Preparation and analysis of multiple source industrial process data

Daniel Gillblad
Per Kreuger
Björn Levin
Åsa Rudström

2005-09-16

SICS Technical Report T2005:10

E-mail: dgi@sics.se, piak@sics.se, blevin@sics.se, asa@sics.se

This work was performed under a grant from the Swedish Foundation for Strategic Research.

Abstract

Industrial process data is often stored in a wide variety of formats and in several different repositories. Efficient methodologies and tools for data preparation and merging are critical for efficient analysis of such data. Experience shows that data analysis projects involving industrial data often spend the major part of their effort on these tasks, leaving little room for model development and generating applications. This paper identifies and classifies the needs and individual steps in data preparation of industrial data. A methodology for data preparation specifically suited for the domain is proposed and a practically useful set of primitive operations to support the methodology is defined. Finally, a proof of concept data preparation system implementing the proposed operations and a scripting facility to support the iterations in the methodology is presented along with a discussion of necessary and desirable properties of such a tool.

Keywords: Data Preparation Methodology, Multiple Source Data Merging, Data Analysis, Data Mining, Data Cleaning, Data Preprocessing

ISSN 1100-3154
ISRN: SICS-T–2005/10-SE

1 Introduction

Data analysis and data mining are traditionally used to extract answers and useful patterns from large amounts of data. In classical statistics, much of the effort goes into deciding what data is necessary, and into designing ways to collect that data. Data mining and knowledge discovery are different in that the analysis has to work with whatever data is available. This data will have been collected for various other purposes and is unlikely to fit the new needs perfectly. Data bases need to be merged, using different formats and ways of identifying items, where special and missing values will most certainly have been encoded differently.

The problem becomes even harder when working with data gathered from industrial production. Such data is generally a mixture of logs from sensors, operations, events and transactions. Several completely different data sources, modes and encodings are not uncommon, making preparation and preprocessing of data essential before an often quite complex merging operation can be performed. Sensor data will reflect that sensors drift or deteriorate and are eventually replaced or serviced, often at irregular intervals. Anomalies showing in one part of the the process may often be due to problems in earlier process steps for which data is sometimes not even available. Still, data analysis is often critical in industrial production, e.g. when trying to increase the efficiency of the production process or finding explanations for the origin of phenomena that are not completely understood.

Data analysis uses an abundance of methods and techniques, but common to all is the need for relevant and well structured data. In practise, many data analysis projects will spend most of their time collecting, sampling, correcting and compiling data. This is even more pronounced when working with data from industrial processes. The quality of the subsequent analysis is also highly dependent on the preparation of data and how the merging of the data sources is performed.

In this paper, we suggest a methodology for data preparation specifically suited for industrial data, based on our experience from analysing data in e.g. the pulp and paper industry, chemical production and steel manufacturing. In particular, the methodology takes into account the iterative nature of the data preparation process and attempts to structure and trace the alternating steps of data preparation and analysis. Based on the methodology, a small set of generic operations is identified such that all transformations in data preparation are special cases of these operations. As proof of concept, a sample implementation of a tool that makes use of the generic operations to support the methodology is presented.

2 Background and related work

Data preparation usually refers to the complete process of constructing well structured data sets for modelling. This includes identification and *acquisi-*

tion of relevant data; *cleaning* data from *noise*, *errors* and *outliers*; *resampling*, *transposition* and other transformations; *dimensionality* or *volume reduction*; and *merging* several data sets into one. All these operations are normally performed in an exploratory and iterative manner. The complete process would benefit greatly from a more structured approach, taking particularities of the domain into account.

Several attempts have been made to describe the entire knowledge discovery and data analysis process [13, 6]. Of these, the CRISP-DM (CRoss-Industry Standard Process for Data Mining) [25] initiative is perhaps the most well known among recent proposals. Other examples include SEMMA, a methodology proposed by SAS Institute Inc. for use with their commercial analysis software [17], and the guidance for performing data mining given in textbooks such as [1, 4]. Most of these process models have roughly the same scope, covering the entire process from problem understanding to model deployment, through steps such as data preparation, modelling and evaluation. The methodology proposed in this paper instead focuses on and details the early parts of the process, referred to in CRISP-DM as *data understanding* and *data preparation*. The guidelines given in [25] for this phase are sensible and well chosen but are described on a level too general to be directly applicable in practise, at least for the specific task of preparing industrial data.

There have also been some more detailed descriptions of data preparation (e.g. [24, 25, 21]) that provide a good understanding of the problem and deliver plenty of tips and descriptions of common practises. However, there is still a lack of a more practical guide to what steps should be performed and when, or how to create an application that supports these steps. These descriptions are generally more focused on corporate databases and business related data than on industrial applications[1].

More recent work also advocates the need for efficient data preparation. Zhang and Yang [29] sum up future directions of data preparation as the construction of interactive and integrated data mining environments, establishing data preparation theories and creating efficient algorithms and systems for single and multiple data sources.

The methodology presented in this paper takes this further with a focus on industrial data, basic methodology and the generic operations supporting the methodology. As far as we know, this is the first attempt to do so. We also emphasize the need for means to document, revoke, and reconstruct a sequence of steps in the data preparation process, and stress the importance of repair and interpretation, effective type analysis, and merging.

[1] Pyle [24] e.g. clearly states that the book deals with corporate databases, while CRISP-DM [25] hints at its business orientation by referring to the first stage of the data mining process as "Business Understanding".

3 A methodology for data preparation

We present a methodology for preparation of industrial data that considers the particularities of the domain and that is likely to generate a useful, complete data set. The different tasks in the preparation of data generally include a number of steps, some of which may have to be iterated several times. It is very common that the need to refine the result of an earlier step becomes apparent only after the completion of a later step, and data preparation is often revisited quite far into the analysis and modelling process. The same applies to early representation choices that turn out to be inappropriate, and late introduction of innovative recoding that simplifies the modelling and analysis.

To handle these problems, the methodology includes a number of distinct operations and explicit iterations. The methodology is intended to provide a structured context to support the data preparation process, and an overview is shown in figure 1. This section describes each step and indicates under what circumstances it will be necessary to back-track to an earlier step.

Problem classification

Problem classification includes identification and classification of the type of data analysis task. Typical tasks in industrial applications include *error* or *anomaly detection*; *fault diagnosis, state classification* or *prediction*; and *proposing corrective actions for errors.* The type of data analysis task entails different choices in encodings and models which will be relevant to later steps. It is usually highly beneficial to have as good an understanding as possible of the problem at this early stage, as this allows for better initial decisions on type determination and representation, reducing the risk of having to revisit and reformulate earlier steps of the process.

Selection of data sources

In industrial applications the collection of data typically involves extracting information from several data bases with different representations and sample rates. Common sources are logs for individual or related groups of sensors, target and measured values for certain parameters related to the type of product produced, and various quality measures. These sources are typically very heterogeneous and need to be individually pre-processed before being merged into a single data source on which the analysis task can be performed. Naturally, the availability and choice of data determines the type of analysis that will later be possible.

Identification of target parameters

For many, although not all data analysis tasks, it is necessary to find a subset of the parameters which uniquely identifies the situations or conditions that the analysis should recognise or predict using the remaining parameters. These

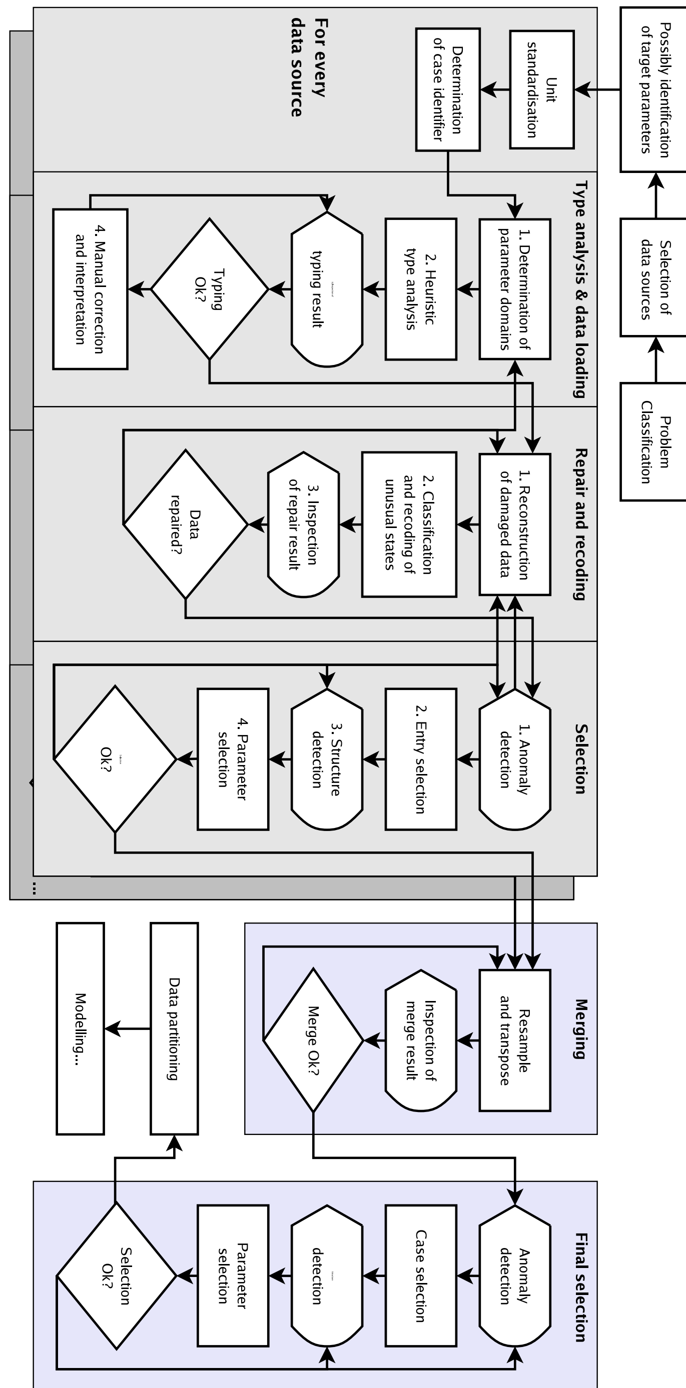

Figure 1: Methodology overview.
Note that the figure illustrates only the main types of iterations and dependencies.

are the *target parameters* for which we seek an explanation, prediction, classification or cause. If it is possible to identify them this at this stage we will benefit from better knowledge of what parameters are important and of suitable transformations.

Unit standardisation in each data source

The standardisation of units in all data sources is necessary to have comparable ranges of values for related parameters in the final merged data set. Usually this is done by a straightforward mapping of values in one unit into another, but often the need to do so is discovered only comparatively late in the process. Because later steps such as anomaly detection and repair may depend on the choices made here, it is still placed early in the data preparation process.

Determination of case identifier

When several data sources have to be merged, it is often necessary to choose and sometimes (re)construct a parameter in each data source that uniquely identifies each case or item in the *final* analysis. This may have to be done in different ways for each data source and may not generate unique entries in every source.

Type analysis of each data source

Analysis of the type, range and format of data represented in each parameter is an often surprisingly difficult problem that will have a significant impact on the result of the later preparation, modelling and analysis steps. For example, the representation and limitations of models of data are often very different depending on whether parameters are continuous or discrete, and results are highly dependent on correct estimates of parameter ranges or domains. Correct type analysis is therefore a very important step in the data preparation process.

Ideally, a domain expert with insight into how the data bases were constructed should be consulted when selecting the most suitable type and range of each parameter. In practise this is seldom feasible, since the domain experts rarely have the necessary time to spare and the number of parameters selected initially can be very large. For this reason it is desirable to simplify and speed up the type analysis step by making it at least semi-automatic.

Type analysis may also be complicated by the occurrence of damaged or missing data, or the use of ad-hoc encodings. A common example of this is the encoding of missing data by using a particular value outside (or even worse, inside) the normal range of the parameter, e.g. string or characters in data that normally consists of integers. This entails that the result of the type analysis step is not only a unique type and range for each parameter, but also input to the repair and recoding step described below. Type analysis involves:

1. Identification of the representation and actual range or domain of each parameter in every data source.

2. Application of some heuristic method to decide if a particular parameter seems to represent samples from a discrete or continuous domain. The heuristic analysis should preferably not only result in a type classification for each parameter, but also give the grounds for this classification. This allows the user to catch type errors at an early stage.

3. Inspection and evaluation of the results.

4. Manual correction of the results and descriptions from the heuristic method using common sense and/or domain knowledge, rudimentary interpretation and possibly classification and encoding of text parameters as e.g. discrete or continuous data and missing values. This step should also include annotation of data that, by the use of domain knowledge, describes both valid parameter ranges and domains and unit/dimension where applicable. This information is highly valuable in later modelling and anomaly detection steps.

The result of the manual correction must again be inspected and evaluated, and the procedure iterated until the result is satisfactory.

Repair and recoding of each data source

The visualisation and inspection of the result of the type analysis generally reveals deficiencies and anomalies in the data that need to be rectified before data preparation and analysis can proceed. This situation is especially pronounced in industrial applications where data often consists of samples of sensor readings which quite frequently drift or report anomalous values, combined with e.g. transmission errors and erroneous manual entries of laboratory measurements and error reports.

Parameters may also be correlated in such a way that the fact that an entry for a particular parameter is missing or is out of range means that values for a subset of the other parameters are useless or should be interpreted in a different way. The detection of such cases generally requires knowledge of the processes involved and/or of idiosyncrasies in the data collection routines. It is also common that the analysis has to proceed quite far before any such anomalies are detected. The repair and recoding step may thus have to be revisited several times, possibly unexpectedly late in the modelling and analysis process.

Methods for automatic detection of possible errors and data cleaning are also highly useful here. Typically, some typographical errors such as misspellings or misplaced decimal points can be comparatively easily detected (see e.g. [22]). If proper operating ranges for parameters have been defined during type analysis, values that fall outside of these can also be found easily. In fact, some errors can be detected just based on the type of the parameter: if a temperature reading shows a value below zero Kelvin, something must be wrong. In process industry data, data reconciliation by e.g. mass and energy balancing may also be useful to detect errors.

In most cases, the repair and interpretation step involves:

1. Reconstruction of missing or obviously damaged parameter data in several data sources.

2. Identification, classification and possibly recoding of process states that give rise to missing and deviant data in several data sources. The representation of missing or irrelevant data should also be taken care of in this step. Not only recoding of parameters, but also introduction of derived parameters should preferably be performed at this point, especially if these parameters are necessary for a later merge operation.

3. Visualisation and validation of repaired data.

All of these steps may have to be iterated and revisited several times.

Selection of parameters and entries in each data source

This step is really a refinement of the choices made in the selection of data sources. The reason to introduce a specific step to revise the choices is that we are, after the type analysis and repair steps, in a better position to assess the quality and usefulness of the data. We may also want to revise or refine the choices made in a first iteration of this step based on results from several steps of preliminary analysis of e.g. correlations between parameters in the data, or exclude subsets of entries identified as anomalies. Obviously, the criteria for selecting entries differ from those used for selecting parameters:

1. Detection of anomalies and artifacts in the data. Automatic or semi-automatic methods for the early detection of anomalies would be highly desirable, but that represents a nontrivial data analysis problem in itself and is out of the scope of this paper. See e.g. [3, 2, 11] for an introduction and further references.

2. Selection and sorting of entries. This generally involves exclusion of entries for which information on the target parameters is missing; filtering of damaged parameters and other artifacts that have not been possible to correct in the repair and recoding step.

3. Structure detection and collection of domain knowledge relevant to parameter selection. This includes correlation analysis, cluster detection and several other techniques (see e.g. [10, 23, 18]).

4. Selection of parameters. This most often based on the result of the type analysis and preliminary structure detection but may also rely on domain knowledge. Different levels of such knowledge should be considered, from naive interpretation of parameter names in the original data sources to more or less complete knowledge of the physical, chemical, economical and social factors involved in the studied process and their representation in the data.

During selection, we might also consider sampling data to reduce the size of the data set. Sampling theory is well described (e.g. [7]) and critical in many data analysis tasks. For industrial data however this is rarely an issue, due to the practical difficulties in acquiring enough data even for a representative view of all aspects of the process.

Merging of several data sources

Merging of data from several data sources is in many cases the single most complex step in the preparation of data. In addition, since data is often given in the form of time series with different sampling rates, there is an obvious need to be able to reliably resample the data using e.g. interpolation, aggregation, integration and related techniques.

Many problems with data in individual data sources become obvious only when merging several data sources into one. Since the merge itself generally is quite a time consuming and error prone operation, it is desirable that the way that the merge is computed is very precisely specified and, most importantly, that it is reproducible in the (very common) case that some earlier data preparation steps have to be revisited.

Common merge problems include cases where one or more data sources contain time series of measurements, possibly with different sampling rates, corresponding to a single entry in another data source. In such cases it is usually necessary to fold (transpose) the time series into a set of additional parameters for the single entry, possibly resampling, integrating and transforming the data in the time series into a more compact or representation believed to be more relevant to the analysis task at hand.

It should be noted that such transformations can easily reduce but also bring out correlations that otherwise could be very difficult to detect by automatic methods. Choices of this kind are really part of the modelling phase, and the correct decisions should be based on thorough knowledge of the problem domain, on the analysis task at hand and on a fair bit of experimentation with alternative representations and models.

Differences and errors in textual representations of identifiers can also cause severe problems while merging. Although such problems are often solvable by defining two corresponding sets of regular expressions for matching, more elaborate methods might have to be applied [15].

Final selection of entries and parameters

At this point, after merging several data sources, it is usually a good idea to revisit the data selection stage, but now for the complete data set. Quite often, a merge operation introduces redundancy in both parameters and entries, and structure that was not visible before the merge (since the data sets were not complete) is now detectable.

The final selection of entries and parameters has the same structure as the one performed for each data source, but some correlations (or lack thereof)

may be difficult to detect before the data sources have been merged. Once the usefulness or uselessness of certain subsets of data has been determined, the selection can either be done for each data source or directly in the merged data. The methodology outlined in figure 1 singles out the second alternative, but going back to the selection step for each data source instead might make sense if the merge operation is simplified by a refined selection on the individual data sources.

Data partitioning

At this stage it is suitable to, if necessary, partition data into *training* (estimation), *validation*, and *testing* data sets for the modelling phase. This partitioning is not necessarily easily made in industrial process data, due to the fact that the production process usually moves quite slowly through a huge state space. The different data sets might have to be separated in time by weeks, or even months, to be useful for estimating a model's actual predictive capabilities. However, estimating the necessary partitioning reliably is difficult, and this step might require a high degree of trial and error.

Modelling

The actual modelling performed after data preparation is not the focus of the work presented in this paper, but, as already pointed out, it is not easy to draw a clear line of where data preparation ends and modelling begins. In fact, we have already touched upon procedures that might be referred to as modelling in earlier steps, perhaps most notably in case and parameter selection, as determining dependencies and finding clusters relies on assumptions and parameterisations of data that are in effect models themselves. For the current discussion it is sufficient to note that at this stage it is often suitable to perform another kind of preparatory modelling: introducing derived parameters, or *features*.

By features we mean new parameters that can, often quite easily, be calculated from other parameters in data. Features may have been introduced earlier to be able to merge data sets with no simple parameter overlap. The reason to introduce them in this step is slightly different in that it is here primarily a means to insert domain knowledge into the data. For example, the temperature over gas volume quotient may very well contain predictive information, and is easily calculated if we know both the volume and temperature. The reason to introduce this feature into the data set is that it represents useful knowledge: although we already know the parameters from which is calculated, a particular analysis method might not easily find this relation by itself.

In industrial data, common features include everything from simple functions such as durations between time points to complex and domain specific correlations, as well as recoding of data in orthogonal bases using e.g. principal component analysis (PCA) [19] or calculating spectral properties using Fourier or wavelet transforms [9]. What kind of derived parameters that are useful for a certain domain is often very difficult to know beforehand. Cataloguing

Preparation step	Operation	Description
Visualisation and inspection	*inspect*	Inspect data
	plot	Plot / visualise data
Type analysis and data loading	*guess*	Guess parameter types
	sample	Extract and load sample of data source
	read	Read complete data
Repair and recoding	*replace*	Replace entries
	derive	Compute derived parameter
	match	Match parameters
Selection	*select*	Select entries and parameters
Merging	*merge*	Resample and merge two data sets

Table 1: Overview of generic operations and transforms

common useful features for a variety of problems and industrial areas would be an interesting research problem in itself. Feature construction and selection, both automatic and manual, are difficult problems, but an introduction to some useful methods can be found in e.g [23, 18].

4 Generic operations and transforms

The methodology outlined in the previous section specifies *what* needs to be done to the data in order to prepare it for modelling and analysis, and proposes an order in which to perform the preparation steps. What remains to describe is *how* to perform the necessary preparations. For this purpose we have identified a small set of generic operations, with a focus on data transformation, of which all data preparation tasks outlined above are special cases. These operations can be composed into a *reproducible sequence of transforms*, thereby making it possible to move back and forth between the preparation steps in a controlled manner.

The set of operations is for practical purposes complete, and although it is deliberately small, we have allowed for some overlap in the functionality of the operations. This implies that a given data preparation task may be accomplished in several ways.

The aim is that, using the generic operations, it should be comparatively quick and easy to get data into a state where anomalies and deficiencies can be subjected to a preliminary analysis. It should then be possible to iteratively devise means to correct and compensate for these anomalies and deficiencies by reiterating and modifying the operations already used, and introducing new operations at the most suitable point in the process. An overview of the generic operations is provided in table 1.

Note that we have listed no supporting operations for problem classification, selection of data sources and identification of target parameters apart from visualisation tools since these steps by their nature require manual inspection

and decisions.

Visualisation, inspection and other supporting operations

Although not described as a separate step in the methodology, visualisation (used here to describe graphical presentation of data) and inspection (used here for viewing data in a textual format) routines are implicitly used by many of the other steps for exploration and verification of results. In fact, throughout the whole data preparation process, visualisation and inspection of both type information and the data itself is essential (see e.g. [12]), not only for discovering structure and dependencies in data, but also to get a more basic understanding of what the attributes represent and on what form. Although visualisation has not been the main focus of our work, a number of operations and application properties that are useful for data preparation can be identified. For very large data sets it can be a good idea to base initial visualisation and inspection on a *sample* of the complete data.

An effective application should include at least mechanisms to *inspect* the type derived for each parameter in the data, and be able to describe and visualise the distribution of both continuous and discrete parameters. Apart from calculating common descriptive statistics measuring e.g. central tendency and dispersion, this includes inspection of the number and distribution of cases for discrete parameters as well as e.g. histograms or dot plots [28] for continuous parameters.

Visualisations, or *plots*, are very useful for determining the nature of the relations in the data and finding potential clusters. Examples of useful visualisations are scatter plots in several dimensions [27], parallel coordinates plots, and simple series plots, possibly combined with correllograms for time dependent data.

Common for all types of visualisations during data preparation is that it should support identification of anomalous data entries and verifying type information. For example, a discrete variable mistakenly represented as continuous can usually easily be identified in a scatter plot, as can potential outliers. Other important properties of the graphics system are interactivity and consistency. Graphics should directly reflect changes to the data or its representation, making it easier for the user to explore the data and explain phenomena in it. Changes made in one visualisation should be noticeable in others that represent the same data.

Visualisation and inspection can be viewed as operations that provide information on useful instantiations of data transformation primitives such as replace, derive and select. This is also true for anomaly detection and structure analysis; these tasks could also be viewed as generic operations on the same level as visualisation and inspection. However, detecting anomalies and structure in data are tasks whose description fall outside the scope of this paper, and a detailed analysis is therefore left to future texts.

Type analysis and data loading

As discussed in the methodology, type analysis is important for understanding, modelling and error correction of the data. Also, just reading data into a database and storing it efficiently generally requires some rudimentary form of interpretation. Since data analysis frequently deals with very large amounts of data it may not be feasible to store e.g. numbers and enumerated types as strings. For these reasons we need a mechanism to specify and preferably also at least semi-automatically derive, or *guess*, parameter types from the data. We also need mechanisms for modifying this type information by explicit specification when necessary. Specification and derivation of field types need to be done for several data sources and there is a need for a mechanism to uniquely refer to the results of such a read operation. This gives us that a generic *read* operation should support

- Sufficiently general input formats, e.g. typical export formats of common database systems and spread sheets.
- The ability to guess the type of each parameter using a suitable heuristic, and then modifying this guess according to a manually generated specification.
- A mechanism to extract only a *sample* of a given data source. This is for practical reasons, as working on complete data sets throughout the whole data preparation process might be too computationally demanding.

These mechanisms can be conveniently provided by parameterising a single read operation for each data source.

Repair, recoding and derivation

There is frequently a need to transform data into another unit or representation. We may also wish to modify a determined type into another one: a discrete parameter into a continuous, a string into an enumerated type or a date etc. This is sometimes most conveniently done by replacing the original value by a value computed from the old by a function, possibly also using values of other parameters as input. Such a mechanism can also be used to discretise a continuous parameter, replacing certain ranges of parameter values with a uniform representation of the fact that the value is missing, or to repair, reconstruct or re-sample data. This operation, *replace*, should

- Use a single transformation function that returns values in accordance with a single resultant type.
- Let the transformation function inspect any number of parameter values in addition to the one being replaced.

Some but not all data analysis methods are capable of automatically detecting correlations between clusters of parameters. However, in many cases it is as already argued it is sometimes useful to introduce derived parameters (features) from ones already present in the data. We wish to allow the analyst the choice of which path to take and therefore provide a mechanism to manually add derived parameters. The primitive operation used to do this, *derive*, should

- Return for each entry in the input data a derived value as a function of a fixed selection of parameters in the old data.
- Return data conforming to a predetermined type, but allow for handling of anomalies in input types.

A special case of replacement and derivation in the above sense is where the input fields contain data that needs to be discretised or classified. One very convenient way to do this is to specify each class as a regular expression that can be matched to the original data. For this reason we propose a specialised form of the derive operation, *match*, which takes a finite list of regular expressions and returns a discrete value corresponding to an enumerated type with as many cases as the number of regular expressions given. Each regular expression corresponds to one discrete class of the original data.

Selection of entries and parameters in a single data source

Frequently the data sources contain redundant or irrelevant parameters, and entries that for some reason or other are not usable. Since this may not be obvious initially it is very useful to be able to select subsets of the data in a traceable and retractable way. Selections of parameters can generally be done using only indexes or parameter names while selection of entries may depend on the actual values in subsets of fields in each entry. The primitive operation *select* allows us to do simultaneous selections of parameters and entries by specifying a list of names or indexes of the parameters to be selected and a boolean filter function used to select entries from the input data.

Merging and resampling

One of the most complex operations in data preparation is the merging of several data sources. It would be almost impossible to provide high level primitives capturing all possible merge and resample needs. For this reason we have chosen to specify a very general mechanism for merging data sources, that for non trivial cases will require programming of functional parameters. It appears, however, to be possible to generalise and with time build up a library of working solutions as generic *merge functions or subroutines.* For the majority of *merge* tasks, we have made the assumption that

- A merge is always done between exactly two data sources. If more than two sources need to be merged this will have to be done by a sequence of

merges[2].

- The entries in the merged data source will always consist of data derivable from exactly one entry in the first input data source and a number of matching entries in the second[3].
- The match should be (efficiently) computable by comparing a specified number of parameter pairs, each pair consisting of a parameter from each input data source.
- For each entry in the first input data source a specified number of derived entries will be computed from the matching entries in the second.

These assumptions allow us to partition the computation of the merge into two steps:

1. Computation of a number of matching entries in the second input data source for each entry in the first input data source. This can be done in a completely general way, using a functional parameter to specify the match criterion.
2. Computation of each derived entry by a supplied merge function from the one entry in the first input data source and the corresponding match of entries in the second. Several types of common merge functions have been identified, such as interpolation, gradients and derivatives.

In short, a primitive merge operation should take as input exactly two data sources, a matching specification and a merge function, and produce a single data source as output.

5 Implementation of a tool for data preparation

The general, primitive operations presented in the previous section could be implemented in many different ways. Several of the operations are similar to those in data base languages such as SQL [8], and if it is the case that the data sources are available as tables in a relational data base the implementation of some of the primitives would be trivial. SQL is, however, in its standard form unsuitable for e.g. advanced forms of merging.

Another common way to execute the primitive operations is to use text processing tools such as *sed*, *awk* and *perl*. Using such tools generally involves a significant amount of programming, and the programs developed may not always

[2] In cases where we wish to re-sample parameters in a single source, we can either use the derive operation or produce a new data source that can be merged with the original one in the manner described here.

[3] This assumption may seem overly restrictive but for all cases we have so far encountered, introducing a derived parameter in one or (rarely) both data sources have allowed us to conveniently use a merge operation with this type of restriction.

be as reusable as could be desired. A system with better integrated transformation primitives, using a higher level of abstraction would be preferable.

To support the methodology in section 3 an interactive tool for data preparation was implemented. This tool is based on a previously developed analysis library, as discussed in [14]. The tool is intended as a framework for data preparation, suitable for both interactive and batch oriented use, facilitating all common preparation tasks. Procedures for cleaning, transformation, selection and integration are provided, implementing the general operations discussed in section 4.

The implementation is directly related to, but takes a different direction than [26], which uses an extension of SQL to provide an environment that supports more general data preparation tasks such as data filtering, reduction and cleaning. [26] makes use of an imperative language, while we propose a functional approach that better suits the use of generic operations, an approach that hopefully will be more general and allow for faster prototyping.

5.1 Requirements specification

A tool for data preparation, analysis and modelling should be designed to fulfil a number of criteria, sometimes in conflict with each other. While keeping the interface simple, it should be possible to handle a large variety of problems. The system should be extensible and adaptable, while maintaining simple data and model abstractions so that problems can be described in a consistent manner. A general data preparation system must also be fully programmable by the user. Almost all data preparation tasks differ slightly from each other, and while *some* subtleties might be possible to handle in a system that cannot be fully programmed or extended, it is bound to one day run into a task that cannot be handled within such a system.

As discussed throughout this paper, data preparation is a highly iterative process and it must be possible to go back and change previous decisions at any step of the process. Hence, a very important requirement is that a support tool should contain mechanisms for documenting each step of the data preparation process in such a way that the process can be easily re-traced and re-run with changed operations.

Another important requirement is that the system should be interactive and support fast prototyping with incremental changes. This implies that a programming language suitable for interpreted use should be chosen for the implementation. This narrows down the scope of possible language families somewhat, but although we would like to use a relatively modern, sound and preferably functional language, we also want to steer clear of languages that strictly use lazy evaluation and instead allow for side effects. Side effects come at a cost, but they make it easier to build a data analysis environment where efficiency in evaluation and memory usage is important. Lazy languages do have problems with liberal stack usage in Data Mining applications [5]. Based on this rationale, the prototype implementation is based on the SCHEME programming language [20], which is both simple, expressive and flexible enough to support

most major programming paradigms.

In addition to the mechanisms dedicated to the data preparation, the implementation should contain facilities for *data abstraction*, *type management* and consistent representation and transparent handling of *missing values.* It should provide a mechanism to refer to the parameters that is independent of the order in which the parameters were given in the data source and allow free naming and index insertion of derived fields. Preferably, values of derived fields should also be computed dynamically and on demand, so that modification of input data automatically results in re-computation of the derived fields whenever their values are requested.

5.2 Design choices

The underlying analysis library [14] of the implementation provides the data abstractions and basic data structures used by all the operations described below. Data is abstracted to a set of *entries* (rows), each consisting of an orthogonal set of *parameters* (columns), the intersection of an entry and a parameter being referred to as a *field.* Each parameter has an associated *type*, used to interpret and encode all the fields of that parameter. The set of types is currently limited to: *continuous* (real), *discrete* (integer), *symbol* (enum), *date*, *time* and *string* (uncoded) although additional annotations of each parameter are provided to encode additional type information such unit, dimension etc.

There are two main types data objects provided by the library. The *storage* data object is designed to provide efficient storage of and access to static data while the *virtual* data object provides a facility to derive new parameters from existing data using a filter construct. This provision is used extensively in the implementation of the data preparation primitives to allow a type of lazy computation of derived parameters and transforms. This also gives us very fast execution of a sequence of operations and inspection of subsets of the data before committing the data to disk.

Most of the data preparation operations described below take an arbitrary data object as input and return a virtual data object as result. This makes the operations clean in the sense that the original data is always unchanged. For most operations it also makes the transformation very fast. However, accessing the data in the resulting virtual data object may be slow. For this reason an explicit *check point* mechanism has also been implemented that will transform the virtual data object to a storage data object and optionally write a representation of it to disk. This means that during the incremental construction of a script to specify a sequence of transformations of the original data, the result of stable parts of the script may be committed to disk. The parts of the script that generate the check point files are then re-executed only when really necessary. The scripting mechanism includes a script compiler and a top level interpreter, which handles the check point mechanism and provides abstractions of the primitive operations as *commands* where the input and output data objects are implicit.

In the following subsections the elements of the implementation most important to data preparation are briefly described and exemplified. The examples are taken from a real case of error detection in steel manufacturing (see [16]). This pilot case was first prepared using e.g. emacs, sed and awk, and then reconstructed using the methods and tools described in this paper. Apart from the time spent on understanding the domain, the time invested in the first iteration of data preparation for this test case was still comparable to the the total time taken to both implement the support tool and preparing the data using the new tool.

5.3 Scripting and maintenance functions

The functions `define-script`,
`script-read`, `script-check` and `script-write` are used to specify a sequence of data transformations as a script, for I/O and for convenience during the iterative development of the final version of the script. Example 5.3 illustrates one typical use of these functions.

```
(define-script (example-1 SaveName)
  (script-read '(((int (0 . 999)) (0 . 3))
                 (string 4 5 6)))
  <command 1>
  <command 2>
  (script-check)
  <command 3>
  ...
  (script-write SaveName))

;;; (example-1 "<file spec>" "<destination path>") ; Call script
```

Example 5.3 defines a script which takes as an argument a file name used to store the result of executing the script. When called it takes in addition and as its first argument an input file specification which is is implicit within in the script but passed on to the `script-read` command. `script-read` should always occur first in the script and will generate the initial data object operated on by the remaining commands in the script.

The `script-read` command will analyse the file indicated by `<file spec>` and generate a tentative typing of all the parameters in the file, modify the typing as specified by its first argument and store the typing information to disk. The type specification in this case indicates that the parameters here given as `0` through `3` should be interpreted as integers in the range `0..999` and the parameters `4`, `5` and `6` as strings of characters. This result is stored to disk as a type file associated with the data file indicated by `<file spec>`. If the script is rerun, the typing will not be performed again unless forced by adding an additional argument to `script-read`. Instead the type-file is used to read in the data.

If the first argument supplied to the script is a sample specification, the sample will be generated and stored to disk. Using either the sample or the original data file the `script-read` command will then generate and pass a storage data object that to `<command 1>`, which will in turn pass its own result as a data object to `<command 2>` and so on.

The result of executing `<command 2>` will be saved as a check point file before being passed to `<command 3>` unless the check-point already exists, in which case neither the `script-read` nor the following commands up to the `script-check` will be executed at all. Instead the the check point file will be used and the result of reading it will be passed to `<command 3>`.

The script returns the result of its last command, in this case a write command with the obvious semantics.

5.4 Data transformations

The following example illustrates a range of simple transformations.

```
(define-script (script-name SaveName)
  ...
  (data-replace (replace-filter >= 0)
    '(cont (0 . 2000)) '(8))
  (data-derive (access -)
    '(int (0 . 172800) "ct0-ch") '(71 29))
  (data-match 85
    '(disc ("No" "hasp.*680" "hasp.*750") "HT"))
  (data-select (lambda (entry) #t)
    '(2 3 4 (8 . 12) 14 (16 . 19) 21 23))
  ...)
```

`data-replace` returns a virtual data object in which the values of all fields for a given set of parameters have been mapped to new values of a corresponding new field type. A transform function given as an argument should take three arguments: the original data object, an entry and a parameter. In example 5.4, the command replaces all values below zero in the parameter `8` with the dedicated symbol for missing values (`#?`) and restricts the type of the parameters to be in the range between `0` and `2000`.

The `data-derive` command returns a virtual data object to which an new parameter of a given type has been added. The values of the fields in the new parameter are derived by a supplied transformation function that is applied to the original data object, an entry and a list of the parameters. In the example, a new parameter named `ct0-ch` is added, with a continuous type with a range between `0` and `172800` computed as the difference of the values of parameters `71` and `29`.

`data-match` adds a new parameter, `HT`, classifying the string values in parameter `85` as one of three values depending on the result of a match with the

regular expressions /hasp.*680/ and /hasp.*750/, defaulting to No if no match is found.

The data-select command, finally, selects a subset of parameters. data-select always returns a virtual data object where only the parameters given and entries that passes a given test function remains. The test function provided in example 5.4 selects the given parameters and leaves all entries of the input data object in the output data object.

As discussed in section 4, merge is one of the most complex operations in data preparation. Therefore, the data-merge command is defined as a general function that in all but trivial cases will require programming of functional parameters.

data-merge merges data from two data-sources, db0 and db1. This is done by adding one or more parameters derived from db1 to a new virtual data object initially identical to db0. First, a match is generated. The match consists of the entries in db1 that match each entry in db0 for each of the parameter pairs given.

The remaining arguments specify how to derive additional parameters. For each parameter specified in db1, a fixed number n of new parameters will be derived from the match. A supplied *value function* computes the value of each new field of a type computed by a corresponding type function. The value function should take four arguments: the data object; a list of entries, which represents the indexes in db1 matched to the current entry in db0; a parameter in db1; and an integer between 0 and n-1. This is typically used to sample or enumerate fields in consecutive entries in db1 to create several new fields in the resulting data object. As an example, to transpose the vector of values of a field in several matched entries into a number of new fields, the following value function could be used:

```
(lambda (db1 entries param i) (db1 (list-ref entries i) param)
```

The next argument is a *type function* is useful for deriving the new field format as a function of the format of the parameter from which its value is derived. To use a unique version of the original parameter name one could e.g. use the following type function:

```
(lambda (param i type dom name)
  (list type dom (format #f "~a-~a" name i)))
```

A small number of useful and convenient merge value functions and utilities are provided by the scripting module. Example 5.4 illustrates the use of data-merge.

```
(define-script (scrpt1 Src1 Src2)
  ...
```

```
;; Merge source 1
(data-merge (scrpt1 ',Src1)  ; Merge subscript 1
  '((0 . 0)(1 . 1))          ; Match parameters
  (lambda (d rws c i)        ; Value function
    (d (list-ref rws i) c))
  (lambda (c i tp dmn nm)    ; Type function
    `(cont (0 . 100) ,nm))
  1 '((2 . 29)))             ; No. of samples & source fields
;; Merge source 2
(data-merge (scpt2 ',Src2)   ; Merge subscript 2
  '((0 . 0)(1 . 1))          ; Match parameters
  (lambda (d rws c i)        ; Value function
    (if (null? rws) "No" "Yes"))
  (lambda (c i tp dmn nm)    ; Type function
    '(disc ("No" "Yes") "Sliver"))
  1 '(0))                    ; No. of samples & source fields
  ...)

;;; (scrpt "<source 0>" "<source 1>" "<source 2>"); Call script
```

The data object produced by the prefix is merged with a data object produced by the script `scrpt1` called with the source file `<source 1>` matching entries for parameters `0` and `1` in both sources. In this case the match is assumed to contain exactly one entry from `<source 1>`, the value of which is appended to the primary data object obtained from `<source 0>` with the same parameter name as in `<source 1>` but with a continuous type with a range between `0` and `100`. Exactly one new parameter for each parameter between `2` and `29` in `<source 1>` is generated.

The result of this operation is then merged with a data object generated by the `scrpt2` called with `<source 2>`, using the same matching parameters, but now adding a boolean parameter depending on if the match is empty or not.

`resample` is a generic sampling utility function for `data-merge` that iterates over the supplied entries and applies a given binary operator to a given parameter `s` of each consecutive entry and a supplied value `l` until it becomes true. It then applies a supplied interpolation function to the resulting entries and parameters.

Typically this is used to search the values of a particular `s` parameter for a value that exceeds the value `l`. The sample function typically interpolates between values of the fields in a given set of parameters for the current and the previous entries using e.g. linear interpolation, gradient, derivatives or splines. Two common sample functions are currently implemented as library routines: `interpolate` that returns a interpolated value of one parameter at which an other parameter would take a given `value`; and `gradient` that returns a gradient of one parameter at which an other parameter would take a given value. Example 5.4 shows the use of `resample`, `interpolate`, and `gradient`.

```
(define-script (scrpt Src1)
  ...
  (data-merge (frnc ',Src1)  ; Merge subscript
    '((0 . 0)(1 . 1))        ; Match parameters
    (lambda (d rws c i)      ; Value function 1
      (resample d rws 5 >= (* (++ i) 100)
      interpolate c 5))
    (lambda (c i tp dmn nm)  ; Type function 1
      `(,tp ,dmn ,(format #f "~a at ~a°C" nm (* (++ i) 100))))
    12 '(3 4 (6 . 14))       ; No. of samples & source fields
  (lambda (d rws c i); Value function 2
    (resample d rws 5 >= (* (++ i) 100) gradient c 3))
  (lambda (c i tp dmn nm)    ; Type function 2
    `(cont ,(if (= c 4) '(0 . 300) '(0 . 30))
    ,(format #f "~a incr/min at ~a°C" nm (* (++ i) 100))))

    12 '(4 5 7 8 9 12 13 14)) ; No. of samples & source fields
  ...)
```

This example illustrates a more complex merge operation with two distinct types of sampling of the data in the secondary data object. In both cases a fixed number of samples (12) are computed from a variable number of matched entries in the secondary object. Both use the library procedure `resample` but with different sample functions (`interpolate` and `gradient`). Most of the data for which the gradient data is computed is also sampled by interpolation.

6 Summary and concluding remarks

In this paper, we have proposed a methodology for data preparation of complex, erroneous data from multiple data sources and shown how the methodology can be supported by a small set of high level and generic operations. By constructing a tool implementing the generic operations we have suggested how the operations can be put into practical use for the preparation of industrial data, prior to modelling and analysis.

Although data preparation and merging is exploratory by nature, the process can benefit greatly from the structured approach suggested in the methodology. Being able to restart the sequence of operations at any point is a major advantage whenever the need to modify, add or delete individual transformation operations is detected later in the preparation or modelling process. This is, as we have already stressed, a very common situation. The implemented tool combines the set of primitives with a scripting facility and checkpoint mechanism as described in section 5, making each step in the process of data preparation *self documenting*, *revocable*, *reproducible* and generally much more efficient in terms of time invested.

Although the primitive operations are derived from the methodology, the methodology itself could be used without the proposed primitives, e.g. in the

context of an SQL data base or when using tools such as *sed*, *awk* or *perl*. However, we argue that a dedicated implementation of this particular set of primitives is an efficient way to reduce the traditionally considerable time spent on data preparation, especially for the analysis of industrial data. The time spent on the preparation phase was reduced by orders of magnitude for our pilot case, and we firmly believe that this will be typical also for future applications. Also, the importance of efficient, semi-automatic type analysis should not be underestimated, as this is usually a difficult problem in this domain.

As for future developments, the methodology and implementation will be applied to other case studies, and based on such results, possibly refined. Different types of automatic or semi-automatic techniques for detection of anomalies and correlations would also clearly facilitate the data preparation process, and could additionally be of use in the modelling steps following the data preparation. For example, extending the use of the type analysis and annotation of parameters with range, domain and unit information would be highly beneficial.

To be truly useful for the general analyst, the implementation of the merge operation requires a library of common merge and interpolation subroutines. Some examples of such library routines were given in section 5.4. Further application of the implementation is expected to result in a larger set of useful such routines being identified. Similarly, a set of useful features for different types of industrial applications should also be identified, along with guidelines of when they are usable.

Future work also includes packaging the software for distribution under GNU General Public License, GPL. Currently, the implementation is available from the authors upon request.

References

[1] P. Adriaans and D. Zantinge. *Data Mining.* Addison-Wesley, Harlow, England, 1996.

[2] C. Aggarwal and P. Yu. Outlier detection for high dimensional data. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 37–46. ACM Press, 2001.

[3] V. Barnett and T. Lewis. *Outliers in Statistical Data.* John Willey, NY USA, 1994.

[4] M. J. A. Berry and G. Linoff. *Data Mining Techniques for Marketing, Sales and Customer Support.* John Wiley & Sons, New York, 1997.

[5] A. Clare and R. D. King. Data mining the yeast genome in a lazy functional language. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, London, 2003. Springer-Verlag.

[6] C. Clifton and B. Thuraisingham. Emerging standards for data mining. *Computer Standards and Interfaces*, 23(3), 2001.

[7] W. G. Cochran. *Sampling Techniques.* Wiley, New York, 1977.

[8] C. J. Date and H. Darwen. *A Guide to the SQL Standard.* Addison-Wesley, Reading, Mass., 1987.

[9] I. Daubechies. The wavelet transform, time-frequency localization and signal analysis. *IEEE Transactions on Information Theory*, 36(5):961–1005, 1990.

[10] R. O. Duda and P. B. Hart. *Pattern Classification and Scene Analysis.* Wiley, New York, 1973.

[11] E. Eskin. Anomaly detection over noisy data using learned probability distributions. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 255–262. Morgan Kaufmann Publishers Inc., 2000.

[12] U. Fayyad, G. Grinstein, and A. Wierse. *Information Visualization in Data Mining and Knowledge Discovery.* Morgan Kaufmann, New York, 2001.

[13] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. Knowledge discovery and data mining: towards a unifying framework. In E. Simoudis, J. Han, and U. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discover and Data Mining*, pages 82–88, Menlo Park, CA, 1996. AAAI Press.

[14] D. Gillblad, A. Holst, P. Kreuger, and B. Levin. The gmdl modeling and analysis system. SICS Technical Report T:2004:17, Swedish Institute of Computer Science, Kista, Sweden, 2004. Available at `ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T-2004-17-SE.pdf`.

[15] M. A. Hernandez and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1997.

[16] A. Holst and P. Kreuger. Butler, fallanalys 1 - outocumpu. Technical Report T2004:07, Swedish Institute of Computer Science SICS, 2004. In Swedish.

[17] SAS Institute Inc. From data to business advantage: Data mining, the semma methodology and the sas system. Sas institute white paper, SAS Institute Inc., Cary, NC, 1998.

[18] A. Jain and D. Zongker. Feature selection: evaluation, application, and small sample performance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(2):153–158, 1997.

[19] I. T. Jolliffe. *Principal Component Analysis.* Springer-Verlag, New York, 1986.

[20] R. Kelsey, W. Clinger, and editors J. Rees. The revised^5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[21] J. Lawrence. Data preparation for a neural network. *AI Expert*, 6:34–41, 1991.

[22] M. L. Lee, H. Lu, T. W. Ling, and Y. T. Ko. Cleansing data for mining and warehousing. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications*, pages 751–760, London, UK, 1999. Springer-Verlag.

[23] H. Liu and H. Motoda. *Feature Selection for Knowledge Discovery and Data Mining*. Kluwer Academic Publishers, Boston, 1998.

[24] D. Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann, New York, 1999.

[25] T. Reinartz, R. Wirth, J. Clinton, T. Khabaza, J. Hejlesen, P. Chapman, and R. Kerber. The current crisp-dm process model for data mining. In F. Wysotzki, P. Geibel, and K Schädler, editors, *Proceedings of the Annual Meeting of the German Machine Learning Group FGML-98*, pages 14–22, Berlin, 1998. Germany Technical Report 98/11 Berlin: Technical University.

[26] K. Sattler and E. Schallehn. A data preparation framework based on a multidatabase language. In *Proc. of Int. Database Engineering and Applications Symposium (IDEAS 2001)*, 2001.

[27] D. F. Swayne, D. Cook, and A. Buja. Xgobi: Interactive dynamic data visualization in the x window system. *Journal of Computational and Graphical Statistics*, 7(1), 1998.

[28] J. Tukey and P. Tukey. Strips displaying empirical distributions: I. textured dot strips. Bellcore technical memorandum, Bellcore, 1990.

[29] S. Zhang, C. Zhang, and Q. Yang. Data preparation for data mining. *Applied Artificial Intelligence*, 17(5–6):375–381, 2003.