A Generic Model for Universal Data Storage and Conversion

Andreas Unterweger, Bernadette Himmelbauer, Simon Kranzer, Peter Ott, Robert Merz, Gerhard Jöchtl
Salzburg University of Applied Sciences
Email: andreas.unterweger@fh-salzburg.ac.at

***Abstract*—We propose a data model which allows storing values of arbitrary types, including inter-data dependencies and meta information. Furthermore, we provide an Extensible Markup Language (XML) based model to describe data formats which allows specifying programs to convert data represented in existing formats both from and to our proposed data model. We will show that these programs are Turing complete, thus allowing the same arbitrarily complex conversions which are possible with Extensible Stylesheet Language Transformations (XSLT) or the C programming language. In addition, we describe the components of a prototypical implementation in form of a validator, a data converter and a data generator. In combination with a data editor, parts of our prototypical implementation are already employed in several use cases in the industry to transform data between different formats.**

Keywords—Data conversion, data model, XML model

I. 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, we present a fully integrated solution which enables storage and transformation of arbitrary data formats.

Our approach is based on two essential models: a generic data model and an XML model. The generic data model allows storage of arbitrary data formats, including meta information and interdependencies. The XML model specifies data formats and serves as transformation language from and to the original data representations. In addition, a universal data converter and a generator are implemented in a prototype framework. Figure 1 shows our complete conversion approach. 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.

Similar to our XML model, a transformation language has been realized in XSLT by the W3C [1]. While both, their and our approach, are Turing complete[2] and allow the conversion of arbitrarily complex data formats, the XSLT version 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 in our approach storage structures are explicitly defined by the generic data model. *biXid*, a bidirectional transformation language, is shown in [3] and designed for the transformation of XML formats only, as is *XMLTrans* proposed in [4]. Another related language targeting model transformation is *DSLTrans*, proposed in [5], which however, is Turing incomplete.

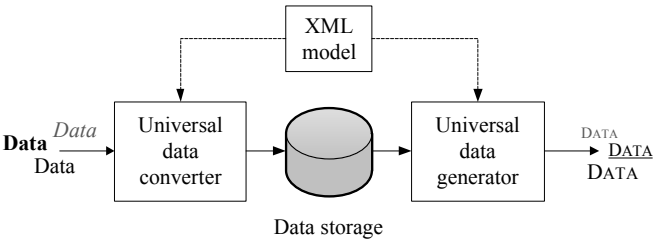


Figure 1. Overview of the data conversion and generation framework based on the proposed data and XML model. Data available in different formats is converted to the data model representation using the data converter which transforms data based on a program specified by the XML model. The data generator enables the export into arbitrary formats.

In addition to the lack of a generic data model (see XSLT above), parser generators like *ANTLR*[6] cannot be used for text generation, while text processing software like *awk*[7] or commercial data converters like *Altova MapForce*[8] impose limits on data formats and/or the complexity of the conversion. As this makes them and all similar solutions either non-universal or not as flexible as our approach, they are not reviewed in detail.

The paper is structured as follows: section II describes the XML model as the core part our approach which allows for data format description and conversion. After showing the Turing completeness of the XML model in section III, the generic data model for storing data values, their dependencies and the corresponding meta information is described in section IV. The prototypical implementation of a universal data converter and data generator based on the two models, i.e. the data and the XML model, is described in section V.

II. THE XML MODEL

In order to specify the format of data to be parsed or generated, we developed an XML model, implemented in form of an XML schema[9] whose complexity is sufficient to model any computer program as shown in section III. By design, programs specified by the model, i.e. data format descriptions, are capable of both parsing and data generation.

As a result, only one description of the data format is required for specification and allows both reading as well as writing data of this format.

For reading a file of a particular format, our 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 as described below. Similarly, our 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 our model. The functionality of a block is specified by its type and additional parameters, including means to access the data base based on the generic data model described below. 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. Additional formats (e.g. the use of "," or "." as comma separator for floating point values) may be specified for convenience. Formats which are currently unsupported can be processed using a similar set of blocks and intermediate operations as the ones described below. 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 conceptual data model.

In addition to the blocks which are processed one after another, various control flow operations are available. Conditional execution can be modeled 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.

For convenience, control flow simplifications have been introduced. A set of consecutive blocks can be combined to a block group with a unique name. Block group references can be used to execute the blocks contained in a group at another point of the program. This is accomplished by referring to the block group by its name, thus avoiding repeated definitions at multiple locations in the code. The nesting of block groups and references is also supported, including the possibility to refer to single blocks, called block references.

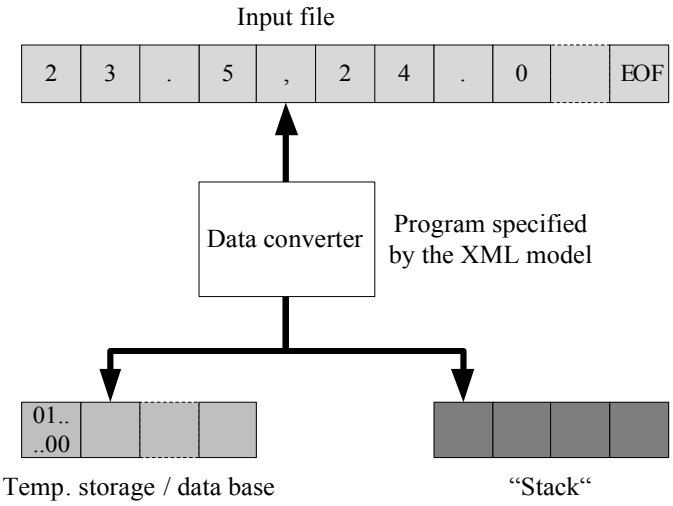


Figure 2. XML model based program illustrated as a Turing machine with three bands and the corresponding read/write heads depicted by arrows. By example, the input band represents the content of an input file with UTF-8 alphabet. The program has processed one input value which is stored in binary form in the data base.

Block group references are similar to function calls in procedural programming languages, although parameters are not explicitly specified. In order to pass or return parameters, an additional stack-like construct can be accessed using a *param* type block for pushing and a *result* type block for popping: This is similar to the call stack used for implementing function calls on most computer architectures[10]. Values of *register* blocks can be assigned to *param* blocks using their block references. Similarly, values of *result* blocks can be assigned to *register* blocks the other way around. The current implementation does not check for stack consistency, although this can be easily implemented and therefore remains future work.

III. 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[11] 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 modeled by a third band of the Turing machine as depicted in Figure 2. 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. The alphabet of the input band is the set of UTF-8 characters [12] 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 prove that any partial μ-recursive function (which is equivalent to a Turing machine in terms of computable functions[13]), 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[14], 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[10]. 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 $h(y, x_1, x_2, \ldots, x_n)$ being $f(x_1, x_2, \ldots, x_n)$ for $y = 0$ and $g(y, h(x_1, x_2, \ldots, x_n), x_1, x_2, \ldots, x_n)$ 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.

IV. The Generic Data Model

Our universal data converter enables the transformation of acquired data sets into a 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 proposed data model for further processing and storage, allowing for both modification and export into other formats using our data generator.

In order to allow the representation of all existing and future data formats, we propose a generic data model which allows storing arbitrarily complex data types in binary form, including metadata which simplifies the conversion from and to the format specified by the model. Due to the metadata 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[15]) or a custom data type, denoted by the data type entity in the entity relationship (ER) model[16] depicted in Figure 4.

To represent list or tree structures, dependencies between single elements can be modeled. Other arbitrary forms of dependencies can be modeled based on dependencies between single elements. In addition, association of the data with physical SI[17] 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 in Figure 3 as described in [18]). 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 kilometers per hour, kilometers 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 millimeters. Creating a *milli* prefix with a factor of 0.001 and combining it to a new unit *mm* (for millimeters), 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 (omitted for the sake of readability), 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

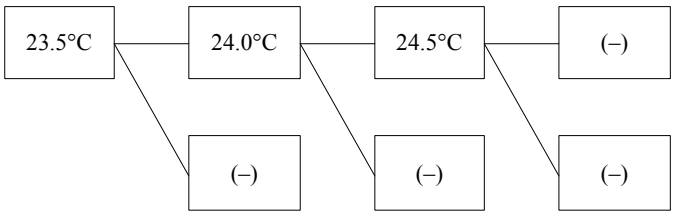


Figure 3. Binary tree representing a list of values. The root node and each top child node, except the last one, contain a value, while the bottom child nodes do not exist, i.e. they are *NULL*. Each value node except the root node has a parent node, including the last *NULL* node whose parent is the last actual value of the list

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. 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. To represent a list, the data model is utilizing a series of dependencies in form of a binary tree. The tree (depicted in Figure 3) 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.

In order to specify the input file format, an XML model is created consisting of a *variable* type block referring to a *float* type data base value and a *static* block representing the constant comma (",") 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 short example.

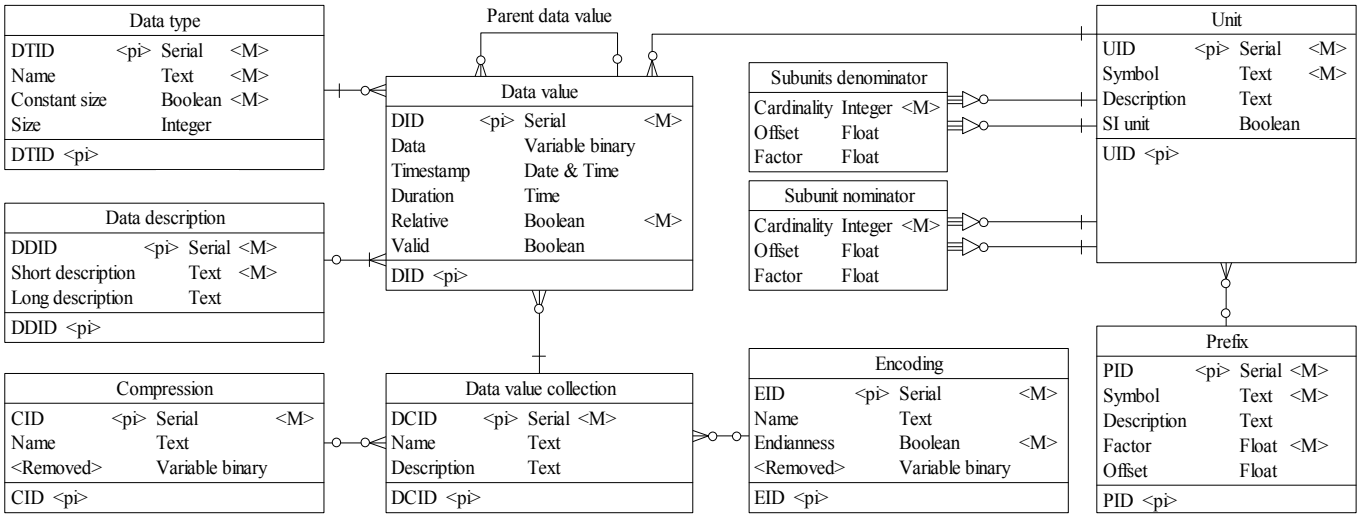


Figure 4. Simplified ER model of the proposed data model. Attributes denoted with <*Removed*> are omitted for the sake of readability, specifying additional properties of the corresponding entities. The central *data value* entity stores a binary representation of a value, the other entities supply meta information.

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

When the data converter with an XML model is applied to an arbitrary file containing temperature values formatted as described above as input, temperature values of this file will be read into the data base. As 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 implicitly specified. 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. This 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*. Therefore, the condition relies on the *eof* operator whose semantics changes depending on the data flow direction (input or output).

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. In our implementation this can be specified by using the data generator GUI.

As described in detail in section III, 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 Figure 5. 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.

(a)

```
23.5,24.0,24.5<EOF>
```

(b)

id	parent id	data type	binary data	unit
id of *23.5*	– (NULL)	float	41BC0000	17 (°C)
id of *24.0*	id of *23.5*	float	41C00000	17 (°C)
id of *24.5*	id of *24.0*	float	41C40000	17 (°C)

(c)

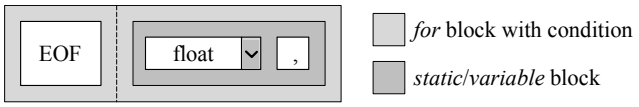
```
<document>
  <temperature>296.65</temperature>
  <temperature>297.15</temperature>
  <temperature>297.65</temperature>
</document>
```

Figure 5. Data model representations of the two example formats described in the text: (a) comma separated temperature values in degrees centigrade; (b) 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; (c) XML formatted output with a list of temperature values in degrees Kelvin.

V. IMPLEMENTATION

For proof of concept, the four components of the framework depicted in Figure 1 have been implemented using C# and MS-SQL Server. In addition, a browser based graphical interface has been constructed with Microsoft ASP.NET and IIS 7.

To feed data into the data converter, the user specifies its structure either by using graphical blocks as depicted in Figure 6 (top) or by defining it directly using our XML model (bottom). As soon as the structure has been defined, the data transformation can be repeated for all data of the same format. Before a program based on our XML model is




```
<for>
  <condition operator="eof" />
  <iterate>
    <block name="Temperature" type="variable">
      <datasource>
        <value type="float" />
        <parent ref="Temperature" />
      </datasource>
    </block>
    <block type="static" optional="true">
      <argument>
        <value type="character">,</value>
      </argument>
    </block>
  </iterate>
</for>
```

Figure 6. Graphical block and control flow representation and the corresponding XML model based program. The *for* block is split into a condition and an iteration partition in both the visualization and the program. Two blocks are contained in the iteration partition, one *variable* type block with the associated *float* type data in the data base and one *static* type block always consisting of a comma as UTF-8 character.

executed to parse or generate files, its structure is validated against our XML schema definition. A parser will check if additional constraints, which are predefined by the XML model, 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 references to blocks of the types which have been described in section II are resolved through a simple symbol table which allows global and local scopes for blocks and block groups. When data are stored in the database, it can be visualized and manipulated using a browser based data editor or any other data mining software. For data export, a data generator has been implemented. It uses the same programs based on our XML model as the data converter to define output formats. Thus, the programs for input and output transformations only need to be defined once.

To demonstrate functionality in industrial settings, a variety of industrial use cases have been defined. Production data from several companies was used to successfully validate the model and the prototype implementation of the framework. Both, disk space utilization and data processing speed, scale with the underlying database management system and the used web server. Parallel execution/conversion is limited in the same way. As the current implementation is a prototype, detailed performance comparisons remain future work.

VI. CONCLUSION

We proposed a generic data model which allows storing of data values and an XML model which enables the transformation from and to arbitrary data formats. As the XML model is Turing complete, all data formats which may be generated or parsed by programs written in any modern programming language, can be read or written by a program specified by our XML model. Due to the generic character of the model, only one program is required to allow both reading and writing of values of a given format. The parts required to perform the data conversion and storage process, including the validation of XML model based programs and the editing of data values within the data model, have been prototypically implemented and are already used in industry. Therefore, our approach enables the use and conversion of distinct and proprietary formats as well as persistent and sustainable storage of the data contained therein. Our approach greatly simplifies the transformation of arbitrary data formats. To our knowledge, no other approach currently provides a similar and comprehensive transformation and storage performance.

ACKNOWLEDGMENT

We like to thank our project partners for their generous support: B&R Industrial Automation Corporation, Bosch AG Hallein, COPA-DATA GmbH, Liebherr-Werk-Bischofshofen GmbH, PALFINGER AG and Wirtschaftskammer Salzburg. We also like to thank the reviewers for their valuable comments that helped to improve this paper.

REFERENCES

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

[2] Brainerd, W. S. and Landweber, L. H., *Theory of Computation*. New York, USA: John. Wiley & Sons, 1974.

[3] 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, 2006.

[4] Walker, D., Petitpierre, D. and Armstrong, S., "XMLTrans: a Java-based XML Transformation Language for Structured Data," in *Proceedings of the 18th International Conference on Computational Linguistics*, Saarbrücken, Germany, 2000, pp. 1136-1140.

[5] Barroca, B., Lucio, L., Amaral, V. Felix, R. and Sousa, V., "DSLTrans: A Turing Incomplete Transformation Language," in *Proceedings of the 3rd International Conference on Software Language Engineering*, Eindhoven, The Netherlands, 2010.

[6] Parr, T. ANTLR Parser Generator. http://www.antlr.org/, 2011.

[7] IEEE, "The Open Group Base Specifications Issue 6 (IEEE Std 1003.1)," Institute of Electrical Engineers and Electronics, 2004.

[8] Altova. Altova MapForce. http://www.altova.com/mapforce.html, 2011.

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

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

[11] Herken, R., *The Universal Turing Machine: A Half-Century Survey*. Wien, New York: Springer-Verlag, 1995.

[12] Yergeau, F., "UTF-8, a transformation format of ISO 10646 (RFC 3629)," Internet Engineering Task Force, 2003.

[13] Singh, A., *Elements of Computation Theory*. London: Springer-Verlag, 2009.

[14] Soare, R. I., *Recursively Enumerable Sets and Degrees*. Berlin, Heidelberg, New York: Springer-Verlag, 1987.

[15] IEEE, "IEEE Standard for Binary Floating-Point Arithmetic for microprocessor systems (IEEE Std 754-1985)," Institute of Electrical Engineers and Electronics, 1985.

[16] Chen, P. P.-S., "The Entity-Relationship Model — Toward a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9-36, 1976.

[17] Bureau International des Poids et Mesures, "The International System of Units (SI) 8th edition," Organisation Intergouvernementale de la Convention du Mètre, 2006.

[18] Ott, P., "Konzeption eines Testframeworks," Salzburg University of Applied Sciences, Puch bei Hallein, Master's thesis 2011.