A Product Line and Model Driven Approach for Interoperable EMR Messages Generation

Deepa Raka

December, 2010

Department of Computer Science
California State University, Fresno
A Product Line and Model Driven Approach for Interoperable EMR Messages Generation

Deepa Raka

December, 2010

Department of Computer Science
California State University, Fresno
Fresno, CA 93740

Project Advisor
Dr. Shih-Hsi Liu

Submitted in partial fulfillment of the requirements for the degree of Master of Science
Abstract

A Health Information System (HIS) is a broad, integrated and standalone information system, designed to manage administrative, financial and medical records within a hospital and/or among hospitals. One of the fundamental parts of this system is called Electronic Medical Records (EMRs). EMRs are computerized legal medical records produced in hospitals. These records can conform to different medical messaging standards such as Health Level 7 (HL7), HL7 CDA (Clinical Document Architecture) or openEHR (Electronic Health Records). In order to achieve the efficient transmission of all such messages across heterogeneous systems, in terms that each of them follows a different standard to record and store EMRs, such messages need to be interoperable and in XML format. This problem may be solved by introducing a domain specific modeling language that defines the common and variable parts among various message standards (HL7, HL7 CDA and openEHR) using the concepts of Software Product Line Engineering (SPLE). Additionally, transformation rules that convert messages from one EMR standard to another is also one of the important features introduced. Our approach will help to build various messages that will conform to different EMR standards and may also help in achieving real-time requirements for the message transmission and reception.

The contributions of this project are three-folds: (1) Users can specify EMR messages in the form of models, making them simple to design and potentially address evolution through model transformation techniques; (2) By applying SPLE concepts, a set of resultant messages sharing common and variable parts may be generated at once; and (3)
The message generation can be extended to other domains as well, making it a more generalized application.
Preface

The idea of this project came from the study of the messages generated in Health Information System (HIS). These messages, known as Electronic Medical Records (EMR), are presented in XML format and represent the critical and confidential information about a patient. Also, they can conform to different available medical message standards, such as HL7 and openEHR. The availability and speed of access of such messages can be very crucial in certain conditions. Also, a patient can have a number of EMRs, for example, records representing his demographic data or clinical observations. Trying to generate such messages one at a time can be a very cumbersome task. Additionally, messages that conform to different standards need to be made interoperable.

A potential solution to this problem is to analyze and separate the message sections as common and variable parts, and develop a technique or process to write the common parts to all the messages and the variable parts to their corresponding messages. Also, because a message is in XML format, care had to be taken to properly place the opening and closing tags in each section, so that the overall message had a meaning. Additionally, if a message is specified in a certain standard, and it has different message schema and tags in another, a mapping has to be provided to automatically generate the message in different standard with its corresponding tags.

To accomplish all the aforementioned objectives, multiple existing technologies such as Software Product Line Engineering (SPLE), Feature Oriented Domain Analysis (FODA) and Model Driven Engineering (MDE) are applied. FODA helps to classify the tags in
the messages as mandatory, optional, alternative or more-of tags. This classification of tags using the FODA method allows users to get all possible and required combination of tags. For example, if a user specifies the tag as mandatory, it has to be present in all the messages while the optional tag may or may not be present in the message(s). If the tags are specified as alternative, only one of them can be present in the message. Depending on this classification, the number of output messages can be determined. In addition to this classification of tags, users can also provide the transformation rules to be applied to them, so that the message(s) are converted from one EMR standard to another.

After the tags are classified, the concept of Software Product Line Engineering is applied to write the common tags to all output messages and the variable tags to the specified messages. The third technology, Model Driven Engineering (MDE), was used to represent the message in the form of a model conforming to the metamodel which represents the rules and semantics of an XML message. Thus the model has to follow the relationships, associations and the restrictions specified in the metamodel to represent the XML message. This has an advantage that the metamodel could be changed anytime, to contain the required rules and every model based on this metamodel and then has to obey those rules. The model, which would represent the real world instance of a message, would then be interpreted using the code generation program done in Java and will give us the output messages in the XML files.

The goal of this project is reached after the message(s) generation was successful by using the above technologies. There are some potential challenges that were addressed during the completion of this project. Initially, developing a metamodel in the tool GME[9] took a while, as the tool was new to me. Also, to identify the different elements
in the metamodel and the relationships between them was a difficult task. The second challenge was to write an interpreter such that the messages were generated, with separate variable and common sections at multiple levels in the hierarchy, using the SPLE concepts. Additionally, generating a message that conforms to the standards, HL7 and openEHR, from a single input model was one of the tough tasks.

After completing this project, I have a better understanding about the different software engineering concepts and technologies and how they can be combined and put together to use to solve any new problem that arises as a part of any project.
CONTENTS

ABSTRACT

PREFACE

CONTENT

CHAPTER 1
PROJECT OVERVIEW
Introduction

CHAPTER 2
BACKGROUND
Electronic Medical Record
EMR Standards
Software Product Line Engineering
Feature Oriented Domain Analysis
FDL Rule Engine
Model Driven Engineering
Generic Model Environment

CHAPTER 3
PROJECT APPROACH
Project Flow
Metamodel
Model
Mapping
Case Study
Complete functionality of the project
Generalization

CHAPTER 4
RELATED WORK
Web Based HL7 message generation and validation
HL7 V3 Java API
Interoperability in HealthCare
Comparison between Related Work and the project
A Health Information System (HIS) [1] is a broad and standalone system used to store clinical and administrative data in a hospital. The HIS stores this information in the form of digital and electronic records called Electronic Medical Records (EMR) [2]. An EMR is represented in XML format and it can conform to different healthcare standards, such as Health Level 7 (HL7) [3] or openEHR (Electronic Health Record) [4], which may cause the issue of interoperability between these messages. Thus, a hospital using one standard for EMRs cannot understand the EMRs that conform to other standards and so the transmission of such messages across various hospitals becomes a challenging task. Developing a potential solution for the above mentioned problem served as a motivation for this project. In order to develop this solution, various existing technologies such as Software Product Line Engineering, (SPLE) [5] using Feature Oriented Domain Analysis (FODA) [6, 7] and Model Driven Engineering (MDE) [8] are applied.

The first step towards this solution is to design a metamodel that could define the grammar of EMR standards-conformed XML messages (for simplicity, we call them XML messages from now on). The metamodel is created such that it includes the syntactical rules and restrictions for the XML messages. For example, an atomic tag in an XML message cannot have sub tags, unlike composite tags, and this rule should be defined while introducing the metamodel. Also, the metamodel also defines syntax that allows users to specify common and variable tags conforming to different standards as
well as necessary structural transformation rules so that output XML messages, combining different tag combinations and structures, could be generated in parallel, and hence still conform to associated standards.

The second step to generate XML messages is to specify an input model which follows the grammar defined in the metamodel. Namely, the metamodel can be considered as a class and the model can be considered as an object of this class from the perspective of object-oriented concept. The input model specifies the XML message tags at different levels, and every tag can have a name, attributes and the text. In order to generate a set of XML messages having common and variable features (tags in this case.) following SPLE concepts, these tags can be classified to either mandatory, optional, alternative, or more-of, according to FODA. Additionally, the report introduces transformation rules that can be applied to the input model, so that the structures of output XML messages can be updated accordingly. For example, the two tags that appear at the same level in first output message standard can become a parent–child tag pair in the other. These transformation rules can be specified in the input model by connecting the desired tag(s) using a connection, which can have one of the following properties: Fold, Inverse, Add or Remove. Users can also apply a rule to change an attribute in one XML message to be a subtag in the other XML message.

After the desired model is created as an input, it is interpreted by the code generation program. This program interprets the input model in a way that it ultimately creates the XML messages(s) in HL7 and openEHR standards. For this purpose, the interpreter first creates the data structures that store the information about each tag, such as its level in the message hierarchy and its association to the parent tag. This is when the transformation rules are applied, so that the data structures for the message, which will
conform to second standard, have the correct association between the connected tags as specified in the model. This serves as a major step in generating the interoperable messages from the given input model. After these data structures are generated, the interpreter makes use of the Feature Definition Language (FDL) Rule Engine [10, 11] written in Java. This rule engine takes the lang file, which is the textual representation of the feature classification, as an input and gives all the possible combinations of the features (or tags) as output.

But, the major challenge in generating the messages from these files is that they just have the simple comma separated list of tags, which shows no association of a tag to its parent and also its level in the message(s). This information is very crucial for making a proper XML message, because it is important that the tags should be properly nested and placed in order to retain their meaning. At this point, the data structures created earlier by the interpreter serve as the solution of the problem. The interpreter uses the output text file of the rule engine and the corresponding data structures in combination in order to generate the XML message(s). This is one of the contributions of this project. Also, the FODA and the transformation rules can be applied to the model simultaneously, which is another significant contribution of the project.

From the above discussion, it can be concluded that any input model will give at least two messages, one for HL7 and the other for openEHR by using the transformation rules and if every tag is classified as mandatory. But if the tags are classified as either optional, alternative, or ‘or’ according to the FODA, then the number of messages will increase depending on the tag combinations and hence these messages will be generated in parallel.
CHAPTER 2

BACKGROUND

This section summarizes the knowledge and techniques utilized in the proposed approach, including EMR, SPLE, FODA, MDE, and a toolkit to realize MDE, called Generic Modeling Environment (GME) [9].

2.1 Electronic Medical Record (EMR)

An Electronic Medical Record (EMR) [2] is a computerized digital medical record generated and used in an organization which delivers care and medical services, such as a hospital. EMRs are likely to be a part of a local individual health information system [1]. Such a system facilitates the storage, retrieval and update of medical records. EMRs can be considered to be more powerful than a simple database, as it offers many functions, such as a combined and total view of patient data, clinical decision support and order entries, integrated communications support, and access to knowledge resources [2]. If EMRs could be interfaced to other systems, such as billing, pharmacy, radiology, and scheduling, among others, it can prove to be much more beneficial and effective. If fully developed and integrated, EMRs along with HIS may meet the requests for real-time data access and assessment of data in a hospital.

EMRs can conform to various standards available for the medical messages. They are present in XML format, making it easier to understand and possible to transmit between the systems. Also, these XML messages can be stored in a computer unlike paper records.
2.1.1 EMR vs. Paper Records

➢ **Cost**

The average cost of storing a medical record in a paper form is much higher than the per unit cost of storage of record on the digital media.

➢ **Storage Space**

The paper records require a significant amount of storage space compared to digital records.

➢ **Integration**

When different records need to be combined and displayed together, it is much more easier to integrate digital records as compared to paper records. Combining and integrating paper records based on certain criteria is a cumbersome and time consuming task. It can be made much simpler and faster if records are stored digitally.

➢ **Transportation costs**

When paper-based records are required in multiple locations, copying, faxing, and transporting costs are significant compared to duplication and transfer of digital records among multiple locations.

2.1.2 EMR Advantages

➢ **Record Sharing**

If EMRs are stored on the central database server, they can be easily shared among different related entities such as patients, physicians, and clinics, among others.

➢ **Faster Retrieval**

Whenever there are critical situations such as accidents, patient information can be retrieved faster.

➢ **Safe Records**
If patient records are stored electronically, they may probably become safe and secure if properly encrypted and stored [12].

➢ Legal Document

In case a conflict arises between any two entities, such as a patient and his/her physician, this record can serve as a legal proof in the court.

2.2 EMR Standards

EMRs generated and used across healthcare organizations can conform to and be presented in different standards. This report concentrates on HL7 and openEHR standards.

2.2.1 Health Level 7 (HL7)

HL7 v2.x [3] represents the medical messages in the plain text form using the Electronic Data Interchange (EDI) format [13], which uses pipes (|) and carets (^) as field separators. Since it was in pre Internet format, the communication exchange was not possible.

HL7 v3 [14] represents messages in XML format. It uses the sophisticated Object Oriented Information Model, called Reference Information Model (RIM) [14], to generate and represent medical messages. It basically compromises six core classes, as shown in Figure 1 below.

![Figure 1: HL7 Core Classes [14].](image-url)
The Act class is used to represent an action that occurs and needs to be documented in the hospital. An Entity represents the physical thing that actually takes part in the entire process. For example, a doctor and a patient become entities for the act of doctor’s appointment. The Role class defines the role that each entity plays while participating in the Act. The same person can play different roles in different Acts. The Participation class provides an association between the Role and Act. The Act Relationship is used to define the relation between two Acts if any. Similarly, the Role Relationship is used to establish a relation between two roles in case they depend on each other.

2.2.2 HL7 Message

HL7 messages are made up of segments, composites and primitive data types. An HL7 message consists of the following data elements:

Message Type:

Every HL7 message has to have a message type ID present in the header section. This ID is a unique identifier for the message. It indicates the purpose and aim of the creation of the message. For example, an HL7 message with the message type ID of ADT is a unique message ID to Patient Administration.

Message Event:

A message event, sometimes also called message trigger, is used to identify the context in which the HL7 message was generated. Similar to message type, a message event is also present in the message header section. A message event is represented using an upper case letter followed by two digits. For example, A01 is for an admission/visit notification HL7 message and A61 is a message event for an HL7 message for changing consulting doctor. Both A01 and A61 are used with ADT messages.
Message Structure

Each message structure is also identified by a unique ID. A message structure is basically a data structure that is used to express an association of a message type with a message event for a particular class of HL7 messages. It structurally consists of a well-defined list of HL7 segments, which can be optional or can repeat for any number of times inside an HL7 message. HL7 messages are triggered and created by an event and sent when an action occurs. HL7 messages are formed and transmitted based on the HL7 standard.

HL7 Segments

HL7 segments are made up of fields and sub-fields that define information related to the HL7 message. Each segment begins with a three-character literal value that identifies it within a message. Segments may be defined as required or optional and may be permitted to repeat. Individual data fields are found in the message by their position within their associated segments. The individual segments are listed as part of the HL7 standard.

2.2.3 openEHR

openEHR [4] is a collection of open specifications for Electronic Health Record (EHR) architecture. The main purpose of openEHR is to allow semantic interoperability among different EHR standards.

Archetypes [14] form a very basic and fundamental part of openEHR. It is used to capture clinical concepts, facts, and information in a structured and organized way. The different types of archetypes are used to support the capture of the data for common clinical activities. These archetypes are associated with some of the principle key building block archetypes which comprise observations, evaluations, instructions and actions. The data built on the basis of these archetypes is stored in larger ‘composition’
structures, which have their own archetypes. Compositions can be considered as a document that results from performing a clinical event e.g., an admission or a discharge summary. Archetypes complexity can range from simple ones, such as temperature, blood pressure or diagnosis, or complex ones, such as capturing the risk to a fetus if a father had a grandmother with Huntingdon’s chorea [4]. Archetypes consist of maximum possible data about each clinical concept and also the context needed to correctly interpret the clinical data. The generation of the archetypes and the templates can be considered as a primary task of clinicians. openEHR archetypes enable clinicians to create the extent and complexity of a health record as per their needs and requirements.

Another important part of openEHR is a template. It is used to capture the data corresponding to a specified clinical observation, such as an ICU discharge summary or antenatal visit record. The information present within a template is essentially self-explainable and does not require significant training for interested clinicians to be able to create templates to record their own observations and data.

Both archetypes and templates can be linked to terminologies or contextually appropriate terminology subsets that will support appropriate term selection by healthcare providers at the point of data entry [4].

2.3 Software Product Line Engineering

Due to the increasing and competitive market, the time and cost for developing and building a product must be less and as minimum as possible. This has led to an ever increasing need even in the field of software engineering to develop various products simultaneously, instead of just developing one product at a time. This problem can be solved by applying product line engineering concept [5] in software domain.
Product line engineering concept takes into consideration common and variable parts to be seen in the resulting products. Once this classification is done, a set of products can be generated such that the common features are put into all the products and then each product is completed by including its own corresponding set of variable features. Thus, various product variants can be extended from the commonly shared core asset, which also creates the opportunity to systematically reuse and differentiate on products in the family. Reusing products or their parts for the creation of new products or to modify existing products can save both time and money. In addition to the above benefits, product line engineering has one more advantage that it has been modeled on a higher abstraction level. This makes possible to apply product line concept to most of the domains and their products, for example, automobiles, cell phones, software products etc.

The SPLE process includes three main phases [5] shown as follows:

(1) **Product Management**

This phase is used to make some important observations and decisions from the economic point of view for building a family of products. It is used to layout a strategic plan and define a scope to specify what should and should not be included in the given product family.

(2) **Domain Engineering**

This phase is used to point out the common and variable parts among the products in the family. The aim is to find out the reusable components and also to separate the products based on their corresponding variability. Thus, the output of this phase is a set of common and variable components for all the products in the family. FODA that will be described in Section 2.3 falls in this phase.
(3) Product Engineering

In this phase, a product is actually engineered by using the commonalities and variability derived from the second phase (domain engineering). The product implements the common reusable components and then adds its own variable requirements or features, thus giving a completely new product. Once this product is fully developed and tested, it can be delivered.

Figure 2 shows the product line development process. Depending on the requirements of the products, the production assets are built which are used to build the final product line and its members. These assets can be rebuilt and reused depending on the changes needed in the product line requirements. Thus it becomes a cyclic process to rebuild and redevelop the products using the production assets which can be changed or reused if the product requirements change.

![Figure 2: Product Line Development [15].](image)

2.4 Feature Oriented Domain Analysis (FODA)

Domain analysis is the method to describe the common and variable properties of a system belonging to a particular domain and the dependencies between the variable properties. Domain analysis basically involves two major activities, namely domain
scoping and domain modeling [6]. The first activity is used to identify the domain of our interest and to define the scope of the domain for a particular application. The domain modeling is used to collect and analyze the data and yield the domain model.

FODA [7] is a method used to identify the features in a domain. Features are the prominent characteristics of a system and allow users to identify the functionalities available in the product(s). Features and feature models, described in more details in the subsequent subsection, generated by applying FODA are used in the domain analysis in order to capture the common and the variable properties of systems in the domain. The output of a feature modeling are reusable assets, such as components, patterns, domain specific languages which can be given as an input in the application engineering phase [7] and ultimately getting the desired software products [16]. The feature model in the FODA [7] comprises four basic components, namely:

a) Feature Model represents a features hierarchy by using something but not exactly a tree structure. It also specifies if a feature is mandatory, alternative, optional or more-of, which are explained in more details in the subsequent subsection.

b) Feature definitions describe all the features and also show if a feature is bound at the compile time, activation time or run time.

c) Composition rules specify valid combinations of the features.

d) Rationale for the features, giving the reasons to select or deselect a particular feature.

### 2.4.1 Feature Model Description

Features define the specifications and functionalities present in a given product. A user can create a custom feature model, consisting only of the features that he/she wants. Features can be classified under one of the following categories:
• **Mandatory:** These features are supposed to be present in all the products, at each time.

• **Optional:** These features may or may not be present in the given product.

• **Alternative (or called One-Of):** These features are such only one of them has to be present in the product. All of them cannot be included in the product at once.

• **Or (or called More-Of):** These features are such that one or more of their combinations can be present in the product.

Consider the following example of a feature model:

![Feature Model Diagram](image)

**Figure 3: FODA Example [17].**

Figure 3 shows an example of FODA. As can be seen in the above feature model diagram, which is used to represent an E-Shop, we can see the following features classification

• Mandatory: Catalogue, Payment, Security

• Optional: Search

• Alternative: High, Standard
• Or: Bank Transfer, Credit Card.

This implies that an E-Shop has to have a Catalogue, a Payment method and Security features. It may or may not have the Search feature. Further, Security can be either High or Standard, but not both. The payment method can be Bank Transfer, Credit Card or both.

Once the feature model is described, it can be used to generate possible combination of outcomes or products.

2.5 Feature Description Language (FDL) Rule Engine

This project has made use of the existing rule engine in Java [11], which is based on the FODA principles. As discussed above, features can be classified as mandatory, optional, alternative, or ‘or’. Once the features are properly classified, a feature model diagram can be constructed. After the feature model is ready, it can be also presented in the textual form. This is suggested in [11], which calls this textual representation as Feature Description Language (FDL). The FDL is capable of expressing everything that can be represented in the feature model and some more information. After a FDL program is written, feature diagram algebra [11] is used to develop and derive the rules and operate on the given input. It consists of four sets of rules, namely,

• Normalization rules: These rules are used to remove duplicate feature entries and reduce the cases of the various constructors.

• Variability rules: These help to count the total number of possible outputs for a given feature diagram.

• Expansion rules: These are used to expand a normalized feature expression into a disjunctive normal form.
• Satisfaction rules: Given a feature expression in disjunctive normal form and given constraints, we can determine which disjuncts satisfy the constraints, specified by the user.

Consider the following feature diagram for any simple car. From this feature model, we can clearly understand how the features have been classified.

The FDL to represent the above feature diagram is as follows:

Car: all( carBody, Transmission, Engine, HorsePower, pullsTrailer? )
Transmission: one-of( automatic, manual )
Engine: more-of (electric, gasoline)
HorsePower: one-of (low Power, mediumPower, highPower)

The code snippet shows that the total numbers of car instances possible are:

1(carBody) * 2(Transmission) * 3(Engine) * 3(HorsePower) * 2(PullsTrailer) = 36.
Based on these principles and information, the Rule Engine [11] is developed such that it takes an FDL program (.lang file) as an input and gives all the possible outcomes (in this case, 36) as output.

The FDL Rule Engine has been slightly modified for this project so that it gives the output of combinations of tags in the text files instead of system console.

2.6 Model Driven Engineering

Model Driven Engineering (MDE) [8] is considered as an evolving branch of software engineering, which focuses on the development of a system model before an actual system is built.

The idea of MDE caused a paradigm shift from the object oriented technology to model engineering [8]. In object oriented technology, everything is considered as objects while in MDE, everything is a model. As object oriented technology is based on the concepts of classes, instances and relations (e.g., instantiation and inheritance), MDE consists of metamodels, models, and the relations between a model and the actual system, as its basic concept. A metamodel is more like a class diagram that specifies the classes, their attributes, the links and relations between the classes and the constraints applied on them. A model is like an object that represents a particular, real world case scenario, by conforming to the metamodel. MDE had two main relations: The first is representation, which uses a model to represent an actual or real world domain, and the second is conformance, which states that a model should conform to a metamodel.

A metamodel can be considered as a model of a model. It is used to define the rules, semantics, specifications and the relations to be followed by a model which conforms to it. If a model does not follow the specified rules and regulations, it cannot be completed.
as the action shows an error. For example, if a metamodel does not have the facility to connect the given two items, the model cannot connect them in the real world scenario. Thus, the metamodel is at the higher level of abstraction and lays the foundation/grammar for the model to be built.

The metamodel itself is a model of some higher metamodel, called Meta Object Facility (MOF) [19]. It can be considered as a grammar to define the metamodels.

The diagram below illustrates the concept of MDE.

![Coordinated metamodels (conformance)](image)

Figure 5: Model Driven Engineering[18].

The diagram shows the basic concepts involved in MDE. As shown in the diagram, the level M0 shows the real world instance or scenario, which is represented by a model at level M1. The model at level M1 should conform to a metamodel at level M2, which lays
the rules and constraints and the relationships between various entities in the model at M1. The metamodel at level M2 also conforms to a higher level metamodel at level M3, which in turn conforms to itself. The level M3 is actually a MOF [19], which is used to represent the very base level of modeling.

2.7 Generic Modeling Environment (GME)

The Generic Model Environment (GME) [9] is a windows-based modeling toolkit that is used to create and evolve domain-specific and multi-aspects models of a system. It provides a user interface, which allows users to design a metamodel and the models that conform to it. It has various features or entities that can be used as per user requirements, such as atoms, models, connections, and attribute, to name a few.

An atom is an indivisible, standalone entity that cannot contain any other parts. A model is used to represent a composite entity, which can act as a container for other parts. Note that models within the GME are a representation, which do not have the same meaning and usage of the aforementioned models in MDE. A connection is used to connect two entities or objects. Attributes are used to specify the properties of an object. Along with this, the GME also gives a way to specify the relations such as inheritance and aggregation.

Other than the modeling facility, the GME also supplies a way to interpret models using code generation tool. The interpreter used in this project is written in Java, and implements the BONComponent of GME [9]. This gives users the access to some very useful built-in functions in the GME components, which help in gathering the information about the given model, as and when needed. For complete reference of GME, please refer to the GME User Manual [9].
CHAPTER 3

PROJECT APPROACH

This project makes use of technologies of SPLE with FODA and MDE with GME in order to generate a family of interoperable XML messages conformed to HL7 and openEHR standards. The basic idea of this project is to allow users to create an input model and to specify the transformation rules with the classification of tags as mandatory, optional, alternative or ‘or’ as per the FODA concept. Then an interpreter is used to interpret this model and generate the EMR message in XML conforming to HL7 and openEHR standards.

3.1 Project Flow

The topmost step for this project was to create a metamodel that defines the static semantic and syntactical rules for the EMR messages in XML format. Then to generate the actual messages using this metamodel, the following five major steps are executed.

Step 1: Create a model

A model is created to specify the classification of tags and the transformation rules. The tags are classified according to the FODA [7] method as the mandatory, optional, alternative or ‘or’. This classification of tags determines the number of message(s) to be generated, with each message having different tag combinations. The transformation
rules are needed to generate the message(s) conforming to both HL7 and openEHR standards.

Step 2: Create data structures (as a sub-step of the interpreter program)

After the model is created, the data structures (vectors) are used to store the information about each tag. In this step, the interpreter program interprets each tag in the model and stores its information, such as its level in the message and the link to its parent tag.

Step 3: Create lang files (as a sub-step of the interpreter program)

The vectors generated in the step 2 are used to create the lang files used as an input to the FDL Rule Engine [11]. The lang files are the textual representation of the tags classification at each level i.e. the mandatory, optional, alternative and ‘or’ tags.

Step 4: Use FDL Rule Engine (as a sub-step of the interpreter program)

The .lang files generated from the step 3 are given as input to the FDL Rule Engine. This engine takes a .lang file and gives all possible combinations of features or tags in this case. The output of this engine is a list of tags combinations, with each combination having a comma separated list of tags.

Step 5: Generate XML Messages (as a sub-step of the interpreter program)

By combining the data structures generated in the step 2 and the output of the FDL Rule Engine in the step 4, the XML messages are generated.

In the steps described above for the project flow, the step 1 which is to create a model is carried out in the GME editor. The remaining step 2 to step 5 are executed within the interpreter, which is a Java program to interpret the current model and to get the corresponding XML messages.
The project flow is summarized in Figure 6.

<table>
<thead>
<tr>
<th>Create a model (using GME)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Classify the tags</td>
</tr>
<tr>
<td>- Specify transformation rules</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Create data structures (in the interpreter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Store each tag object, their desired level in the XML message and their association with the parent using vectors</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Create lang files (in the interpreter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- For both standards, HL7 and openEHR create a .lang file i.e. the textual representation of tags classification at each level</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use FDL Rule Engine (in the interpreter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Give the 2 lang files as an input to the FDL Rule Engine</td>
</tr>
<tr>
<td>- The output gives the comma separated list of each possible features combination</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generate XML messages (in the interpreter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Using the comma separated list output from the FDL Rule Engine (step 4) and the data structures created earlier (step 2), the XML messages are generated.</td>
</tr>
</tbody>
</table>

Figure 6: Project flow.

3.2 Metamodel

As a first step in the project, a metamodel was designed using the GME tool. Basically, this metamodel defines the syntactical and static semantic rules for EMR messages in XML format. In general, all XML messages have tags, which can further have attributes and their values, sub tags, and/or the tag text. A simple example of an XML message is given as follows:

```xml
<xml version="1.0">
  <patient>
    <name> ABC </name>
    <birthdate> 10-10-1980 </birthdate>
  </patient>
</xml>
```
Here `<xml>` tag has attribute “version” and its value as “1.0”. Both `<xml>` and `<patient>` are composite tags, which mean they have sub tags. Tags `<name>` and `<birthdate>` are atomic tags having no sub tag, with their corresponding tag-text. Note that the end tags of each tag have to be properly placed in order to create a meaningful nested XML message. Considering these characteristics of an XML message, the metamodel consists of the following parts as shown below:

3.2.1 Project_Root

Project_Root acts like a root container at the very top level of the metamodel and contains all the other elements included in the metamodel. It is basically a model, having further models, atoms, attributes and connections. Since it is a root model, it has to be specified as “root” in its attributes panel shown at the bottom right of Figure 7. This can be done by setting its property “Is root folder?” as “true”.

![Figure 7 The metamodel of the project](image-url)
3.2.2 Element

Element is like an abstract class, which can have both ‘child’ (atomic) and ‘parent’ (composite) tags/subclasses. In GME, this has to be a model to contain other models. Element acts as a base class for the “Child” and “Parent” models.

3.2.3 Child

Child is constructed by a model used to represent instances of atomic tags in the message. Even if Child represents an atomic tag, it has to be a model so that it could contain an atom Attributes, which represents the attributes present in the tag.

3.2.4 Parent

Parent is constructed by a model, which is used to represent instances of composite tags of the message. It can contain other ‘child’ (atomic tags) or ‘parent’ (composite tags) models. Therefore, we can see a containment relationship between “Element” and ‘Parent’.

3.2.5 Attributes

Attributes is an atom used to represent the different attributes that can be presented in an XML tag. An XML tag can have zero or more attributes, which is represented in the metamodel, by using the cardinality of 0..*.

3.2.6 operation

‘operation’ in the metamodel is a connection used to specify the transformation rules, which are necessary to generate the interoperable messages in HL7 and openEHR from a single model. Connection is used to connect one or more tags (child or parent) to specify
the transformation among them by setting its property to ‘fold’, ‘inverse’, ‘add’, or ‘remove,’ as described in the subsequent subsections.

3.2.6.1 Fold

fold implies that the source of the connection becomes the parent of the destination.

Suppose, if the original message conformed to original or first standard (e.g., HL7) is like:

```xml
<P>
  <A> </A>
  <B> </B>
</P>
```

And we need to get the message conformed to the second standard (e.g., openEHR) as:

```xml
<P>
  <A>
    <B> </B>
  </A>
</P>
```

Then in this case, the ‘fold’ property of the ‘operation’ connection is used. This is used by connection A to B in the model and the setting the connection to ‘fold’. This tells the interpreter to generate the messages in two different standards as shown above. This can be applied to a chain of tags such as A – B – C – D. In this case, the message looks like:

```xml
<A>
  <B>
    <C>
      <D> </D>
    </C>
  </B>
</A>
```

3.2.6.2 Inverse

This is the opposite of fold. In ‘inverse’ connection, the destination of the connection becomes the parent of the source of the connection.
3.2.6.3 Add

If a tag is connected with an operation and its property is specified as add, it implies that this is the tag that is missing in the first/original message needs to be added in the message(s) conformed to the second standard. Suppose if the original message or first standard message (e.g., HL7) is as follows:

```xml
<P>
  <A>
    <B> </B>
  </A>
</P>
```

And the second standard (e.g., openEHR) message that needs some extra tags look like:

```xml
<P>
  <P1> </P1>
  <P2> </P2>
  <A>
    <B> </B>
    <C> </C>
  </A>
</P>
```

This can be specified in the model by adding the extra tags (P1, P2, C) at the desired levels and connecting them by the connection and setting its property to ‘add’.

3.2.6.4 Remove

The tag connected by using the operation with the property ‘remove’ implies that this tag needs to be removed from the message(s) conformed to the second standard. The remove property is thus complementary to the add connection described above.

A major advantage of these properties is that these properties can be combined. For example, we can both add an element at a level and make it as a parent for the other elements at that level.
For example, supposed the message in the first standard (e.g., HL7) is:

```xml
<P>
    <A> </A>
    <B> </B>
</P>
```

And the second standard (e.g., openEHR) message is:

```xml
<P>
    <P1>
        <A> </A>
        <B> </B>
    </P1>
</P>
```

This shows that the tag “P1” needs to have both the connections, ‘add’ and ‘fold’. This is possible by using one connection set to ‘add’, and the other two connections set to ‘fold’, with `<P1>` as source and `<A>` and `<B>` as destinations, respectively. The snapshot of such a combination is shown in Figure 8:

![Figure 8: Using “add” property of connection.](image)

In Figure 8 above, the connection which is appearing as highlighted bold black font is set to ‘add’ property, and is used to connect the child tag P1. This means the tag P1 should
not be present in the HL7 message but it should be added to the message conforming to openEHR.

In Figure 9 shown below, the highlighted bold black connection is set to have the property ‘fold’. It is connected from tag P1 (child) to the tag B (child). This implies that the tag P1 should become a parent of tag B in the message conforming to openEHR standard. Similarly, there is a connection from P1 to A also. Thus, the transformation rules enable to create the interoperable messages in HL7 and openEHR from one model. Also, as discussed above, the connection properties can be combined, similar to tag P1 which has both the ‘add’ and ‘fold’ connection properties.

![Figure 9: Using “fold” property of connection](image)

In addition to the aforementioned properties of the connection ‘operation’, the metamodel also specifies the properties for the models “Child” and “Parent” and the atom “Attributes.” More details will be described in the following subsection.
3.2.7 Properties of Child and Parent models:

This subsection describes the properties of the models Child and Parent created in the metamodel. Child and Parent have three properties which are as follows:

3.2.7.1 Tag_Name:

This field is used to specify the name of the tag in the message.

3.2.7.2 Tag_Text:

If a tag has text, this field can be used to include the textual contents for the particular tag.

3.2.7.3 Type:

Type is an important property for a tag and is the one that is used to specify the tag as mandatory, optional, alternative or ‘or’ as per the FODA [7]. By default, each tag is mandatory.

![Figure 10: Classifying the tags according to FODA](image)

Selecting Type property to classify the features using FODA
As shown in the figure 10 above, the right hand side corner has a drop down list to select from mandatory, optional, one-of (alternative) and more-of(or) , which are based on FODA concept.

3.2.8 Properties of atom Attributes:

The atom “Attributes” in the metamodel, which are used to represent the attributes in an XML tag, has the properties described in the following subsections.

3.2.8.1 A_Name

This is used to specify the name of the attribute

3.2.8.2 A_Value

The value of the given attribute

3.2.8.3 Add_Remove

This property is used to add or remove an attribute from a message. It is similar to the add and remove properties of the connection discussed earlier. The default value is set to Default. Figure 11 shows how the Add/Remove property can be set by choosing from the drop down menu.
3.2.8.4 Make_Tag

This is another important property which helps in achieving interoperability between messages in the two standards. If the property “Make_Tag” of an attribute is specified as 1 in the given model, it changes from an attribute to a tag in the second standard message(s).

Supposed the message in first standard (e.g., HL7) is:

```xml
<P A1="V1" A2="V2">
    <A> </A>
</P>
```

And in the second standard (e.g., openEHR) message, it is:

```xml
<P A1="V1">
    <A2> V2 <A2>
    <A> </A>
</P>
```
This can be done by setting the “Make_Tag” property of the attribute “A1” to 1. By default, this property is set to 0 for all attributes.

In Figure 12, the right hand bottom corner shows that the attribute property ‘Make_Tag’ can be set to 0 or 1, to change the attribute to a tag as discussed above. The default value for the ‘Make_Tag’ property is 0. Thus, this property also helps in specifying the transformation rule like the connection ‘operation’ as discussed earlier.

In summary, the metamodel allows users to specify the tags classification as per FODA and the transformation rules, by creating a model.

3.3 Model

A model can be created, conforming to the aforementioned metamodel and specifying the different combination of features as desired by users. As already discussed, a model can have any number of levels and hierarchy, and the tags at each level can be classified
by using their properties. Once the model is designed, the interpreter (code generation program) can be run to get the resulting family of messages.

3.4 Mapping

The two standards being used in this project are HL7 and openEHR. But the mapping between these two standards is still in the developing phase and not completed yet [20]. This project is a step further, after the mapping will be completely defined. For illustration purpose, a mapping has been assumed in this project by studying some instances of the messages in both standards, namely, HL7 and openEHR.

3.5 Case Study

As discussed earlier, users need to create a model specifying the tags classification and transformation rules in order to get the desired messages conforming to HL7 and openEHR standards. After the model is created, an interpreter program written in Java is used to interpret the model and generate the data structures storing the information about the model, then create the lang files and use them as an input to the FDL Rule Engine [11] to get all possible combinations of tags. The output from this rule engine and the data structures created earlier are then used to generate the XML messages.

Let us consider a part of messages from the HL7 and openEHR Observations messages example [21]. The expected messages to be generated are as follows:
**HL7 message:**

```xml
<observationEvent>
    <code display="sitting systolic blood pressure"
        codeSystemName="SNOMED-CT"
        codeSystem="2.16.840.1.113883.19.6.96"
        code="407554009"></code>
    <value units="mm[Hg]" value="120"></value>
</observationEvent>
```

**openEHR message:**

```xml
<ELEMENT archetype_node_id="at0004">
    <name>
        <mappings>
            <target>
                <terminology_id>SNOMED-CT</terminology_id>
                <code_string>407554009</code_string>
            </target>
            <match>at0004</match>
        </mappings>
        <valuenew>systolic</valuenew>
    </name>
    <value>
        <units>mm[Hg]</units>
        <magnitude>120</magnitude>
    </value>
</ELEMENT>
```

The five major steps for generating the above messages are described in the following subsections.

### 3.5.1 Create a Model

As mentioned above, the first step to get the HL7 and openEHR messages is to design a model. The model is created in GME editor window using the components of the metamodel (see Section 3.2) designed for this project. Users can drag and drop the required entities in the main window. To add children to an element, it has to be double
clicked and again the desired children element are dragged and dropped in the current window.

To get the above messages, the model will have the following specifications:

Figure 13: Model specification summary

Figure 13 shows a way to create the model for the case study example. The message on the left conforms to HL7 and the message on the right conforms to openEHR. In the model, the root tag is `<observationEvent>` which is a Parent model (see Section 3.2.4). The observationEvent model will further have the following entities:

- **Parent model code**: `<code>` contains four attributes which are code, codeSystem, codeSystemName, displayName. The attributes, codeSystem and displayName will have the Add/Remove property set to Remove_Attr as we do not want them in the openEHR message. The tag mapping is from code to target and mapping for the attributes code and codeSystemName is code_string and terminology_id respectively.
• **Parent model value**: `<value>` tag is a model that contains two attributes, units and value which are mapped as units and magnitude, respectively in openEHR message. Since these attributes become tags in the openEHR message, they have the property Make_Tag set to 1.

• **Other objects**: Along with models code and value, the root model `<observationEvent>` also has four child models (see Section 3.2.6), namely, mappings, match, name, and value. Since these tags are needed only in the openEHR message, they are connected with the property add (See 3.2.6.3), which implies that they should be included only in the openEHR message. Also, the child model mappings is connected to the models code and match using the fold (see Section 3.2.6.1). This implies that the tag mappings will become the parent tag for tags code (target) and match in the openEHR message. Similarly, the code name becomes the parent for name and value child models.

Figure 14 shows a snapshot of the model created as described above.

![Figure 14: Model](image)
After the model is created, a Java interpreter program is used to execute the further steps, described in Sections 3.5.2, 3.5.3, and 3.5.4.

### 3.5.2 Create Data Structures

The interpreter uses vectors to store the information about the model. HL7 and openEHR messages use two vectors, each of which keeps the information for every level of the model such as the objects at that level and the link to the index of the parent level.

For HL7 message, vectors named ‘models’ and ‘levels’ are used. For the above model, vectors models and levels are shown in Table 1.

<table>
<thead>
<tr>
<th>index</th>
<th>models (vector) (type: objects)</th>
<th>levels (vector) (type: integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>observationEvent</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>Value</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Code</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Vectors for HL7 message

Table 1 shows that the vector models and levels have 3 entries. The object at index 0 is always the root object representing the root tag, which in this case is observationEvent.

Then the interpreter goes through the model children of the root object, which are two models (value and code) for the HL7 message. We add them next in the models vector, with their corresponding levels as 0. The level indicates that the value and code are supposed to appear as the children of the object at level 0 i.e., observationEvent tag in the HL7 message. The above procedure is repeated for all the entries in the models vector.

But since value and code do not have any model children, no new information is entered.
in the vectors. Thus, this represents the complete information about the given model for HL7 message. The vector models actually store the GME objects, e.g., models, which represent the corresponding tags. These objects can be retrieved from the vector and used to get the properties of that object, for example, Tag Name and Tag Text for the model objects, Parent and Child (see Section 3.2). For simplicity and understanding purpose, Table 1 above shows the objects tag names instead.

For openEHR message, the vectors named ‘models1’ and ‘levels1’ are used. These vectors have to be created by considering all the transformation rules. For the model created above, the vectors models1 and levels1 are as shown in Table 2:

<table>
<thead>
<tr>
<th>Index</th>
<th>models1 (vector)</th>
<th>levels1 (vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(type: objects)</td>
<td>(type: integer)</td>
</tr>
<tr>
<td>0</td>
<td>observationEvent</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>Mappings</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Code</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Name</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Match</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Value</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Value</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2: Vectors for openEHR message

As discussed during the model creation, observationEvent is the root model or tag for the message. Hence it appears as the first entry in the models1 vector i.e., at index 0. Also as it does not have any parent tag, its corresponding entry in the levels1 vector is -1, which
is used to represent the index of the parent tag for the current entry. -1 implies that it is the topmost tag in the message with no parent tag as the vector does not have index -1.

Then we get all the children of the root object, in this case, the Parent models value and code and the Child models name, match, mappings, value. So, it will imply that the corresponding levels for each of these children should be 0. But since the transformation rules are applied, the levels change. Note that HL7 vectors in table1 did not have mappings, value, code and match child models. This is because these models have been connected with an add connection, which implies that they should only be included in the openEHR message. Also, there is a fold connection from mappings to parent model code and child model match. This suggests that the level for code and match should be equal to the index of the mappings object, which is 1. Similarly, there is a fold connection from name child model to the mappings and value child models. So the level for value and mappings is the index of name child model, which is 3. In order to get the correct information in the vectors, the interpreter checks for every child and parent model if it is connected by a connection and which type (fold, inverse, add or remove). If the model object is connected, the interpreter determines the type of connection and checks if the object already exists in the models1 vector. Depending on the result of this check, the object is either added to the models1 vector if it not already in it or its level is updated if it is the destination of fold connection or source of inverse connection. The level needs to be updated with the index number of the source in case of fold and the index number of the destination in case of inverse. If no transformation rule is applied to the object, it is simply added in the vector models1 with the level as the index of the current object being processed.
Thus the vectors are generated which gives the brief view of the hierarchy and arrangement of tags objects in the model.

3.5.3 Creating .lang files

The lang files are the textual representation of tags at each level and their classification as mandatory, optional, alternative or ‘or’ as per FODA method. In the example above, all the tags in both the messages are mandatory. An example combining both the tags classification and transformation rules will be shown in Section 3.6.

The lang files are created by using the vectors generated in the previous step. HL7 and openEHR messages both have one lang file created in order to represent the tags classification at each level. The interpreter starts from the first or root element in the vectors models (HL7) or models1 (openEHR) and get all the entries in the same vector whose level indicates that they are the subtags or children of the element being processed currently, in this case the root element. Then depending on the tags classification, it generates the textual representation of tags for that element. Similarly, the steps are repeated for each element and the final lang file is generated. The lang files for the current example are shown in Figure 15.
Figure 15: Lang Files

As seen in Figure 15, the lang file for HL7 shows that the observationEvent has two mandatory children, Value and Code. Since Value and Code have no further children, the file has only one line of text. For openEHR, the text file has three lines of text, representing the observationEvent, Name and Mappings respectively. The entries in this text file have to be in proper order i.e., Name has to come before Mappings as Name is the parent of Mappings. Please note that the Value in the first line of observationEvent is the parent model while the value in the second line of Name is the child model. These are two different objects with same tag name.

Also, the ‘all’ keyword before the tags represent that all the tags are mandatory. For alternative, optional and ‘or’ classified tags, the keywords are one-of, more-of and opt respectively. Thus the lang files are created and given as an input to the FDL Rule Engine [11].
3.5.4 Use FDL Rule Engine:

The FDL Rule Engine [11] is used to generate all the possible combinations of tags from the given input lang file. The output is the comma separated list of tags for each unique possible combination. This project uses a batch file to give the lang files as input and get the output files.

For the lang files in Figure 15, the FDL Rule engine generates the output shown in Figure 16.

![Figure 16; Output of FDL Rule Engine](image)

3.5.5 Generating XML messages:

As shown in Figure 16, the output files of FDL Rule Engine give all the possible combinations of tags as per their FODA classification. But there is no way to relate the tags to their parent tags from these files, as they do not show the parent tags at all. This also does not give us any idea of the levels at which each tag is present. All this information is necessary in order to generate the meaningful and complete XML messages corresponding to these output files. In order to solve this challenge, the vectors, models and levels for HL7 message(s) and models1 and levels1 for openEHR message(s)
generated earlier by the interpreter are used. The interpreter parses through one combination of tags at a time. The interpreter travels through the vector models (HL7) or models1 (openEHR) and check if that entry is present in the current string of combination of tags being considered. If yes, it adds all the sub tags and attributes for the current tag or object. Thus the parent tags are added to the message. Also, at this step the interpreter checks if any attributes are supposed to be converted into tags by checking their Make_Tag property value and converts it into the tag form and appends to the message. It also checks if any attributes are to be added or removed from the openEHR message. These steps are repeated for every entry in the model and the message gets generated using the recursive form of this function. Along with these vectors, the mapping of tags and attributes is used, which is stored in the Hash table in the interpreter program itself. This is done for every list of tags combination that was obtained from FDL Rule engine till the end of file is reached. Thus the total number of messages generated is equal to the number of combinations in the output text files.

Thus this project has major contribution in terms of combining the FODA and transformation rules to help generate interoperable messages in HL7 and openEHR from the single model.

3.6 Complete functionality of the project

As mentioned earlier, the project allows to classify the tags using FODA as either mandatory, optional, alternative or ‘or’ along with the transformation rules. In order to illustrate all the possible combination of functionalities in the project, this example was created.
In Figure 17, observation is the root tag. Event, BloodPressure, Patient, and type are mandatory tags while Visitors is an optional tag. Subtags systolic and diastolic of BloodPressure are alternative tags while subtags Name and ID of Patient are ‘Or’ tags. There is a fold connection from Event to Patient, which indicates that Event should be the parent tag of Patient. The inverse connection between examiner and type implies that examiner should be the subtag of type. Additionally, examiner has an add connection which means examiner tag should not be present in the HL7 message but should in openEHR as a subtag of type as mentioned earlier. The ID tag is connected by a remove connection, so it should not be present in openEHR but should be the subtag of Patient in HL7. Fname and Lname attributes should be converted to subtags of name in openEHR as they have the Make_Tag property set to 1. Also, please note that Lname property
Add/Remove is set to Add_Attr which implies that it should not appear in any HL7 messages. The snapshots of the model created for the above example is shown in Figure 18.

Figure 18: Model showing root element for the example in figure 17

Figure 18 shows the first element (observation) of the model. Since it contains other elements, it is a parent model. It is dragged and dropped from the left side window pane. To add children to this element, it is double clicked. A new window is opened and the children are dragged and dropped in the new window. Users can go to any number of levels using this method in order to specify the entire message hierarchy. Additionally, as mentioned earlier, the elements can be classified as either mandatory, optional, alternative or ‘or’ and the transformation rules can also be applied. Figure 19 shows the snapshot of the model inside the observation element.
As shown in Figure 19, the root element (observation) has six children, Event, Patient, BloodPressure, Visitors, Examiner and type. There is a fold connection between Event and Patient (Patient becomes subtag of Event) and an inverse connection between Examiner and Type (Examiner becomes subtag of Type). Additionally, examiner has an add connection, which shows that it should only appear in openEHR messages(s) and not in HL7 message(s). Every tag is mandatory here except Visitors, which is an optional tag. This is specified in the bottom right window in the Type property. It is a drop down menu with the possible values as mandatory, optional, one-of or more-of.

Similarly, the children of BloodPressure, diastolic and systolic are specified as one-of and the children of Patient, name and ID are specified as more-of. Patient also has an additional attribute SSN which is connected by remove connection, making it to be available in only in HL7 and not in openEHR message(s). Name further has attributes Fname and Lname with their Make_Tag property set to 1.
3.6.1 Create data structures

The data structures for the above message model is created in the same way as in section 3.5.2. They are used to store the level of each object and the link to the index of their parents.

3.6.2 Lang Files

The lang files for the above message model is generated as shown in the figure 20 below.

![Lang Files for example](image)

Thus the lang files above shows the tags classification at each level. These files represent mandatory (all), optional (opt), alternative (one-of) and ‘or’ (more-of) features classified at each level in the message. For example, Visitors tag become optional, the systolic and diastolic become one-of and Name and Id become more-of. Additionally, the openehr file has Type as the parent of Examiner and Event as the parent of Patient. Also, it does not have SSN which is present in the HL7 file.
3.6.3. Using FDL Rule Engine

The lang files generated above are given as input to the FDL Rule Engine and the output obtained is as shown in Figure 21.

In Figure 21, the output of the FDL Rule Engine shows the possible combination of tags as per their classification. For example, Visitors is optional in some combinations. Also, diastolic and systolic never appear in one combination, because they are classified as
alternative. Similarly, Id and Name appear are classified as more-of and so at least one of them appears in the message.

### 3.6.4 Generating XML messages

The XML messages are generated for each comma separated list of tags as shown in Figure 21 above. Thus, there should be 12 messages for HL7 and 12 messages for openEHR. The output files neither give any information of the level of tags nor does it show the parent to which a tag belongs to. This information is very important for generating the meaningful XML messages. The problem is solved by using the output files in Figure 21 and the data structures created earlier (please refer to Section 3.5). Figure 22 shows four of the twelve HL7 messages generated. And Figure 23 shows four of the twelve openEHR messages generated.

![Figure 22: 4 of 12 generated HL7 messages](image)
Figure 23: 4 of 12 generated openEHR messages

Figure 24 shows the two HL7 and two openEHR messages and highlights the transformations applied to the messages

Figure 24: 2 HL7 and 2 openEHR messages
3.7 Generalization

Nowadays XML messages are used in almost every domain in order to overcome the issue of transporting the data among heterogeneous systems. As such, given the abilities of this project, it can be applied to many other domains and can be customized in order to include their own mappings.

For example, in networking domain, users can have some common elements in a message such as message header and footer, while the message can either be a sender, receiver or an acknowledged message. In such case, we can classify the features as per FODA into mandatory and alternative for this example. Also, the mapping can help us to get a more sophisticated group of messages generated from a single model.

Thus the model created by specifying the tags classification and transformation rules can generate the interoperable XML messages from HL7 and openEHR standards. The total number of messages generated depends on the way tags are classified. At least 2 messages are generated for every model when every tag classification is left to mandatory, which is default.
CHAPTER 4

RELATED WORK

4.1 Web based HL7 Message Generation and Validation

The initial versions of HL7 used the Electronic Data Interchange (EDI) format to represent a message, using the pipes (|) and delimiters, such as ^. The HL7 message has segments starting with a segment ID, which in turn has data fields separated by (|) and the fields further has component and sub components separated by ^ . The segment finally ends with a carriage return. Web based HL7 message generation and validation system [22] takes advantage of this representation of the HL7 message to validate it and convert it into XML format.

The system basically uses five important modules to validate and generate the HL7 message. The HL7 parser module reads the message, one character at a time and distinguishes it as either a datum or a separator. It also stores the segments and fields in the buffer memory for later use. The second module, called as database module, is used to store the valid definitions of HL7 specification. The third and important module, called as data validation module, is used to check the data obtained from the parser module against the data stored in the database. It carries out validations such as the presence of required segments and their sequence, the required fields, their lengths and the data types. The fourth module, XML module, is used to actually carry out the conversion of the message from the EDI format to the XML format. The last module in the system is the I/O interface module, which can be used by the user to upload the original message file.
for validation and transformation to the server using the HTTP protocol. It also has the facility for online entry and editing of the message and validating it. Thus, the message can be entered step by step, also helping in learning the HL7 message standard in a better way.

4.2 HL7 Java SIG Project

The HL7 Java SIG Project [23] introduces Java API to parse and build the HL7 V3 messages. To generate these messages the API does not need any knowledge of the existing message structures. The API makes use of the HL7 data model of Reference Information Model (RIM) [14], the HL7 data types and the meta files of message definition. The API can process the actual given HL7 V3 XML message independent of the message type and contents with the help of the corresponding meta file. This will enable the API to process the newly added messages with their meta files without actually changing the API itself.

The API actually has Java classes for representing the HL7 data types and RIM objects. The RIM object classes have the get and set functions for the message structures stores in the memory thus helping to create and retrieve the HL7 V3 Refined Message Information Model (RMIM) [13] instances. The HL7 V3 RIM has total of 29 data types, many of which has rich methods, serving as data type handlers to support API tools. Additionally, the API contains two important components which are message builder and message parser. The message builder builds the HL7 V3 XML message from the instance of RMIM in the memory. It uses Hierarchical Message Description (HMD) [13] to make sure that the XML message have all the correct elements in the right order with the correct attributes. The message parser is complementary to message builder. The parser
reads the HL7 V3 XML message and generates the in memory RMIM instance. It uses
the HMD to create the right data structures from the XML message. The data in the XML
message is evaluated using the specific handlers of the data types.
Thus HL7 Java SIG project provides functionalities to create and parse the HL7 V3 XML
messages from the RIM data models and data types of HL7.

4.3 Interoperability in HealthCare

Medical messages in the electronic form can be present in different standards such as
HL7 or openEHR. Different departments within a hospital or different hospitals can
generate messages that conform to different varied standards. The major problem in the
communication of such messages is interoperability.

The thesis by Dr. Amanda Joanne Ducrou of University of Wollongong [13] talks about
the use of design science research [24] to solve the issue of interoperability in healthcare.
This thesis emphasizes on the use of three main standards and terminologies, namely,
HL7, openEHR and SNOMED CT (Clinical Terminology) [25]. Ontology mapping
among these different standards and terminologies is used as a base to solve the issue of
semantic interoperability and enable the communication between the various systems
using the different standards. The server terminologies from the SNOMED CT is first
converted to XML format, stored in eXist Open Source Native XML Database [26]. This
is then queried using the XQuery [27] in order to get the required results depending on
the SNOMED concept ID. This helps in easier mapping among SNOMED CT and HL7
and openEHR, as all of them are ultimately represented in XML format. Once the
semantic interoperability is achieved, the technical interoperability is needed in order to
communicate and send these messages in their available format. The thesis mentions the
use of a framework based on JINI technology [28] to achieve the technical interoperability between the messages. JINI offers a number of services that could be used to for the communication in a network. The users can use the easy plug-ins in order to connect to these services. But, this framework could not provide the flexibility to include other message formats and also had the issue with the scalability. As a result, a final interoperability solution was built based on the MULE [29] enterprise service bus (ESB). This solution was called as Health Service Bus (HSB) [13]. The HSB provided the technical interoperability for the semantic interoperable messages by providing the reliable messaging system. Along with these features, the HSB also provided the way for process interoperability, which is the social engineering aspect of interoperability. The system with all the interoperability, namely, technical, semantic and process interoperability is supposed to be the complete system and thus, HSB provided a complete solution by offering all kinds of interoperability. HSB uses canonical message model, which is based on the ontology mapping between HL7 and openEHR to attain the semantic interoperability. The technical interoperability between the messages was achieved by using the MULE message oriented middleware core which provides reliable and secure transportation for different message formats. Finally, the HSB used the intelligent routing characteristics of ESB to achieve the process interoperability.

4.4 Comparison between Related Work and The Project

As discussed above, the related work described in Section 4.1 (i.e., web based HL7 message generation and validation) tries to create and validate the HL7 XML messages. It parses the HL7 EDI message and gets the message components to check them against the database having HL7 components definitions for message validation and also
generates the XML form of the message from the components. The related work in Section 4.2 (i.e., HL7 Java SIG Project) makes use of the HL7 V3 RIM data model, HL7 data types and the meta files to create the XML message from the RMIM instance present in the memory. It can also generate the RMIM instance from the XML message, thus giving two ways of transformation of XML messages and RMIM instances. The third related work in Section 4.3, which is the thesis titles interoperability in healthcare, defines a way to tackle the semantic, technical and process interoperability of HL7 and openEHR messages. It uses the ontology mapping and XQuery and XSLT to develop a system to achieve the above mentioned interoperability. This project introduces the combination of SPLE with FODA and MDE with GME in order to generate the interoperable HL7 and openEHR messages. The FODA is used to classify the tags in the messages as mandatory, optional, alternative or ‘or’. This helps to get all the possible combination of tags. Each combination represents an XML message to be generated. Also, the transformation rules introduced in the project using GME based on MDE helps to specify all the transformations that should be specified in order to get the messages in HL7 and openEHR standards simultaneously. The FODA classification and the transformation rules can be specified simultaneously in a single model using this project, which helps in generating all the possible XML messages with the unique tags combinations in HL7 and openEHR standards.
CHAPTER 5

CONCLUSION

This project helps to generate a family of different EMR messages (XML) with the help of features classification based on SPLE and FODA principles and also by using transformation rules. As such, the messages can be generated by specifying the features (tags) as either mandatory, optional, alternative or more-of. It uses concepts of software product line, analyzed by using FODA, in order to build the common parts (tags) in all the messages and then separately write the variant tags to their corresponding output messages. Also, the project is designed using MDE concepts, which facilitates simple design and better communication. A significant aspect of this project is that users have the flexibility to define the interoperable relations between the tags other than simply applying the FODA classification to them. The tags can be related such that one can become a parent of another, or it can be added or removed from the other standard message(s). Also, attributes can become tags in the other standard if specified so by users. Once users give all such specifications, the model can be interpreted using the code generation program written in Java to generate the desired XML messages. Since XML is used in many other domains, this application can be used to generate the messages in that domain too, thus making it a more generalized tool.
References


