ZHULEB

Ivan Kurmanov

August-September 1999

Document info:

status: proposal;

version: preliminary release;

state: incomplete.

1 Introduction

ReDIF is a text-file format for metadata expression. It has been invented few years ago, but it hasn't been stable over this period. A number of minor corrections and important enhancements took place.
The ReDIF data files processing applications are significantly dependent on the ReDIF parser, implemented in Perl programming language. Parser reads data files and processes it and passes the ReDIF templates (records) to an application as a ready for use in-memory data structure. The ReDIF templates validation and preprocessing is performed on the fly and all invalid templates are filtered out.
Readers of this document are supposed to be aquainted with main notions of ReDIF and its parsing. If you are not, or not sure about it, I suggest reading the Parsing ReDIF document at:
http://openlib.org/acmes/wolgo/Parsing.html

1.1 Zhuleb invention rationale

Zhuleb shall replace the 'redif.spec' functionality and and encourage distributed development of enhancements and extensions to ReDIF, preserving the Parser and it's applications' interface without changes. It shall facilate reuse of data types design and programming and ensure wide then otherwise possible compatibility of higher-level applications.
This will allow to continue to use existing ReDIF-aware applications, which rely upon existing parser 'rr.pm', but to apply them to a wider set of input data. In a longer term this will allow to create portable and extensible applications for ReDIF data processing, presentation, archiving.

1.2 Zhuleb design principles and requirements

Zuhleb shall be:

- more powerful and flexible then its predecessor (redif.spec)
- compatibility with existing standards is desireable
- expressed in XML

And Zhuleb shall provide for the following:

- mechanism to define ReDIF data structures (records and elements) and content (data) types.
- mechanism for inheritance between the element types (including records) and content (data) types.
- mechanism for embedded documentation
- mechanism to program (influence) the parser's behaviour in a number of type-dependant events (including all main stages of validation of each record or its part and possible external events and actions on the data).
- set of predefined simple data types, e.g. dates, integer, so on.

- mechanism to define part-of relationships between otherwise-independently defined structures (defining collection-of types)
- compiler (or translator) of Zhuleb specification shall produce verbose error descriptions and warnings (at user option).

2 ReDIF data format

The ReDIF data format, before any Zhuleb specification, is a text-file record-based element/value format. Each record has a certain type.
Depending on its type, a record can contain different sets of elements. Each element has a name (an identifier), which occurs in the data. Elements are either simple or complex. Simple (basic) elements must have a textually-expressed content (or value), which may or may not be empty (depending on the element type). Complex elements may or must (depending on the element type) contain some other elements.

2.1 Terms compatibility

'Simple element' is the same as 'attribute' in older ReDIF terms.
'Complex element' is the same as 'cluster' in older ReDIF terms.
'ReDIF record' is the same as 'template' in older ReDIF terms.

2.2 Parser and Zhuleb integration

To process ReDIF data files all (known to the author) ReDIF applications use the ReDIF parsing engine. This is assumed, that core role of the parser in a processing ReDIF data will continue.
'Zhuleb' is just a language, which is supposed to replace functionality of 'redif.spec' in RePEc (ReDIF). It's implementation will require an upgrade of the existing parser (rr.pm). Zhuleb and parsing are tightly connected. Some requirements for the parser are mentioned in a section below.
The ReDIF parsing engine will use Zhuleb files to find out syntax and semantics of the data files. Lexical analysis rules and global syntax rules will be hard-coded into the parser.
The intelligence and responsibility, needed for processing ReDIF data, will be divided between the parser and a specification written in Zhuleb.

2.3 Role of the ReDIF parser

The parser will be responsible for extracting records, elements and values from the data. Parser will identify the records' types and validate records upon the appropriate specification data.
Parser shall be able to support and recognize different low-level syntaxes of the data. It must support at least old ReDIF attribute/value template syntax and (possibly at a later stage of implementation) XML syntax.
This imposes high requirements on the parsing engine internal design. It shall be designed with extensibility in mind (internal modularity, good (appropriate) level of abstraction and interface simplicity).
Support for other syntaxes may be provided in the future.
While parsing data, the parser will identify and validate different data structures against the Zhuleb specification.
Upon validation will generate appropriate events or actions. At such events or actions an appropriate code fragment (function) from the Zhuleb specification will be executed by the parser if specified.
In such case, the parser's behaviour will be influenced by the result of the function execution, or by the function call from the executed code fragment or by modifed (during execution) variables.
This way extensible checking and processing rules will be achieved.

2.4 Role of Zhuleb specification

The specification written in Zhuleb will describe the record types and the element types. The notion of type is quite wide, it includes: data structure requirements and constraints and value requirements and properties, procedures for data processing, manipulation and presentation.
Zhuleb will be used to define the existing ReDIF format and it will allow to create extensions to ReDIF or even to introduce completely new data types with the same ReDIF parser and related tools.

3 General Zhuleb features

3.1 Data types

This section will try to clarify the use of term 'type' in this document.
Type as used hereinafter is an abstract notion, close to the same notion in classic programming languages. (Some modern programming languages, e.g. Perl, have a relatively limited usage of data types.)
We may define a data type as a set of properties, which determine possible range of values and a range of manipulations with a piece of data and the results of such manipulations.
Object-oriented programming approach adds behaviour to the notion of type. That is: a type may define the data structure and the coded behaviour.
We will treat units of data as objects.
An object is always of a certain type. In other words, an object always belongs to a class. An object contains data and the code. Each object can contain its own unique data. The code that objects contains is always common for all objects of the class.
In Zhuleb we will apply such object-oriented approach both to the record types and the basic element value types.

3.2 Complex elements

Complex elements are such that do contain or can contain other elements. A complex element can be refered to as a 'parent' to the elements that it contains. And an element can be refered to as a 'child' to the complex element which it is contained in.
Each complex element has a type. The type will be identified by the parser.
Complex elements will be accounted as valid if its structure corresponds to the element type definition and all its children elements are valid.

3.3 Records

A record in ReDIF is a most important unit of information.
A record is a complex element.
A record has a type and must conform the its type requirements. Otherwise a record cannot be accounted to as a valid ReDIF record and will be ignored by the parser, not being transfered to an application.

3.4 Basic elements and their types

Each basic element in data has a type. Depending on the element type, various restrictions, rules and procedures for it's value will be activated. The type of an element will be identified through its root complex element type specification.
Default element type will be the free-text which means no restrictions on the value, including possibility of an empty value.
Several primitive built-in types for numeric and character data will be provided by Zhuleb. For example, a date type.
Creation of new user-defined value types will be allowed.
One of the options for the user-defined types will be the limited dictionary of string values: all possible values for the type may be listed in the specification text or in an external file.
There shall be a possibility for the specification to require that an element value of a certain type will have to have a non-empty (non-zero) value to be valid.
There shall be a possibility for the specification to provide a default value for an element value type.
New user-defined types may be introduced through defining methods: routines, written in a programming language, that will be executed by the parsing engine when validation, value output or a similar operation is necessary. See more details below.
Support for Perl 5 regular expressions to define the value types shall be included into Zhuleb.

4 Object-Oriented features of Zhuleb

4.1 Introduction to object-oriented

Zhuleb will be designed as an object-oriented language.
In object-oriented computing objects are conceptual software-internal representations for real-life objects. Objects are grouped in classes by their types. In other words, classes are formally defined types of objects.
An object can be viewed as a combination of mutually related data and code. Defined classes contain the exhaustive formal description of the data fields and routines of all objects of the class. Such descriptions are expressed in an

object-oriented programming language.
The code, contained in a class definition is said to describe the object's behaviour. This or that behaviour is triggered by the operations on the object or object method invocaton, in the OOP (object-oriented programming) terminology.
Hierarchies of classes may be created. One class may inherit it's properties from another class, and add to the later certain functionality or provide more detailed description of the related objects. The OOP terms 'Child class' and 'Parent class' are used to describe that kind of relationship between the two classes. Parent class is always more abstract.
For a more elaborate and detailed introductions and tutorials to object-oriented technology I suggest the following links:
- What is Object-Oriented Software? by Terry Montlick http://www.soft-design.com/softinfo/objects.html (A nice short tutorial for anybody)
- Object-Oriented FAQ - Object FAQ http://www.cyberdyne-object-sys.com/oofaq/ http://www.cyberdyne-object-sys.com/oofaq2/ (Detailed discussions of all major aspects)
- What is "Object-Oriented Programming"? (1991 revised version) by Bjarne Stroustrup http://www.programmersheaven.com/files/misc/tex (A paper by a classic scientist in the area - PDF format.)
All data types in Zhuleb will be classes.
Two groups of classes will exist: for simple element value types and for complex element types. At this stage it is not clear yet how these two groups will be differentiated. Possibly they will inherit from different base classes.
A class definition shall consist of several parts. First part we will call 'class declaration'. It will provide most important general information of the class. It shall contain (at least): class name (identifier), human-readable class purpose and usage description, inheritance information (parent class name) and the class' data elements configuration (for entity classes).
Possibly class declaration will also be required to hold a list of all object methods (their names) with their purpose and usage desciption.
Basic element value type class declaration shall (in addition to the general class info) contain general settings for the type, e.g. non-empty value requirement.
Second part of the class definition shall contain the methods defintion for a class.
At this stage the Perl 5 programming language is chosen as a language for methods definition. Methods are simply perl functions, which will be executed in an object context with some special parameters.
Methods will be executed by the Parser either at the data retrieval or by a request of a higher-level application (interface). Interface description for methods will be provided at a later stage.
As a general rule will be taken that all methods are optional and none of the methods is required to be defined.
At this stage it is not clear, where Zhuleb shall allow multiple inheritance or not. (Mulitple inheritance allows for a child class to have several parent classes. This is a useful feature sometimes, but it's implementation can bring significant additional internal complexity to the processing software.) One possible way around the problem is a concept of an interface, as it is defined in Java.

4.2 CLASS RELATIONSHIPS

Some support to define class relationships other than inheritance and containment shall be designed. At least support for hierarchical referencing shall be built-in to enable hierarchical multi-component handles and to enable an option of the records' cross-referencing.
This must bring automated support for series to archive and paper to series relationships, expressed in the handles structure.

4.3 XML Schemas

XML schemas is a proposed concept of a language to be used instead of DTD to define structure of XML documents and to impose some rules on the values of XML elements.
Different dialects of Schemas exist: XML-Data, SOX (version 1.0 and 2.0), DCD and others.
Zhuleb will be based on one of these dialects, and will inherit from it most of the features.

5 Special Zhuleb features

5.1 Special adaptations for the old ReDIF templates

As a general requirement, Zhuleb and new parsing engine shall support old ReDIF 1.0 templates.

5.2 Zhuleb specification library

"Zhuleb" shall allow to store and maintain the ReDIF specification in a number of separate files. Main or root specification file shall exist, which will hold global definitions and possible option settings. Other files may be included into the specification iether implicitly by their naming and directory placement or explicitly by a kind of 'include' directive.
Ideally a mechanizm shall exist that will allow to collect specification files from the various points on the web automatically to build a library of Zhuleb specifications.
(Possibly this can be integrated with an existing ReDIF mirroring tool remi.)
A special file and directory naming convention will be needed to exclude or minimize chances for a name-conflict. This also concerns name identifiers used in the specification files themselves.
The classes from all sources of the specification files shall be organized into a consitent hierarchy.
This will allow an improved error-localization and allow distributed maintenance of the specification: various people will maintain the separate pieces they are responsible for.
Specification files shall be compiled each time after they have been changed. This shall be done either by an external compiler or by a routine built-into the parsing software. This shall eliminate the need to read and translate specification each time some data needs to be accessed: compiled files shall be much quicker to load.
template types and allowed/required attributes of them 2) attributes, allowed and/or required to appear in template 3)
but each cluster has to be independent of other. Each level and/or branch of the hierarchy is of a certain type and of a certain name.
Eg., a template of the 'article' type, can contain multiple 'author' clusters, each of the 'person' type. In that case 'author' will be the name, 'person' will be the type.
Several clusters of the same name (and type) are syntactically delimited and it is always possible to find-out where one cluster ends and another one starts.
And, finally, a cluster can contain simple attributes and/or other clusters.
Existing template types, the attribute and cluster names and types that can occure within each template type and their occurence constraints and requirements are defined by the 'redif.spec' file.
Some data values constraints and pre-processing and value checking instructions can be coded in 'redif.spec' as well. Perl programming language is used for that.
)