Syntax Recovery for Uniface as a Domain Specific Language

Majd Zohri Yafi
Computer Science and Electrical Engineering
University of Essex
Colchester, United Kingdom
mzohri@essex.ac.uk

Arooj Fatima
Computing and Technology
Anglia Ruskin University
Cambridge, United Kingdom
arooj.fatima@anglia.ac.uk

***Abstract* - This paper discusses the problems faced by the organisations who are running domain specific 4GL systems to deploy their core business logic. Given the fact that it is often not realistic to find new engineers for these not-widespread languages, the paper proposes a method to extract useful artefacts from 4GL systems which have the data stored in XML like format such as Uniface system. In this work, the authors show how to use Encapsulated Document Object Model to read Uniface XML and scan the content to extract the custom code. In addition, this paper introduces how to restore the code schema and visualise it.**

Keywords - Uniface, domain specific languages, XML parsing, EDOM

I. NTRODUCTION

There are many programming languages currently in use, evolved from first generation programming languages to the sixth generation. In the first generation, the language is largely unstructured and written with very low level statements that control the hardware. Assembly language is an example of the first generation language. The second generation is also unstructured but it innovated the concept of records e.g. FORTRAN language. In the third generation programming languages, the concept of object oriented programming had been presented. Pascal implements this concept. The fourth generation programming languages are domain specific languages with higher abstraction in comparison to the previous generations. These languages are functional and database centric. The fifth generation is more focused on solving a specific problem rather writing an application that runs an algorithm. This technology is commonly used with artificial intelligence applications. The sixth generation programming languages mimic spoken languages to instruct the computer to perform a specific action. This paper focuses on the fourth generation languages because many languages that became legacy belong to this generation [1]. Fourth Generation Programming Languages renovates the functionality of third generation programming languages to work with keywords. There are common traits among those keywords. Firstly, they cover applications whereby each of those keywords encapsulates a sum of atomic instructions which could be written in lower level language as lengthy procedures. Secondly, those keywords are as close to natural language words as it could be. Many companies used different 4GLs to develop their business and expressed their entire business logic in one particular 4GL. However, eventually every 4GL decommissioned and this caused problems for the companies using them i.e. re-engineering the whole logic or migrating to another language [1]. Also, it may be a challenge to find new programmers who are familiar with the language. One of the possible solutions is the redevelopment of a decommissioned 4GL's compiler with modern technologies. Since 4GLs are procedural languages, modelling them using class diagrams or sequence diagrams is not feasible. This calls for a new solution to parse and represent a 4GL system. A tool support that goes beyond a simple parser or compiler is essential to ensure that other people adopt your language [17].

The authors used Uniface as a case study to find a workaround the above problem statement. The choice of Uniface is based on the fact that it is largely platform independent [13]. The language Workbench tools enhance the usage of a language and make it easier to avoid or resolve errors [11, 12]. However, unearthing the targeted syntax is the preliminary requirement since it needs to be updated to change the business logic. In this paper, the authors show how to extract Uniface Syntax from the code embedded in XML (Extensible Markup Language) [15] format. The extracted syntax is then transformed to a valid grammar that can be used by the XML parsers without errors.

Section II describes the problem statement that became the base of this research. Section III explains the proposed solutions by the authors. Section IV gives a brief summary of the paper. In addition, the design recovery method which is applied to Domain Specific Languages can demystify the process of extracting the data model and expose the business logic from within the system.

II. PROBLEM STATEMENT

The authors started their research by analyzing Uniface 4GL language to understand its structure. Figure 1 explains the code distribution for a Uniface file. Uniface stores content plus code in XML format while using XSD [10] for schema and XLS for styling. An example of exported codebase of Unifaces in XML format has been shown in Figure 2. Since Uniface codebase was exported as XML, the authors tried different XML parsers to visualize the document structure and schema. However, it failed to pass the XML validation test when using XML validators. In XML the term markup is used to determine tags where tags are nodes which start with an opening bracket which is "less than" symbol and closing bracket which is "greater than" symbol. The element consisting of less than symbol, content and greater than symbol is referred to as a tag. Each XML

0-7695-6405-4/18/$31.00 ©2018 IEEE
DOI 10.1109/UKSim.2018.00023

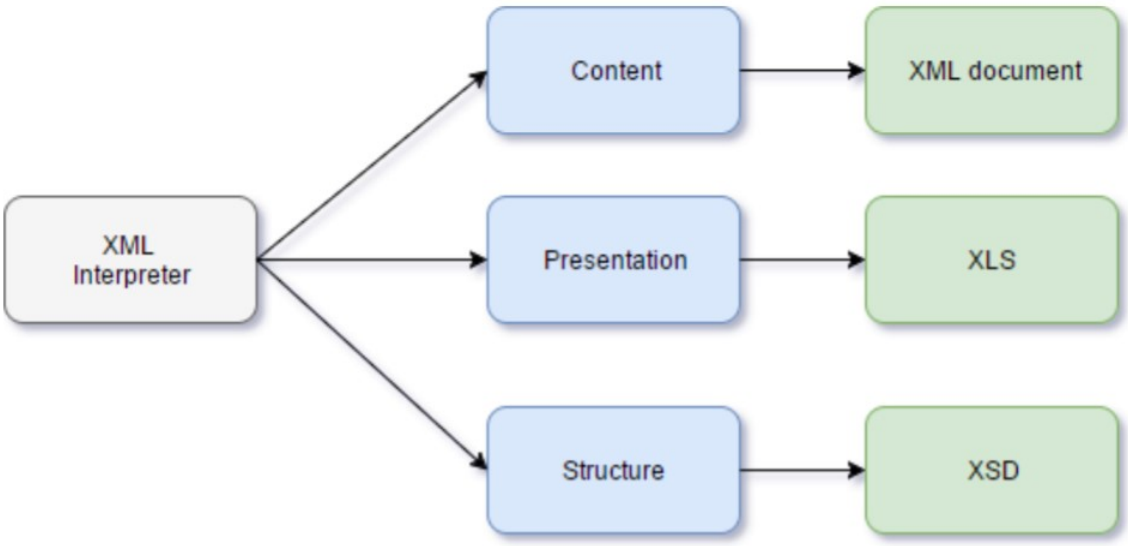

Figure 1. Extracted Data Structure from Uniface

document must be formatted in a way that tags are nested inside the root tag which defines the beginning and end of the document. If the document satisfies this requirement, then the document is flagged as a well-formed document. According to [6] "A node is a well-formed XML node if its serialized form, without doing any transformation during its serialization, matches its respective production in [XML 1.0] or [XML 1.1] (depending on the XML version in use) with all well-formedness constraints related to that production, and if the entities which are referenced within the node are also well-formed. If namespaces for XML are in use, the node must also be namespace well-formed". The XML parsers are designed to parse and iterate well-formed XMLs only.

There are a number of tools designed to tolerate inconsistencies in some node structures such as HTML interpreters. HTML document is a special case of XML like node structure which starts with and ends with HTML tag. All other HTML elements are nested in the HTML element. Therefore, XML and HTML have different Internet media type (aka. MIME Type) [7]. HTML interpreters are designed to detect unclosed tags and close them and dismiss the absence of root nodes. In contrary, XML interpreters are very strict and do not tolerate any inconsistency. Parsing Uniface XML data is not as straightforward as parsing a normal XML syntax. In fact, the attempt to parse the XML data resulted with errors due to ambiguity and use of Unicode characters and special characters that are needed to express specific functionality. This leads to malfunction of most of the off the shelf parsers regardless of which language and technique they were written in. In many cases, there are special characters which are typed within the content of the node e.g. symbols for question mark (?), exclamation mark (!), greater than (>) and less than (<). This deceives the interpreter which assumes '<' is a beginning of new node and '>' is a closure of the node. However, Uniface generates those characters for internal purpose only. The solution to this problem is to build an XML interpreter which could either parse the file as a string or use an advanced reader which stringifies XML rather manipulating it as an object

ALGORITHM 1:

```
Input:
    file: uniface extracted file
    n = 0
Output:
    files: files array

if(file)
  while readFile(file) ≠ EOF do
    node = readNode()
    if node.type == Table
      if node Is Not blank
        while readNode().type = Occ do
          temp_file_name = "Occ"+table+file
          xml = writeXML(temp_file_name, node.text)
          files[n] = xml
        end while
      else
        continue
      end if
    else
      continue
    end if
  end while
  writeIndex(files)
else
   showError()
end if
```

node.

III. Proposed System

Uniface code exported as XML is a bulk of code that is a mix of XML, HTML and code specific to Uniface (as shown in Figure 2. The Uniface program code is enclosed within nested nodes and elements. As discussed in Section II, the exported XML cannot be validated by an XML parser if it uses special characters in the Uniface code. Also, parsing large files beset with difficulties and leverage uncertainty whilst running custom procedures and methods on it. This makes difficult to debug when errors happen. Therefore, the perception is that creating smaller chunks of files which contain less code and may be narrowed down to avoid the causes of errors.

```
<?xml version="1.0" ?>
<!-- Created by UNIFACE - (C) -->
<UNIFACE release="9.5" xmlengine="2.0">
  <TABLE>
    <DSC name="UFORM" model="DICT"  ...>
      <FLD name="UTIMESTAMP"  ... />
      <FLD name="UCOMPSTAMP"  ... />
      <FLD name="ULABEL"  ...  />
      <FLD name="FTYP" seqno="4"  ... />
    </DSC>
    <OCC>
      <DAT name="UTIMESTAMP">2015-06-24T.
      <DAT name="UCOMPSTAMP">2015-06-24T.
      <DAT name="ULABEL">MAJD_TEST</DAT>
      <DAT name="WVPOS">3</DAT>
      <DAT name="WHPOS">1</DAT>
      <DAT name="WVSIZ">35</DAT>
      <DAT name="WHSIZ">85</DAT>
      <DAT name="CLRSCRN">N</DAT>
      <DAT name="UBORDER">N</DAT>
      <DAT name="STORE" xml:space="preser
          if ($status &lt; 0)
             message $text(1500)     ; err
             rollback
          else
            if ($status = 1)
              message $text(1723)    ; no
            else
              commit
              if ($status &lt; 0)
                rollback
              else
```

Figure 2. Example Uniface Codebase

Initially, the authors planned to divide the exported XML into equal small sized chunks based on the actual file size. However, while gathering the initial observations, the authors realised the fact that the Uniface XML file contains program code and interrupting it at any position means that statements fully or partially would be truncated. This would cause incomplete chunks of code e.g. an incomplete form or an incomplete if-else condition. Consequently, the program code cannot be properly visualized, updated, or upgraded. Therefore, the authors have used the schema to truncate only at closing tags for specific nodes. To split the file at specific nodes i.e. <Table>, a different parser is required that can split the file without parsing, extract the required nodes and write these to a new external file. The authors found this operation less efficient and very expensive in term of resource consumption.

The authors explored different techniques to speed up the process of splitting a large file and minimize memory consumption i.e. DOM (Document Object Model) [4], SAX (Simple API for XML) [5] and EDOM (Encapsulated Document Object Model). This research led the authors to propose Algorithm 1 to read a uniface code base. To read nodes in the exported XML from uniface, the authors used EDOM (Encapsulated Document Object Model) technique. EDOM is not a de facto standard in terms of parsing XML documents. However, it does not only provide a mechanism to parse, create and modify a document but also simplifies DOM interfaces [3]. The authors have chosen EDOM because of its high efficiency to parse node structures. Unlike DOM (that loads a whole file in the memory to be parsed), EDOM loads the document as strings which decreases the load on memory and speed up the process [2]. EDOM maintains an index array for the strings extracted from a document. At this stage, the parser does not validate the document against well-formed document standards. This feature relieves the developers from the complexity involved parse DOM, SAX, DTD (Document Type Definition) Schema and XSL (Extensible Stylesheet Language) [16] manipulation. In order to avoid stack overflow issue for devices which have limited resources, EDOM replaced the classic recursion method with element stack method which behaves like memory stack to index string elements [14]. However, the content is still stored in the memory heap. The aforementioned technique is built for devices with limited resources and also for huge files which consume the host hardware. It iterates over the document from start to end and stack code as strings. Since the code is read line by line, it does not need to load the whole file in the memory. Consequently, it saves memory to be overflown.

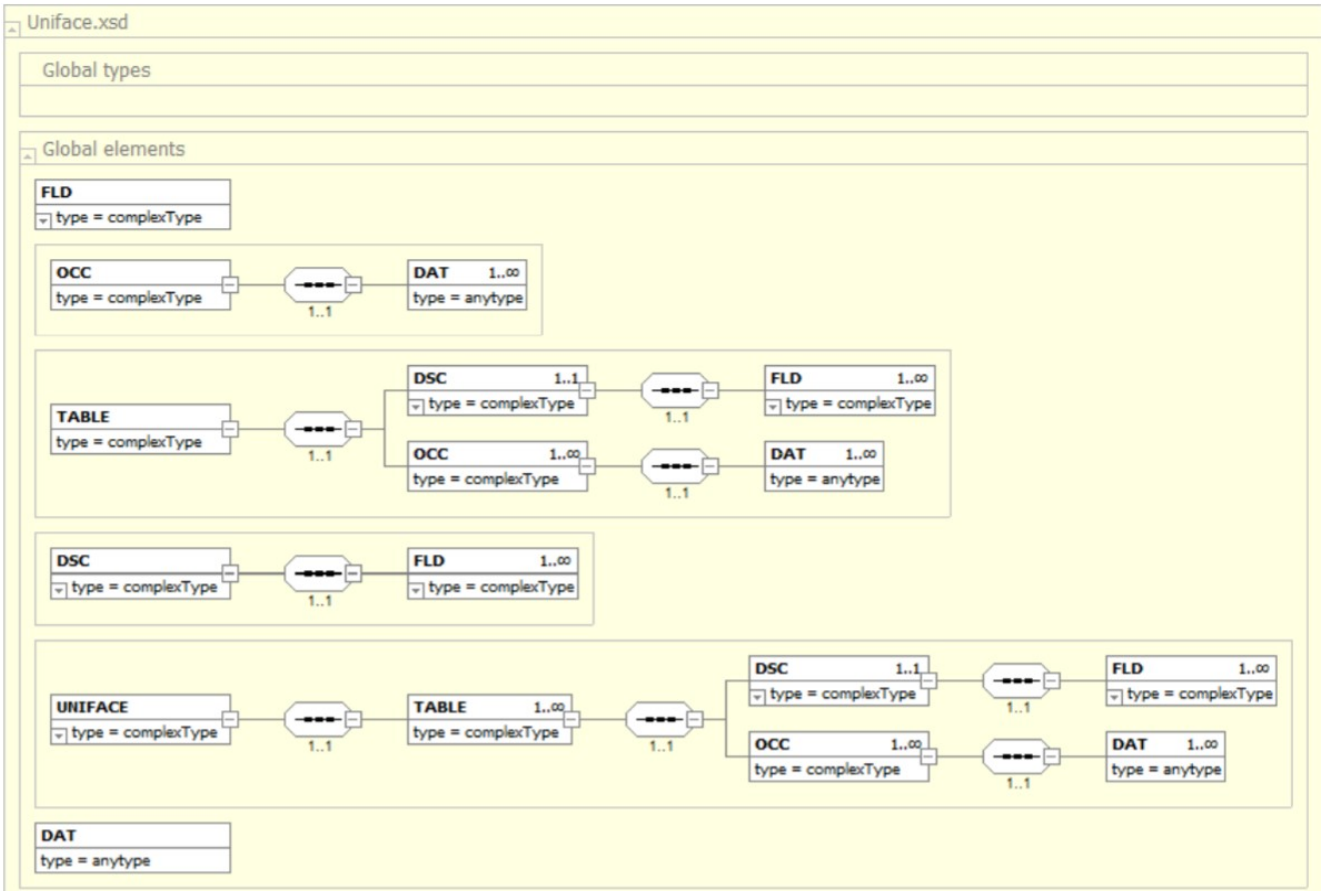

Figure 3. Uniface XML Schema Definition

The authors also noted a drawback for the approach to read uniface codebase and save in small chunks i.e. more storage space to accommodate files and a track record of file indexes. Also, it may be time-consuming to link the extracted chunks with their schema individually to verify their validity as well as to modify and combine the code together. Despite the limitations, the authors found that the plus points for this approach (i.e. the final outcome of being able to extract uniface code from an XML codebase), outweighed its limitation.

A. Explanation of Algorithm 1

The algorithm reads the exported uniface file from start to end. The readNode() function utilises EDOM to reads the file as text strings and identifies nodes. The first check it performs is to determine the tag type. If the character at the current position is less than symbol, it assumes that it is a tag. Then it performs a check against the next character whereby if the next one is slash then it is a closing tag. An exclamation mark denotes comments, CDATA section or DOCTYPE. A question mark coupled with angle bracket identifies a tag describing XML. The algorithm reads all nodes in the file to locate table nodes. Uniface exported XML contains a lot of unwanted information e.g. comments, holder code and entity nodes. Since this is not the interest of this work to process this data, the algorithm ignores all such data or nodes. The table nodes are identified by <Table> tag. It loops through each table tag and finds all OCC tags. OCC has been named after the word 'occurrence' that is a special tag in uniface XML to describe all occurrences of code in a table. Having the uniface language code, the OCC tags are the target nodes for the proposed algorithm. Before reading the <OCC> nodes, the algorithm checks for empty nodes. If a node is empty, it will be ignored by the algorithm. Ignoring empty occurrences helps for following three reasons

- It speeds up the process by reducing the number of occurrences to be processed.
- It saves the memory by not creating an XML for empty occurrence and not saving an index for the file.
- It also saves the system to read empty files in future.

Hence the algorithm reads all non-empty OCC nodes in each table and writes each occurrence to a separate XML file. Each XML is given a unique name combined with table number and occurrence number. All file names are recorded in an index file so that these can be retrieved in an order.

B. Testing Algorithm 1

To test the algorithm, the authors parsed multiple XML files from Uniface code base. A part of an example XML has been shown in Figure 2. The algorithm successfully read the exported XML and extracted uniface code.

To verify the extracted code, the authors utilized DTD (Document Type Definition) technique [8]. Visualising node structures like Uniface as DTD is commonly used to describe documents' meta-data [9]. Moreover, it validates the content of XML documents against used vocabulary. In addition, using DTD makes it possible to perform grammar check and validate syntax. The extracted document will be flagged as valid if and only if it strictly complies with the DTD definition. Figure 5 provides an overview for XML interpreted as DTD definition. DTD does not only define the elements that could be written in the XML document but also describe the order of these elements and their associated attributes. It can define attributes to be mandatory or optional. Hence DTD does not support name-spacing but only string datatype. The authors used DTD to validate the exported XML to verify if it matches the Uniface source code files used for testing.

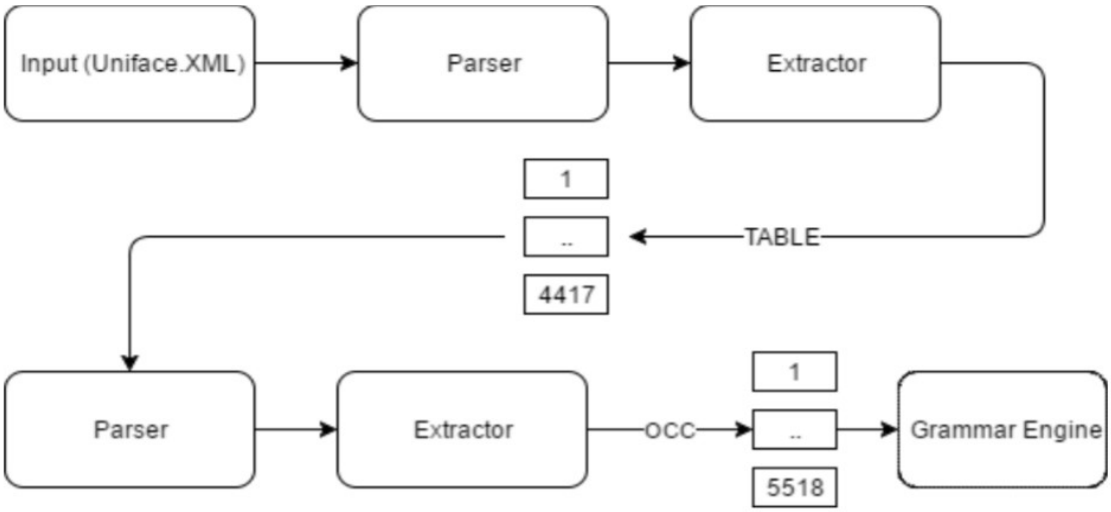

Figure 4. Proposed Uniface Processor

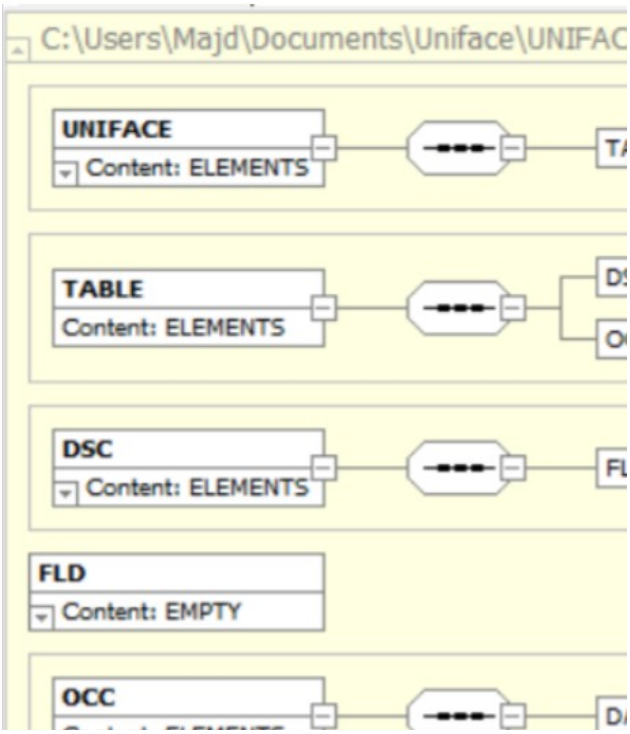

Figure 5. Uniface Data Type Definition

XSD (XML Schema Definition) is another way to validate XML and check if the grammar has precisely been applied to the input. DTD does the same job to detect well-formed documents and valid structure. However, XSD is a standard representation for XML structure and uses XML syntax which makes it extendible for further addition. Also, it supports name-spaces. Considering the above features of XSD, the authors generated Uniface XSD schema that helped to understand the structure of the code and vocabulary used to form it. Figure 3 shows the graphical representation of the extracted XSD from Uniface code.

The authors also successfully performed reverse testing to combine all code chunks with their XSDs to regenerate a complete uniface codebase. Figure 4 shows a graphical representation of the proposed system where it takes a uniface exported XML, reads it and extracts all Table entities. From each Table entity, it extracts all non-empty OCC nodes that can be passed to the Grammar engine for reverse testing or further processing.

IV. SUMMARY AND CONCLUSION

In this paper, the authors showed a handy method that improves the readability of Unicode XML format with special characters and illustrated the mechanism of

extracting the language syntax by omitting the special characters and unwanted nodes based on the generated schema.

The next phase to this research is to further process the extracted code from uniface codebase to translate into a more expressive language such as C#. In addition, we plan to generate an Abstract Syntax Tree for the extracted code and demonstrate it using Backus normal form and Syntax Diagram (Railroad Diagrams).

REFERENCES

[1] Z. Vandim, 2017. Incremental Coverage of Legacy Software Languages. In Proceedings of the Third Edition of the Programming Experience Workshop.

[2] Y. Shu, T. Chenggong, L. Quan and P. Xiaohong, 2009. Research of optimizing device description technology based on XML in EPA. In Electronic Commerce and Security, 2009. ISECS'09 IEEE. Second International Symposium on, vol. 1, pp. 561-564.

[3] Y. Yang, F. Lixin, Z. Chongquan and L. Xinkai, 2007. Optimization and Application of EPA Device Description Based on XML. In Control Conference, 2007. Chinese, pp. 175-179. IEEE, 2007.

[4] T. H. Gnech, S. Koenig, O. Petrik, and J. Roehrig, 2018. "Optimized read/write access to a document object model.. U.S. Patent 9,858,250, issued January 2, 2018.

[5] P. M. Gavali, 2018. Investigation of Mining Association Rules on XML Document. International Journal of Computer Engineeringin Research Trends (IJCERT), vol (5), 12-15.

[6] L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne, 2004. Document object model (dom) level 3 core specification, V 1.0. W3C.

[7] S. Faulkner, A. Eicholz, T. Leithead, A. Danilo, 2010. Document object model (dom) level 3 core specification, https://www.w3.org/TR/html51/ introduction.html#html-vs-xhtml.

[8] P. B. Monday, 2002. Tagged markup language interface with document type definition to access data in object oriented database. U.S. Patent 6,480,860, issued November 12, 2002.

[9] I. H. Witten, B. David, P. Gordon and B. Stefan, 2002. Importing documents and metadata into digital libraries: Requirements analysis and an extensible architecture. In the proceeding of International Conference on Theory and Practice of Digital Libraries, pp. 390-405. Springer, Berlin, Heidelberg, 2002.

[10] M. Keith, S. Merrick and N. Massimo, 2018. XML Mapping Files. In Pro JPA 2 in Java EE 8, pp. 593-654. Apress, Berkeley, CA, 2018.

[11] S. Erdweg, T. V. D. Storm, M. Völter, M. Boersma and R. Bosman, 2013. The State of the Art in Language Workbenches — Conclusions from the Language Workbench Challenge. In Proceedings of the Sixth International Conference on Software Language Engineering (SLE) (LNCS), Vol. 8225. Springer, 197–217.

[12] M. Fowler, 2005. Language Workbenches: The Killer-App for Domain Specific Languages. June, 2005.

[13] P. Lancaster, 1996. Technology Briefing Report RAD (Rapid Application Development). V 1.0, 31st July, 1996.

[14] FirstObject, 2014. EDOM documentation. http://www.firstobject.com

[15] A. Grinberg, 2018. Introducing XML. In XML and JSON Recipes for SQL Server, pp. 3-22. Apress, Berkeley:CA.

[16] D. T. Judd, A. B. Jason, P. M. Michael and J. L. David, 2017. Content management and transformation system for digital content. U.S. Patent 9,686,378, issued June 20, 2017.

[17] E. Moritz and H. Behrens, 2010. Xtext: implement your language faster than the quick and dirty way. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (pp. 307-309).