MAIME: A Maintenance Manager for ETL Processes

Dariuš Butkevičius[*]
Vilnius University
darius.butkeviciu@gmail.com

Philipp D. Freiberger
Aalborg University
{pfreib15

Frederik M. Halberg
Aalborg University
fhalbe12

Jacob B. Hansen
Aalborg University
jh12

Søren Jensen
Aalborg University
sajens12

Michael Tarp
Aalborg University
mtarp09}@student.aau.dk

ABSTRACT

The proliferation of business intelligence applications moves most organizations into an era where data becomes an essential part of the success factors. More and more business focus has thus been added to the integration and processing of data in the enterprise environment. Developing and maintaining Extraction-Transform-Load (ETL) processes becomes critical in most data-driven organizations. External Data Sources (EDSs) often change their schema which potentially leaves the ETL processes that extract data from those EDSs invalid. Repairing these ETL processes is time-consuming and tedious. As a remedy, we propose *MAIME* as a tool to (semi-)automatically maintain ETL processes. MAIME works with SQL Server Integration Services (SSIS) and uses a graph model as a layer of abstraction on top of SSIS Data Flow tasks (ETL processes). We introduce a graph alteration algorithm which propagates detected EDS schema changes through the graph. Modifications done to a graph are directly applied to the underlying ETL process. It can be configured how MAIME handles EDS schema changes for different SSIS transformations. For the considered set of transformations, MAIME can maintain SSIS Data Flow tasks (semi-)automatically. Compared to doing this manually, the amount of user inputs is decreased by a factor of 9.5 and the spent time is reduced by a factor of 9.8 in an evaluation.

Categories and Subject Descriptors

Information systems [**Information integration**]: Extraction, transformation and loading

1. INTRODUCTION

Business Intelligence (BI) is an essential set of techniques and tools for a business to provide analytics and reporting in order to give decision support. The BI techniques and tools use a Data Warehouse (DW) containing the data for the decision making of a business. A DW contains transformed data from one or more External Data Sources (EDSs) [4]. In order to populate a DW, an Extract-Transform-Load (ETL) process is used. An ETL process extracts data from one or more EDSs, transforms the data into the desired format by cleansing it (i.e., correcting or removing corrupt/inaccurate data), conforming from multiple sources, deriving new values, etc., and finally loads it into the target DW.

Construction of an ETL process is very time-consuming [4]. Maintaining ETL processes *after* deployment is, however, also very time-consuming. The challenges in the maintenance of ETL process are very well demonstrated by the following examples which we have obtained from Danish organizations. The data warehouse solution at a pension and insurance firm needs to cope with unknown changes in the main pension system when weekly hotfixes are deployed. The database administrator at a facility management company has to manage more than 10,000 ETL jobs that execute daily. The ETL team at an online gaming-engine vendor has to manage the daily format changes in the web services delivering sales and marketing data. The data warehouse team at a financial institution strives between the challenges of variances of source data formats and a fixed monthly release window for ETL programs. To summarize, when the number of ETL programs becomes overwhelming for the IT team, management of these programs becomes time-consuming. In such a situation, unexpected changes in source system databases or source system deliveries add more complexity and risk in the maintenance of the ETL programs.

The impact of an EDS schema change depends on both the type of the EDS schema change and how ETL processes use the changed EDSs. Maintenance of ETL processes thus requires manual work, is very time-consuming, and is quite error-prone. To remedy these problems, we propose the tool MAIME which can (1) detect schema changes in EDSs and (2) semi-automatically repair the affected ETL processes. MAIME works with SQL Server Integration Services (SSIS) [5] and supports Microsoft SQL Server. SSIS is chosen as the ETL platform since it is in the top three of the most used tools by businesses for data integration [12], has an easy to use graphical tool (SSIS Designer [2]), and includes an API, which allows access to modify ETL processes through third party programs like MAIME. To maintain ETL processes, we formalize and implement a graph model as a layer of abstraction on top of SSIS Data Flow tasks. By doing so, we only have to modify the graph which then automatically modifies the corresponding SSIS Data Flow task. Running

[*]Work done at Aalborg University

ETL processes is often an extensive and time-consuming task, and a single ETL process not being able to execute could cause a considerable amount of time wasted since the administrator would have to repair the ETL process and run the ETL processes once again. Based on this observation, MAIME can be configured to repair ETL processes even if this requires deletion of parts/transformations of the ETL processes.

In an evaluation MAIME is shown to be able to successfully repair ETL processes in response to EDS schema changes. For the implemented set of transformations, a comparison between resolving EDS schema changes in MAIME and doing it manually in the SSIS Designer tool shows that MAIME is on average 9.8 times faster and required 9.5 times less input from the user to maintain.

The rest of the paper is organized as follows. Section 2 provides an overview of MAIME and how it detects EDS schema changes. Section 3 formalizes the graph model and describes its usage. Section 4 describes how graph alterations are done. Section 5 describes the implementation. Section 6 shows how MAIME compares to doing the maintenance manually. Section 7 covers related work. Section 8 concludes and provides directions for future work.

2. OVERVIEW OF MAIME

In this section, we give an overview of the SSIS components covered by our solution. Then, we explain how MAIME detects EDS schema changes. Finally, we give a description of the architecture of MAIME.

Knight et al. [5] provide an overview of the SSIS architecture. To briefly summarize, a *package* is a core component of SSIS which connects all of its tasks together. The package is stored in a file which the SSIS engine can load to execute the contained *control flows*. A control flow of a package controls the order and execution of its contained *tasks*. This work focuses on *Data Flow tasks*. A Data Flow task can extract data from one or more sources, transform the data, and load it into one or more destinations. It can be observed that the actions performed in a Data Flow task thus resemble those of the general notion of an ETL process. We therefore regard a Data Flow task as an ETL process. A Data Flow task can make use of several types of transformations referred to as *Data Flow components* in SSIS. The current prototype of MAIME covers the following subset of common transformations: *Aggregate*, *Conditional Split*, *Data Conversion*, *Derived Column*, *Lookup*, *Sort*, and *Union All*. To extract and load data, we use *OLE DB Source* and *OLE DB Destination*. For convenience, we extend the term "transformations" to also cover OLE DB Source and OLE DB Destination even though they do not transform data.

We now consider how to maintain ETL processes using these transformations. The first problem is to detect EDS schema changes. We do not want MAIME to rely on third party tools for this. One possibility is then to use triggers. They would, however, need to be created first and for many source systems, there would be many of them to handle both deletions, renames, and additions for hundreds of tables. Further, they would fire for each individual modification and not only after all modifications are done. Instead, we therefore just extract metadata from SQL Server's Information Schema Views with plain SQL statements (other DBMSs typically offer something similar such that support for them also could be added). We thus store a snapshot of how the source databases look and when MAIME is run, it creates a new snapshot and compares that to the previous snapshot to detect which changes have occurred.

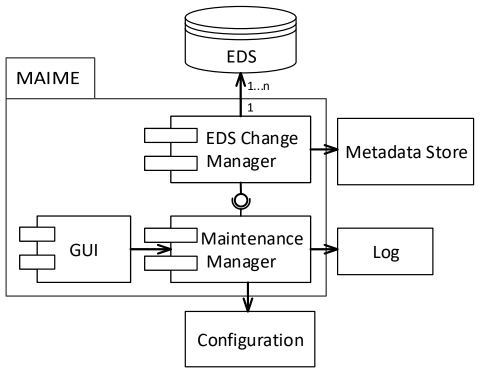


Figure 1: Architecture of MAIME.

Figure 1 shows the architecture of MAIME. We first provide short descriptions for MAIME's surrounding elements that are used by the core MAIME components. EDSs correspond to the external data sources of which schema changes are detected. In Figure 1, the *Metadata Store* corresponds to a local directory. Each captured EDS metadata snapshot is stored as a JSON file in the Metadata Store. The *Log* is a collection files that log every change done by MAIME so that an administrator can analyze the maintenance. The *Configuration* is a JSON file which stores the user-defined configurations.

The *EDS Change Manager* (ECM) captures metadata from the EDSs by using the Information Schema Views. The metadata snapshots are stored in the Metadata Store. Before repairing an ETL process, the current snapshot is compared with the previous snapshot. As a result, a list of EDS schema changes is produced which is accessible to the *Maintenance Manager* (MM). The core logic of MAIME resides in the MM. The MM loads specifications of ETL processes (i.e., SSIS Data Flow tasks) and creates corresponding *graphs*. One graph is responsible for updating one underlying ETL process. The MM contains multiple graphs and operates on them when the graph alteration algorithm (explained in Section 4.2) is called for the found EDS schema changes. Modifications made by the graph alteration algorithm to the graphs are done semi-automatically since the user can be prompted. How exactly this is done depends on the configurations described in Section 4.1. The *GUI* is accessed by an administrator to maintain ETL processes. It involves: (1) Selecting ETL processes to maintain, (2) adjusting administrator configurations, (3) confirming or denying changes by answering to prompts, (4) visualizing applied changes to the ETL processes, and (5) displaying log entries for the applied changes.

3. GRAPH MODEL

To analyze ETL processes and the effects of EDS schema changes, the ETL processes are modeled using graphs which provide a level of abstraction over the SSIS packages. When we make a change in our graph, corresponding changes are also applied to the underlying SSIS package. Afterwards, we save the SSIS package which is now in an executable state. An advantage of using graphs for modeling an ETL process lies in the capability of easily representing dependencies between columns for all transformations and handling cascading changes using graph traversal. In this section, we describe our graph model in details.

Each ETL process is represented as an acyclic *property graph* $G = (V, E)$ where a vertex $v \in V$ represents a transformation and an edge $e \in E$ represents that columns are transferred from one transformation to another. A property graph is a labeled, attributed, and directed multi-graph [10]. Each vertex and edge can have multiple properties. A property is a key-value pair. To refer to the property *name* of vertex v_1, we use the notation $v_1.name$. The set C represents all *columns* used in an ETL process (i.e., columns from an EDS and columns created during the ETL process). Each column $c \in C$ is a 3-tuple $(id, name, type)$ where *id* is a unique number, *name* is the name of the column, and *type* holds the data type of the column. *id* was included because *name* and *type* are not enough to uniquely identify a column. Each edge $e \in E$ is a 3-tuple $(v_1, v_2, columns)$ where $v_1, v_2 \in V$ and $columns \subseteq C$ is the set of the columns being transferred from v_1 to v_2. Putting columns on the edge is particularly advantageous for transformations which can have multiple outgoing edges where each edge can transfer a different set of columns, such as Aggregate in SSIS.

A vertex v representing a transformation has a set of properties depending on the type of transformation. The properties all vertices have in common are: *name*, *type*, and *dependencies*. The only exception is a vertex for an OLE DB Destination which does not have the *dependencies* property. *name* is a unique name used to identify vertex v. *type* denotes which kind of transformation v represents (e.g., Conditional Split or Aggregate). *dependencies* shows how columns depend on each other. If, for example, we extract the column c from an EDS and use an Aggregate transformation that takes the average of c and outputs c', we say that $c' \mapsto \{c\}$. In the case of the Aggregate transformation, c' originates from c and is therefore dependent on c. Any modifications such as deletion of c or a modification of its data type can result in a similar change to c'. Formally, *dependencies* is a mapping from an *output column* $o \in C$ to a set of *input columns* $\{c_1, \ldots, c_n\} \subseteq C$. We say that o is dependent on $\{c_1, \ldots, c_n\}$ and denote this as: $o \mapsto \{c_1, \ldots, c_n\}$. The output columns are defined as the columns that a vertex sends to another vertex through an outgoing edge, such as an OLE DB Source transferring columns to an Aggregate. The input columns are defined as the columns that a vertex receives from another vertex through an incoming edge. The main purpose of dependencies is to detect whether an EDS schema change has any cascading effects. Due to this, the graph contains some trivial dependencies if a transformation does not affect a column, e.g., a dependency such as: $c \mapsto \{c\}$ where c is dependent on itself. One example of an output column depending on multiple input columns is a Derived Column transformation where a derived output column o is dependent on the input columns i_1 and i_2 if they were used to derive the value of o. Thus $o \mapsto \{i_1, i_2\}$ showing that if an EDS schema change affects i_1 or i_2 then it may also affect o.

Additional properties of a vertex are defined by the type of the vertex. For instance, *v.aggregations* is a necessary property for *Aggregate*, whereas *OLE DB Source* does not use aggregations and does not have this property. Table 1 shows all properties specific to each transformation, and the number of incoming and outgoing edges. The following describes the definitions of properties in more detail and how *dependencies* is specified for each type of transformation.

Table 1: Transformations, their properties, and number of allowed incoming and outgoing edges.

Transformation	Specific properties	In	Out
OLE DB Source	`database`, `table`, and `columns`	0	1
OLE DB Destin.	`database`, `table`, and `columns`	1	0
Aggregate	`aggregations`	1	many
Conditional split	`conditions`	1	many
Data conversion	`conversions`	1	1
Derived column	`derivations`	1	1
Lookup	`database`, `table`, `joins`, `columns`, and `outputcolumns`	1	2
Sort	`sortings` and `passthrough`	1	1
Union all	`inputedges` and `unions`	many	1

An **OLE DB Source** is used to extract data from a table or view of an EDS. It provides data for upcoming transformations and is thus represented by a vertex with no incoming edges. An OLE DB Source has some additional properties. *database* is the name of the database data is extracted from. *table* is the name of the table data is extracted from. *columns* is the list of columns extracted from the table. *dependencies* for OLE DB Source is trivial as for each column $c \in columns$, $c \mapsto \emptyset$. Each column is dependent on nothing in the OLE DB Source, as this is the first time the columns appear in the graph.

An **OLE DB Destination** represents an ending point in the graph and is responsible for loading data into a DW. OLE DB Destination has the same properties as OLE DB Source, here representing where *columns* are loaded to. Since OLE DB Destination does not have any outgoing edges, *dependencies* does not exist for OLE DB Destination.

An **Aggregate** applies aggregate functions such as SUM to values of columns which results in new outputs with the aggregated values. Formally, the only property specific to an Aggregate vertex is defined as $aggregations = \{(f_1, input_1, output_1, dest_1), ..., (f_n, input_n, output_n, dest_n)\}$ where $f_i \in \{COUNT, COUNT\ DISTINCT, GROUP\ BY, SUM, AVG, MIN, MAX\}$ is the aggregate function, $input_i \in C$ is the input column that the i^{th} output column is computed from, and $output_i \in C$ is the result of the i^{th} aggregation and output of aggregate i, respectively. $dest_i \in V$ is the vertex receiving the output column $output_i$. A given destination *dest* can appear in multiple tuples of *aggregations*, which shows that the *dest* receives multiple columns through a single edge. For each tuple i in *aggregations*, we have the dependency $output_i \mapsto \{input_i\}$.

A **Conditional Split** applies expressions to rows and can thereby route them to different destinations based on which conditions they satisfy. Expressions are for example $SUBSTRING(title, 1, 3) == "Mr."$ or $salary > 30000$. Formally, the only property specific to the Conditional Split vertex is defined as $conditions = \{(expr_1, p_1, dest_1), \ldots, (expr_n, p_n, dest_n)\}$ where $expr_i$ is a predicate that specifies which rows are routed to $dest_i$, $p_i \in \mathbb{N}^+$ is a priority which indicates in which order the conditions are evaluated in (where 1 is the highest priority). The set of priorities $\{p_1, \ldots, p_n\}$ is a gap-free series of numbers that range from 1 to n, where n is the number of conditions. $dest_i \in V$ is the vertex that receives rows (that have not been taken by a higher priority condition) for which $expr_i$ holds. In Conditional Split the set of output columns on each outgoing edge is equal to the set of input columns. This is because a Conditional Split transformation is not able to add, delete, or otherwise change columns, but is only able to filter rows. Thus, each column depends on itself.

A **Data Conversion** casts the value of a column into a new column with the new data type. Formally, the only specific property of the data conversion vertex is defined as $conversions = \{(input_1, output_1), \ldots, (input_n, output_n)\}$ where $input_i \in C$ is the input column of the vertex and $output_i \in C$ is a new column that has been created through the Data Conversion from $input_i$. The specific conversions of data types are present in the type of $output_i$. A Data Conversion creates a new column rather than replacing the existing column and thus both $input_i$ and $output_i$ are output columns. *dependencies* is simple for Data Conversion, as a dependency is made for each new $output_i$ such that $output_i \mapsto \{input_i\}$ for all tuples in *conversions* and since all input columns are sent to next vertex, each depends on itself.

A **Derived Column** creates new columns based on input columns and expressions applied to those inputs. Formally, the only property specific to the Derived Column vertex is defined as $derivations = \{(expr_1, output_1), \ldots, (expr_n, output_n)\}$ where $expr_i$ is an expression used for computing the new values of $output_i$, and $output_i \in C$ is a newly created column. *dependencies* for Derived Column is similar to Data Conversion in that each input column is dependent on itself. For all output columns in *derivations* we define the dependency $output_i \mapsto \{c_{i,1}, \ldots, c_{i,j}\}$ where $\{c_{i,1}, \ldots, c_{i,j}\}$ is the set of all columns used in $expr_i$.

A **Lookup** extracts additional data from a database by equi-joining with given input columns. Formally, the properties of a Lookup vertex are defined as follows: *database* is the name of the database to lookup additional columns from. *table* is the name of the table used. $columns \subseteq C$ represents all columns in the table. $outputcolumns \subseteq columns$ is the columns extracted from the table. $joins = \{(input_1, lookup_1), \ldots, (input_n, lookup_n)\}$ where *joins* represents the equi-join of the Lookup. $input_i$ is a column from the preceding vertex used for the join condition and $lookup_i$ is a column from *columns*. The equi-join is used to extract the columns in *outputcolumns*. Each input column has a dependency to itself. Each new column in *outputcolumns* is dependent on all the input columns used in the join conditions of the Lookup. In other words, for each output column $output \in outputcolumns$, we have that $output \mapsto \{input_1, \ldots, input_n\}$ where $input_i$ is the input column of the i^{th} tuple in *joins*.

A **Sort** sorts the input rows in either ascending or descending order and creates new output columns for the sorted rows. It is possible to sort on multiple columns where a certain priority has to be given. Formally, the properties of a Sort vertex are defined as follows. $sortings = \{(input_1, output_1, sorttype_1, order_1), \ldots, (input_n, output_n, sorttype_n, order_n)\}$ where $input_i, output_i \in C$, $sorttype_i \in \{ascending, descending\}$, and $order_i \in \mathbb{N}^+$ indicates in which order the columns are sorted (where 1 is the highest priority). The set

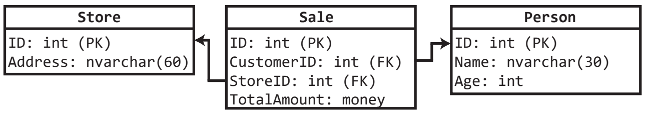


Figure 2: Example of an EDS schema.

of orders $\{order_1, \ldots, order_n\}$ is a gap-free series of numbers from 1 to n. $passthrough = \{c_1, \ldots, c_j\}$ is the set of columns passed through the transformation. This does not include columns that are used for sorting. For dependencies, every output column is dependent on itself.

A **Union All** combines rows from multiple vertices. Formally, the properties of a Union All vertex are defined as follows. $inputedges = (e_1, \ldots, e_j)$ where e_i is the i^{th} incoming edge of the Union All and j is the amount of incoming edges. $unions = \{(output_1, input_1), \ldots, (output_n, input_n)\}$ where $input_i = (c_1, \ldots, c_j)$. Each element in *unions* shows how multiple input columns are combined into a single output column. $c_i \in C \cup \epsilon$ where ϵ is a convention used to indicate that for a given union, no input is taken from the corresponding edge. For example, the tuple $(output_i, input_i)$ where $input_i = (c_1, \epsilon, c_3)$ indicates that the unioned $output_i$ uses c_1 from e_1, c_3 from e_3, but no column is used from e_2. $output_i$ is the column generated by unioning the columns in the i^{th} *input* tuple. A column from a table can only be part of a single union. For the i^{th} tuple of *unions*, $output_i \mapsto \{c_1, \ldots, c_n\}$ such that the new column, which is a result of the union, is dependent on all columns that were used for the union. This dependency is derived for all tuples in *unions*.

EXAMPLE 1. *Figure 2 shows the schema of an EDS called* `SourceDB`. *Now consider the SSIS Data Flow task shown in Figure 3 where*
– ***OLE DB Source*** *extracts the columns ID, Name, and Age from the Person table.*
– ***Lookup*** *extracts the TotalAmount from the Sale table by joining Sale.CustomerID with Person.ID.*
– ***Derived Column*** *derives the new column AmountTimes10 which is derived from the derivation (TotalAmount * 10, AmountTimes10).*
– ***Conditional Split*** *splits the rows into two directions based on {(Age > 40, 1, Aggregate),(TotalAmount > 10000, 2, OLE DB Destination)}.*
– ***Aggregate*** *computes into AvgAmount the average of AmountTimes10 when grouping by Age.*
– ***OLE DB Destination*** *loads ID, Age, and AmountTimes10 into the DW table PersonSalesData.*

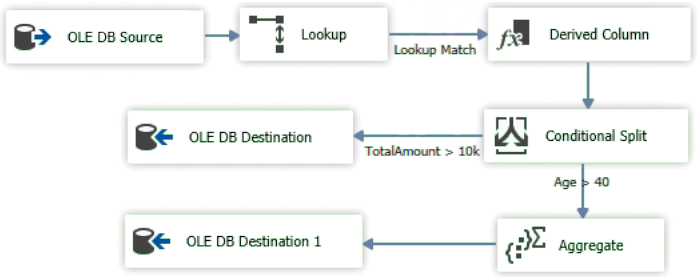


Figure 3: An ETL process.

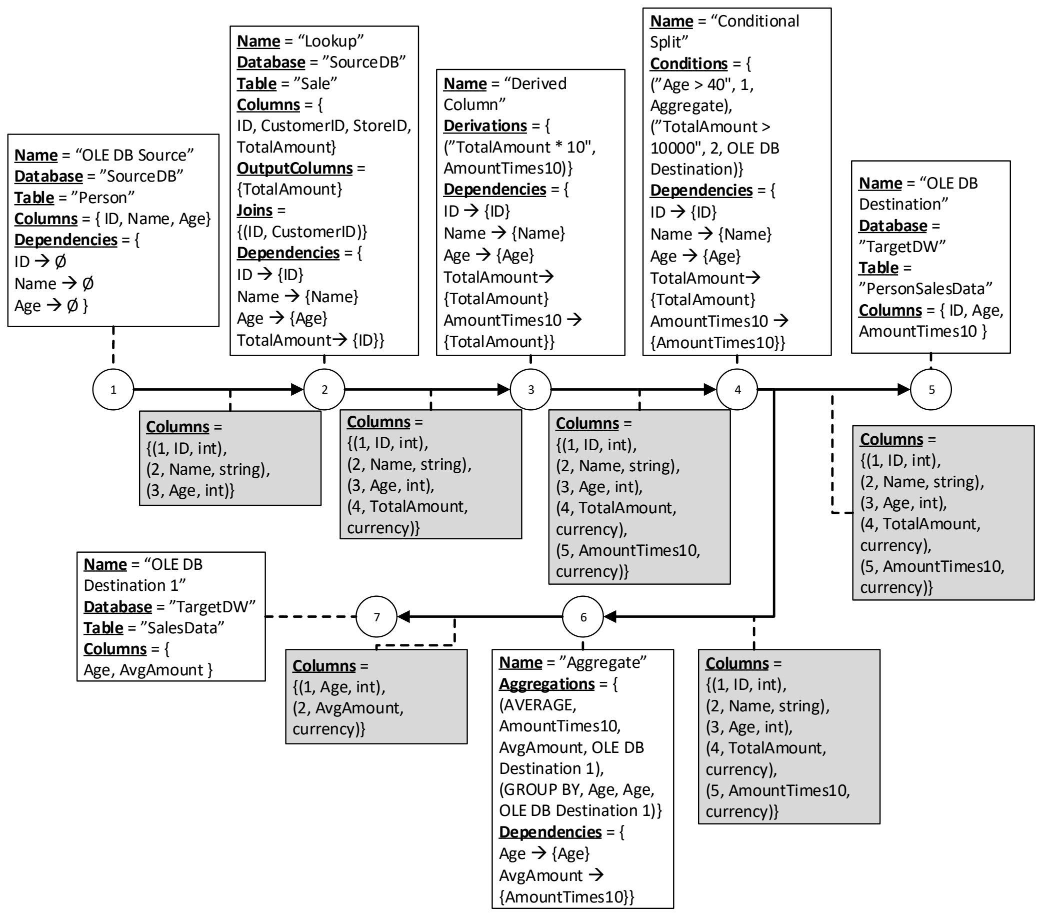


Figure 4: A MAIME graph for the Data Flow task in Figure 3.

– ***OLE DB Destination 1*** *loads AvgAmount and Age into the DW table SalesData.*

Figure 4 shows the corresponding MAIME graph where there are seven vertices, i.e., one for each transformation (the types are not shown since they are obvious from the names of the vertices). As an example of a dependency, we have AmountTimes10 $\mapsto$ *{TotalAmount} in Derived Column. This comes from the derivation TotalAmount * 10. This shows that any change to TotalAmount can affect AmountTimes10, and if TotalAmount is deleted, AmountTimes-10 can no longer be derived and therefore also needs to be deleted.*

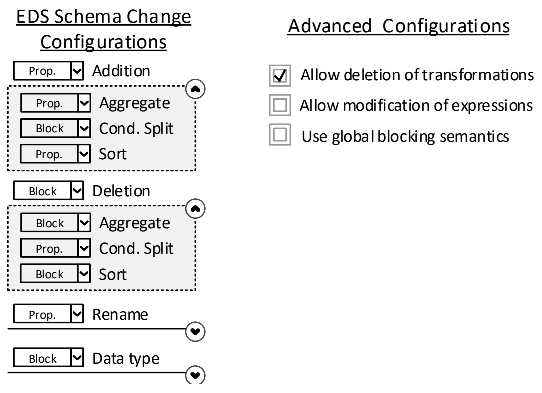
4. GRAPH ALTERATION

In this section, we describe how the MAIME graph is updated when changes happen at the EDSs. First, we describe MAIME's user-configurable settings. Then we describe the algorithm that updates the graph (which in turn updates the underlying SSIS Data Flow task).

4.1 Administrator Configurations

This section describes how an administrator can configure which modifications MAIME is allowed to perform in case of EDS schema changes. As an example, an administrator can define that only renames in the EDS should be propagated to the SSIS Data Flow tasks automatically, and MAIME should *block* (i.e., not propagate) any other changes in the EDS which then have to be handled manually. If MAIME is given free reign, it can, however, make any number of modifications to ensure that ETL processes execute successfully. In the following, we describe the *configurations* that an administrator can specify before executing MAIME. There are two types of configurations: (1) *EDS schema change configurations*, and (2) *Advanced configurations*. To properly understand the configurations, we first explain the different kinds of EDS schema changes. The schema changes that we consider are the following. **Addition** which represents a new column in a schema; **Deletion** which represents a deleted column in a schema; **Rename** which represents a renamed column in a schema; **Data Type Change** which represents a column where the data type has changed (incl. properties such as length).

For an EDS schema change $ch \in$ {Addition, Deletion, Rename, Data Type Change} and vertex type t, a policy $p(t, ch) \in$ {Propagate, Block, Prompt} can be specified by the administrator. The **Propagate** policy defines that when $p(t, ch) =$ Propagate, reparation of vertices of type t is allowed for EDS schema changes of type ch. The **Block** policy says that when $p(t, ch) =$ Block then for every vertex v where $v.type = t$, the alteration algorithm (explained in Section 4.2) is not allowed to make modifications to v or to any successor v_{succ} of v, even if $p(v_{succ}.type, ch) =$ Propagate. The **Prompt** policy defines that the choice of whether to block or propagate the change is deferred to runtime where the user is prompted to make the choice. Figure 5 shows how the administrator can set a policy for a type of an EDS schema change, or be more specific and choose a policy for each type of vertex for a given EDS change. For example, she can configure that for all Conditional Split vertices, deletion of columns should be propagated, but all other EDS changes should be blocked.



EDS Schema Change Configurations

Prop. Addition
- Prop. Aggregate
- Block Cond. Split
- Prop. Sort

Block Deletion
- Block Aggregate
- Prop. Cond. Split
- Block Sort

Prop. Rename

Block Data type

Advanced Configurations

- ✓ Allow deletion of transformations
- ☐ Allow modification of expressions
- ☐ Use global blocking semantics

Figure 5: Configuration of MAIME.

As can be seen in Figure 5, *advanced configurations* can also be specified. MAIME is based on the idea that ETL processes should be repaired such that all of them can execute afterwards. We now explain some more difficult cases of maintaining ETL processes and which configurations the administrator can use for such cases. Considering the case of a deletion of a column, we need to take into account that both columns and vertices can be dependent on the deleted column. Columns such as a derived column is dependent on the columns that it was derived from. Vertices are dependent on columns if the columns are a part of that vertex's input, expression, condition, or used in some other way by the vertex.

The **Use global blocking semantics** option dictates whether the alteration algorithm should terminate if the policy for any vertex is Block. If *Use global blocking semantics* is enabled and $p(t, ch) =$ Block then an ETL process containing a vertex of type t will not be considered for reparation whenever the EDS schema change is of type ch.

The **Allow deletion of transformations** option allows deletion of vertices in a graph (and thus transformations in a SSIS Data Flow task). With this option enabled, MAIME can modify graphs to such an extent that large portions of a graph are removed, but the corresponding updated ETL process can execute. Consider, for example, if a Sort vertex uses column a for sorting. If a is deleted from the EDS, it would render the vertex invalid. With *Allow deletion of transformations* option enabled, the alteration algorithm attempts to repair the process by deleting the Sort vertex and any successive vertices, thereby making the process executable. The algorithm described in Section 4.2 further explains this principle.

The **Allow modification of expressions** option allows the alteration algorithm to modify expressions in vertices (such as Conditional Split and Derived Column) in the event of deletions or data type changes of columns. Consider, for example, a Conditional Split with input columns a and b and one condition with the expression $a < 20$ && $b < 40$ and assume that an EDS schema change of b being deleted occurs. If *Allow modification of expressions* is enabled, the alteration algorithm attempts to update the expression into $a < 20$. This would make the process run successfully without removing the whole expression, but the semantics is different. If *Allow modification of expressions* is disabled, the alteration algorithm would instead remove the expression. Modification of expressions is always allowed if the EDS schema change is the renaming of a column. This is because it is very simple to preserve the original semantics by replacing the old name of the renamed column with the new name in the expression.

4.2 The Graph Alteration Algorithm

This section details how our graph model adapts to EDS schema changes. As described above, the administrator can choose between the Propagate, Block, and Prompt policies. Only a propagation (which can also occur through a prompt) alters the graph. This section goes into depth on how to propagate an EDS schema change through the graph. How propagation is handled for a given type of EDS schema change is specific to each type of transformation. For space reasons, details about the different actions are not provided here, but are available elsewhere [3]. However, just applying actions to each vertex independently is not enough, as a change in one vertex can also affect successive vertices. The `Alter-graph` algorithm takes in a single EDS schema change ch and our property graph G. To be able to handle multiple EDS schema changes, the `Alter-graph` algorithm is called for each EDS schema change. For simplicity, we only present the case where *Allow deletion of transformations* is enabled and neither Prompt nor data type change is used.

The order in which we traverse the graph matters as visiting a vertex also affects successive vertices. For this, topological sorting is used on Line 1 such that when a vertex is visited, it is guaranteed that all its predecessors have been visited beforehand. The list of topologically sorted vertices is referred to as L. On Lines 2–6, the algorithm handles the case where *Use global blocking semantics* is enabled. This entails checking if there exists a vertex in L with the Block policy defined for the EDS change type ch. In case such a vertex exists, the algorithm returns an unchanged graph. On Line 7, we start traversing each vertex v of L. On Lines 8–11, we check if the policy for v given ch is Block. If this is the case, we can disregard this branch of the graph for the traversal, which is why we remove all successors of v from the sorted list of vertices. Line 12 updates the dependencies of a vertex. This is done by going through each dependency in $v.dependencies$ for a given vertex v and checking that all of the involved columns are still present in the incoming edges. The reason for this is that some columns might have been removed from v's incoming edges when traversing the preceding vertices, such that $v.dependencies$ refers to columns that no longer exist in v's input.

As stated before, it is not sufficient to just look at vertices independently when going through the output columns of each vertex. Lines 13–19 show the case for deleting columns with no dependencies. As an example, consider a vertex v whose preceding transformation is a Derived Column with the input column c, which is used to derive a new column d through the expression $d = c + 42$. Now, v will receive c and d as input columns, but if the EDS schema change (ch) is a deletion of c, not only will c not be available to v

Algorithm 1: Algorithm for propagating EDS schema changes to the graph

```
   Name: Alter-graph
   Input: EDS-Change ch, Graph G
   Output: Graph G
1  List L = topological-sort(G)
2  if Use global blocking semantics is enabled then
3      foreach Vertex v ∈ L do
4          Policy p = lookup-policy(v.type, ch.type)
5          if p = Block then
6              return G
7  foreach Vertex v ∈ L (in topological order) do
8      Policy p = lookup-policy(v.type, ch.type)
9      if p = Block then
10         Remove all successors of v from L
11         continue
12     Update v's dependencies to not include deleted
       columns
13     foreach Column c ∈ v's outgoing edges do
14         if v.dependencies(c) = ∅ AND v.type ≠ OLE
           DB Source then
15             if v is fully dependent on c then
16                 Delete v and v's incoming and
                   outgoing edges from G
17                 Break inner loop
18             else
19                 Delete c from v's corresponding
                   outgoing edges and dependencies
20     if v.type is OLE DB Destination AND v has no
       incoming edges then
21         Delete v
22     if v was not deleted AND ch affects v then
23         G = alter(G, v, ch)
24 return G
```

anymore, d will also be deleted, as it is no longer possible to derive it from the Derived Column without c. The way to find out that d is no longer computable is by seeing that d is dependent on $\emptyset$ (Line 14). This signifies that d was dependent on column(s) which have been deleted. The exception is an OLE DB Source, which is the only vertex for which *dependencies* maps columns to $\emptyset$, as explained in Section 3 (OLE DB Source is the only transformation allowing no incoming edges). Beginning on Line 15, it is considered if it is possible to delete only the given column d or if it is necessary to delete the entire vertex v. This narrows down to whether v is *fully dependent* on d or not. We say that a vertex v is fully dependent on some column c, when v would be rendered invalid if c was deleted. For instance, if v is an Aggregation and c is the only remaining column that is being used for aggregations, then v is fully dependent on c. The deletion of c would result in $v.aggregations = \emptyset$, which is not a valid transformation. What qualifies a vertex as being fully dependent on a column is specific for each type of transformation, for details see [3]. Lines 16–17 show the case of deleting a whole vertex and all of its incoming and outgoing edges, if they are no longer used. This iteration of the loop breaks because it would not make sense to continue iterating over the output columns of a vertex that has been deleted. The other case of only deleting the given column is shown in Lines 18–19. Since a vertex of type OLE DB Destination does not have any outgoing edges, it is not considered in the previous loop on Lines 13–19. However, we still want to delete the vertex if it is invalid, i.e., if it has no incoming edge. This is performed on Lines 20–21. Afterwards on Lines 22–23, if v was not deleted at an earlier point in the algorithm, then the corresponding propagation action for the EDS schema change and transformation type is invoked. Finally, the fully altered graph is returned. Recall that the underlying SSIS Data Flow tasks gets updated automatically when the graph is updated.

Table 2: Currently supported EDS changes.

	Deletion	Rename
OLE DB Source	✓	✓
OLE DB Destination	✓	✓
Aggregate	✓	✓
Conditional Split	✓	✓
Data Conversion	✗	✗
Derived Column	✓	✓
Lookup	✓	✓
Sort	✗	✗
Union All	✗	✗

5. IMPLEMENTATION

This section describes the implementation of the prototype. MAIME was developed in C# for Microsoft SQL Server 2014 Developer Edition and with the SSIS Designer as the data integration tool. The SSIS Designer is part of the SQL Server Data Tools (v. 14.0.60203.0). The codebase for the implementation includes around 7,700 lines of code, 85 classes, and 410 members. The implementation is open source and is available from https://github.com/sajens/MAIME.

The advanced configurations, *Allow deletion of transformations* and *Use global blocking semantics* are implemented. We did not implement *Allow modification of expressions* yet. We focused our attention primarily on *Allow deletion of transformations* and *Use global blocking semantics*, since they seemed to be the most impactful configurations for MAIME. Currently, *Allow deletion of transformations* is always enabled. The Propagate and Block policies are implemented as described previously. Table 2 shows which transformations have been implemented with respect to deletion and renaming of EDS columns in the current prototype.

We now consider how the graph is represented internally in MAIME. Our graph is implemented as a layer on top of SSIS. Therefore, our graph constructs have references to the corresponding SSIS constructs. To be more exact, each vertex refers to a SSIS component (transformation), each edge refers to a SSIS path, and each column has a corresponding reference to a so-called *SSIS IDTS object*. The SSIS IDTS object covers both input and output columns in SSIS. By having these references, we can easily propagate changes to the underlying ETL process during execution of the graph alteration algorithm. For the graph, we have implemented the classes Graph, Vertex, Edge, Column, and a class extending Vertex for each supported transformation such that specific dependencies and properties can be represented.

In order to instantiate the graph, we have to translate the components from SSIS[1]. Initially, a SSIS package is loaded by parsing its `.dtsx` file conforming to XML specification. Then, we extract its SSIS Data Flow tasks. Note that every Data Flow task is stored as a graph in a `.dtsx` file. Construction of a MAIME graph is done in two iterations. During the first iteration, we extract one Data Flow task and go through its components to create corresponding vertices with references to the Data Flow task components. With the set of vertices V and references to the SSIS components, we can extract information for each vertex such as *v.name* and *v.type*. Afterwards, every SSIS path is translated into an edge. This gives us the set of edges E of our graph. It is important to note that columns are not stored in the edges in this iteration, since they do not exist on SSIS paths. After the first iteration is complete, we have created the basic structure of the graph of MAIME. In the second iteration, we deduce dependencies, assign columns to edges, and create transformation specific properties.

6. EVALUATION

We now evaluate the efficiency of MAIME. In the evaluation, we compare how much time and how many user inputs are required to resolve a series of EDS schema changes when using MAIME and when repairing the ETL flow manually. A user input is defined to be a mouse click or a keystroke done by the user (both are recorded separately). We discussed and validated this approach with practitioners in field of ETL development.

We consider three SSIS packages and each has a single Data Flow task. For each considered SSIS package, both MAIME and the manual work end up having the same end result, i.e. the same state of the maintained SSIS package. Only the process of achieving that end result differs. The EDS schema changes used for the evaluation include deletion and renaming.

For the manual approach, the user performing the evaluation (one of the authors) is given the EDS schema changes and two versions of the ETL process: One in its initial state, and another in the desired result state. The user then performs maintenance of each package three times using Visual Studio 2015 Community edition and SQL Server Data Tools. The first evaluation gives an indication of how long it takes to maintain an ETL process without knowing every step to maintain the ETL process. The second and third attempt are performed in order to provide a best-case scenario, since the user learns the quickest way to maintain the ETL process through repetition. The time of the evaluation starts when the user begins repairing the ETL process after having seen the given information (i.e., EDS schema changes and the two versions of the ETL process). During the test, the duration and amount of user inputs used are recorded using a software tool.

When considering MAIME, the user interacts with MAIME to (1) accept the configurations, currently loaded ETL processes, and connected EDSs, and (2) start the maintenance process. The EDS schema change configurations of MAIME for all evaluations are all set to propagate. For the advanced configurations, *Allow deletion of transformations* is enabled, *Allow modification of expressions* is disabled, and *Use global blocking semantics* is disabled. The time of the evaluation starts when the user begins to use MAIME and ends when the SSIS package has been repaired. MAIME is also used to repair the package three times and the duration and amount of user inputs are again recorded using a software tool.

[1]msdn.microsoft.com/en-us/library/ms403344.aspx

We acknowledge that the user knows MAIME very well and thus easily can use it. That is why the user also is given the desired resulting SSIS package for the manual approach such that he does not have to think about what needs to be changed. This is thus an overly optimistic way to measure the needed time for the manual approach (the amounts of needed clicks and keystrokes are not affected, though).

We now elaborate on one of our test cases and a given list of EDS schema changes. In test case 1, we consider the scenario from Example 1 and assume the following changes to the EDS: (1) Age is renamed to RenamedAge in the *Person* table and (2) TotalAmount is deleted from the *Sale* table. The state of the graph after it has been maintained is shown in Figure 6 in the SSIS designer tool, and with MAIME's graph in Figure 7. The latter shows that Age is successfully renamed on all edges. The deletion of TotalAmount is slightly more complicated since the condition TotalAmount > 10000 in the Conditional Split involves the column. This condition is no longer valid which results in the removal of the outgoing edge containing it. We therefore delete OLE DB Destination, as it no longer has any incoming edges. Derived Column derives the column AmountTimes10 from TotalAmount and can therefore no longer be derived. We delete this derivation but still retain our Derived Column transformation since it can exist without doing any derivations. Another possibility would be to let MAIME delete the Derived Column and connect the Lookup and Conditional Split transformations. The Aggregate transformation takes the average of AmountTimes10, which no longer exists and this aggregation is therefore also deleted. OLE DB Destination 1 no longer loads the aggregated AvgAmount from the Aggregate transformation into the DW. Both TotalAmount, AmountTimes10, and AvgAmount are deleted from the edges.

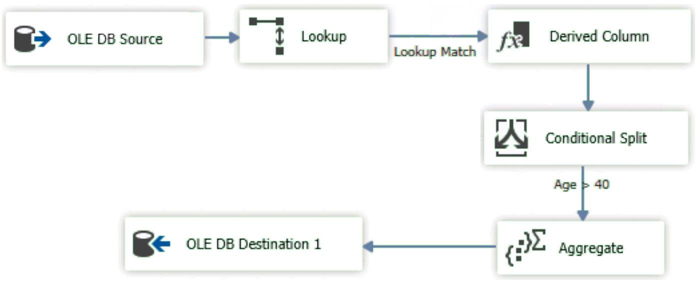


Figure 6: The Data Flow task of the SSIS package for test case 1 after the EDS schema changes.

Test cases 2 and 3 are not shown here for space reasons, but only briefly described. In test case 2, TotalAmount is deleted, and both Address and Person.ID are renamed. In test case 3, Name is deleted and Age is renamed.

For each test case, the used time is shown in Table 3 and the amounts of user inputs are shown in Table 4. For time-consumption, MAIME took on average 4 seconds across all 3 test cases while manually resolving changes took 39.3 seconds on average across all 3 cases for the third attempt. This means MAIME was on average of 9.8 times faster for resolv-

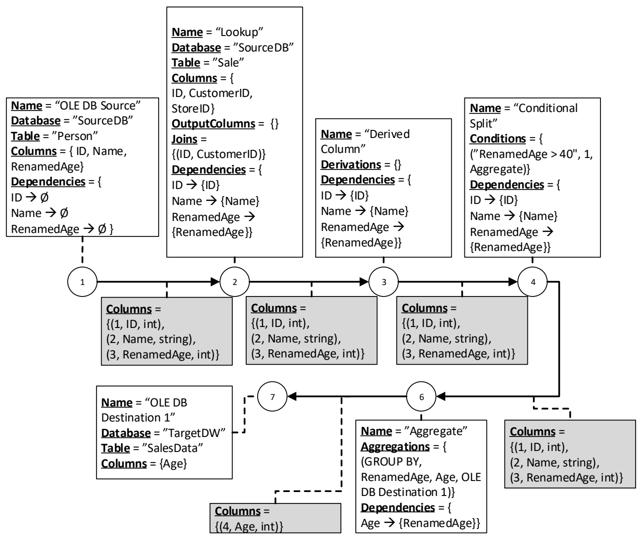


Figure 7: The MAIME graph for the Data Flow task in Figure 6.

Table 3: Time needed for handling EDS schema changes for the three test cases.

Time Elapsed	MAIME	Manual
Test Case 1	4 sec, 4 sec, 4 sec	187 sec, 159 sec, 59 sec
Test Case 2	4 sec, 4 sec, 4 sec	154 sec, 60 sec, 49 sec
Test Case 3	4 sec, 4 sec, 4 sec	23 sec, 13 sec, 10 sec

ing EDS schema changes. For user input, MAIME required on average 4 user inputs, while manually resolving changes required 38 user inputs on average for the third attempt across all 3 cases. This means MAIME on average required 9.5 times less inputs for resolving EDS schema changes.

For the manual work, it took significantly more time the first time the user had to maintain the ETL process. The user did, however, get progressively quicker at maintaining the ETL processes for the second and third attempt, with the best result being from the third attempt. The number of both mouse clicks and keystrokes showed similar results. For these evaluations MAIME is thus able to significantly reduce the amount of time and user input needed for maintaining ETL processes.

In a real life scenario, a company would have to maintain a multitude of ETL processes, which for each could take on average 39.3 seconds to repair as our evaluation showed. However, the timings shown for MAIME in Table 3 (4 seconds) include the time to first load the ETL processes, detect EDS schema changes, and the time the user spent on clicking buttons in the GUI. If there were many ETL processes, it would still be enough to do this once. The reparation algorithm of MAIME uses less than a second in all the cases, and we thus argue that the evaluation setup actually gives the manual process an advantage.

7. RELATED WORK

Related work exists on the topic of maintaining ETL processes when the schema of EDSs change. The framework *Hecataeus* by Papastefanatos et al. [7, 8, 9] analyzes ETL processes by detecting changes to the schema of the EDSs

Table 4: User inputs needed for handling EDS schema changes for the three test cases.

User Inputs	MAIME	Manual
Test Case 1	Keystrokes: 0, 0, 0 Mouse clicks: 4, 4, 4	Keystrokes: 23, 15, 12 Mouse clicks: 88, 85, 38
Test Case 2	Keystrokes: 0, 0, 0 Mouse clicks: 4, 4, 4	Keystrokes: 9, 9, 8 Mouse clicks: 92, 48, 45
Test Case 3	Keystrokes: 0, 0, 0 Mouse clicks: 4, 4, 4	Keystrokes: 0, 0, 0 Mouse clicks: 16, 11, 11

(called evolution events) and proposes changes to the ETL processes based on defined *policies*. The Hecataeus framework abstracts ETL processes as SQL queries and views which are used in a graph to represent the activities. An activity resembles an ETL transformation. Hecataeus's graph is captured by an *ETL Summary* which can have multiple subgraphs for each activity. Each activity can further be broken down, as it includes nodes or entire subgraphs for relations, SQL queries, conditions, and uses views, e.g., for input and output. The types of evolution events taken into account are addition, deletion, and modification. An administrator can annotate nodes and edges with policies for each type of evolution event, while in MAIME policies are provided for each EDS change type and not specifically for each node or edge. The approach taken by MAIME would be preferable if the administrator had to repair a lot of ETL processes. The three Hecataeus policies dictate how ETL processes should be adjusted when an evolution event occurs: (1) *Propagate* readjusts the graph to reflect the new semantics according to the evolution event throughout the rest of the graph, (2) *Block* retains the old semantics, and (3) *Prompt* asks the administrator to choose Propagate or Block. Both Propagate and Prompt are similar in MAIME, whereas their Block attempts to retain the semantics. An extension of Hecataeus has been made for what-if analysis of evolutions of relations, views, or queries in a data-intensive ecosystem [6]. While MAIME models columns on edges, Hecataeus models input and output schemata as subgraphs.

Another framework is *E-ETL* [14, 15] by Wojciechowski which is also able to semi-automatically adapt ETL processes to EDS schema changes. In order to adapt the ETL processes, there are different methods which define how reparations are propagated. E-ETL has the same three policies as Hecataeus, with the exception of the *Block* policy. In E-ETL the Block policy ignores the EDS schema change and does not attempt to modify the graph. The E-ETL framework has multiple ways to handle an EDS schema change. These include *Defined rules* in which the administrator can define rules himself for nodes and edges; *Standard rules* which are the default rules used if the administrator did not define anything; and *Alternative scenarios* where case-based reasoning is used to adjust the ETL process based on solutions to similar problems experienced previously. How these algorithms are used together is not specified. Like Hecataeus, E-ETL models ETL processes through SQL queries.

Related work also exists on how to conceptually model ETL processes using transformations that are commonly used in ETL processes. Trujillo et al. [13] do this by defining a small set of transformations modeled through UML diagrams. Examples of the mechanism include *Aggregation*, *Conversion*, and *Join*. This is in contrast to other frameworks, such as Hecataeus and E-ETL, where ETL processes are described by SQL queries and views, and represented with graphs. Using transformations rather than SQL queries can make conceptual modeling simpler and more maintainable as each transformation has a clear responsibility and provides a higher level of abstraction compared to SQL queries, but at the cost of expressiveness. Trujillo et al. provide a set of general ETL processes that are not specific to any platform, while our transformations are specific to SSIS. Furthermore, instead of using UML diagrams we use property graphs to more closely relate to the internal graphs of SSIS. In [11], conceptual models are specified with a kind of diagrams which then are mapped to logical models where graphs also are used.

Business Intelligence Markup Language (BIML) [1] is an XML dialect that can be used to generate SSIS packages. It can be used to create templates and reusable code snippets to make the creation of SSIS packages easier and less time-consuming. BIML does, however, not handle changes, but can rather re-generate packages when a change occurs.

8. CONCLUSION AND FUTURE WORK

ETL processes can be complex and time-consuming to repair manually. We presented MAIME as a tool to reduce the amount of errors and time spent on maintaining ETL processes. To accomplish this task, we introduced and implemented a graph model as a layer on top of SSIS Data Flow tasks, which simplifies the handling of EDS schema changes.

As we have seen in the evaluation, MAIME required 9.5 times less input from the user and was 9.8 times faster compared to doing it manually with the SSIS Designer tool. MAIME can thus ease the burden of maintaining ETL processes.

There are a number of possible directions for future work. The paper presents an adaptive way of handling source data changes in ETL programs. One direction for future work is to make MAIME more robust against the variance of source types and change types. The adaptivity very often depends on manual decisions made by the ETL administrator. Thus, an interesting extension of this work is to model or formalize the pattern of the manual decisions and predict the next decisions such that the system execution process involves less manual interruptions. Since the system starts to make automatic changes to ETL programs, it is very natural that the ETL administrators considers to "roll-back" the execution of certain groups of ETL programs to the previous state. How to define the transaction process in such an ETL program and how to ensure the possibility of the roll-back action is another direction to explore.

We have so far disregarded propagating changes to the DW, e.g., adding a column to the relevant DW schema. However, an administrator might want to do proper adjustments to the DW in order to capture the correct semantics of an EDS schema change. For example, when a column was inserted in an EDS, it might be reasonable to insert a corresponding column into the DW. This is another direction to investigate. It could also be interesting to investigate how to maintain ETL programs when the target DW is changed.

Another direction for future work is to extend the current prototype. Currently, we extract only one SSIS Data Flow task from the current SSIS package. An improvement would be to extract all SSIS Data Flow tasks from each SSIS package. Furthermore, support for more SSIS transformations could be implemented as well as support for SSIS Control Flow tasks. For example, it would be interesting to have support for more kinds of data sources, such as XML (where a schema can be explicitly given) and a CSV (where a schema typically has to be inferred). We note that a few local software vendors and IT service providers have been in the progress of adopting and extending MAIME into their product and service offerings.

9. ADDITIONAL AUTHORS

Harry Xuegang Huang (Danmarks Nationalbank, email: `xh@nationalbanken.dk`) and Christian Thomsen (Dept. of Comp. Science, Aalborg University, email: `chr@cs.aau.dk`).

10. REFERENCES

[1] Business Intelligence Markup Language. https://www.varigence.com/biml (Last accessed 2016-10-31).

[2] SSIS Designer. https://msdn.microsoft.com/en-us/library/ms137973(v=sql.120).aspx (Last accessed 2016-10-31).

[3] D. Butkevičius et al. MAIME - Maintenance Manager for ETL. Student project d802f16, Dept of Comp. Science, Aalborg University, August 2016. https://github.com/sajens/MAIME/blob/master/Technical-Report.pdf.

[4] R. Kimball and M. Ross. *The Data Warehouse Toolkit*. John Wiley & Sons, 2011.

[5] B. Knight et al. *Professional Microsoft SQL Server 2012 Integration Services*. John Wiley & Sons, 2012.

[6] P. Manousis, P. Vassiliadis, and G. Papastefanatos. Impact Analysis and Policy-Conforming Rewriting of Evolving Data-Intensive Ecosystems. *Journal on Data Semantics*, 4(4), 2015.

[7] G. Papastefanatos et al. Rule-Based Management of Schema Changes at ETL Sources. *ADBIS*, 2010.

[8] G. Papastefanatos et al. What-If Analysis for Data Warehouse Evolution. *DaWaK*, 2007.

[9] G. Papastefanatos et al. Policy-Regulated Management of ETL Evolution. *Journal on Data Semantics XIII*, 2009.

[10] M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 2010.

[11] A. Simitsis. Mapping Conceptual to Logical Models for ETL Processes. *DOLAP*, 2005.

[12] E. Thoo and M. A. Beyer. Gartner's Magic Quadrant for Data Integration Tools. 2014.

[13] J. Trujillo and S. Luján-Mora. A UML Based Approach for Modeling ETL Processes in Data Warehouses. *ER*, 2003.

[14] A. Wojciechowski. E-ETL: Framework for Managing Evolving ETL Workflows. *Foundations of Computing and Decision Sciences*, 38(2), 2013.

[15] A. Wojciechowski. E-ETL Framework: ETL Process Reparation Algorithms Using Case-Based Reasoning. *ADBIS Short Papers and Workshops*, 2015.