Efficient development of data migration transformations

Paulo Carreira
Oblog Consulting and FCUL
paulo.carreira@oblog.pt

Helena Galhardas
INESC-ID and IST
hig@inesc-id.pt

1 Introduction

Nowadays, the business landscape changes very fast. Organizations merge and joint-ventures have become common headlines. This reality requires system reengineering, information integration and migration of legacy data. In this paper, we address the data migration issue.

Current data migration applications aim at converting legacy data stored in sources with a certain schema into target data sources whose schema is predefined. Organizations often buy applicational packages (like SAP, for instance) that replace existing ones (e.g., supplier management). This situation leads to data migration projects that must transform the data model underlying old applications into a new data model that supports new applications. The migration process is first exhaustively tested and then applied in a one-shot operation, usually during a weekend. The original data sources become obsolete once the migration is performed. The transformation step of the ETL (Extract-Transform-Load) process involved in large-scale data migration projects has two kinds of requirements. The first one concerns the specification of migration transformations. The second deals with the project development and management.

Several issues arise when specifying data migration transformations. First, migration programs require more powerful languages than those supported by most commercial ETL tools currently available. In fact, those languages are usually not powerful enough to represent the semantics of the transformation rules involved. Typically, complex transformations are handled by ad-hoc programs coded outside the tools. Second, data migration programs need more than simple programmers. People that write migration code are often business experts as well. They prefer to use high-level constructs that can be easily composed. Third, the cost involved in the production and maintainability of migration programs must be minimized. Migration code must be short, concise and easily modifiable.

Data migration projects deal with large amounts of data and potentially involve a considerable number of transformations. Therefore, data migration programs are iteratively developed. In real world projects, easy prototyping is thus an imperative requirement. Moreover, as in any other software development effort, code and data must be logically organized into distinct packages. Managing such information is crucial for the success of the initiative.

Finally, migration processes deal with critical data. This means that project auditing is frequent and strict. Auditors want to be sure that the entire set of source data is migrated, i.e., that the migration transformations cover all source records. To ensure this, they need a tool that measures the progress of the migration, and reports which source fields have been migrated and which target fields have been populated.

Data Fusion is a data transformation platform developed and commercialized by Oblog Consulting. It addresses the requirements of generic data transformation applications. In this paper, we describe Data Fusion DM which is the Data Fusion data migration component that has evolved from the requirements of real data migration problems.

1.1 The Data Fusion DM component

Data Fusion offers a domain-specific language named DTL (standing for *Data Transformation Language*) for writing concise and short programs. It also provides an Interactive Development Environment (IDE) for efficiently producing and maintaining code. In the rest of the paper, we will focus on the DTL primitives and IDE features supported by Data Fusion DM.

DTL provides a set of abstractions appropriate for expressing the semantics associated to data transformations. The basic concept is a *mapper* that may enclose several rules. A *rule* encloses transformations with similar logics, e.g., populate fields with the *null* value (see Section 3 for an example of a mapper). The choice of providing such domain-specific language brings several advantages. First, migration solutions can be expressed in a language close to the problem domain. Second, programs are usually concise and easy to read and maintain. Due to these two features, DTL is appropriate for easy prototyping and testing, which are major requirements of data migration applications. Third, the compiler can check if the specific vocabulary is correctly used. In DTL, for example, a target

attribute cannot be assigned twice. Since DTL embodies domain knowledge, a number of optimizations that could not be identified otherwise, can be introduced. Finally, a debugger facility can be developed for data migration programs. The debugger facility implementation of Data Fusion DM is in progress.

The Data Fusion DM IDE supports the development of data migration projects. It follows the trend of modern environments for software development (like e.g. Eclipse or Visual Studio). It includes a text editor that supports known functionalities such as syntax highlighting and code templates. Moreover, the DTL compiler is integrated within the IDE and provides helpful hints when compilation errors occur. The user can parameterize Data Fusion DM through the IDE, in order to differentiate among production and development modes. The types of errors that are allowed when writing and testing a migration application are not the same as the ones that may occur when migrating real data.

The IDE also supports project management. First, the code produced is organized into packages according to the functionality provided. This feature is extremely important in large-scale projects as is the case of data migration. Second, the IDE provides a project tracking facility that shows to be very useful in real data migration applications. The information to migrate is precious in the sense that every source record must be migrated and every slot of the target schema must be filled in. Auditing a data migration project is a very common activity. People owning data to be migrated frequently ask for periodically checking the progress of the data migration process. The IDE reports the state of all source and target fields, i.e., the association between all target and source fields, the percentage of source and target data already migrated, etc.

1.2 Related work

The commercial ETL tools currently available usually either provide an incredible number of operators (e.g., Sagent [Sag]) for transforming data or only a small set of operators (e.g. DataJunction [Dat]). The first group of tools is not easy to use, given the large number of abstractions that the programmer must be able to handle. In the second group of tools, complex transformation logics must be developed as external ad-hoc functions through programming interfaces. This solution has several drawbacks. First, programmers must be aware of at least two programming languages: the transformation language supplied by the tool and the programming language (usually Java or C) for writing external code. Second, migration programs that handle rich transformation semantics turn to be complex and difficult to optimize. Furthermore, debugging of migration transformations tha invoke external code is difficult and further delays the data migration development cycle. The Data Fusion DM approach defends that functions should be defined in the transformation language to allow integrated development and debugging without having to switch among development environments and tracking down bugs through archaic mechanisms.

Commercial tools provide a GUI to specify the source-target mappings, often imposing weak forms of interaction when compared with a modern programming language editor. Finally, some of these tools (e.g., Compuware FileAid/Express [Fil]) neglect the development environment in favor of a more powerful set of data transformations. The application of Data Fusion DM to solve real world data migration problems confirmed our expectations about the IDE usefulness.

Several research data transformation tools have been proposed in the last years. Potter's Wheel [RH01] is a tool for discrepancy detection that allows the user to successively apply simple schema and data transforms. However, the class of data transformations expressible with these transformation operators is limited and does not cover all data migration requirements. The semantics of the AJAX [GFS+01] map operator is similar to the semantics of a Data Fusion DM rule, but rules can enclose more complex logics due to the expressiveness of DTL when compared to the map let clause. Express [SHT+77] is an early prototype for data transformation. As Data Fusion DM, it offers a language for specifying transformations of source files into target files. However, unlike DTL, it does not support recursion. Clio [MHH+01] is a tool for interactive development of schema mappings. However, the set of transformations supported is a subset of SQL. As it will be shown in Section 3, Data Fusion DM DTL can express transformations that cannot be written in SQL.

DTL was designed for capturing the semantics of arbitrarily complex data migrations. It is a domain specific language because it is oriented to a particular problem domain [vDKV00]. Many domain-specific languages have been used over the years (SQL, ASN.1, Makefiles, among others [vDKV00]), and the subject has been receiving increased attention from the research community [Kam97]. MedMaker MSL [PGMU96] and Squirrel ISL [ZHK96] are data integration languages whose main goal is to fusion data from several sources. DTL is intended for specifying a larger class of data transformations.

Data Fusion DM assumes that the source-target schema mappings are known. The tool does not offer any facility for discovering schema mappings as it is the case of COMA [DR02], TranScm [MZ98] and others.

2 Architecture

The Data Fusion platform follows the client-server architecture depicted in Figure 1. On the client side,

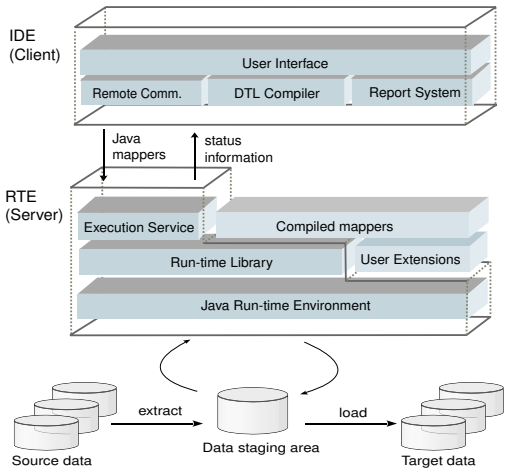

Figure 1: Architecture of Data Fusion

the *Integrated Development Environment* (IDE) allows users to work in multiple data migration projects. On the server side, the *Run-Time Environment* (RTE) is responsible for compiling and parallelizing the data migration requests submitted from IDE instances. This client-server architecture attains scalability. An instance of the IDE may submit requests to multiple RTE instances and an instance of the RTE may run in parallel accepted submissions from multiple IDE instances.

The IDE is constituted by *(i)* the graphical user interface, which is a development environment for DTL specifications *(ii)* the remote communication subsystem in charge of submitting the compiled mappers and receiving the migration progress information, *(iii)* the DTL compiler that generates Java code from DTL mappers, and *(iv)* the report system that is responsible for displaying project tracking and auditing information.

The RTE is composed by *(i)* an execution service responsible for processing submission requests by compiling, launching and monitoring the execution of mappers, *(ii)* a run-time library that implements the semantic concepts of DTL and *(iii)* the Java Run-time Environment which is responsible for executing the Java code.

The transformations are executed by the RTE on a data staging area which can be supported by any RDBMS with a JDBC connection. Data extraction and loading are performed by third-party tools (e.g., Oracle SQL*Loader).

3 The Data Transformation Language

To motivate the unique features of the DTL language, we present a simple example which is a simplified version of typical real world problems found when migrating legacy data. The example show problems that are solved in a concise and self-contained way using DTL. To the best of our knowledge, complex restructuring sequences or manual coding would have to be used if tackled with currently available data transformation frameworks.

The first example illustrates the ability to reason about source record ordering, in particular to variations of record ordering. The second example illustrates cardinality control.

3.1 Migration of currency rates

In this example, the data source view `RATE_LOG` (see Figure 2) stores currency rates that enter the information system at regular intervals. The column `RATE` is the currency conversion rate and `HOUR` contains the hour at which the currency rate entered the system. The goal is to migrate these data into another table where only the start value, turning point values (ascending to descending and descending to ascending) and final value must be kept.

Domain experts elicited the following requirements:

1. Migrate from `RATE` into `RTVAL` the first rate value (1,2 in the figure), the values after turning points (1,6, and 1,4 in the figure) and the last value (1,8 in the figure).

2. The column `HOUR` is mapped directly into column `RTHOUR`.

The DTL mapper that performs this conversion is shown on the right of Fig.2 using the DTL syntax. Note that each requirement is associated with a rule. In particular, the non-trivial requirement for the column `RTVAL` is implemented using a self-contained rule.

The rule works as follows. The variable `V` is used to keep the previous value of `RATE`. It is persistent across rule firings. For each source record, the **foreach** statement of the rule is executed. The **if** statement decides whether to migrate using the value of the `RATE` column. The condition checks if the view cursor is located at the first record, at the last record or if there is a rate variation. Variations are detected using the built-in boolean operator **varying**. This operator evaluates an expression (`signal` in this case) for two consecutive source records. If the returned value is not the same for both records, the operator returns `true`.

Achieving the same effect through a general purpose language involves instructions for moving the cursor forward and backward and an extra temporary variable for holding the value returned by the `signal` function.

Supporting the claim of section 1, the **varying** operator simplifies the migration logic and greatly improves code conciseness and readability.

3.2 Migration of loan information

The source view `LOANS`, in Figure 3, stores the details of loans requested per account. The source column

RATE_LOG

RATE	HOUR
1,2	08:05
1,3	12:05
1,6	16:05
1,5	08:05
1,4	12:05
1,6	16:05
1,7	08:05
1,8	12:05

RTVAL	RTHOUR
1,2	08:05
1,5	16:05
1,6	12:05
1,8	12:05

```
mapper RateConvert
  import master RATE_LOG
  export RATE_EVOL

  RTVAL = rule
    var V: numeric(8,4) = null
    foreach
      if atfirst RATE_LOG
       or atlast RATE_LOG
       or varying signal(RATE - V)
      then
        @@ = RATE
      else
        exclude
      end if
      V = RATE
    end foreach
  end rule

  RTHOUR = HOUR
end mapper
```

Figure 2: Specification of the RateConvert mapper

ACCT is the account number, LOAN is the loan number for each account and AMT is the amount requested. The target system does not support loan amounts superior to 100. When a loan amount greater than 100 is found in the source, it must be split into several loan payment entries in the target. In the target view PAYMENTS, LOANNO is the loan number and AMOUNT is the amount to be payed. The mapping requirements are as follows:

1. The column LOANNO is mapped by concatenating ACCT with LOAN.

2. The column AMOUNT is obtained by breaking down the value of AMT into multiple records with a maximum value of 100, in such a way that the sum of amounts for the same LOANNO is equal to the source amount for the same loan.

The mapper that implements these requirements is shown on the right side of Figure 3. The first requirement is implemented in the rule[1] that assigns the concatenation of the source columns ACCT and LOAN to the column LOANNO.

To implement the second requirement, an auxiliary variable rec_amnt is initialized with the value of AMT and is used to partition the total amount into parcels of 100. The dynamic creation of records is achieved by nesting an **insert** statement into a **while** loop. Each time an **insert** is executed, a new value for the target column is associated with the rule. Internally, values produced by the rules are represented by nodes in a graph. After executing all the rules for a source record, the values contained in the nodes are combined by a graph traversal algorithm to produce target records. In Figure 3, for each iteration of the loop, a node AMOUNT is loaded with 100. After the loop, an additional node AMOUNT is filled in with the remaining value. When both rules are executed for each source

[1]No **rule** keyword is required when the rule is composed of a single statement.

LOANS

ACCT	LOAN	AMT
123	001	20,00
123	002	140,00
456	001	250,00

PAYMENTS

LOANNO	AMOUNT
123001	20,00
123002	100,00
123002	40,00
456001	100,00
456001	100,00
456001	50,00

```
mapper LoanConvert
  import master LOANS
  export PAYMENTS

  LOANNO = ACCT || LOAN

  AMOUNT = rule
    var rec_amnt: numeric
    rec_amnt = AMT
    while rec_amnt > 100 do
      @@ = 100
      rec_amnt = rec_amnt - 100
      insert
    end while
    @@ = rec_amnt
    insert
  end rule
end mapper
```

Figure 3: Specification of the LoanConvert mapper

record, the values stored in node LOANNO and in nodes AMOUNT (for each LOANNO node, several AMOUNT nodes may exist) are combined to generate several records in the target view PAYMENTS.

The distinguishing feature illustrated by this example is as follows. The mapping logics used to load the target columns whose value is fixed (LOANNO in this example) is kept outside the loop. This is highly beneficial because in real-world examples, we often encounter target tables with tens of columns. By nesting all rules inside the loop would compromise their readability.

4 Scenario demonstrated

Data Fusion DM has been used in real data migration projects. For example, it was applied by the Spanish software house INDRA [ind] to migrate financial data, and by Siemens to integrate three databases storing Portuguese public administration information.

In data migration projects, there is a common pattern. Proprietary applications are discontinued in favor of applicational packages which means that legacy data is migrated into a fixed target schema. The migration project that we will demonstrate intends to illustrate the generic characteristics of a data migration project driven by these requirements.

Due to confidentiality restrictions we cannot present real data used in our projects. Therefore, the scenario demonstrated is a constructed example of a banking migration. The Banking information system is composed of four applications: Clients, Accounts, Loans and Credit-cards. The data handled by these applications must be migrated into a pre-defined target schema.

With this demonstration, we want to outline the following points:

1. Complex legacy data transformations – We will illustrate a set of data migration transformations expressible in DTL that are either not tackled or are impractical in existing tools and frameworks.

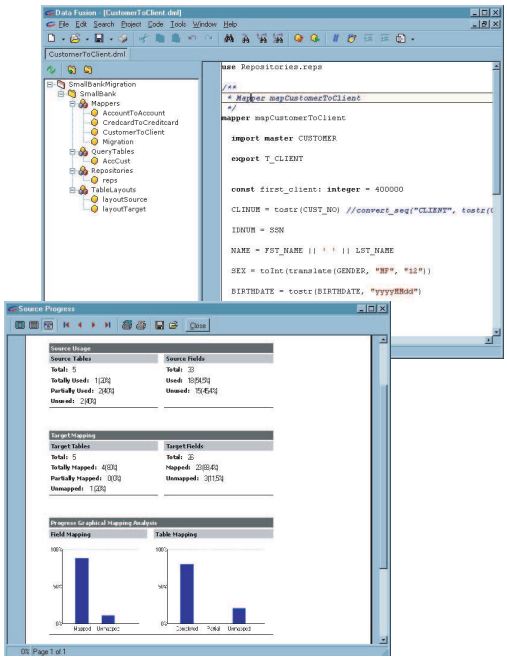

Figure 4: Snapshot of DATA FUSION IDE

2. IDE – We will show our development environment for DTL specifications (see a snapshot in Figure 4). In particular, we will present how the IDE project management handles the migration of real world financial data systems with thousands of tables.

3. Project tracking and auditing – By taking advantage of data dependency information supplied by the DTL compiler we are able: *(i)* to compute coverage metrics for source and target schema and *(ii)* to develop data dependency reports for source and target fields. We show how coverage metrics indicate the progress of rule coding. We also show how auditors take advantage of the data dependency reports to gain insight and confidence about the migration specification.

References

[Dat] DataJunction. http://www.datajunction.com.

[DR02] Hong-Hai Do and Erhard Rahm. Coma – a system for flexible combination of schema matching approches. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'02)*, Hong-Kong, August 2002.

[Fil] Compuware FileAid/Express. http://www.compuware.com/products/fileaid/express.htm.

[GFS+01] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative Data Cleaning: Language, Model, and Algorithms. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'01)*, Rome, Italy, September 2001.

[ind] Indra. http://www.indra.es.

[Kam97] S. Kamin, editor. *First ACM SIGPLAN Workshop on Domain-Specific Languages (DSL'97)*, January 1997.

[MHH+01] R. J. Miller, L. M. Haas, M. Hernandéz, C. T. H. Ho, R. Fagin, and L. Popa. The Clio Project: Managing Heterogeneity. *SIGMOD Record*, 1(30), March 2001.

[MZ98] T. Milo and S. Zhoar. Using schema matching to simplify heterogeneous data translation. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'98)*, New York, USA, August 1998.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A Mediator System Based on Declarative Specifications. In *Proc. Int'l. Conf. on Data Engineering*, Naharia, Israel, March 1996.

[RH01] V. Raman and J. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'01)*, Roma, Italy, 2001.

[Sag] Sagent. http://www.sagent.com.

[SHT+77] N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh, and V. Y. Lum. EXPRESS: A Data EXtraction, Processing and REStructuring System. *ACM Transactions on Database Systems*, 2(2):134–174, June 1977.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[ZHK96] G. Zhou, R. Hull, and R. King. Generating Data Integration Mediators That Use Materialization. *Journal of Intelligent Information Systems*, 6(2/3):199–221, 1996.