DART: A Data Acquisition and Repairing Tool

Bettina Fazzinga, Sergio Flesca, Filippo Furfaro, and Francesco Parisi

DEIS - Università della Calabria
Via Bucci - 87036 Rende (CS) ITALY
{bfazzinga, flesca, furfaro, fparisi}@deis.unical.it

Abstract. An architecture is proposed providing robust data acquisition facilities from input documents containing tabular data. This architecture is based on a data-repairing framework exploiting integrity constraints defined on the input data to support the detection and the repair of inconsistencies in the data arising from errors occurring in the acquisition phase. In particular, a specific but expressive form of integrity constraints (*steady aggregate constraints*) is defined which enables the computation of a repair to be expressed as a mixed integer linear programming problem.

1 Introduction

The need to acquire data from different sources of information often arises in many application scenarios, such as e-procurement, competitor analysis, business intelligence. In several cases these sources are heterogenous documents, possibly represented according to different formats, ranging from paper documents to electronic ones (PDF, MSWord, HTML files). In order to be exploited to provide valuable knowledge, information must be extracted from the original documents and re-organized into a machine-readable format. The problem of defining efficient and effective approaches accomplishing this task is a challenging issue in the context of Information Extraction (IE). Most of traditional IE techniques focus on efficiency, providing unsupervised extraction algorithms which automatically extract records from documents. However, it frequently happens that some of the extracted records are not correctly recognized, i.e. the value of one (or more) field has been misspelled. In several contexts (such as balance analysis) extracted information must be 100% error free in order to be profitably exploited, thus unsupervised approaches are not well-suited. In these cases, data transcription from input documents into a machine-readable format requires massive human intervention, thus compromising efficiency and making valuable resources be wasted. Human intervention is mainly devoted to verifying the correctness of acquired data by comparing them with the content of source documents.

Indeed, if integrity constraints are defined on the input data, this kind of human intervention can be reduced by automatically verifying whether acquired data satisfy these constraints, thus limiting manual corrections to those pieces of acquired data which do not satisfy them. In fact current approaches exploiting integrity constraints on source documents require inconsistent acquired data to be manually edited by a human operator. This editing task is likely to be onerous, since a large amount of data in the input documents need to be accessed and compared with the acquired ones.

The idea underlying this paper is that human intervention can be reduced by exploiting some repairing technique to suggest the "most likely" way of fixing inconsistent data. We introduce the architecture of a system (namely, *DART* - Data Acquisition and Repairing Tool) based on this idea. The motivation of this work and the contribution provided by this system can be better understood after reading the following example, describing a specific application scenario (that is, data acquisition from balance sheets).

Example 1. The balance sheet is a financial statement of a company providing information on what the company owns (its assets), what it owes (its liabilities), and the value of the business to its stockholders. A thorough analysis of a company balance sheet is extremely important for both stock and bond investors, since it allows potential liquidity problems to be detected, thus determining the company financial reliability as well as its ability to satisfy financial obligations.

Figure 1 is a portion of a document containing two *cash budgets* for a firm, each of them related to a year. Each cash budget is a summary of cash flows (receipts, disbursements, and cash balances) over the specified periods.

2003	Receipts	beginning cash	20
		cash sales	100
		receivables	120
		total cash receipts	220
	Disbursements	payment of accounts	120
		capital expenditure	0
		long-term financing	40
		total disbursements	160
	Balance	net cash inflow	60
		ending cash balance	80

2004	Receipts	beginning cash	80
		cash sales	100
		receivables	100
		total cash receipts	200
	Disbursements	payment of accounts	130
		capital expenditure	40
		long-term financing	20
		total disbursements	190
	Balance	net cash inflow	10
		ending cash balance	90

Fig. 1. An input document

This cash budget satisfies the following integrity constraints:

a) for each year, the sum of *cash sales* and *receivables* in section *Receipts* must be equal to *total cash receipts*;
b) for each year, the sum of *payment of accounts*, *capital expenditure* and *long-term financing* must be equal to *total disbursements* (in section *Disbursements*);
c) for each year, the *net cash inflow* must be equal to the difference between *total cash receipts* and *total disbursements*;
d) for each year, the *ending cash balance* must be equal to the sum of the *beginning cash* and the *net cash inflow*;

Generally balance sheets like the ones depicted in Figure 1 are available as paper documents, thus they cannot be automatically processed by balance analysis tools, since these work only on electronic data. In fact, some companies do business acquiring electronic balance data and reselling them in a format suitable for being processed by commercial analysis tools. Currently electronic versions are obtained by means of either human transcriptions or OCR acquisition tools. Both these approaches are likely to result in erroneous acquisition, thus compromising the reliability of the analysis task.

An example of numerical value recognition error occurring during the acquisition phase is the recognition of the value 250 instead of 220 for "total cash receipts" in the

year 2003. Consequently, some constraints are not satisfied on the acquired data for year 2003:

i) in section *Receipts*, the value of *total cash receipts* is not equal to the sum of values of *cash sales* and *receivables*;
ii) the value of *net cash inflow* is not to equal the difference between *total cash receipts* and *total disbursements*.

Furthermore, some symbol recognition errors in non-numerical strings may occur in the acquisition phase. For instance, the item "bgnning cesh" could be recognized instead of "beginning cash". □

DART is a system supporting the acquisition of heterogeneous documents and the supervised repairing of the acquired data. With respect to Example 1, DART will suggest to change the "total cash receipts" value for year 2003 from 250 (i.e. the acquired value) to 220, thus reducing the human intervention, as the human operator is no longer required to access the whole input document to fix acquisition errors making integrity constraints violated. In particular, DART is based on the notion of *card*-minimal repair introduced in [16], where the problem of repairing numerical data which are inconsistent w.r.t. aggregate constraints is addressed. Aggregate constraints defined in [16] can express constraints like those defined in the context of balance-sheet data. The notion of *card*-minimal repair is well-suited for our context, where data inconsistency is due to bad symbol recognition during the acquisition phase. Indeed, applying the *card*-minimal semantics means searching for repairs changing the minimum number of acquired values, which corresponds to the assumption that the minimum number of errors occurred in the acquisition phase.

This work stems from a specific application context, where data to be acquired are balance sheets. In this scenario, the relevant information is formatted according to a tabular layout. Therefore, our acquisition approach is targeted to tabular data. However, observe that this feature does not limit DART to the acquisition of balance sheets, as tabular data often occur in many different application contexts, such as web sites publishing product catalogs.

Related Work

The most widely used notion of repair and consistent query answer on inconsistent data is that of [2]: a repair of an inconsistent database D is a database D' satisfying the given integrity constraints and which is minimally different from D. The consistent answer of a query q posed on D is the answer which is in every result of q on each repair D'. Different approaches to the problem of extracting reliable information from inconsistent data had been introduced in [1, 8].

Based on the notions of repair and consistent query answer introduced in [2], several works investigated more expressive classes of queries and constraints. In [3] extended disjunctive logic programs with exceptions were used for the computation of repairs, and in [4] the evaluation of aggregate queries on inconsistent data was investigated. A further generalization was proposed in [19], where the authors defined a sound and complete technique (in presence of universally quantified constraints) based on the rewriting of constraints into extended disjunctive rules with two different forms of negation

(negation as failure and classical negation). In [9, 10] a practical framework for computing consistent query answer for large relational database has been presented, and the system *Hippo* supporting projection-free relational algebra queries and denial integrity constraints was presented.

All the above-cited approaches assume that tuple insertions and deletions are the basic primitives for repairing inconsistent data. More recently, in [11] a repairing strategy using only tuple deletions was proposed, and in [7, 24, 25] repairs consisting of also value-update operations were considered. The latter are the first approaches performing repairs at the attribute-value level.

In [6] the problem of repairing databases by fixing numerical data at attribute level was investigated in presence of both denial constraints (where built-in comparison predicates are allowed) and a non-linear form of multi-attribute aggregate constraints (when constraints of this form are defined, the repair existence problem was shown to be undecidable). In [16] the problem of repairing and extracting reliable information from data violating a given set of aggregate constraints was investigated. These constraints consist of linear inequalities on aggregate-sum queries issued on measure values stored in the database. This syntactic form enables meaningful constraints to be expressed, such as those of Example 1 as well as other forms which often occur in practice.

In this work we define a restricted class of aggregate constraints and provide a method to compute a *card*-minimal repair defined in [16] (according to the *card*-minimal semantics, a repaired database D' minimally differs from the original database D iff the number of value updates yielding D' is minimum w.r.t. all other possible repairs). We exploit this computation method in the DART system where data are acquired by means of acquisition tool and information is extracted and transformed by a wrapping system.

There has been a lot of research work related to web information extraction. Specialized information extraction procedures, called *wrappers*, represent an effective solution to capture text contents of interest from a source-native format and encode such contents into a structured format suitable for further application-oriented processing. Web wrappers typically exploit markup-tag and lexical token information to infer the template structuring the contents in a web page.

Traditional issues concerning wrapper systems are the development of powerful languages for expressing extraction patterns and the ability of generating these patterns with the lowest human effort [5, 13]. Several systems for generating web wrappers have been recently proposed. We mention here DEByE [20], XWRAP [21], Lixto [5], SCRAP [15, 17], RoadRunner [13]. All these systems do not provide any facility for effectively handling tabular data. Indeed, there are no systems that address data extraction from HTML tables in a satisfactory way. In [14] data extraction from HTML tables with unknown structure is addressed. This system fails when dealing with small tables and in finding mappings related to numeric attributes. A wrapper-learning system called WL2 is presented in [12]. It uses very specific extraction rules which can be applied only to documents which are structurally similar to the documents in the training example.

Main contributions

In this work we introduce a system architecture aiming at supervised acquiring of information encoded into tabular data inside documents with possibly heterogeneous formats. Main novelties of our proposal are the following:

1. Our system embeds a wrapping module for extracting information from tabular data. This module can manage tables having "variable" structures, i.e. tables whose cells can span multiple rows and columns, according to no pre-determined scheme. This is a valuable feature, as all existing wrapping techniques do not work at all or are far from being satisfactory on tabular data without a "rigid" structure.
2. A framework for computing *card*-minimal repairs on wrongly acquired data is introduced to drive the data validation process. This framework exploits a specific form of aggregate constraints (namely, *steady aggregate constraints*) defined on the source documents to check the consistency of the acquired data and computing a repair.

Describing our wrapping technique in detail is out of the scope of this paper. Here we will focus on presenting the architecture of our system and the technique adopted for computing repairs.

2 DART in a nutshell

DART (*Data Acquisition and Repairing Tool*) is a system providing robust data acquisition facilities. It takes as input documents containing tabular data, and it exploits integrity constraints defined on the input data to support the detecting and the repairing of inconsistencies due to errors occurring in the acquisition phase. If acquisition errors are detected, the system proposes a way to correct these errors. Proposed corrections are validated by means of human intervention. In order to detect and repair inconsistencies, integrity constraints are considered expressing algebraic relations among the numerical data reported in the cells of the input tables. These constraints are exploited only to fix the acquired numerical values. Moreover, a dictionary of the terms used in the specific scenario which the input documents refer to is exploited to provide spelling error corrections on non-numerical strings.

Two kinds of user interact with DART, namely the *acquisition designer* and the *operator*. The former is an expert on the application context and specifies the metadata which are used to support both the extraction of tabular data and the repairing process. The latter interacts with the system during the acquisition of each document: if the acquired data need to be corrected, he is prompted to validate proposed corrections.

As shown in Figure 2, DART consists of two macro-modules. The first module takes as input documents containing tabular data and returns a relational database where the extracted tabular data are stored. It performs three steps: it loads the input document and convert it in HTML format, it extracts the tabular data from the HTML document and it transforms them into a database instance. This module exploits metadata specified by the acquisition designer, which describe the structure and the semantics of the input documents[1]. The second module takes as input the database instance D generated by the *acquisition and extraction module*. It locates possible inconsistencies in D and returns a repair for D. Both the inconsistency detection and the repair computation are

[1] As it will be clear in the following, designing an extraction module taking as input HTML documents will make it possible to exploit its features also in Web applications, where the problem of automatically extracting information from HTML pages often arises in many scenarios.

accomplished according to a set of aggregate constraints $\mathcal{AC}$ defined by acquisition designer and represented in the metadata. In more detail, the *repairing module* transforms the problem of finding a *card*-minimal repair[2] for D w.r.t. $\mathcal{AC}$ into an MILP instance (Mixed-Integer Linear Programming problem) and solves it providing a repair for D. The proposed repair is then validated by the operator, who either accepts it or requires to compute a different repair. In fact, it can be the case that the proposed repair is unsatisfactory since the operator realizes that it consists of value updates which do not correspond to the actual content of the source document. In this case the operator inserts further constraints on the acquired data. Basically, he drives the repairing process by specifying the exact values that some pieces of the repaired data must take.

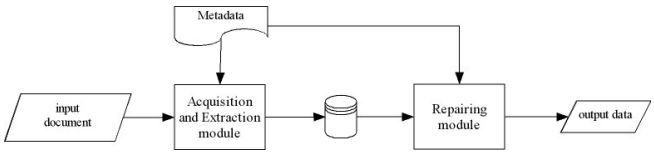

Fig. 2. Data flow in DART

3 Preliminaries

We assume classical notions of database scheme, relational scheme, and relations. In the following we will also use a logical formalism to represent relational databases, and relational schemes will be represented by means of sorted predicates of the form $R(A_1 : \Delta_1, \ldots, A_n : \Delta_n)$, where $A_1, \ldots, A_n$ are attribute names and $\Delta_1, \ldots, \Delta_n$ are the corresponding domains. Each Δ_i can be either $\mathbb{Z}$ (infinite domain of integers), $\mathbb{R}$ (reals), or $\mathbb{S}$ (strings). Domains $\mathbb{R}$ and $\mathbb{Z}$ will be said to be *numerical domains*, and attributes defined over $\mathbb{R}$ or $\mathbb{Z}$ will be said to be *numerical attributes*. Given a ground atom t denoting a tuple, the value of attribute A of t will be denoted as $t[A]$.

Given a database scheme $\mathcal{D}$, we will denote as $\mathcal{M}_\mathcal{D}$ (namely, *Measure attributes*) the set of numerical attributes representing measure data. That is, $\mathcal{M}_\mathcal{D}$ specifies the set of attributes representing measure values, such as weights, lengths, prices, etc. For instance, in Figure 3, $\mathcal{M}_\mathcal{D}$ consists of the only attribute *Value*.

3.1 Aggregate constraints

Given a relational scheme $R(A_1 : \Delta_1, \ldots, A_n : \Delta_n)$, an *attribute expression* on R is defined recursively as follows:

- a numerical constant is an attribute expression;
- each A_i (with $i \in [1..n]$) is an attribute expression;
- $e_1 \psi e_2$ is an attribute expression on R, if e_1, e_2 are attribute expressions on R and ψ is an arithmetic operator in $\{+, -\}$;
- $c \times (e)$ is an attribute expressions on R, if e is an attribute expression on R and c a numerical constant.

[2] As it will be shown in Section 3.2, a *card*-minimal repair for a database is a repair changing the minimum number of values w.r.t. all possible repairs.

Let R be a relational scheme and e an attribute expression on R. An *aggregation function* on R is a function $\chi : (\Lambda_1 \times \cdots \times \Lambda_k) \rightarrow \mathbb{R}$, where each Λ_i is either $\mathbb{Z}$, or $\mathbb{R}$, or $\mathbb{S}$, and it is defined as follows:

$$\chi(x_1, \ldots, x_k) = \begin{array}{l} \texttt{SELECT sum(e)} \\ \texttt{FROM R} \\ \texttt{WHERE } \alpha(x_1, \ldots, x_k) \end{array}$$

where $\alpha(x_1, \ldots, x_k)$ is a boolean formula on $x_1, \ldots, x_k$, constants and attributes of R.

Example 2. Consider the database scheme $\mathcal{D}$ consisting of the single relation scheme *CashBudget(Year, Section, Subsection, Type, Value)*, and its instance reported in Figure 3. This instance represents a possible output of the *acquisition and extraction module* when DART takes as input the document in Figure 1 (it results from the case that a symbol recognition error occurred in the acquisition phase, so that the acquired value of *total cash receipts* is 250 instead of 220). Values '*det*', '*aggr*' and '*drv*' in column *Type* stand for *detail*, *aggregate* and *derived*, respectively. In particular, an item of the table is *aggregate* if it is obtained by aggregating items of type *detail* of the same section, whereas a *derived* item is an item whose value can be computed using the values of other items of any type and belonging to any section.

Year	*Section*	*Subsection*	*Type*	*Value*
2003	Receipts	beginning cash	drv	20
2003	Receipts	cash sales	det	100
2003	Receipts	receivables	det	120
2003	Receipts	total cash receipts	aggr	250
2003	Disbursements	payment of accounts	det	120
2003	Disbursements	capital expenditure	det	0
2003	Disbursements	long-term financing	det	40
2003	Disbursements	total disbursements	aggr	160
2003	Balance	net cash inflow	drv	60
2003	Balance	ending cash balance	drv	80
...	...	...	...	...
2004	Receipts	beginning cash	drv	80
2004	Receipts	cash sales	det	100
2004	Receipts	receivables	det	100
2004	Receipts	total cash receipts	aggr	200
2004	Disbursements	payment of accounts	det	130
2004	Disbursements	capital expenditure	det	40
2004	Disbursements	long-term financing	det	20
2004	Disbursements	total disbursements	aggr	190
2004	Balance	net cash inflow	drv	10
2004	Balance	ending cash balance	drv	90

Fig. 3. A cash budget

The following aggregation functions are defined on the relational scheme *CashBudget*:

$$\chi_1(x, y, z) = \begin{array}{l} \texttt{SELECT sum(Value)} \\ \texttt{FROM CashBudget} \\ \texttt{WHERE Section=x} \\ \quad \texttt{AND Year=y AND Type=z} \end{array} \qquad \chi_2(x, y) = \begin{array}{l} \texttt{SELECT sum(Value)} \\ \texttt{FROM CashBudget} \\ \texttt{WHERE Year = x} \\ \quad \texttt{AND Subsection=y} \end{array}$$

Function χ_1 returns the sum of *Value* of all the tuples having *Section* x, *Year* y and *Type* z. For instance, χ_1('Receipts', '2003', 'det') returns $100 + 120 = 220$, whereas χ_1('Disbursements', '2003', 'aggr') returns 160. Function χ_2 returns the sum of *Value* of all the tuples where *Year=x* and *Subsection=y*. In our running example, as the pair *Year*, *Subsection* is a key for the tuples of *CashBudget*, the sum returned by χ_2 is an attribute value of a single tuple. For instance, χ_2('2003', 'cash sales') returns 100, whereas χ_2('2004', 'net cash inflow') returns 10. □

Definition 1 (Aggregate constraint). *Given a database scheme $\mathcal{D}$, an aggregate constraint on $\mathcal{D}$ is an expression of the form:*

$$\forall x_1, \ldots, x_k \left(\phi(x_1, \ldots, x_k) \implies \sum_{i=1}^{n} c_i \cdot \chi_i(X_i) \leq K \right) \quad (1)$$

where:

1. $c_1, \ldots, c_n, K$ *are constants;*
2. $\phi(x_1, \ldots, x_k)$ *is a conjunction of atoms containing the variables* $x_1, \ldots, x_k$*;*
3. *each* $\chi_i(X_i)$ *is an aggregation function, where* X_i *is a list of variables and constants, and variables appearing in* X_i *are a subset of* $\{x_1, \ldots, x_k\}$.

Given a database D and a set of aggregate constraints $\mathcal{AC}$, we will use the notation $D \models \mathcal{AC}$ [resp. $D \not\models \mathcal{AC}$] to say that D is consistent [resp. inconsistent] w.r.t. $\mathcal{AC}$. Observe that aggregate constraints enable equalities to be expressed as well, since an equality can be viewed as a pair of inequalities. For the sake of brevity, in the following equalities will be written explicitly.

Example 3. Constraints a) and b) defined in Example 1 can be expressed as: for each section and year, the sum of the values of all *detail* items must be equal to the value of the *aggregate* item of the same section and year, that is:
Constraint 1:
$\forall\, x, y, s, t, v \quad CashBudget(y, x, s, t, v) \implies \chi_1(x, y, \text{'det'}) - \chi_1(x, y, \text{'aggr'}) = 0$
□

For the sake of simplicity, in the following we will use a shorter notation for denoting aggregate constraints, where universal quantification is implied and variables in ϕ which do not occur in any aggregation function are replaced with the symbol '_'. For instance, Constraint 1 of Example 3 can be written as:
$CashBudget(y, x, \_, \_, \_) \implies \chi_1(x, y, \text{'}det\text{'}) - \chi_1(x, y, \text{'}aggr\text{'}) = 0$

Example 4. Constraints c) and d) of Example 1 can be expressed as follows:
Constraint 2: *CashBudget*$(x, \_, \_, \_, \_) \implies$
$\chi_2(x, \text{'net cash inflow'}) - (\chi_2(x, \text{'total cash receipts'}) - \chi_2(x, \text{'total disbursements'})) = 0$
Constraint 3: *CashBudget*$(x, \_, \_, \_, \_) \implies$
$\chi_2(x, \text{'ending cash balance'}) - (\chi_2(x, \text{'beginning cash'}) + \chi_2(x, \text{'net cash balance'})) = 0$

3.2 Repairing inconsistent databases

Updates at attribute-level will be used in the following as the basic primitives for repairing data violating aggregate constraints. Given a relational scheme R in the database scheme $\mathcal{D}$, let $\mathcal{M}_R = \{A_1, \ldots, A_k\}$ be the subset of $\mathcal{M}_\mathcal{D}$ containing all the attributes in R belonging to $\mathcal{M}_\mathcal{D}$.

Definition 2 (Atomic update). *Let* $t = R(v_1, \ldots, v_n)$ *be a tuple on the relational scheme* $R(A_1 : \Delta_1, \ldots, A_n : \Delta_n)$*. An* atomic update *on* t *is a triplet* $< t, A_i, v_i' >$*, where* $A_i \in \mathcal{M}_R$ *and* v_i' *is a value in* Δ_i *and* $v_i' \neq v_i$.

Update $u =< t, A_i, v_i' >$ replaces $t[A_i]$ with v_i', thus yielding the tuple $u(t) = R(v_1, \ldots, v_{i-1}, v_i', v_{i+1}, \ldots, v_n)$.

Observe that atomic updates work on the set $\mathcal{M}_R$ of measure attributes, as our framework is based on the assumption that data inconsistency is due to errors in the acquisition phase. Therefore we only consider repairs aiming at re-constructing the correct measure data.

Example 5. Update $u =< t, \textit{Value}, 130 >$ issued on the following tuple:
$t = \textit{CashBudget}(2003, \text{'Receipts'}, \text{'cash sales'}, \text{'det'}, 100)$
returns the tuple: $u(t) = \textit{CashBudget}(2003, \text{'Receipts'}, \text{'cash sales'}, \text{'det'}, 130)$. □

Given an update u, we denote the pair $< \textit{tuple}, \textit{attribute} >$ updated by u as $\lambda(u)$. That is, if $u = <\ t,\ A_i,\ v >$ then $\lambda(u) =< t, A_i >$.

Definition 3 (Consistent database update). *Let D be a database and $U = \{u_1, \ldots, u_n\}$ be a set of atomic updates on tuples of D. The set U is said to be a* consistent database update *iff $\forall\ j, k \in [1..n]$ if $j \neq k$ then $\lambda(u_j) \neq \lambda(u_k)$.*

Informally, a set of atomic updates U is a consistent database update iff for each pair of updates $u_1, u_2 \in U$, u_1 and u_2 do not work on the same tuples, or they change different attributes of the same tuple.

The set of pairs $< \textit{tuple}, \textit{attribute} >$ updated by a consistent database update U will be denoted as $\lambda(U) = \cup_{u_i \in U}\{\lambda(u_i)\}$.

Given a database D and a consistent database update U, performing U on D results in the database $U(D)$ obtained by applying all atomic updates in U.

Definition 4 (Repair). *Let $\mathcal{D}$ be a database scheme, $\mathcal{AC}$ a set of aggregate constraints on $\mathcal{D}$, and D an instance of $\mathcal{D}$ such that $D \not\models \mathcal{AC}$. A* repair *$\rho$ for D is a consistent database update such that $\rho(D) \models \mathcal{AC}$.*

Example 6. A repair ρ for *CashBudget* w.r.t. constraints 1), 2) and 3) consists in decreasing attribute *Value* in the tuple: $t = \textit{CashBudget}(2003, \text{'Receipts'}, \text{'total cash receipts'}, \text{'aggr'}, 250)$ down to 220; that is, $\rho = \{ < t, \textit{Value}, 220 > \}$. □

If a repair exists, different repairs can be performed on D yielding a new database consistent w.r.t. $\mathcal{AC}$, although not all of them can be considered "reasonable". For instance, if a repair exists for D changing only one value in one tuple of D, any repair updating all values in all tuples of D can be reasonably disregarded. To evaluate whether a repair should be considered "relevant" or not, we use an ordering criteria stating that a repair ρ_1 is preferred w.r.t. a repair ρ_2 if the number of changes issued by ρ_1 is less than ρ_2.

Example 7. Another repair for *CashBudget* is: $\rho' = \{\langle t_1, \textit{Value}, 130\rangle, \langle t_2, \textit{Value}, 70\rangle, \langle t_3, \textit{Value}, 190\rangle\}$, where:
$t_1 = \textit{CashBudget}(\ 2003, \text{'Receipts'}, \text{cash sales'}, \text{'det'}, 100)$,
$t_2 = \textit{CashBudget}(\ 2003, \text{'Disbursements'}, \text{'long-term financing'}, \text{'det'}, 40)$,
$t_3 = \textit{CashBudget}\ (\ 2003, \text{'Disbursements'}, \text{'total disbursements'}, \text{'aggr'}, 160)$.
Observe that $\rho < \rho'$, where ρ is the repair defined in Example 6. □

Definition 5 (*Card*-minimal repair). *Let $\mathcal{D}$ be a database scheme, $\mathcal{AC}$ a set of aggregate constraints on $\mathcal{D}$, and D an instance of $\mathcal{D}$. A repair ρ for D w.r.t. $\mathcal{AC}$ is* card*-minimal repair iff there is no repair ρ' for D w.r.t. $\mathcal{AC}$ such that $|\lambda(\rho')| < |\lambda(\rho)|$.*

Example 8. Repair ρ of Example 6 is a *card*-minimal repair. □

Given a database D which is not consistent w.r.t. a set of aggregate constraints $\mathcal{AC}$, different *card*-minimal repairs can exist on D. In our running example, repair ρ of Example 6 is the unique *card*-minimal repair.

In [16] the problem of repairing and extracting reliable information from data violating a given set of aggregate constraints has been investigated. It has been shown that 1) given a database D violating a set of aggregate constraints, deciding whether a repair for D exists is NP-complete, and 2) given a database D violating a set of aggregate constraints and a repair ρ for D, deciding whether ρ is a *card*-minimal repair is coNP-complete. Furthermore, the consistent query answer under both the set-minimal and the *card*-minimal semantics has been studied.
Observe that, as the repair-existence problem is NP-complete, there is no ε-approximation algorithm $\mathcal{A}$ [23] for the computation of a *card*-minimal repair for D, unless $P = NP$. Otherwise, running $\mathcal{A}$ would result in obtaining a possible repair for D (not necessarily a *card*-minimal one) in polynomial time.

4 Steady aggregate constraints

In this section we introduce a restricted form of aggregate constraints, namely *steady aggregate constraints*. On the one hand, steady aggregate constraints are less expressive than (general) aggregate constraints, but, on the other hand, computing a *card*-minimal repair w.r.t. a set of steady aggregate constraints can be accomplished by solving an instance of an MILP (Mixed Integer Linear Programming) problem. This allows us to adopt standard techniques addressing MILP problems to accomplish the computation of a *card*-minimal repair (as it will be clear in the following, this would not be possible for general aggregate constraints). However, observe that the loss in expressiveness is not dramatic, as steady aggregate constraints suffice to express relevant integrity constraints in many real-life scenarios. For instance, all the aggregate constraints introduced in our running example can be expressed by means of steady aggregate constraints.

Before providing the formal definition of steady aggregate constraint, we introduce some preliminary notations.

Given a relational scheme $R(A_1, \ldots, A_n)$ and a conjunction of atoms ϕ containing the atom $R(x_1, \ldots, x_n)$, we say that the attribute A_j corresponds to the variable x_j, for each $j \in [1..n]$. Given an aggregation function χ_i, we will denote as $\mathcal{W}(\chi_i)$ the union of the set of the attributes appearing in the WHERE clause of χ_i and the set of attributes corresponding to variables appearing in the WHERE clause of χ_i. Given an aggregate constraint κ where the aggregation functions $\chi_1, \ldots, \chi_n$ occur, we will denote as $\mathcal{A}(\kappa)$ the set of attributes $\bigcup_{i=1}^{n} \mathcal{W}(\chi_i)$. Given an aggregate constraint κ, we will denote as $\mathcal{J}(\kappa)$ the set of attributes such that for each $A \in \mathcal{J}(\kappa)$ there are two atoms $R_i(x_{i_1}, \ldots, x_{i_n})$ and $R_j(x_{j_1}, \ldots, x_{j_m})$ in $\phi(x_1, \ldots, x_k)$ satisfying both the following conditions:

1. there are $i_l \in [i_1..i_n]$ and $j_h \in [j_1..j_m]$ such that $x_{i_l} = x_{j_h}$;
2. A corresponds to either x_{i_l} or x_{j_h}.

Basically, $\mathcal{J}(\kappa)$ contains attributes A corresponding to variables shared by two atoms in ϕ.

The reason why sets $\mathcal{A}(\kappa)$ and $\mathcal{J}(\kappa)$ have been introduced is that they allow us to detect a useful property. In fact, in the case that $\mathcal{A}(\kappa) \cup \mathcal{J}(\kappa)$ does not contain any

measure attribute, the tuples in the database instance D which are "involved" in κ (i.e. the tuples where ϕ and the WHERE clauses of the aggregation functions in κ evaluate to true) can be detected without looking at the values of their measure attributes. As it will be clear in the following, if this syntactic property holds we can translate κ into a set of linear inequalities and then express the computation of a *card*-minimal repair w.r.t. κ as an instance of MILP.

Definition 6 (Steady aggregate constraint). *Let $\mathcal{D}$ be a database scheme, $\mathcal{M}_\mathcal{D}$ the set of measure attributes of $\mathcal{D}$ and κ an aggregate constraint on $\mathcal{D}$. An aggregate constraint κ is said to be a* steady aggregate constraint *if:*

$$(\mathcal{A}(\kappa) \cup \mathcal{J}(\kappa)) \cap \mathcal{M}_\mathcal{D} = \emptyset \tag{2}$$

Example 9. Consider a database scheme $\mathcal{D}$ containing the relational schemes $R_1(A_1, A_2, A_3)$ and $R_2(A_4, A_5, A_6)$, where $\mathcal{M}_\mathcal{D} = \{A_2, A_4\}$. Let κ be the following aggregate constraint on $\mathcal{D}$:

$$\forall\, x_1, x_2, x_3, x_4, x_5\ (R_1(x_1, x_2, x_3), R_2(x_3, x_4, x_5) \implies \chi(x_2) \leq K) \tag{3}$$

where:

$$\chi(x) = \begin{array}{l}\texttt{SELECT sum}(\texttt{A}_6) \\ \texttt{FROM} \quad \texttt{R}_2 \\ \texttt{WHERE} \quad A_5 = x\end{array}$$

We have that $\mathcal{A}(\kappa) = \{A_5, A_2\}$ and $\mathcal{J}(\kappa) = \{A_3, A_4\}$, therefore κ is not a steady aggregate constraint.

Consider *Constraint 1* of our running example. We have that $\mathcal{A}$(*Constraint 1*) $= \{Year, Section, Type\}$ and $\mathcal{J}$(*Constraint 1*) $= \emptyset$. Since $\mathcal{M}_\mathcal{D} = \{Value\}$, Constraint 1 is a steady aggregate constraint. Similarly, it is straightforward to show that also constraints 2) and 3) are steady aggregate constraints. □

5 Computing a *card*-minimal repair

Several theoretical issues regarding the consistent query answer (CQA) problem have been widely investigated for different classes of constraints, and some techniques for evaluating the CQA have been proposed too (see Related Work section). It can be shown that all complexity results (characterizing either the repair existence problem and the consistent query answer problem) given in [16] (where general aggregate constraints were considered) are still valid for our restricted class of aggregate constraints.

Indeed, in our specific application scenario, we are more interested in computing a repair (fixing all the acquired values) than evaluating whether a single acquired value is "reliable". The main contribution of this section is the definition of a technique for computing a *card*-minimal repair for a database w.r.t a set of steady aggregate constraints, which is based on the translation of the repair-evaluation problem into an instance of a mixed-integer linear programming (MILP) problem [18]. Our technique exploits the restrictions imposed on steady aggregate constraints w.r.t. general aggregate constraints to accomplish the computation of a repair. As it will be clear later, this approach does not work for (general) aggregate constraints.

Consider a database scheme $\mathcal{D}$ and a set of steady aggregate constraints $\mathcal{AC}$ on $\mathcal{D}$. In this case, we can model the problem of finding a *card*-minimal repair as MILP problem (if the domain of numerical attributes is restricted to $\mathbb{Z}$ then it can be formulated as an ILP problem).

We first show how a steady aggregate constraint can be expressed by a set of linear inequalities.

Consider the steady aggregate constraint κ:

$$\forall x_1, \ldots, x_k \left(\phi(x_1, \ldots, x_k) \implies \sum_{i=1}^{n} c_i \cdot \chi_i(y_{i_1}, \ldots, y_{i_{m_i}}) \leq K \right) \tag{4}$$

where $\cup_{i=1}^{n} \{y_{i_1}, \ldots y_{i_{m_i}}\}$ is a subset of $\{x_1, \ldots, x_k\}$ and for each $i \in [1..n]$:

$$\chi_i(y_{i_1}, \ldots y_{i_{m_i}}) = \begin{array}{ll} \texttt{SELECT} & \texttt{sum(e}_\texttt{i}\texttt{)} \\ \texttt{FROM} & \texttt{R}_{\chi_\texttt{i}} \\ \texttt{WHERE} & \alpha_i(y_{i_1}, \ldots, y_{i_{m_i}}) \end{array}$$

Without loss of generality, we assume that each attribute expression e_i occurring in the aggregation function χ_i is either an attribute or a constant.

We associate a variable z_{t,A_j} to each database value $t[A_j]$, where t is a tuple in the database instance D and A_j is an attribute in $\mathcal{M}_\mathcal{D}$. z_{t,A_j} is defined on the same domain as A_j. For every ground substitution θ of $x_1, \ldots, x_k$ such that $\phi(\theta x_1, \ldots, \theta x_k)$ is true, we will denote as T_{χ_i} the set of the tuples involved in the aggregation function χ_i, that is $T_{\chi_i} = \{t : t \models \alpha_i(\theta y_{i_1}, \ldots, \theta y_{i_{m_i}})\}$.
The translation of χ_i, denoted as $\mathcal{P}(\chi_i)$, is defined as follows:

$$\mathcal{P}(\chi_i) = \begin{cases} \sum_{t \in T_{\chi_i}} z_{t,A_j} & \text{if } e_i = A_j; \\ \\ e_i \cdot |T_{\chi_i}| & \text{if } e_i \text{ is a constant.} \end{cases}$$

Starting from $\mathcal{P}(\chi_i)$, the whole constraint κ can be expressed as a set $\mathcal{S}$ of linear inequalities as follows. For every ground substitution θ of $x_1, \ldots, x_k$ such that $\phi(\theta x_1, \ldots, \theta x_k)$ is true, $\mathcal{S}$ contains the following inequality:

$$\sum_{i=1}^{n} c_i \cdot \mathcal{P}(\chi_i) \leq K \tag{5}$$

Observe that this construction is not possible for a non-steady aggregate constraint since, given a database instance D and an aggregation function χ_i in the constraint, we cannot determine T_{χ_i}: changing a measure value might result in changing the set of the tuples involved the aggregation function.

For the sake of simplicity, in the following we associate to each pair $\langle t, A_j \rangle$ an integer index i, therefore we write z_i instead of z_{t,A_j}. If we assume that the number of values involved in constraints in $\mathcal{AC}$ concerning the given database instance D is N then the index i will take values in $[1..N]$.

As shown above, we can translate each steady aggregate constraint into a system linear inequalities. The translation of all aggregate constraints in $\mathcal{AC}$ produces the system of linear inequalities $A \cdot Z \leq B$, where $Z = [z_1, z_2, \ldots, z_N]^T$. This system will be denoted as $\mathcal{S}(\mathcal{AC})$.

Example 10. Consider the database scheme $\mathcal{D}$ of our running example, the database instance in Figure 3 and the set of aggregate constraints $\mathcal{AC}$ consisting of constraints 1), 2) and 3). The values involved in constraints in $\mathcal{AC}$ w.r.t. the given database instance in Figure 3 are as many as the number of tuples, that is $N = 20$. Therefore, z_i, with $i \in [1..20]$ is the variable associated to the database value $t[Value]$, where t is the i-th tuple in Figure 3. For instance, z_2 is the variable associated with the value of attribute *Value* in the tuple $t = \textit{CashBudget}(2003, \text{'Receipts'}, \text{'cash sales'}, \text{'det'}, 100)$.

The translation of constraints 1), 2) and 3) is the following, where we explicitly write equalities instead of inequalities:

$$1) \begin{cases} z_2 + z_3 = z_4 \\ z_5 + z_6 + z_7 = z_8 \\ z_{12} + z_{13} = z_{14} \\ z_{15} + z_{16} + z_{17} = z_{18} \end{cases} \qquad 2) \begin{cases} z_4 - z_8 = z_9 \\ z_{14} - z_{18} = z_{19} \end{cases} \qquad 3) \begin{cases} z_1 - z_9 = z_{10} \\ z_{11} - z_{19} = z_{20} \end{cases}$$

$\mathcal{S}(\mathcal{AC})$ consists of the system obtained by assembling all the equalities reported above (basically, it is the intersection of systems 1,2,3). □

In the following we will denote the current database value corresponding to the variable z_i as v_i. That is, if z_i is associated with $t[A_j]$, then $v_i = t[A_j]$. Every solution s of $\mathcal{S}(\mathcal{AC})$ corresponds to a (possibly non-minimal) repair $\rho(s)$ of D w.r.t. $\mathcal{AC}$. In particular, for each variable z_i which is assigned a value different from v_i, repair $\rho(s)$ contains an atomic update assigning the value z_i to the database item corresponding to z_i.

In order to decide whether a solution s of $\mathcal{S}(\mathcal{AC})$ corresponds to a *card*-minimal repair, we must count the number of variables of s which are assigned a value different from the corresponding source value in D. This is accomplished as follows. For each $i \in [1..N]$, we define a variable $y_i = z_i - v_i$ on the same domain as z_i. Consider the following system of linear inequalities, which will be denoted as $\mathcal{S}'(\mathcal{AC})$:

$$\begin{cases} AZ \leq B \\ y_i = z_i - v_i \quad \forall i \in [1..N] \end{cases} \tag{6}$$

As shown in [22], if a system of equalities has a solution, it has also a solution where each variable takes a value in $[-M, M]$, where M is a constant equal to $n \cdot (ma)^{2m+1}$, where m is the number of equalities, n is the number of variables and a is the maximum value among the modules of the system coefficients. It is straightforward to see that $\mathcal{S}'(\mathcal{AC})$ can be translated into a system of linear equalities in augmented form with $m = N + r$ and $n = 2 \cdot N + r$, where r is the number of rows of A[3].

In order to detect if a variable z_i is assigned (for each solution of $\mathcal{S}'(\mathcal{AC})$ bounded by M) a value different from the original value v_i (that is, if $|y_i| > 0$), a new binary variable δ_i will be defined. δ_i will have value 1 if the value of z_i differs from v_i, 0 otherwise. To express this condition, we add the following constraints to $\mathcal{S}'(\mathcal{AC})$:

$$\begin{cases} y_i \leq M\delta_i \quad \forall i \in [1..N] \\ -M\delta_i \leq y_i \quad \forall i \in [1..N] \\ \delta_i \in \{0, 1\} \quad \forall i \in [1..N] \end{cases} \tag{7}$$

[3] Observe that the size of M is polynomial in the size of the database, as it is bounded by $\log n + (2 \cdot m + 1) \cdot \log(ma)$.

The system obtained by assembling $\mathcal{S}'(\mathcal{AC})$ with inequalities (7) will be denoted as $\mathcal{S}''(\mathcal{AC})$. For each solution s'' of $\mathcal{S}''(\mathcal{AC})$, the following hold: 1) for each z_i which is assigned in s'' a value greater than v_i, the variable δ_i is assigned 1 (this is entailed by constraint $y_i \leq M\delta_i$); 2) for each z_i which is assigned in s'' a value less than v_i, the variable δ_i is assigned 1 (this is entailed by constraint $-M\delta_i \leq y_i$). Moreover, for each z_i which is assigned in s'' the same value as v_i (that is, $y_i = 0$), variable δ_i is assigned either 0 or 1.

Obviously each solution of $\mathcal{S}''(\mathcal{AC})$ corresponds to exactly one solution for $\mathcal{S}(\mathcal{AC})$ (or, analogously, for $\mathcal{S}'(\mathcal{AC})$) with the same values for variables z_i, and, vice versa, for each solution of $\mathcal{S}(\mathcal{AC})$ whose variables are bounded by M there is at least one solution of $\mathcal{S}''(\mathcal{AC})$ with the same values for variables z_i. As solutions of $\mathcal{S}(\mathcal{AC})$ correspond to repairs for D, each solution of $\mathcal{S}''(\mathcal{AC})$ corresponds to a repair ρ for D w.r.t. $\mathcal{AC}$ such that, for each update $u = \langle t, A, v\rangle$ in ρ it holds that $|v| \leq M$. Repairs satisfying this property will be said to be *M-bounded repairs*.

In order to consider only the solutions of $\mathcal{S}''(\mathcal{AC})$ where each δ_i is 0 if $y_i = 0$, we consider the following optimization problem $\mathcal{S}^*(\mathcal{AC})$, whose goal is minimizing the sum of the values assigned to the variables $\delta_1, \ldots, \delta_N$:

$$\min \textstyle\sum_{i=1}^{N} \delta_i \quad \begin{cases} AZ \leq B \\ y_i = z_i - v_i \quad \forall i \in [1..N] \\ y_i - M\delta_i \leq 0 \quad \forall i \in [1..N] \\ -y_i - M\delta_i \leq 0 \quad \forall i \in [1..N] \\ z_i, y_i \in \mathbb{R} \quad \forall i \in I_{\mathbb{R}} \\ z_i, y_i \in \mathbb{Z} \quad \forall i \in I_{\mathbb{Z}} \\ \delta_i \in \{0, 1\} \quad \forall i \in [1..N] \end{cases} \tag{8}$$

where $I_{\mathbb{R}} \subseteq \{1, \ldots, N\}$ and $I_{\mathbb{Z}} \subseteq \{1, \ldots, N\}$ are the sets of the indexes of the variables $z_1, \ldots, z_N$ (and, equivalently, $y_1, \ldots, y_N$) defined on the domains $\mathbb{R}$ and $\mathbb{Z}$, respectively.

It is straightforward to see that any solution of $\mathcal{S}^*(\mathcal{AC})$ corresponds to an M-bounded repair ρ for D w.r.t. $\mathcal{AC}$ having minimum cardinality w.r.t. all M-bounded repairs for D w.r.t. $\mathcal{AC}$. It can be shown that if there is a repair for D w.r.t. $\mathcal{AC}$, then there is an M-bounded *card*-minimal repair ρ^* for D (this follows from Lemma 1 in [16]). This implies that any solution of $\mathcal{S}^*(\mathcal{AC})$ corresponds to a *card*-minimal repair for D w.r.t. $\mathcal{AC}$.

Basically, the minimum value of the objective function of $\mathcal{S}^*(\mathcal{AC})$ represents the number of atomic updates performed by any *card*-minimal repair, whereas the values of variables $z_1, \ldots, z_N, y_1, \ldots, y_N, \delta_1, \ldots, \delta_N$ corresponding to an optimum solution s^* of $\mathcal{S}^*(\mathcal{AC})$ define the atomic updates performed by the *card*-minimal repair $\rho(s^*)$.

Example 11. The optimization problem obtained starting from the database in the Figure 3 of our running example and from the set of steady aggregate constraints consisting of 1), 2) and 3) is shown in Figure 4. Specifically, since it is assumed that the domain of attribute $Value$ of relation $CashBudget$ is $\mathbb{Z}$, then $I_{\mathbb{Z}} = \{1, \ldots, 20\}$ and $I_{\mathbb{R}} = \emptyset$. The value of the constant M is $20 \cdot (28 \cdot 250)^{2 \cdot 28+1}$.

The minimum value of the objective function of this optimization problem is 1 (only $\delta_4 = 1$). This problem admits only one optimum solution where the value of each variable $y_1, \ldots, y_{20}$ is 0 except for y_4 that takes value -30. The *card*-minimal repair corresponding to this solution is that of Example 6. □

$$\min \sum_{i=1}^{20} \delta_i$$

$$\begin{cases} z_2 + z_3 = z_4 & y_4 = z_4 - 250 & y_{15} = z_{15} - 130 \\ z_5 + z_6 + z_7 = z_8 & y_5 = z_5 - 120 & y_{16} = z_{16} - 40 \\ z_{12} + z_{13} = z_{14} & y_6 = z_6 - 0 & y_{17} = z_{17} - 20 \\ z_{15} + z_{16} + z_{17} = z_{18} & y_7 = z_7 - 40 & y_{18} = z_{18} - 190 \\ z_4 - z_8 = z_9 & y_8 = z_8 - 160 & y_{19} = z_{19} - 10 \\ z_{14} - z_{18} = z_{19} & y_9 = z_9 - 60 & y_{20} = z_{20} - 90 \\ z_1 - z_9 = z_{10} & y_{10} = z_{10} - 80 & y_i - M\delta_i \leq 0 \quad \forall i \in [1..20] \\ z_{11} - z_{19} = z_{20} & y_{11} = z_{11} - 80 & -y_i - M\delta_i \leq 0 \quad \forall i \in [1..20] \\ y_1 = z_1 - 20 & y_{12} = z_{12} - 100 & z_i, y_i \in \mathbb{Z} \quad \forall i \in [1..20] \\ y_2 = z_2 - 100 & y_{13} = z_{13} - 100 & \delta_i \in \{0, 1\} \quad \forall i \in [1..20] \\ y_3 = z_3 - 120 & y_{14} = z_{14} - 200 & \end{cases}$$

Fig. 4. MILP-problem instance for the running example

6 DART architecture

The DART architecture is shown in Figure 5, where the organization of both the *Acquisition and extraction module* and the *Repairing module* of Figure 2 are described in more detail. In the following we discuss the tasks accomplished by these modules.

6.1 Acquisition module

This module performs the task of acquiring the information contained in the (either electronic or paper) input documents, and represents it into an electronic document whose format is suitable for the extraction phase accomplished by the *Data Extraction Module*. As the current implementation of DART embeds a wrapper working on HTML documents, input documents which are not already in this format are converted into an HTML document by means of a format-conversion tool (in the current implementation this tool supports the conversion of PDF, MSWord, RTF documents). In particular, paper documents are first digitized and processed by means of an OCR tool (yielding PDF documents) whose output is then processed by the converter.

6.2 Data extraction module

The *Data extraction module* carries out both the information extraction and the database generation tasks. The former task is accomplished by a wrapping sub-module which takes as input the HTML document generated by the *Acquisition module* as well as a set of extraction metadata providing information on the semantics and the structure of data contained into the input document.

Wrapper

Data to be extracted from the input HTML document are contained into tables whose position inside the document is specified inside the extraction metadata. The information encoded into each table is extracted by evaluating whether its rows match some patterns (namely *row patterns*) defining structure and content of the data to be extracted.

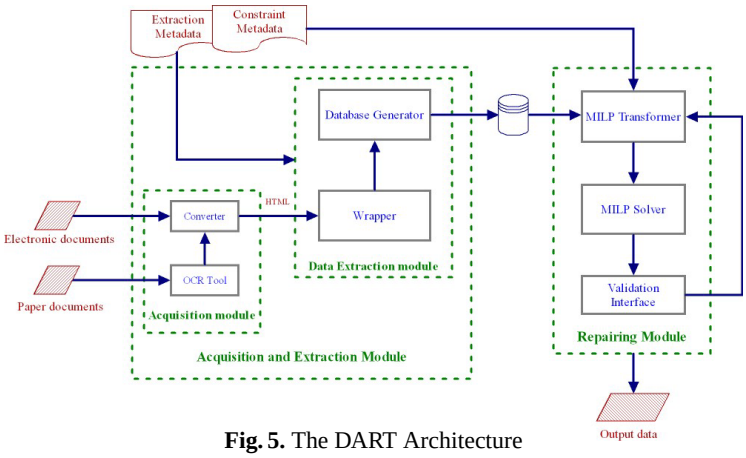

Fig. 5. The DART Architecture

Before explaining how the wrapping sub-module works, we give some details about the set of extraction metadata.

This set contains *domain descriptions*, *row patterns* and *hierarchical relationships*. *Domain descriptions* specify a set of domains and the sets of lexical items that belongs to each domain. For instance, considering the balance sheet analysis context, *Section* and *Subsection* are domains. Some lexical items belonging to the former are "Receipts", "Disbursements", "Balance", whereas some lexical items belonging to the latter are "beginning cash", "receivables", "payment of accounts" and "capital expenditure". In the following we will denote the set of these domains as Dom. *Hierarchical relationships* are relations among lexical items belonging to different domains. For instance, the items "beginning cash", "cash sales", "receivables" and "total cash receipts" are specializations of "Receipts". Figure 6 depicts some domains, some lexical items belonging to them and some hierarchical relationships represented by means of arrows.

A *row pattern* specifies the structure and the content of a table row. The structure is

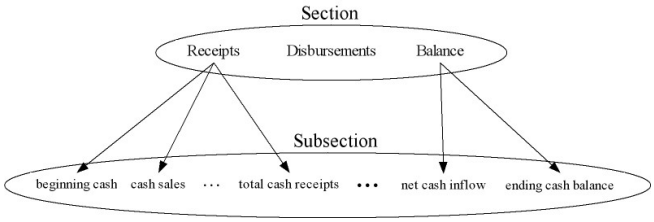

Fig. 6. Domains and hierarchical relationships

given specifying an ordered set of cells. The content of a cell is either a domain belonging to Dom or a *standard domain* such as *Integer*, *String*, etc. A row pattern r matches a row r_t of a table in an input document if r and r_t have the same number of cells and if the content of the i-th cell of r_t matches the domain specified into the i-th cell of r. A row pattern contains an headline indicating the semantics of the domains specified in the cells. The headline will be exploited in the database generation task to construct a relation scheme. In a row pattern, hierarchical relationships can be specified among

lexical items expected in some cells. For instance, it is possible to require that a lexical item expected in a cell must be a generalization of another lexical item required in another cell.

Example 12. Consider the row pattern shown in Figure 7(a). The headline consists of the cells with the dashed border. The row pattern indicates that the rows which must be extracted from the input table consist of 4 cells. In particular, both the first and the last cells specify that a value of type *Integer* is required, and the headline specifies that the first value is interpreted as *Year* and the last as *Value*. The second cell indicates that a lexical item s_1 belonging to the *Section* domain is expected. The third cell imposes a hierarchical relationship, indicated by an arrow. It specifies that a lexical item s_2 belonging to the *Subsection* domain is required, and that s_2 must be specialization of s_1. □

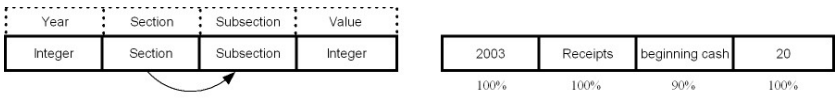

Fig. 7. a)A row pattern b)A row pattern instance

The wrapper takes as input a set of row patterns and the HTML document returned by the acquisition module, and returns a set of *row pattern instances*. A row pattern instance is the result of the matching between a table row and the set of row patterns. First, for each row r_t of the input table, the wrapper identifies the row pattern r that matches r_t at best, i.e. it chooses the row pattern having the most similar structure and the most compatible content w.r.t. r_t. After this choice the wrapper constructs the row pattern instance p relative to r.

In more detail, the evaluation of the matching between a table row and a row pattern yields a score representing the matching degree. The matching is performed comparing the table cells and the corresponding row pattern cells. The comparison between a row pattern cell and an input table cell yields a cell matching score. The whole row pattern instance is associated with a score obtained by applying a suitable *t*-norm to all the matching scores of its cells.

Each cell matching score results from "validating" the string s in the table cell w.r.t the domain d specified in the cell of the row pattern. The validation of s w.r.t. d is accomplished by identifying the item s' in d which is the most similar [4] to s, and returning the similarity degree between s' and s. Given a string s and a domain d we denote the item in d which is the most similar to s as *msi*(d, s). The string [resp. the domain] contained in the i-th cell of a document row r_t [resp. row pattern r] will be denoted as $r_t(i)$ [resp. $r(i)$].

For each document row r_t, the row pattern r for which the matching degree is maximum is chosen. Then a row pattern instance p is constructed, where the i-th cell of p contains the item *msi*$(r(i), r_t(i))$.

Observe that the construction of the row pattern instance is a form of repair on the input data. Indeed, incorrect items in the input tables (i.e. items which do not belong to the corresponding domain in the specified row pattern) are transformed into the most similar valid lexical items.

[4] s' must also satisfy the hierarchical relationships specified in the row pattern.

Finally, we obtain a set of row pattern instances such that each document row is mapped on a row pattern instance.

Example 13. Consider the document in Figure 1 and the row pattern in Figure 7(a). Assume that a symbol recognition error in non-numerical string occurs, like the recognizing of the item "bgnning cesh" instead of "beginning cash". The matching between the first document row and this row pattern returns the row pattern instance in Figure 7(b), where *Integer* in the first cell is bound to "2003", *Section* to "Receipts", *Subsection* to "beginning cash" and *Integer* in the last cell is bound to "20". In Figure 7(b) the matching scores for the cells are also depicted. The third cell score (90%) is lower than the others (100%), since it comes from a non-exact match.

Note that the value "2003" is coded into a multi-row cell of the input table, and it is bound in this row pattern instance since the wrapper considers this value associated to all the document rows which are adjacent to the multi-row cell. □

Database generator

The *Database generator* sub-module takes as input the set of row pattern instances returned by the wrapper module and returns a database instance D conforming to the database scheme defined in the extraction metadata.

Extraction metadata specify also *classification information* providing classification of lexical items depending on the role they play in aggregation constraints. For instance, in Example 1 lexical items in the domain *Subsection* are classified as *detail*, *aggregate* and *derived* items (the meaning of these classes has been defined in Example 2).

The definition of the database scheme contained in the extraction metadata contains both the definition of the relational scheme (that is, the name of the relations and, for each relation, the names of its attributes) and the correspondence between each relation scheme and the row pattern instances taken as input. For instance in our running example the relational scheme specified in the extraction metadata consists of *CashBudget(Year, Section, Subsection, Type, Value)*. Moreover, the extraction metadata contain the specification that attributes *Year, Section, Subsection, Value* correspond to the cells of the row pattern instances described by the same names in the headline, whereas the attribute *Type* is determined by classification information.

Each row pattern instance taken as input is exploited to insert a new tuple in the corresponding relation. For instance, each tuple t in Figure 3 is obtained from a row pattern instance r returned by the wrapper. In particular, the values of the attributes *Year, Section, Subsection, Value* in t are taken from the corresponding cells of the row pattern instance r. Moreover the value of the attribute *Type* is implied by the value of the attribute *Subsection* according to classification information.

6.3 Repairing module

The input of the repairing module is the database D obtained by the data extraction module and a set $\mathcal{AC}$ of steady aggregate constraints implied by the constraint metadata. The repairing module returns a *card*-minimal repair for D w.r.t. $\mathcal{AC}$. This is accomplished by means two phases: first, the problem of finding a *card*-minimal repair for D w.r.t. $\mathcal{AC}$ is translated into an instance of an MILP problem (as we have shown in Section 5), and then such an obtained MILP instance is solved by means of an MILP solver, which is implemented using LINDO API 4.0 (available at `www.lindo.com`).

Validation Interface
The *Validation Interface* is the component allowing the *operator* to interact with DART. When a document is processed, the *Validation Interface* displays the repair computed by the *Repairing module* by showing the suggested set of value updates. Then, the operator examines the proposed repair by comparing every updated value with the corresponding source value in the input document. If the operator verifies that the suggested updated values are equal to the corresponding source values, then the repair is accepted and the repaired data is considered as consistent. Otherwise, a new repair is computed by the *Repairing module* according to operator "instructions". That is, for each suggested update u which has not been accepted by the operator, the operator can specify the actual source value v corresponding to the database item d changed by u. Then an aggregate constraint is added to the set of constraints inputted into the MILP transformer, forcing the value of d to be equal to v. Similarly, accepting an update u on the database item d is translated into an aggregate constraint forcing the value of d to be equal to the value suggested by the repair. After this, a new repair is computed, corresponding to the solution of the new MILP instance obtained by assembling the aggregate constraints resulting from *Constraint Metadata* with those resulting from operator validation. This process goes on until the generated repair is accepted by the operator.

At each iteration, the operator is not requested to validate values which had been already validated in a previous iteration. Moreover, the computation of a repair can be re-started after validating only some of the suggested updates. Every repair is proposed to the operator by displaying its updates in a specific order. That is, an update u_2 is displayed before another update u_2 if the database item d_1 changed by u_1 is involved in a larger number of ground aggregate constraints than the database item d_2 changed by u_2 (i.e. if the variable corresponding to d_1 occurs in the MILP instance in a larger number of inequalities than the variable corresponding to d_2). This ordered displaying is an heuristics which is useful in the case that the operator chooses to re-start the repair computation after a small number of validations, and it aims at finding an acceptable repair in a small number of iterations.

7 Conclusions and future works

DART is currently being developed. Both the *Acquisition and extraction* module and the *Repairing* module have been implemented, but no user-friendly interface is currently available. Preliminary tests show that DART effectively supports the acquisition of balance data, providing the correct repair of wrongly acquired data in a few iterations in most cases. A more extensive experimental evaluation of system effectiveness will be accomplished on larger data sets when a user-friendly visual interface will be available.

References

1. Agarwal, S., Keller, A. M., Wiederhold, G., Saraswat, K., Flexible Relation: An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases, *Proc. International Conference on Data Engineering (ICDE)*, 495–504, 1995.
2. Arenas, M., Bertossi, L. E., Chomicki, J., Consistent Query Answers in Inconsistent Databases, *Proc. Symposium on Principles of Database Systems (PODS)*, 68–79, 1999.
3. Arenas, M., Bertossi, L. E., Chomicki, J., Specifying and Querying Database Repairs using Logic Programs with Exceptions, *Proc. International Conference on Flexible Query Answering Systems (FQAS)*, 27–41, 2000.

4. Arenas, M., Bertossi, L. E., Chomicki, J., He, X., Raghavan, V., Spinrad, J., Scalar aggregation in inconsistent databases, *Theoretical Computer Science*, Vol. 3(296), 405–434, 2003.
5. Baumgartner, R., Flesca, S., Gottlob, G., Visual Web Information Extraction with Lixto, *Proc. International Conference on Very Large Data Bases (VLDB)*, 119–128, 2001.
6. Bertossi, L., Bravo, L., Franconi, E., Lopatenko, A., Complexity and Approximation of Fixing Numerical Attributes in Databases Under Integrity Constraints, *Proc. International Symposium on Database Programming Languages (DBPL)*, 262–278, 2005.
7. Bohannon, P., Flaster, M., Fan, W., Rastogi, R., A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification, *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 143–154, 2005.
8. Bry, F., Query Answering in Information Systems with Integrity Constraints, *IFIP WG 11.5 Working Conference on Integrity and Control in Information Systems*, 113–130, 1997.
9. Chomicki, J., Marcinkowski, J., Staworko, S., Computing consistent query answers using conflict hypergraphs, *Proc. International Conference on Information and Knowledge Management (CIKM)*, 417–426, 2004.
10. Chomicki, J., Marcinkowski, J., Staworko, S., Hippo: A System for Computing Consistent Answers to a Class of SQL Queries, *Proc. International Conference on Extending Database Technology (EDBT)*, 841–844, 2004.
11. Chomicki, J., Marcinkowski, J., Minimal-Change Integrity Maintenance Using Tuple Deletions, *Information and Computation (IC)*, Vol. 197(1-2), 90–121, 2005.
12. Cohen, W. W., Hurst, M., Jensen, L. S., A flexible learning system for wrapping tables and lists in HTML documents, *Proc. International World Wide Web Conference (WWW)*, 232–241, 2002.
13. Crescenzi, V., Mecca, G., Merialdo, P., RoadRunner: Towards Automatic Data Extraction from Large Web Sites, *Proc. International Conference on Very Large Data Bases (VLDB)*, 109–118, 2001.
14. Embley, D. W., Tao, C., Liddle, S. W., Automating the extraction of data from HTML tables with unknown structure, *Data & Knowledge Engineering*, Vol. 54(1) 3–28, 2005.
15. Fazzinga, B., Flesca, S., Tagarelli, A., Learning Robust Web Wrappers, *Proc. International Conference on Database and Expert Systems Applications (DEXA)*, 736–745, 2005.
16. Flesca, S., Furfaro, F., Parisi, F., Consistent Query Answer on Numerical Databases under Aggregate Constraint, *Proc. International Symposium on Database Programming Languages (DBPL)*, 279–294 2005.
17. Flesca, S., Tagarelli, A., Schema-Based Web Wrapping, *Proc. International Conference on Conceptual Modeling (ER)*, 286–299, 2004.
18. Gass, S.I., Linear Programming Methods and Applications, *McGrawHill*, 1985.
19. Greco, G., Greco, S., Zumpano, E., A Logical Framework for Querying and Repairing Inconsistent Databases, *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, Vol. 15(6), 1389–1408, 2003.
20. Laender, A. H. F., Ribeiro-Neto, B. A., da Silva, A. S., DEByE - Data Extraction By Example, *Data & Knowledge Engineering*, Vol. 40(2), 121–154, 2002.
21. Liu, L., Pu, C., Han, W., XWRAP: An XML-Enabled Wrapper Construction System for Web Information Sources, *Proc. International Conference on Data Engineering (ICDE)*, 611–621, 2000.
22. Papadimitriou, C. H., On the complexity of integer programming, *Journal of the Association for Computing Machinery (JACM)*, Vol. 28(4), 765–768, 1981.
23. Papadimitriou, C. H., Computational Complexity, *Addison-Wesley*, 1994.
24. Wijsen, J., Condensed Representation of Database Repairs for Consistent Query Answering, *Proc. International Conference on Database Theory (ICDT)*, 378–393, 2003.
25. Wijsen, J., Making More Out of an Inconsistent Database, *Proc. International Conference on Advances in Databases and Information Systems (ADBIS)*, 291–305, 2004.