An XSLT Translator for the openEHR Archetype Definition Language

Abstract

The cityEHR Electronic Health Records system is a pure XML application for managing patient health records, using open standards. The structure of the health record follows the definition in the ISO 13606 standard, which is used in cityEHR as a basis for clinicians to develop specific information models for the patient data they gather for clinical and research purposes. In cityEHR these models are represented as OWL/XML ontologies. The most widely adopted approach to modelling patient data in accordance with ISO 13606 is openEHR, which uses its own Archetype Definition Language to specify the information models used in compliant health records systems. This paper describes a translator for the Archetype Definition Language, implemented using XSLT and XML pipeline processing, which generates OWL/XML suitable for use in cityEHR.

Introduction

A typical electronic health record contains an array of information about a patient, built up over their lifetime; some information is very structured (laboratory text results, for example), some almost completely unstructured (e.g. notes on a consultation in clinic). The ISO-13606 standard [1] includes a specification for the structured representation of the health record as a set of Compositions, containing Sections with Entries containing Elements and Clusters of Elements. The Compositions can (optionally) be organised in Folders, so that the whole health record resembles a filing cabinet (the EHR_Extract) containing folders of paper documents, as shown in Figure 1.

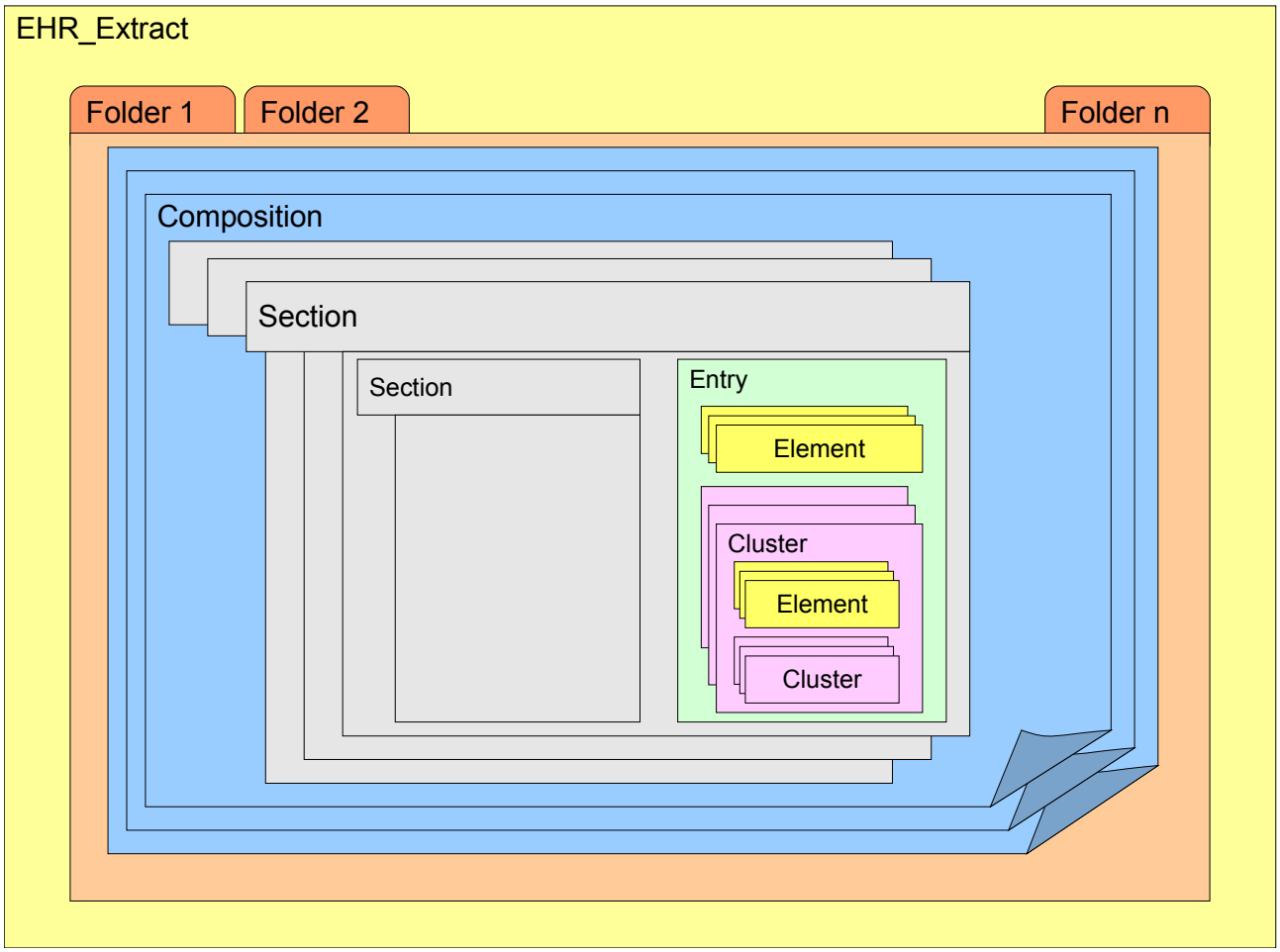

Figure 1. Structure of an Electronic Health Record in ISO-13606

The HL7 CDA standard (Health Level Seven, Clinical Document Architecture) [2] defines an XML vocabulary for the markup of a clinical document and follows the same basic structure as the ISO-13606 standard, starting at the composition level (a Composition in ISO-13606 is equivalent to the document element cda:ClinicalDocument in HL7 CDA).

The structure of the health record defined by ISO-13606 and HL7 CDA is completely generic; the same structure applies to a record of vital signs made during an operation, to a set of laboratory text results sent from the lab to a general practitioner or to questions about lifestyle answered by a patient when attending a clinic.

The cityEHR Electronic Health Records system, which has been developed since 2010, is an XRX (XForms, REST, XQuery) application for managing electronic health records [3]. The system runs as an Enterprise Java application, using the Orbeon Forms framework [4], accessed through a web browser and web services interfaces. It is designed as a completely open system, using XML and healthcare-specific open standards, and although running through a Java framework, it contains no Java code of its own; all code is XML, XSLT, XQuery, XForms, XHTML or CSS. The health record in cityEHR is stored as a collection of HL7 CDA XML documents in the eXist native XML database [5].

cityEHR uses a multi-stage modelling approach to define the clinical data set that will be gathered in any specific clinical setting. The intention is that clinicians themselves should design the information models that define the entries and elements in the dataset for their specific requirements, which are usually a combination of the requirements for their clinical specialty (e.g. Rheumatology, Respiratory, Liver and Bilary, etc) and the requirements for secondary use of the data in clinical studies. The stages of modelling in cityEHR are shown in Figure 2.

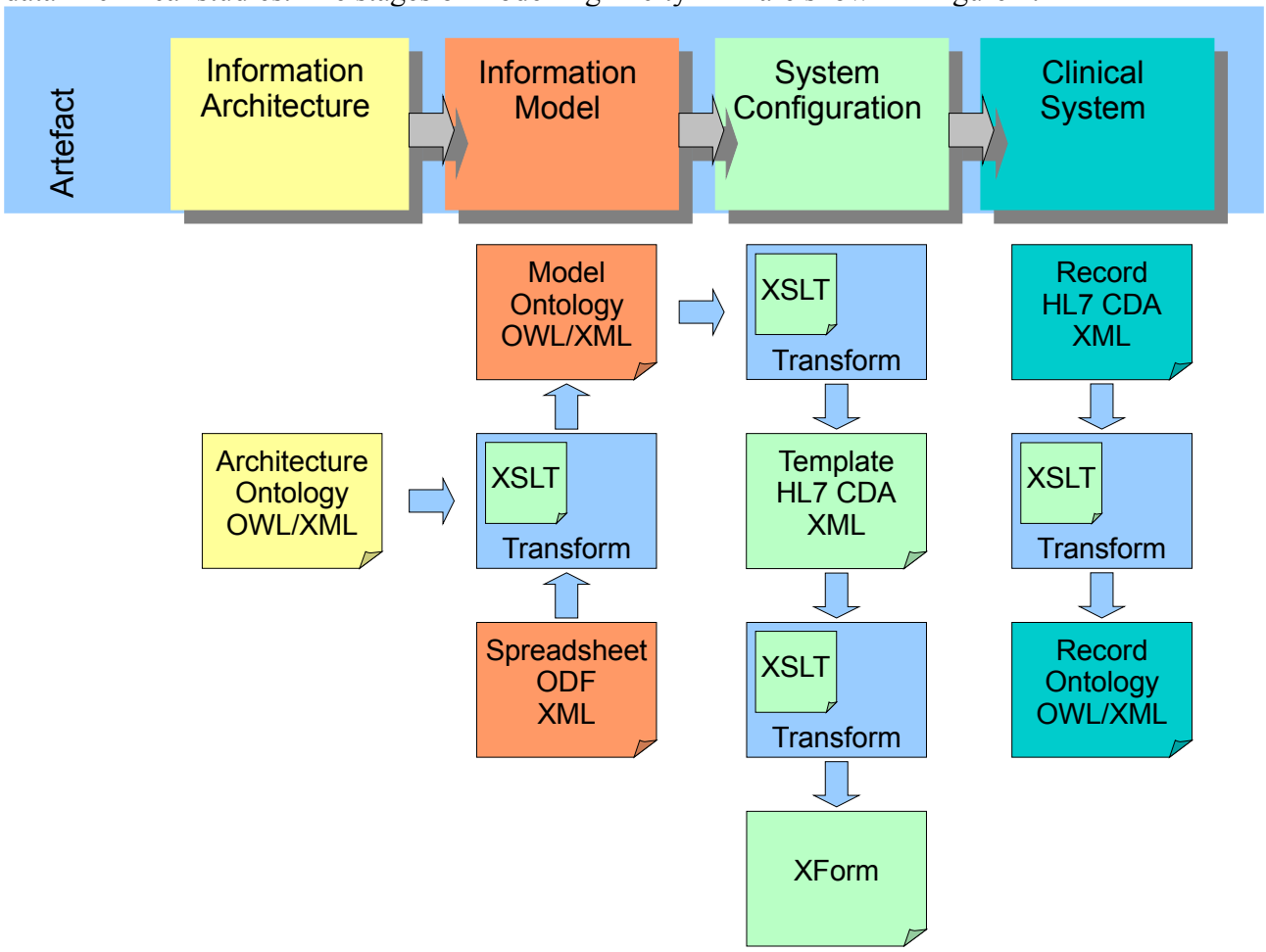

Figure 2. Information Models for the Electronic Health Record in cityEHR

The multi-stage modelling approach used in cityEHR is inspired by an earlier initiative, whose roots can be traced back almost thirty years. openEHR (openehr.org) is an open framework for modelling healthcare information, in a form suitable for the design and implementation of Electronic Health Records systems. It builds upon a number of funded European initiatives, from the early 1990's onwards, and the ISO 13606 standard that arose from those initiatives [6].

openEHR has an abstract reference model for healthcare information that is based on, and extends, ISO-13606. It uses a two-stage modelling process whereby the models for specific clinical settings, called Archetypes, are specified using the Archetype Definition Language [7] which applies the constraints of the Archetype Object Model. In its simplest form, an 'archetype' corresponds to an Entry in the ISO-13606 model (cda:Entry in HL7 CDA), but a collection of entries can be grouped into a 'template' which corresponds to a Composition in ISO-13606 (cda:ClinicalDocument); to achieve this, a template may include other externally defined archetypes. A 'template_overlay' is an Archetype that is used to overload, or specialize, an existing template (for example an internationally published template may be adapted for use in a particular local setting); an 'operational_template' is a template that has been 'flattened' by expanding any included archetypes and applying any overlays. All four types – archetype, template, template_overlay and operational_template – are referred to generically as Archetypes.

The Archetype Definition Language, in its original form, was an example of a domain-specific language [8], created to describe specific clinical information models that conform to a more general model of the structure of an Electronic Health Record. Since it was first conceived, it has been generalized, so that it can be used to define any set of models that conform to any underlying reference model. This generaliztion has made ADL similar to an XML schema language, except with its own unique syntax, and complicates the task of creating a parser in XSLT. However, since XSLT2 is a Turing complete language [9], it should, theoretically, be possible to build such a parser.

In recent years there has been a growth in the number of published archetypes available through public repositories and there are now a number of large-scale EHR systems, both commercial and open source, that can be configured using them (so in that sense can be said to be openEHR compliant systems). In consequence, it was decided that the cityEHR should be enhanced with the ability to consume and generate information models expressed using ADL.

Although there are existing Java implementations of ADL parsers and AOM serializers [10], the design principles of cityEHR prohibit the use of third-party Java code, outside the unmodified code of Orbeon and eXist. To maintain this principle, an ADL translator has been developed in XSLT 2 so that cityEHR can now be configured using information models published using ADL.

Methods

The ADL translator is implemented using the principles of multi-pass compilation, whereby the target code is constructed in stages, through multiple passes through the source code [11]. The multiple stages are implemented using the XML pipeline implementation in Orbeon Forms, and follow the four classical stages of a multi-pass complier: lexical analysis, syntax parsing, semantic analysis and target code generation.

Each of these stages is implemented in XSLT2, using the regular expression support in the XPath functions tokenize and matches, together with the XSLT analyze-string element and a number of recursive templates.

The output of the first three stages is an Abstract Syntax Tree (AST), in XML, representing the source ADL at increasing levels of detail. This is similar to the approach described by Badros [12] in generation of JavaML, an XML representation of Java source code, and more generally by

Maletic et al [13]. By the final stage, the input AST is complete enough to generate the target code, which is a set of OWL/XML assertions suitable for loading in cityEHR, to define its information model.

The regular expression (regex) capability of the XSLT analyze-string element can be used to parse parts of the source ADL. However, the mathematical model underlying regex is that of a Finite State Automaton – a process which consumes the source code string by matching patterns and at each point is in a single state, from a finite set of defined possibilities. This model is not sufficient to parse the repeated nested patterns that exist in some sections of ADL; these require a more sophisticated Pushdown Automaton, which uses a 'stack' to keep track of the state of nested patterns which could lead, in theory at least, to an arbitrary number of states. One method of implementing a Pushdown Automaton in XSLT is to use a recursive template to 'push' the current state onto the stack, which is passed as a parameter to the template.

Structure of the Archetype Definition Language

An ADL source code file can be one of four types – archetype, template, template_overlay, operational_template – which is specified in an initial ADL Declaration. Following this declaration, the overall structure is a sequence of sections; the sections are always in the same order but some are included, excluded or optional, depending on the declared type.

The ADL Declaration has its own syntax and within each subsequent section the syntax follows one of three defined sub-languages; cADL (constraint ADL) [14], ODIN (Object Data Instance Notation) [15] or FOPL (First Order Predicate Logic). In addition, there are several 'sub-languages' that are used at defined points in the ADL code for referencing objects in the model (the ADL Path Language) and the The overall structure of the ADL source code for an archetype of type 'template' is shown in Figure 3.

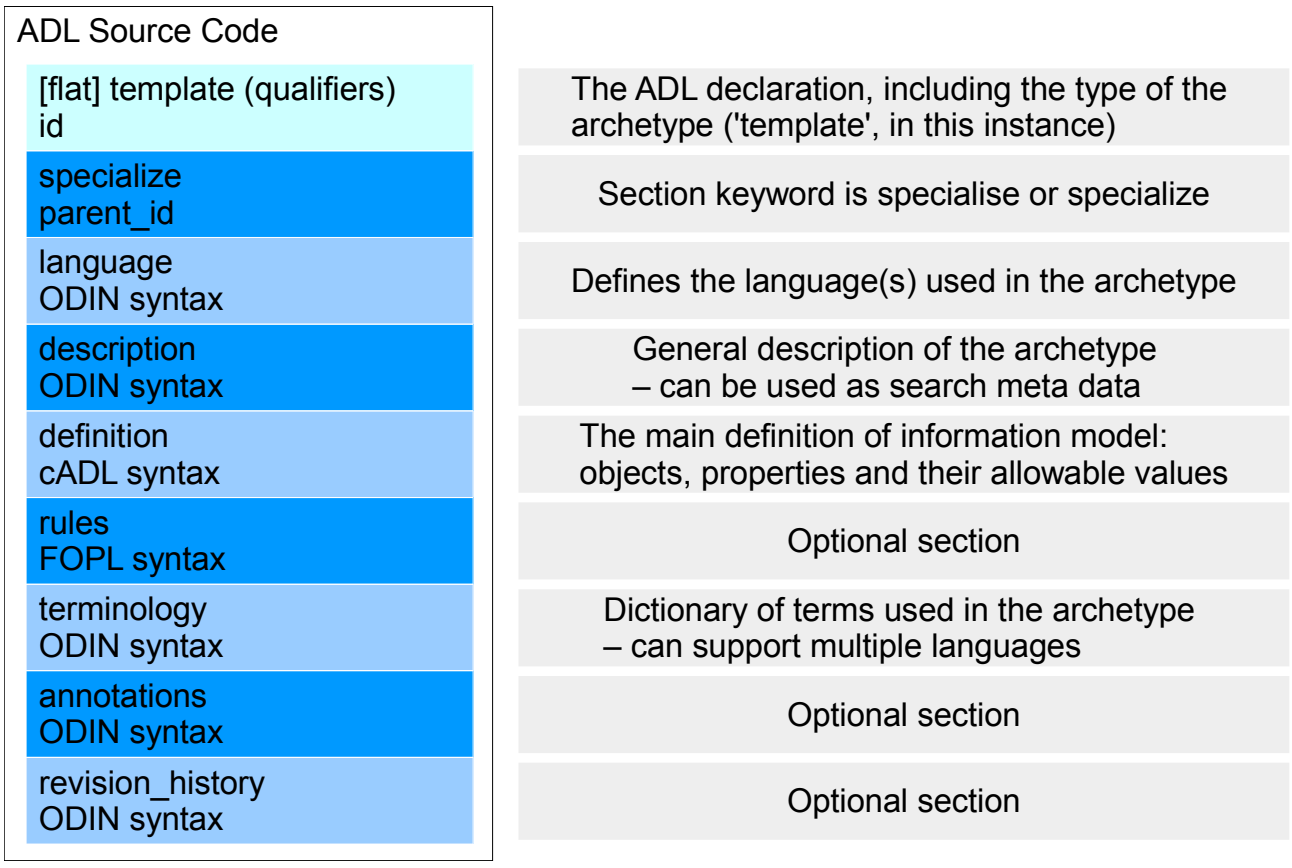

Figure 3. Overall structure of ADL Source Code for a 'template' archetype.

Preparation of ADL Source Code

The ADL source code is read as a plain text from the file system and prepared as the text content of an XML document within a single document element. However, there are a couple of steps that must be taken before this document is ready for input to the lexical analysis.

The syntax of ADL includes some code that can look like XML markup, including heavy use of angled brackets and the allowance of ampersand in text strings and comments. These must be escaped using `<` and `&` which is achieved by the Orbeon pipeline processor called url-generator which reads a plain text file from a URL and escapes possible markup, working in the same way as the XSLT function `unparsed-text()`.

The ADL specification states that ADL files are stored in Unicode UTF-8 encoding. As such, an ADL source code file is likely to start with a Byte Order Mark (BOM) which is a standard way to signify the Unicode encoding to processors. In UTF-8, the BOM is the sequence of three bytes `0xEF,0xBB,0xBF` and if present these three bytes are stripped from the ADL source code before processing using the XPath function `substring-after`.

Lexical analysis

The initial stage, lexical analysis, takes the ADL source code as input and returns a simple XML document with the source split into sections and any ADL comments removed. The type of the archetype is found using the XPath `matches` and `substring-before` functions.

Based on the type of the archetype, a string variable is set, representing the finite state model for that type. The ADL source code is then split into individual lines using the XPath tokenize function, which makes it easier to identify ADL keywords (which appear in isolation at the start of a line) and to strip comments which appear from any unescaped occurrence of the string -- until the end of the line.

The main parsing is achieved by a tail-recursive named template, parseSections, which takes as its parameters the tokenized sequence of line of source code, the finite state pattern for the archetype type, the current state, tokenized sequence of next allowable states and the content of the current section which builds on each recursive call.

When one of the keywords for the next state is encountered, a `section` element is generated for the current state, and the state machine moves forward to the new state.

The resulting document is a set of sections, each containing the relevant ADL source code, with comments removed.

Syntax analysis

The syntax analysis parses each section of the document generated from the lexical analysis, using a separate parser for each of the three sub-languages. The parsers for the cADL and ODIN syntax are implemented in the same named template, since both are essentially nested block structures; the syntax is different, but the basic structure is the same.

The main syntax parsing is achieved by a named template called parseADL which is passed a parameter determining whether the section is cADL or ODIN syntax. Depending on the syntax, variables are set for the block delimiters ({ } in cDAL < > in ODIN) and for the regular expression that matches the pre-amble for a block in each language.

The pre-amble pattern is then used in an xsl:analysze-string element to find the start of each block and the unmatched source code then consists of the delimited code for that block, followed by the code for subsequent blocks. The unmatched source code is read from the start to the end of the block using a named function called extractBlockContent which is called recursively to read the

source until the opening and closing delimiters are matched. A block element is then output, with its content generated by a recursive call to the parseADL template (this is not a tail-recursive call, but blocks in cADL or ODIN are rarely nested more than four deep, so the XSLT code is unlikely to run out of stack space in this recursion). The remaining unmatched source code is then processed using a tail-recursive call to the parseADL template.

Semantic analysis

The general goal of semantic analysis in a multi-stage complier is to ensure that source code which follows the correct basic syntax of a language also adheres to the higher level semantic rules governing the relationships between objects in the language. Hence semantic analysis can only take place once the source code has successfully passed lexical and syntax analysis.

In this ADL Translator, the semantic analysis plays three roles. The first is to check that the parsed syntax form the previous stage meets the requirements and constraints of the openEHR Reference Model; second is to further parse the assertions and path expressions and to resolve references (both internal and external); the third is to reconcile differences between the openEHR Reference Model and the model architecture of cityEHR.

The output from the semantic analysis is an XML document representing the Abstract Syntax Tree of the ADL source code at a level of detail sufficient to generate code for a cityEHR information model.

Code generation

The code generation stage is a relatively straightforward, compared with syntax and semantic analysis. The Abstract Syntax Tree output from semantic analysis is matched using simple templates which generate the equivalent OWL/XML assertions.

Conclusions

It has proved feasible to implement a translator for the Archetype Definition Language using XML pipeline processing and XSLT 2. The target language for the translation was OXL/XML, using assertions specific to the cityEHR architecture, but target code could also be generated in other representations, if required.

The motivation for implementing using pure XML technology was to avoid dependence on third-party Java packages and achieving this is probably the main advantage offered by the methods outlined in this paper.

There is a large body of ADL archetypes and templates available in public repositories and the next step in this work will be to test the translator against a sample of these test cases, which will inevitably lead to refinement and improvement of the first version described here.

Although the translator described here is sufficient to generate valid OWL/XML information models for cityEHR in a practical subset of ADL archetypes, it is envisaged that more thorough testing will lead to additional refinement, especially of the semantic analysis stage, in order to accurately translate any published ADL archetype.

References

1. ISO 13606-1:2008 Health informatics - Electronic health record communication - Part 1: Reference model. https://www.iso.org/standard/40784.html
2. Dolin RH, Alschuler L, Boyer S et al. "HL7 Clinical Document Architecture, Release 2." J Am Med Inform Assoc. 2006;13(1):30-9.
3. Chelsom, J.J., Summers, R., Pande, I. and Gaywood, I., 2011, June. Ontology-driven

development of a clinical research information system. In *Computer-Based Medical Systems (CBMS), 2011 24th International Symposium on* (pp. 1-4). IEEE.
4. Bruchez, E., 2006. Orbeon, Inc.[Online]. XTech 2006: XForms: an Alternative to Ajax. XTech.
5. Meier, W., 2002. eXist: An open source native XML database. In *Web, Web-Services, and Database Systems* (pp. 169-183). Springer Berlin Heidelberg.
6. Kalra, D., 2006. Electronic health record standards. Schattauer GMBH-Verlag.
7. Beale, T., 2002, November. Archetypes: Constraint-based domain models for future-proof information systems. In OOPSLA 2002 workshop on behavioural semantics (Vol. 105).
8. Mernik, M., Heering, J. and Sloane, A.M., 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, *37*(4), pp.316-344.
9. Kepser, S., 2002. *A proof of the Turing-completeness of XSLT and XQuery*. Technical report SFB 441, Eberhard Karls Universitat Tubingen.
10. Chen, R. and Klein, G., 2007. The openEHR Java reference implementation project. *Studies in health technology and informatics*, *129*(1), p.58.
11. Aho, A.V. and Ullman, J.D., 1972. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc.
12. Badros, G.J., 2000. JavaML: a markup language for Java source code. *Computer Networks*, *33*(1), pp.159-177.
13. Maletic, J.I., Collard, M.L. and Marcus, A., 2002. Source code files as structured documents. In *Program comprehension, 2002. proceedings. 10th international workshop on* (pp. 289-292). IEEE.
14. The openEHR Foundation. Archetype Definition Language ADL 2, Version 2.06, 2016. http://www.openehr.org/releases/AM/latest/docs/ADL2/ADL2.html
15. The openEHR Foundation. Object Data Instance Notation (ODIN), Version 1.5.2, 2017. http://openehr.org/releases/BASE/latest/docs/odin/odin.html