exists in the target of a reference relationship.
- whenever the destination of an external instance is resolved to an existing entity and it is determined that the relationship already exists between them.

The default merge process will not delete relationships in the target database. A merge function can be used to override this behaviour.

5. Summary

The data transformation facilities described in this paper are part of the ToolBuilder product and are in production use. The various functions which may be supplied to customise transforms are written in an interpreted language (called EASEL [4]) which is provided as part of the product. These may be replaced by functions written in C.

Where the host and target data models are the same and data is to be copied, without change, from the host database to the target database, the transform is very simple and can be provided by default. Such a transform is used by ToolBuilder's database compacting facilities, where the data is dumped out of the host database and loaded into an empty target database with the same data model.

The intermediate file format is text. This was chosen because it can be further edited or processed as required, using the many text processing tools available. It also simplifies distribution when the host and target repositories are not physically co-located. Combined with facilities recording changes made to a database, this has been exploited to implement configuration control for a user with two physically separate sites. Batches of approved modifications are applied to a master database defining the system under development.

As well as their originally intended use of allowing users to upgrade to new versions of generated CASE tools based on modified databases, the data transformation facilities have been use to transfer information between generated CASE tools supporting different methods.

References

[1] Alderson A. (1991) Meta-CASE Technology, in *Software Development Environments and CASE Technology*, Lecture Notes In Computer Science, 509, A.Endres and H.Weber (Eds.), Springer-Verlag, 1991.

[2] Cartmell J. and Alderson A. (1988) The Eclipse Data Model - A Functional Account, in *Software Engineering Environments*, P.Brereton (Ed.), Ellis Horwood, 1988.

[3] *Eclipse: An Integrated Project Support Environment*, Bott M.F. (Ed.), Peter Peregrinus Ltd., 1989

[4] Shipman D.W., The functional data model and the data language DAPLEX, ACM Transactions on Database Systems 6(1), March 1981.

[5] Buneman P., Frankel R.E. and Nikhik R., An implementation technique for database query languages, ACM Transactions on Database Systems 7(2), June 1982.