MARSHALL PLAN SCHOLARSHIP

Report

prepared for the
Austrian Marshall Plan Foundation

submitted by:
Peter Ott

Supervisor UMD:	Prof. Shuvra S. Bhattacharyya, Ph.D. Will Plishker, Ph.D.
Supervisor FHS:	FH-Prof. DI Dr. Gerhard Jöchtl

Salzburg, September 2011

Acknowledgement

First and foremost, I would like to thank Professor Shuvra Bhattacharyya for the great opportunity to be in his research group and his guidance throughout the course of my research. I would like to thank Dr. William Plishker for his thoughtful advices.

I am very grateful towards the Austrian Marshall Plan Foundation and University of Maryland for making this overwhelming experience possible.

I would like to give a special thanks to FH-Prof. Mag.^{a} Dr.^{in} Gabriele Abermann and Mag.^{a} Teresa Rieger, MPA for their support with the application, as well as my advisor FH-Prof. DI Dr. Gerhard Jöchtl for his assistance.

I would like to thank Akta, LaShanna, Vicky and Kristin for taking good care of me, bringing me for lunch and showing me how to behave like an American. Furthermore, I would like to thank Mary for letting me sublet her room.

Special thanks goes to my awesome roommate Sarah, her family, her boyfriend Mike, Mike's family, especially Robert and Christine, as well as Mike's best friend Jimmy. You guys showed me an unforgettable time.

Details

First Name, Surname:	Peter Ott
University:	University of Maryland
Department:	Department of Electrical and Computer Engineering (ECE)
Research Group:	DSPCAD
Supervisor UMD:	Prof. Shuvra S. Bhattacharyya, Ph.D. Will Plishker, Ph.D.
Supervisor FHS:	FH-Prof. DI Dr. Gerhard Jöchtl

Keywords

1^{st} Keyword:	data conversion
2^{nd} Keyword:	data format description
3^{rd} Keyword:	data generation

Abstract

This report describes the conception and prototypical implementation of a graphical web-based data conversion toolkit. A generic data model allows storing values of arbitrary types, including inter-data dependencies and meta information. Furthermore, an Extensible Markup Language (XML) based model is provided to describe data formats. It enables toolkit components to convert data represented in existing formats both from and to our proposed data model. It is shown that the XML model is Turing complete. In addition, the components of a prototypical implementation are described. It comprises a validator, a data converter and a data generator. In combination with a data editor, parts of our prototypical implementation are employed in a use-case scenario for an industrial application.

Contents

List of Figures

List of Tables

1

Introduction

Whenever huge quantities of information from industrial applications need to be stored, transformation and manipulation of data is a complex issue. Vast amount of data combined with proprietary data formats, which are generated from different data sources, pose a great challenge for data handling. To address this issue, this report presents a fully integrated solution which enables storage and transformation of arbitrary data formats.

The presented solution is developed as part of the *Simulation and Test Automation* Project at the Industrial Information Technology research group at the Salzburg University of Applied Sciences, Salzburg, Austria. It targets the need of generic test management system, which covers the entire product life-cycle. Available solutions on the market focus on specific task and cover parts of the test process chain. These gaps within the test chain slow down the entire process. The project deals with a *Test Framework*, which offers a modular approach in order to adapt to the companies infrastructure. It comprises different solutions to build a consistent test chain, providing a single graphical user interface in order to omit the shift between the programs.

The knowledge about data flow processing and the description of data flow formats at the DSPCAD research group at University of Maryland, College Park, MD, USA helps to improve the innovative approach.

1.1 Related work

Similar to the generic data format conversion approach presented within this report, a transformation language has been realized in XSLT by the W3C [10]. While both allow the conversion of arbitrarily complex data formats, XSLT requires a different data format specification for each transformation direction (input and output). Furthermore, the XSLT implementation does not specify how storage inside the database is handled, while storage structures of the the presented approach are explicitly defined by a generic data model. Therefore, the complexity of the transformations process is reduced to the definition of a XML Model, which is presented in 2.1. Further transformation languages like *biXid* shown in [4], provide similar functionality, whereas *biXid* was designed for the transformation of XML formats only. The same restrictions apply to *XMLTrans* proposed in [12].

Besides that, many proprietary tools for data conversion exist, both open and closed source. Examples of these kind are *XML Convert*[1], *Altova MapForce*[2] or *ETL Suite*[3], which all offer data integration solutions for business applications. While the first one offers XML as the underlying data description form, it lacks filter options, decision-based integration and a graphical user interface. The latter ones rely on direct conversion approaches, mapping the input directly to the output. While this approach might be suitable for one-time data migration scenarios, for continuous data integration a unified data base scheme has to be provided in order to allow the storage of all possible data structures within the company. The extension of an additional data format might result in a change of the underlying business data base.

Most of these tools are capable of converting only between fixed data formats while the presented approach enables users to specify their own formats for input and output. This eliminates the design of a data base scheme by utilizing a generic data model presented in 2.3. Moreover, the transformation language specified by an XML model only requires information about the desired input or output format, respectively. A web-based graphical editor presented in section 2.1 facilitates the usage.

[1] `http://www.unidex.com/xflat.htm`
[2] `http://www.altova.com/mapforce.html`
[3] `http://www.adeptia.com/products/data_transformation.html`

2

Universal Data Conversion and Generation

The graphical conversion approach is based on two essential models: an Extensible Markup Language (XML) model serving as the transformation language and a generic data model data base scheme. The latter allows storage of arbitrary data formats, including meta information and interdependencies. The XML model specifies data formats and serves as the transformation language from and to the original data representations. In addition, a universal data converter and a generator are implemented in order to process arbitrary data both from and to the generic data model using the XML format specification. Fig. 2.1 shows the complete conversion approach.

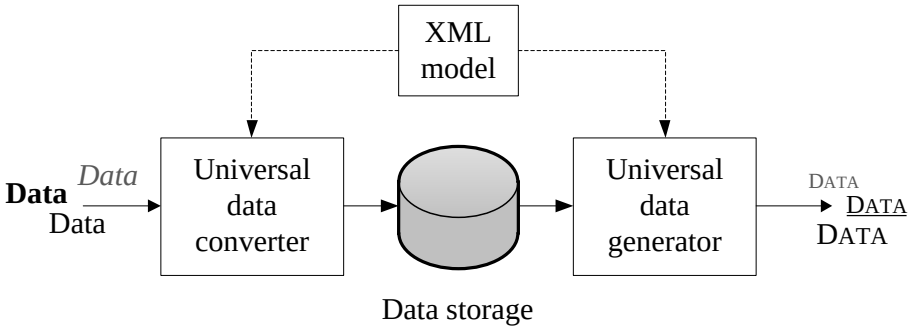

Figure 2.1: Overview of the data conversion and generation framework.

Based on the XML model, a program for transformation only needs to be specified once for a particular data format. The same model will be used for both conversion and generation. Data available in different formats is converted to the data model

representation using the *data converter* which transforms data based on a program specified by the XML model. While the data editor allows for further manipulation of the stored data, the *data generator* enables export into arbitrary formats. In combination with the relational data base utilizing the generic data model, vast amount of data can be processed online directly from the source to the desired output format. Hence, the data conversion tool not only suits one-time data migration purposes, but also allows online processing within heterogeneous data infrastructures.

2.1 The XML Model

In order to specify the format of data to be parsed or generated, an XML model is developed, implemented in form of an XML schema [9] whose complexity is sufficient to model any computer program. By design, programs specified by the model, i.e. data format descriptions, are capable of both parsing and generating data. As a result, only one description of the data format is required for specification and allows both reading and writing data of this format.

For reading a file of a particular format, the data converter parses the file according to the program specified by the XML model, storing all relevant data in the data base, which can be accessed by the program. Similarly, the data generator reads data from the data base and writes the output to a file according to the format specification. As described below, both the data converter and the data generator rely on the same programs which are specified by the XML model.

The XML model describes a set of blocks and block groups which form the core part of the XML model. The functionality of a block is specified by its type and additional parameters, including means to access the data base based on the conceptual generic data model described in section 2.3. Depending on the context (parsing or generation), blocks may behave differently. E.g., when used in a reading context, a certain block will parse a constant character X from an input file. When used in a writing context, the same block will generate a constant character X in an output file.

The default block types which have been specified allow for reading and writing of

constants as well as default data types supported by the data model. In order to ensure the Turing completeness of the specified programs, so called *register* blocks are introduced which can temporarily store values of a predefined type which may be altered by *change* operations. For permanent storage, the data base connection available for the data of each block can be used, relying on a physical data base model of our proposed data model.

In addition to the blocks which are processed one after another, various control flow operations are available. Conditional execution can be modelled by *if* sections which allow for the values of blocks to be compared. This includes data base values, constants and data input and output. Similar to any programming language, the *true* or *false* branch is processed depending on the result of the comparison. Loops can be defined by *for* sections which include a condition for termination similar to *if* sections as well as an *iterate* section which is executed repeatedly until the termination condition occurs.

Since the definition of a data format description via the XML model requires prior knowledge, a visual web-based editor has been implemented as depicted in Fig. 2.2.

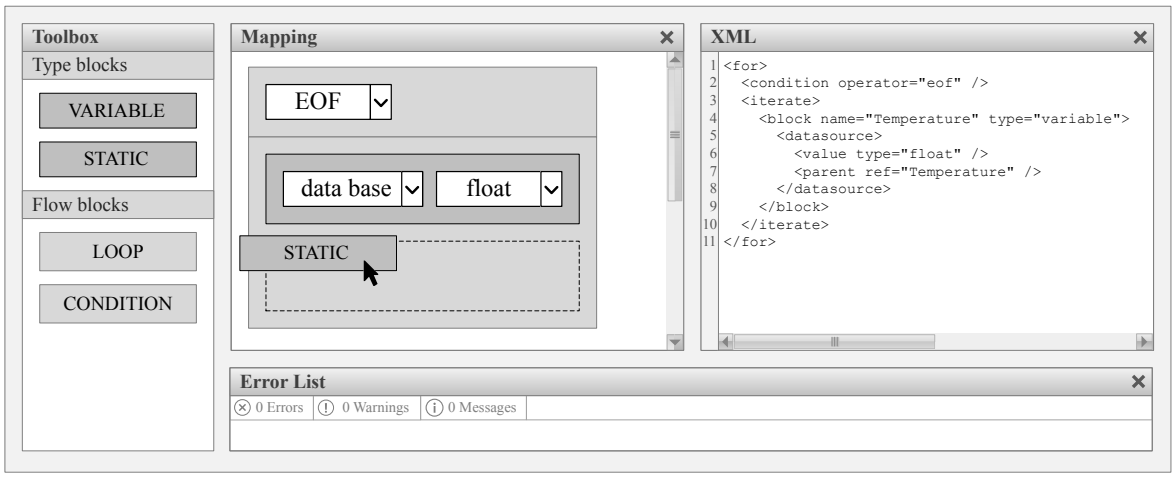

Figure 2.2: Conceptual graphical user interface for the data template creation.

The block concept is directly reflected in the graphical component of the editor. For each block, the required parameters are displayed graphically. Graphical sequencing, nesting and grouping of blocks allow easy prototyping of the XML model only requiring minor programming skills. On the client side, the data format description is visually constructed, while the XML model representation is generated automatically on the server side, using asynchronous web requests [11]. A post model construction validator

ensures full XML model compliance. Validation against our XML schema definition is performed, followed by a parser checking whether additional constraints are met. E.g., such constraints are the string representation of the set of data types allowed by default by the model (e.g. *int32* is supported, while *int128* is not).

The server components are implemented with Microsoft ASP.NET using C# [5]. Therefore, the services are hosted as a web application on an IIS 7 webserver. Javascript on the client-side provides user interface interaction.

The user specifies a data format's structure either by using graphical blocks as depicted in Fig. 2.3 or by defining it directly using our XML model. As soon as the structure has been defined, the data transformation can be repeated for all data of the same format.

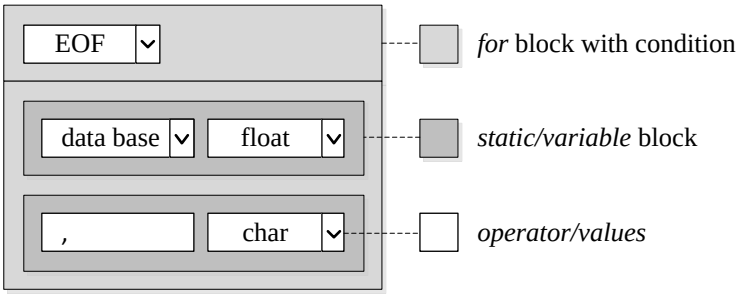

Figure 2.3: Graphical block and control flow representation for a given data format.

2.2 Turing Completeness of the XML Model

The XML model allows specifying both input and output formats with one single description. When only considering it as the specification of an input format, it describes a program or, more abstract, a Turing machine [2] with two bands – one for reading input from an arbitrary file and one for writing intermediate results to a temporary storage or a data base. This includes space for a stack-like structure for the parameters of operations similar to function calls. The latter is included solely for convenience and may also be modelled by a third band of the Turing machine as depicted in Fig. 2.4.

The two (or three) bands are of finite length as they are limited by the number of bytes in the input file and the memory available for storing intermediate results, respectively.

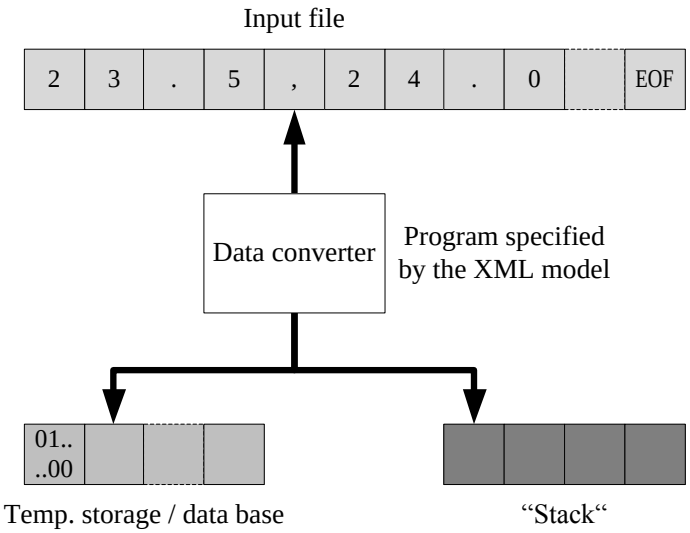

Figure 2.4: XML model based program illustrated as a Turing machine

The alphabet of the input band is the set of UTF-8 characters [14] for text files or single bits or bytes in the case of binary files. While one alphabet would be sufficient, the second one is provided for the sake of convenience. Differently, the band used for temporary storage uses an alphabet which is able to represent all available data types in binary form, i.e. $\{0,1\}^*$ when omitting the limitation of a finite amount of available memory used for representation.

To show the Turing completeness of the XML model, it is sufficient to proof that any partial µ-recursive function (which is equivalent to a Turing machine in terms of computable functions [6]), can be computed by a program specified by the model. This is demonstrated in the subsequent paragraphs by showing that the building blocks of partial µ-recursive functions, i.e. three functions and three operators [7], can be represented by the XML model. The blocks in the XML model are able to store data of different data types, such as integers, floats and many more. However, partial µ-recursive functions operate on natural numbers which can be represented using *uint64* data types for a limited range or using a custom data type of arbitrary size which stores values only limited by the size of the available memory.

Considering partial µ-recursive functions, the constant function can be easily represented by a *register* block with the specified value. It can be referred to in other parts of the program. E.g., to model the successor function, a *change* block will modify

the value of the *register* block by using the *increment* operator. Other operators are allowed by the XML model for convenience, although they are not required for Turing completeness.

Reading data from the input band, i.e. the input file being processed, the projection function of the *outermost* function call can be modeled as a read operation from the input band with a constant or variable offset, specified by the type of the corresponding block. Although the read head of the input band can only be advanced but not reversed, it is possible to save any information from the input band into *register* blocks and retrieve it from them at a later stage. The projection function in general can be modeled by a block group which is invoked by a block group reference. The actual parameters of the function are passed using a *param* type block indicating the number of arguments and the value of each parameter which are pushed onto a stack [13]. As described earlier, this corresponds to the third band of the Turing machine. Function evaluation relies on the return values of the appropriate block group references, passed by *result* type blocks.

Composition can be modeled by repeated invocation of block group references, i.e. a block group reference within a block group reference. Passing parameters is performed in the same way as described above. Similarly, primitive recursion with $h(y, x1, x2, ..., xn)$ being $f(x1, x2, ..., xn)$ for $y = 0$ and $g(y, h(x1, x2, ..., xn), x1, x2, ..., xn)$ otherwise may be represented by a block group consisting of an *if* condition comparing the first parameter (y) to zero. Depending on the result, either a sequence of blocks representing function f (or a block group reference to it) is evaluated, or function h is evaluated repeatedly through recursion and composed with g.

The minimization operator µ can be implemented using a *for* loop. The *for* loop uses two *register* blocks; one for the loop counter and another one for the *exit* flag, respectively. The *exit* flag is a Boolean *register* block which is initialized to *false*. The *for* loop iterates until a condition changes the value of the *exit* flag to *true*. The minimization operator µ is implemented by using the loop counter as an input argument for the function to be minimized. The loop counter is incremented in each iteration of the loop and the function is evaluated. If the result of the function is zero the minimum of the function has been found.

The comparison to zero can be implemented using an *equal* operator within an *if* section, comparing the returned value to the constant zero (specified as described above for constant functions in general). In the case of a successful comparison, a *change* block switches the value of the *exit* flag to *true*. By doing so, the program flow exits the *for* loop at the point at which the function is zero with the loop counter as its argument. This leaves the return value of the minimization operator (i.e., the argument of the function at its minimum) in the loop counter *register* block for further evaluation, e.g. for a *result* block to use it as the return value of a block group.

Since the XML model allows the specification of Turing complete programs, all input files which can be interpreted by computer programs can be parsed by a program which is specified by the proposed XML model. Turing machine equivalence also holds for the generation of files, with the only difference being that the input and output band are switched. Note that the input band in the data generation scenario may refer to the data base and the data stored therein, specifying data access using appropriate blocks. The additional band for function call parameters is optional and remains for convenience, although it could be included into the second band as described above.

2.3 The Generic Data Model

In order to allow the representation of all existing and future data formats, a generic data model is presented in the following. It allows storing arbitrarily complex data types in binary form, including meta data which simplifies the conversion from and to the format specified by the model. Due to the meta data being stored in addition to the actual data, it is possible to represent encrypted and compressed data as well as binary values of arbitrary size and endianness. The binary values represent the actual data which is associated with a predefined (e.g. *int16*, *int32*, *float* according to IEEE 754 [3]) or a custom data type.

To represent list or tree structures, dependencies between single elements can be modelled. Other arbitrary forms of dependencies can be modelled based on dependencies between single elements. To represent a list, the data model is utilizing a series of dependencies in form of a binary tree. The tree (depicted in Fig. 2.5) is constructed

in such a way, that one child node is containing an empty element (*NULL*), while the other child node is containing a value and a sub-tree consisting of all subsequent elements.

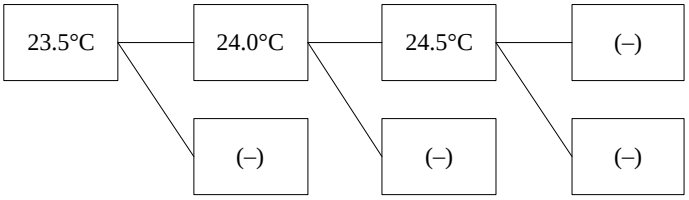

Figure 2.5: Binary tree representation of a list of values.

In addition, association of the data with physical SI [1] units allows for a description of the stored values and automated data conversion, which can be done implicitly up to a certain degree.

Consider the following example: the SI units for meters and seconds are stored by default in the unit table (in a data base derived from the ER model). To specify velocity in meters per second as a new unit, meters are used as nominator and seconds are used as denominator, respectively, both with a cardinality of one and no additional factors.

In a similar way, to represent acceleration in meters per second squared meters are used in the nominator with a cardinality of one and seconds are used in the denominator with a cardinality of two. This approach to represent the units of data values also allows for implicit conversion. E.g., to convert from meters per second to kilometres per hour, kilometres per hour are specified as a new unit with the *meters* per second unit in the nominator with a factor of 3.6 and an offset of zero. This is all the information required for the conversion between data values with those two units.

In addition, each unit can be combined with a prefix, e.g. *micro*, *Mega* etc., which is specified by a prefix symbol and a corresponding factor. The prefix can then be associated with a unit. The combination of both can then again be used to derive new units in the same fashion as described above. Consider the example of a unit specifying cable cross-sections in square millimetres. Creating a *milli* prefix with a factor of 0.001 and combining it to a new unit *mm* (for millimetres), the unit *mm squared* can be derived by using *mm* with a cardinality of two as nominator.

As our proposed data model also allows storing both absolute and relative measurement errors for each data value, it is predestined to represent data sets acquired from measurements of industrial processes, e.g. captured temperature values or voltages. In addition, the data model is able to store whether a single data value is valid and if it specifies an absolute or a relative value. Furthermore, a date and time stamp for each measurement, as well as the sampling interval for a series of measurements can be stored and used for calculating derivatives and accumulations, e.g., the amount of water flowing through a tube in a specified time interval.

To illustrate the process of data storage the following simple example is considered: a comma separated list of temperature values in degrees centigrade is available in form of a plain text file as depicted in Fig. 2.6 (a). In order to read the temperature values into the data base, an XML model for the given file format is specified and used by the data converter. The input file format is specified by an description language using the XML model presented in section 2.1 and called *data template* in the following.

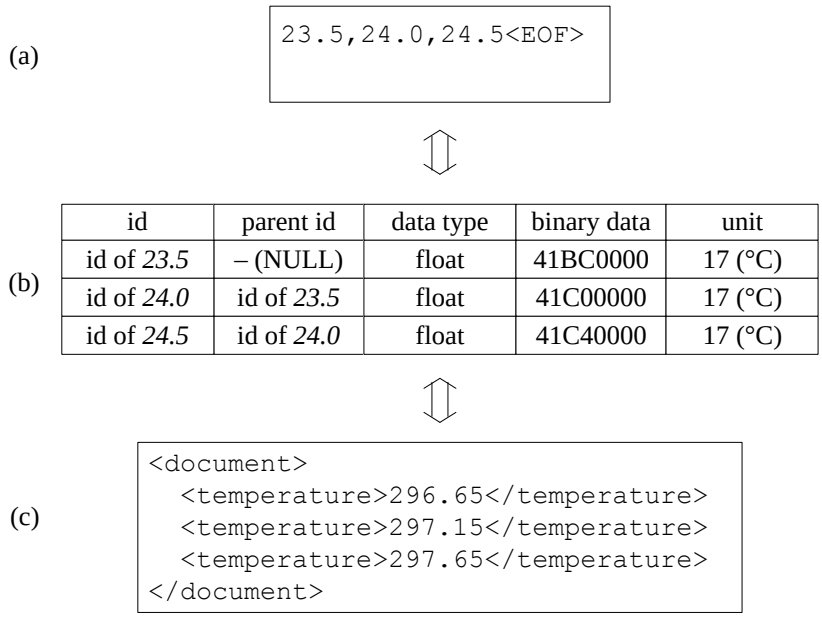

Figure 2.6: Data model representations of two example formats.

The universal data converter enables the transformation of acquired data sets into the generic data model. It uses a parser which is specified graphically or by programming based on the underlying XML model. The data which is processed by the program is then stored in a database which is based on the generic data model for further

processing and storage. This allows for both modification and export into other formats using the data generator.

Using a data template consisting of a *variable* type block referring to a *float* type data base value and a *static* block representing the constant comma (","), allows separating the values in the input file. These two blocks are enclosed by a *for* loop which iterates until the end of the file (*eof*) is reached. The *variable* type block parses the float values and uses the value from the last iteration as the parent element for the value which is read in the current iteration (requiring the data converter to save this value temporarily). Thus, it stores a list in form of a tree as described above. Note, that this example does not consider platform dependent representations of floating point values as this would exceed the scope of this example. Fig. 2.6 (b) shows a representation of temperature values in the proposed data model, including both the data values (represented as binary values according to IEEE754, as hexadecimal values) and SI unit meta data.

When the data converter and an XML model is applied to an arbitrary file containing temperature values which are formatted as described above as input, temperature values of this file will be read into the data base. Since the XML model allows describing the SI unit of each value, the information that all temperature values are stored in degrees centigrade in the data base is included as meta information. The conversion of units, e.g., degrees centigrade to Kelvin, is done implicitly. The basic SI units are stored by default in the database, their arithmetic relation to derive units is in this case specified by an offset (273.15) and a multiplicative factor (1) and used to convert one unit into the other.

The temperature values which are stored in the data base can now be transformed into any data format and any unit representation using the data generator and a corresponding XML specification containing the desired file format and unit specification. Suppose, the desired output is an XML file with temperature values in Kelvin. The output of the given example is shown in Fig. 2.6 (c). The corresponding data template requires an XML model specifying a *document* start and end tag using *static* type blocks and a *for* loop which iterates until there are no child elements left, i.e., the content of the element is *NULL*. Within the *for* loop, there are two *static* type

blocks for the *temperature* start and end tags, respectively, as well as a *variable* block as described in the input example above. The *variable* block specifies a list structure of temperature values of type *float*.

In contrast to the previous example, the SI unit of the block data is set to Kelvin for which an implicit conversion from degrees centigrade exists as described above. Given this XML model, the data generator can write an XML style list of temperature values in Kelvin, only requiring the first temperature value from the data base as a starting point. This can be specified by using the data generator GUI depicted in Fig. 2.2.

The specification of the two formats can also be used to reverse the conversion process, i.e. reading an XML style list of temperature values in Kelvin and writing them as a comma separated list of values in degrees centigrade to a plain text file as depicted in Fig. 2.6. Aside from text and XML files, arbitrarily complex input and output data formats can be processed, including post script files, graphics and many more. Since a single definition of a format through an XML model is being used for both input and output, respectively, the specification process is greatly simplified resulting in significant savings of time and resources.

During the interpretation of the XML file which specifies a data format, the file's elements, i.e. blocks, are represented in form of a tree consisting of their "child" blocks and their properties. Each node is then processed sequentially by processing its child nodes recursively, thereby parsing or generating files, depending on the input direction. For each type of node (*static* block, *for* loop etc.), the software implements the required actions (e.g. reading from or writing to the data base). It continues processing the next node as specified by the data format, e.g. by jumping to another block group reference's node. If all required nodes have been processed and all input/output operations are finished, the parsing or generating process is complete.

The specification of the two formats can also be used to reverse the conversion process, i.e. reading an XML style list of temperature values in Kelvin and writing them as a comma separated list of values in degrees centigrade to a plain text file as depicted in Fig. 2.6. Aside from text and XML files, arbitrarily complex input and output data formats can be processed, including post script files, graphics and many more. Since a single definition of a format through an XML model is being used for both input and output, respectively, the specification process is greatly simplified resulting in significant savings of time and resources.

Data which is stored within the presented data model can be grouped, augmented and changed in an intuitive way. To facilitate this, a graphical component for data manipulation is developed, which endows a broad set of statistical functions to enhance stored data and generate new data sets. E.g., this can be used to supply test data. Unwanted or incomplete data can easily be enriched, changed or padded.

Fig. 2.7 depicts a simplified ER model of the generic data model. Attributes denoted with <Removed> are omitted for the sake of readability, specifying additional properties of the corresponding entities. The central data value entity stores a binary representation of a value, the other entities supply meta information.

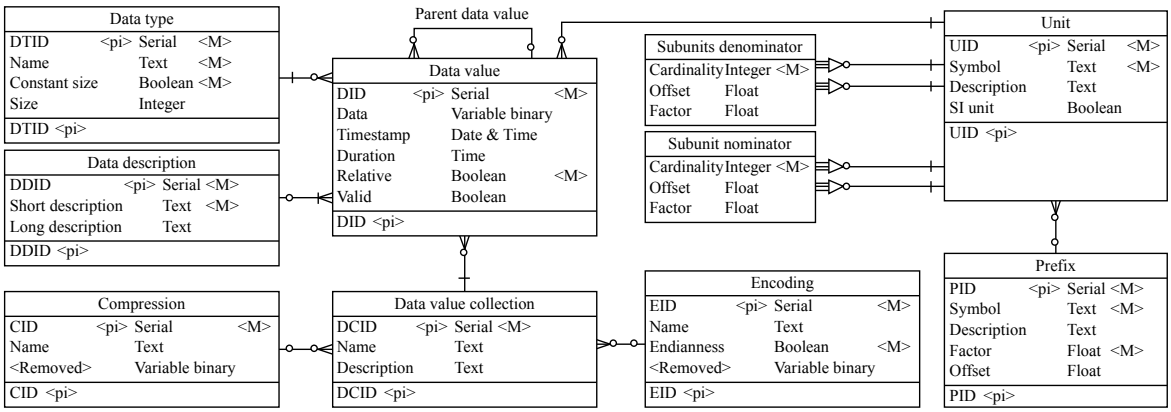

Figure 2.7: Simplified ER model of the generic data model.

2.4 Implementation

A prototypical implementation of the web-based graphical data converter was accomplished using state-of-the-art technology.

The web application is based on the Microsoft ASP.NET technology hosted on an Internet Information Services webserver. The ASP.NET technology allows the creation of dynamic web pages, using any .NET programming language on the server-side and JavaScript (JS) on the client-side. For the server-side C# is chosen.

The repository for the acquired data sets utilizes a relational Data Base Management System (DBMS). In this case a Microsoft SQL Server 2008 R2 is used. The the connection to the DBMS is established by using the Microsoft Language Integrated Query (LINQ) technology. It allows data base queries using native C# syntax. Furthermore, C# allows the native serialization of object instances to a binary format in the data base. Hence, no additional data transformation is required to store the data. An additional security enhancement is given, as the serialized objects need the corresponding class definition to deserialize them.

The graphical user interface presented in section 2.1 is based on the ExtJS[1] framework. It offers a comprehensive toolset of graphical user controls for web applications. It features rapid-prototyping capability and cross-browser support. Therefore, the ExtJS framework facilitates the graphical user interface design. A better integration of the ExtJS framework into the ASP.NET environment is given by the Ext.NET[2] framework, which provides the ExtJS functionality as ASP.NET markup.

Fig. 2.8 depicts the prototypical graphical user interface of the data template editor. On the left side within the toolbox are three example blocks given, to model a data format. Further blocks should follow. The blocks can be dragged into the body area of the transformation language generator. Configurations can be made according to the block type. The configuration is context sensitive to the previously made configurations to prevent possible model errors and warnings. An XML view is available, which is not illustrated within Fig. 2.8.

[1] `http://www.sencha.com/products/extjs`
[2] `http://www.ext.net`

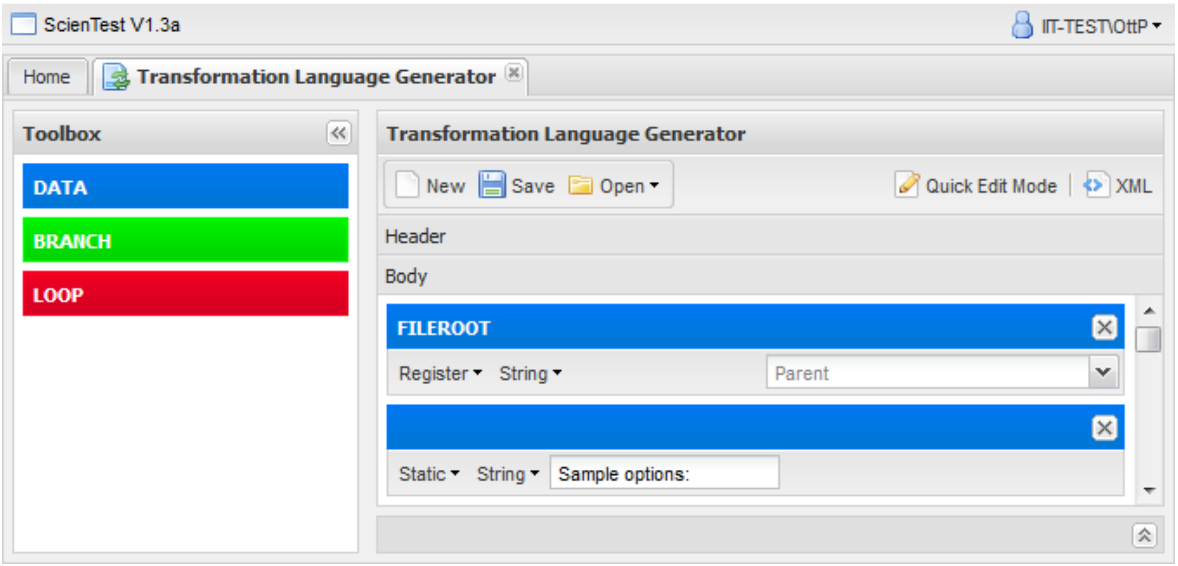

Figure 2.8: Graphical user interface of the web-based data template editor.

A data model validator is depicted in Fig. 2.9 to verify the compliance of the data template against the XML model. The validator is realized as an asynchronous web request. The XML data of the data template is posted to the service and processed on the server-side. The result are three different categories of messages. The first provides additional processing information and design improvements. The second category comprise warnings which may cause minor drawbacks or design flaws. The third category covers errors, which causes unstable behaviour of the data converter and generator. Hence, data templates with errors can not be used for the conversion process.

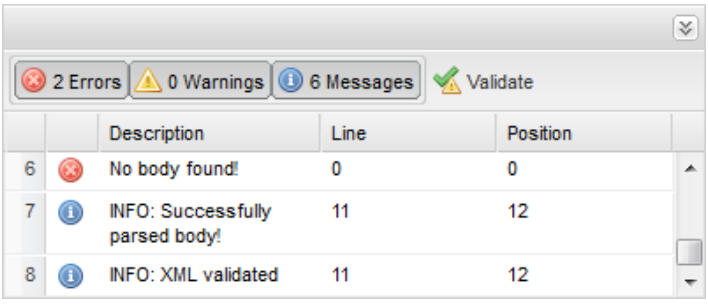

Figure 2.9: Graphical user interface of the validator within the data template editor.

The graphical user interface of the data conversion process depicted in Fig. 2.10 is utilizing HTML5 features to give the user a desktop application like experience. It is possible to drag files onto the web application user interface. This facilitates the

uploading and processing of data files. For non-HTML5 compliant browsers, a fall-back mechanism is implemented, which checks the browser of the supported upload types. Available upload mechanisms in order of best experience are HTML5, Flash, Silverlight and standard HTML file upload. Status updates for each individual task of the current processing step is pushed to the client user interface using Reverse Ajax technology. This omits the need of page refreshes.

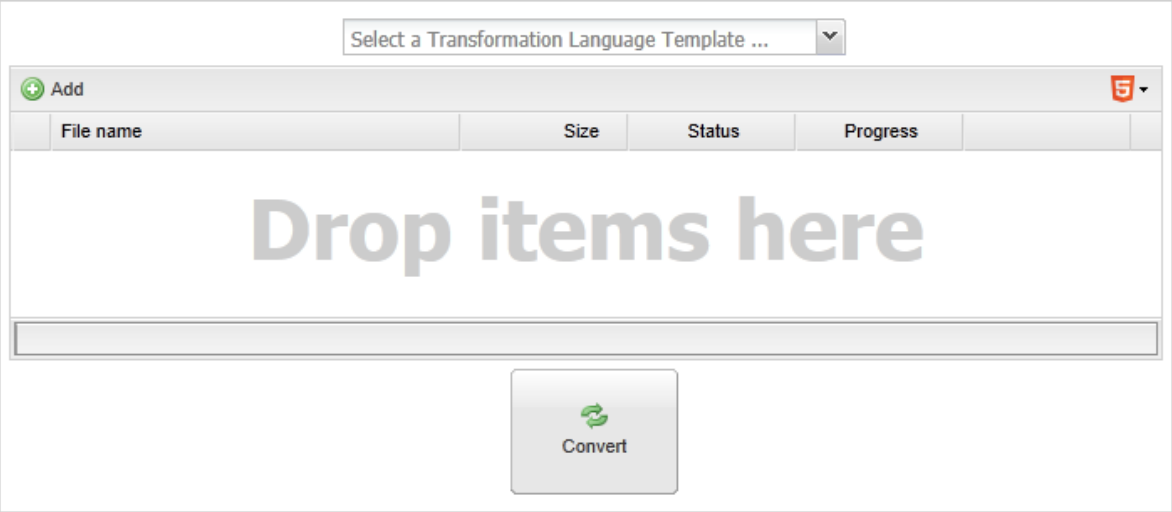

Figure 2.10: Graphical user interface of the data converter.

2.4.1 Use-case

An industrial use-case has been designed for the data conversion process in cooperation with a company, specialized in manufacturing semi-conductors. The use-case describes the conversion of test results in plain text format and the representation of the test results as a report. The tabular organization of the test results within the plain text format consists of seven columns with different data types. Table 2.4.1 depicts an anonymized sample data set.

ID	Date	Item	ItemId	XCoord	YCoord	Status
1	2011-07-01, 08:00:00	Foo1	10 000	0F	0F	PASS
2	2011-07-01, 08:00:01	Foo1	10 000	0A	0F	FAIL
...						

Table 2.1: Sample data set for use-case scenario

A corresponding data template can be found in listing A.1.

2.4.2 Performance

The use-case was reviewed towards performance aspects. A data set of 2000 tuples with seven values each, are converted and stored within the generic data base. The data was processed on a Sun X2200 server with two dual-core AMD Opteron processors and 24 GB RAM.

A cold start scenario results in a processing time of 1:30 minutes for the sample data set. The best case using a warm start was registered with 47.13 seconds. These results are based on iterations of 20 times. The data base commit commands of the LINQ technology (System.Data.Linq.DataContext.SubmitChanges()) take up to 60% - 70% of time consumed by the entire program. The conversion process itself takes about 23% - 28% of the entire processing time. Averagely it takes about 23 seconds for the example data set, which means about 11.5 ms per tuple.

2.4.3 Optimization approaches

The first optimization step would be an evaluation to replace the LINQ technology with the traditional ADO.NET technology. Another approach would be to accumulate write tasks and perform a batch commit.

The conversion process can be enhanced by optimizing the data converter for parallel operation. Right now, the data converter sequentially converts each data item and builds up the tree structure within the generic data base. Since the tree structure is specified by the data template, parallel tasks can be identified prior to the conversion. The data converter also treats each data item individually without any knowledge about predecessor and successor elements. This poses a major performance drawback for tabular structures.

3

Conclusion

This paper describes the prototypical, graphical implementation of a generic data converter and generator. It enables the conversion and storage of data from and to arbitrary data formats. The XML model specifies data formats for the transformation in both directions, i.e., reading and writing, respectively. Since this XML model is Turing-complete, any format parsed or generated by a computer program can be processed. In addition, the parsed data can be stored in a data base whose structure is defined by the proposed generic data model. Data which is stored within the generic data model can be grouped, augmented and changed in an intuitive way. To facilitate this, a graphical component for data manipulation has been developed. Incomplete stored data can easily be enriched, changed and padded.

The parts required to perform the data conversion and storage process have been prototypically implemented. A web-based approach gurantees platform independency, as well as minimum requirements on the client-side. The presented approach enables the use and conversion of distinct and proprietary formats as well as persistent and sustainable storage of the data contained therein. It greatly simplifies the transformation of arbitrary data formats.

3.1 Findings

The conversion process has major performance issues as described briefly in section 2.4.2. Especially the data base commits for each data value consumes most of the processing time. Therefore the LINQ technology may not be suitable for this application. An evaluation for a better technology is suggested. Moreover the conversion process may be enhanced by parallelism and *intelligent* algorithms to use synergy effects within tabular data formats.

3.2 Future work

The data conversion process can somehow be seen as a data flow. The actor is the data converter itself, which needs some kind of input. Once enough input tokens are available, the data converter fires the conversion process. The system therefore schedules the conversion task. An improvement of the scheduling approach might improve the overall performance. The stream-based function approach described in [8], enables the actor to choose from a pool of suitable functions for the next step, depending on the state of the actor. This kind of choosing the best method is similar to the human perception and pattern recognition. It might enable the data converter to have artificial intelligence. Hence, less interaction with the user would be required and the specification of a data format might be obsolete.

Bibliography

[1] Bureau International des Poids et Mesures: *The International System of Units (SI) 8th edition*, 2006.

[2] Herken, Rolf: *The Universal Turing Machine: A Half-Century Survey.* Oxford University Press, Inc., New York, USA, 1992.

[3] IEEE: *IEEE Standard for Binary Floating-Point Arithmetic for microprocessor systems (IEEE Std 754-1985)*, 1985.

[4] Kawanaka, S. and Hosoya, H.: *biXid: A Bidirectional Transformation Language for XML.* In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, New York, USA, January 2006.

[5] Liberty, Jesse, Maharry, Dan, and Hurwitz, Dan: *Programming ASP.NET 3.5.* O'Reilly Media, Inc., Sebastopol, USA, 4th edition, 2008.

[6] Singh, Arindama: *Elements of Computation Theory.* Springer Publishing Company, Incorporated, 2009, ISBN 1848824963, 9781848824966.

[7] Soare, Robert I.: *Recursively enumerable sets and degrees.* Springer-Verlag New York, Inc., New York, USA, 1987.

[8] Sriram, Sundararajan and Bhattacharyya, Shuvra S.: *Embedded Multiprocessors: Scheduling and Synchronization.* Marcel Dekker, Inc., New York, NY, USA, 1st edition, 2000.

[9] W3C: *XML Schema Part 0: Primer Second Edition*, 2004. `http://www.w3.org/TR/xmlschema-0`.

[10] W3C: *XSL Transformations (XSLT) Version 2.0*, 2007. `http://www.w3.org/TR/xslt20`.

[11] W3C: *XMLHttpRequest*, 2010. `http://www.w3.org/TR/XMLHttpRequest/`.

[12] Walker, Derek, Petitpierre, Dominique, and Armstrong, Susan: *XMLTrans: a Java-based XML transformation language for structured data*. In *Proceedings of the 18th conference on Computational linguistics - Volume 2*, COLING '00, pages 1136–1140, SaarbrÃ¼cken, Germany, 2000. Association for Computational Linguistics.

[13] Wirth, N.: *Compiler Construction*. Addison-Wesley, Wokingham, UK, 1996.

[14] Yergeau, F.: *UTF-8, a transformation format of ISO 10646 (RFC 3629)*, 2003.

List of Abbreviations

DBMS Data Base Management System

JS JavaScript

LINQ Language Integrated Query

XML Extensible Markup Language

XSLT Extensible Stylesheet Language Transformations

Appendix

A

Use-Case Data Template

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <document xmlns="http://iit-test.fh-salzburg.ac.at/TR/
       DataTemplateSchema.xsd" xmlns:xsi="http://www.w3.org/2001/
       XMLSchema-instance">
3    <head>
4      <name>Sample Use Case</name>
5      <description>Data template for sample test case</description>
6      <author>IIT-Team</author>
7      <company>FH Salzburg</company>
8      <department>Informationstechnik &amp;amp; System-Management</
           department>
9      <date>2011-07-31</date>
10   </head>
11   <body>
12     <block type="static">
13       <argument>
14         <value type="string">ID&amp;#9;Date&amp;#9;Item&amp;#9;ItemId&amp;#9;XCoord
               &amp;#9;YCoord&amp;#9;Status&amp;#13;&amp;#10;</value>
15       </argument>
16     </block>
17     <block type="register" name="FileData">
18       <datasource>
19         <value type="string" />
20       </datasource>
21     </block>
22     <for>
23       <condition operator="eof" />
24       <iterate>
```

```
25          <block type="register" name="DataEntry">
26            <datasource>
27              <value type="string" />
28              <parent ref="FileData" />
29            </datasource>
30          </block>
31          <block type="register" name="ID">
32            <datasource>
33              <value type="string" />
34              <parent ref="DataEntry" />
35            </datasource>
36          </block>
37          <block type="dynamic">
38            <datasource>
39              <value type="uint8" />
40              <parent ref="ID" />
41            </datasource>
42          </block>
43          <block type="static">
44            <argument>
45              <value type="char">&amp;#9;</value>
46            </argument>
47          </block>
48          <block type="register" name="Date">
49            <datasource>
50              <value type="string" />
51              <parent ref="DataEntry" />
52            </datasource>
53          </block>
54          <block type="dynamic">
55            <datasource>
56              <value type="datetime" />
57              <parent ref="Date" />
```

```
58            </datasource>
59          </block>
60          <block type="static">
61            <argument>
62              <value type="char">&amp;#9;</value>
63            </argument>
64          </block>
65          <block type="register" name="Item">
66            <datasource>
67              <value type="string" />
68              <parent ref="DataEntry" />
69            </datasource>
70          </block>
71          <block type="dynamic">
72            <datasource>
73              <value type="string" />
74              <parent ref="Item" />
75            </datasource>
76          </block>
77          <block type="static">
78            <argument>
79              <value type="char">&amp;#9;</value>
80            </argument>
81          </block>
82          <block type="register" name="ItemId">
83            <datasource>
84              <value type="string" />
85              <parent ref="DataEntry" />
86            </datasource>
87          </block>
88          <block type="dynamic">
89            <datasource>
90              <value type="string" />
```

```
91            <parent ref="ItemId" />
92          </datasource>
93        </block>
94        <block type="static">
95          <argument>
96            <value type="char">&amp;#9;</value>
97          </argument>
98        </block>
99        <block type="register" name="XCoord">
100         <datasource>
101           <value type="string" />
102           <parent ref="DataEntry" />
103         </datasource>
104       </block>
105       <block type="dynamic">
106         <datasource>
107           <value type="string" />
108           <parent ref="XCoord" />
109         </datasource>
110       </block>
111       <block type="static">
112         <argument>
113           <value type="char">&amp;#9;</value>
114         </argument>
115       </block>
116       <block type="register" name="YCoord">
117         <datasource>
118           <value type="string" />
119           <parent ref="DataEntry" />
120         </datasource>
121       </block>
122       <block type="dynamic">
123         <datasource>
```

```
124             <value type="string" />
125             <parent ref="YCoord" />
126           </datasource>
127         </block>
128         <block type="static">
129           <argument>
130             <value type="char">&#9;</value>
131           </argument>
132         </block>
133         <block type="register" name="Status">
134           <datasource>
135             <value type="string" />
136             <parent ref="DataEntry" />
137           </datasource>
138         </block>
139         <block type="dynamic">
140           <datasource>
141             <value type="string" />
142             <parent ref="Status" />
143           </datasource>
144         </block>
145         <block type="static">
146           <argument>
147             <value type="string">&#13;&#10;</value>
148           </argument>
149         </block>
150       </iterate>
151     </for>
152   </body>
153 </document>
```

Listing A.1: Data template for example data set shown in section 2.4.1