CDRGen: A Clinical Data Registry Generator

Pedro Miguel da Cunha Alves
Instituto Superior Técnico
Universidade de Lisboa, Portugal
pedro.cunha.alves@tecnico.ulisboa.pt

Abstract

In the health sector, data analysis is typically performed over clinical data that is stored in Clinical Data Registries (CDRs). Currently, CDRs tend to be supported by a software application composed of a user interface, a database and the logic that provides the communication between these two. As far as we could find, in Portugal there are twelve CDRs covering seven medical specialties, which does not satisfy the physicians' demand from other medical specialties. This demand eventually causes the creation of new CDRs, but these are usually created from scratch. This results in a waste of resources since CDRs have common characteristics: their user interfaces support common functionalities and their collected data covers common topics. Therefore, this thesis focus on the development of a Clinical Data Registry Generator (CDRGen) software system that generates CDRs with a minimum effort in terms of software design and development. CDRGen creates CDRs more efficiently, enabling to satisfy the physicians' requirements in a timely manner. CDRGen also facilitates the developers' work since, in general, they only need to specify the data to collect and CDRGen generates a CDR based on it. Concretely, CDRGen receives a high-level specification of the data to collect, parses it and generates the three components of a CDR: user interface, database and the logic that provides the communication between these two. Before developing CDRGen, we perform an analysis of the existing Portuguese CDRs regarding collected data, functionalities and user interface. When developing CDRGen, we use one existing CDR. After its development, we use two other CDRs to evaluate CDRGen. For these two CDRs, we measure the time required to develop a new CDR. In addition, we assess the quality of the generated CDR regarding its database and functionalities.

Keywords: CDRGen, Clinical data registry, Database, User interface

1. Introduction

In the health sector, the concept of *Clinical Data Registry (CDR)* is crucial [1]. A CDR records data about patients and the health care they receive over time. Typically, a CDR is focused on a given medical specialty or a set of diseases. They prove to be remarkable for conducting scientific research that addresses clinical problems [2]. This scientific research is based on data analysis performed over data collected and stored in CDRs.

Very often, physicians use Excel files to build their own CDR. However, Excel files have limitations, such as poor scaling, for example. Given these limitations, over the years there has been an effort to build more robust CDRs that are composed of three components: (i) a user interface that enables to enter data through forms and then to access data; (ii) a database to store the entered data; and (iii) the logic that provides the communication between the user interface and the database.

In Portugal, there have been some initiatives regarding the creation of CDRs. However, the CDRs that currently exist do not cover all medical specialties. In fact, we were able to find 12 CDRs that cover 7 medical specialties (all described in Section 2). The existing Portuguese CDRs collect data and types of data that are common to various medical specialties (for instance, the patient's name as a string and the birth date as a date). In addition, as far as we know, the data that is collected and stored by half of the Portuguese CDRs is structured (e.g., it is stored in a relational database). This characteristic provides advantages, since storing structured data facilitates the execution of data analysis.

1.1. Problem

Since the Portuguese CDRs do not cover all medical specialties, new CDRs tend to be created. Due to this, typically new software applications are built from scratch in order to support each of these new CDRs. This is a waste of resources since a lot of work is repeated given the fact that CDRs have similar characteristics. Namely, their user interface provides common functionalities (such as enter and navigating through the data) and their col-

lected data covers common topics (e.g., patient's characteristics and medical examinations). The repeated work increases substantially if the way in which the user interacts with their user interfaces is always the same. Derma.pt is an exceptional example of a CDR that, we believe, has been built reusing the results of Reuma.pt CDR. To overcome this built from scratch process, we could use Low-Code Development Platforms (LCDPs) to build applications that support CDRs and also reuse the work performed between them. However, we would always have to make a lot of changes if we wanted to generate another CDR that collects different specific data and that provided the same interaction with the user interface.

1.2. Objectives

The main objective of this thesis is to design and develop a *Clinical Data Registry Generator* (CDRGen) software system to generate a CDR based on a high-level specification. Using CDRGen, we want to generate CDRs with a minimum effort in terms of software design and development.

To develop CDRGen, we focus on the following three sub-objectives:

- Identification of the collected data, the functionalities and the user interface characteristics taking into account the existing Portuguese CDRs.
- Syntax of the specification to provide to CDRGen for generating a new CDR.
- Generation of the software logic required to create and manage the database, the user interface and the logic that provides the communication between these two components of the generated CDR.

1.3. Solution

The overview of the solution to generate a CDR is illustrated in Figure 1 and is described as follows:

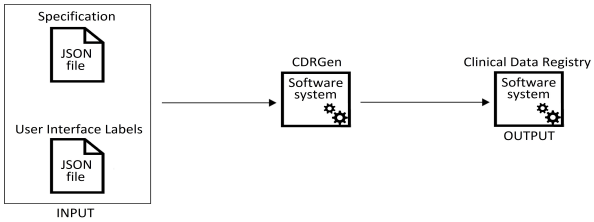

Figure 1: Overview of the solution to generate a CDR.

- The input of *CDRGen* is two JavaScript Object Notation (JSON) files called *specification* and *user interface labels*. The specification is mainly composed of the data to be collected by the CDR. The user interface labels describe the labels of the user interface that are not related with the data to be collected.
- *CDRGen* reads these two JSON files, parses them, and generates the code to create and manage the generated CDR.
- A *CDR* is the output of *CDRGen*.

1.4. Validation

To assess CDRGen, we use two existing CDRs and, for each CDR, we measure:

- The *time* required to develop a CDR with CDRGen.
- The *ratio* of tables, their relationships and attributes that exist in the generated database in relation to the expected database based on the data collected by the existing CDR.
- The *ratio* of functionalities that the generated CDR can perform in relation to the functionalities provided by the existing CDR.

1.5. Document Outline

This document is organized as follows. Section 2 details the related work regarding the existing Portuguese CDRs and LCDPs in general. In Section 3, we perform a requirements analysis based on what is currently supported by the Portuguese CDRs that we approach in Section 2. Section 4 details CDRGen and, in Section 5, we assess it. Finally, Section 6 presents the conclusions of this thesis, an approach to its limitations and a vision of future work.

2. Related Work

In this section, we describe the related work regarding two aspects: (i) the twelve Portuguese CDRs that we were able to find (Section 2.1); and (ii) Low-Code Development Platforms (Section 2.2).

2.1. Portuguese Clinical Data Registries

We organize the twelve Portuguese CDRs that we were able to find in three groups of CDRs:

1. *Specialty-specific*, composed of the following 4 CDRs that cover different medical specialties: (i) Reuma.pt[1]; (ii) Derma.pt[2]; (iii) National Cancer Registry (RON)[3]; and (iv) Information System for HIV/AIDS infection (SI.VIDA)
2. *Cardiological*, composed of the following 6 CDRs that address different diseases of Cardiology: (i) Portuguese Registry of Acute Coronary Syndromes (ProACS) [4]; (ii) National Registry of Arrhythmogenic Right Ven-

[1] http://www.reuma.pt/
[2] http://www.derma.pt/
[3] https://dre.pt/application/file/a/107688306/
[4] https://registos.spc.pt/RegistoSCA/

Table 1: Summary of the Portuguese CDRs

Property \ Group	Specialty-specific CDRs	Cardiological CDRs		Research CDRs
Clinical data registries	- Reuma.pt - Derma.pt - RON - SI.VIDA	- ProACS - RNMAVD - PRo-HCM	- PMR - PRNC - RNCI	- Umedicine - PRECISE Stroke
Medical specialties	- Rheumatology - Dermatology - Oncology - Infectiology	- Cardiology		- Urology - Neurology
Data collected	- Patient's characteristics - Diagnosis - Examinations - Disease-specific - Treatment - Questionnaires	- Patient's characteristics - Patient's history - Family history - Diagnosis - Examinations - Treatment - Patient's follow-up		- Patient's characteristics - Patient's history - Examinations - Treatment - Patient's follow-up - Symptoms - Questionnaires
Structured data	- Yes (25%) - Unknown (75%)	- Yes (50%) - Unknown (50%)		- Yes
Accessible to	- Health workers - Patients	- Physicians (75%) - Unknown (25%)		- Physicians - Patients - Clerks

tricular Myocardiopathy (RNMAVD)[5]; (iii) Portuguese Registry of Hypertrophic Cardiomyopathy (PRo-HCM)[6]; (iv) Portuguese Myocarditis Registry (PMR)[7]; (v) Portuguese Registry of Non-compaction Cardiomyopathy (PRNC)[8]; and (vi) Portuguese Registry on Interventional Cardiology (RNCI)[9]

3. *Research*, composed of the following 2 CDRs built in the context of research projects: (i) Umedicine; and (ii) PRECISE Stroke[10].

Table 1 summarizes the Portuguese CDRs, and describes, for each of the three groups mentioned, the concerned CDRs, its medical specialties, the data collected, whether the data is structured, and by whom it is accessible.

All the CDRs are primarily focused on data entry, and the data analysis is usually performed by external tools to the CDRs. In addition, Reuma.pt in conjunction with Umedicine are the only CDRs that are accessible both to physicians and patients, contrarily to the other CDRs that can only be accessed by physicians or by other type of health workers.

These CDRs collect data common to various topics, such as patient's characteristics, diagnosis, treatments, medical examinations and patient's follow-up. The data to be collected is adjusted in relation to the medical specialty of each CDR. In six of the CDRs the data is structured, being composed largely by enumerated fields and by a few free text fields.

Finally, during the research about the Portuguese CDRs, we verified that there are no CDRs for some medical specialties. This is a problem, since physicians require CDRs for their medical specialties.

2.2. Low-Code Development Platforms (LCDPs)

LCDPs are software platforms composed of tools that allow developers to create applications efficiently and with a minimum amount of written code. In order to develop applications, the LCDPs provide a high-level environment that is mainly composed of graphical user interfaces. LCDPs usually provide a lower cost compared to the standard process of creating applications. Still, suppose we wanted to develop several applications that provide the same functionality but whose data to store is different. When using LCDPs, we could always create a new application and develop everything from scratch. Although, we would be wasting resources since work would be repeated given the common aspects of the applications. On the other hand, we could reuse the first application developed and change certain aspects. However, we would not only have to change the appli-

[5] http://registos.spc.pt/RegistoMAVD/
[6] http://registos.spc.pt/RegistosMiocardiopatia/
[7] http://registos.spc.pt/RegistosMiocardite/
[8] http://registos.spc.pt/RegistoMNC/
[9] http://registos.spc.pt/RegistosNacionaisSPC/
[10] https://stroke.precisemed.org/home/

cation database (which was expected), but also its user interface in order to accommodate the different data. This way, we always had to spend time and, therefore, resources. This would be a problem that fits into the creation of new CDRs, since they usually provide common functionalities (such as adding and navigating through the data) and collect different specific data.

3. Requirements Analysis

In this section we present the requirements analysis that we perform in order to decide what we will support in any CDR to be generated. The requirements gathering is performed with respect to three topics: (i) data to store (Section 3.1); (ii) functionalities (Section 3.2); and (iii) user interface (Section 3.3). This analysis is performed based on the existing Portuguese CDRs, that we described in Section 2.1. Since we do not have full access to all these CDRs, we rely on those that we have information on at least one of the topics.

3.1. Data to Store

In terms of the data to store, we first identify the common entity groups with respect to the data collected by the 12 CDRs addressed in Section 2.1. In concrete, we identify the following six entity groups:

- `Person`, representing the patient, namely her characteristics
- `Diagnosis`, representing the symptoms and diseases
- `Treatment`, representing the various treatments that a patient can perform
- `Questionnaire`, representing the questionnaires that a patient fills in
- `Medical examination`, representing medical examinations
- `Medical appointment`, representing the patient's follow-up in medical visits.

These entity groups will be the basis for indicating the data to be collected by a CDR to generate. An *entity group* is identified by a name and is composed of one or more entities. An *entity* is also identified by a name and is composed of one or more attributes. An *attribute* is identified by a name and has a value. The values of all attributes represent the data that is collected by a CDR.

With these entity groups identified, we focus on the four CDRs that we have information about the data collected: RON, RNMAVD, PRNC and Umedicine. As we found in these four CDRs, in a CDR to generate we will have relationships between the entity that represents the patient and the other entities. These relationships may have a one-to-one or a one-to-many multiplicity. In addition, as we found in Umedicine, we will also support the existence of relationships between the entities that represent[11]: `symptom` and `disease`; `treatment` and `medical examination`; `treatment` and `disease`; `medicine` and `disease`; and `medicine` and `medication`.

Concerning the attribute types, in a CDR to generate we will support the following attributes types: `string`, `numeric` (integer and float), `date`, `enumerated`, `list` (of one of the previous types), `boolean`, and `image`. This set of attribute types results mainly of the union of the sets of attribute types that we found in the four CDRs that we focused on. As we also found, for an enumerated-type attribute it will be possible to collect another value other than the values that are predefined. In addition, for `string` and `numeric` attributes we will enable to define a limit of characters/digits.

Finally, we will support the constraints between attributes stating that:

- The value of an attribute can determine the assignment of values to other attributes (founded in all four CDRs).
- The value of an attribute must be one of the values that were already given to another attribute (founded in three of the four CDRs)[12].

3.2. Functionalities

Regarding the functionalities that are supported by CDRs, we analyse five CDRs: Umedicine, Reuma.pt, RON, PRNC and RNMAVD. The CDR to be generated will be able to be accessed by four different user types: (i) `patient`; (ii) `clerk`; (iii) `administrator physician`; and (iv) `non-administrator physician`. We chose these user types since we believe that they cover the different user types that may exist. The `administrator physician` user type always exists for any generated CDR, while the other three user types may not necessarily exist. These user types will be able to perform different functionalities, as follows:

- The `patient` user will be able to:
  - Add/visualize/edit/delete her data from any of the six entity groups.
- The `clerk` user will be able to:
  - Add `patient` users
  - Add a subset of personal data of new patients.

[11] This will not be included in the system implementation
[12] This will not be included in the system implementation

- The `administrator physician` user will be able to:
  - Add/visualize/edit/delete data from any of the six entity groups that concerns a patient
  - Search for patients by name or by other kind of identification
  - Generate reports regarding modifications (add/edit/delete) on the data of a particular patient
  - Add users of any type.
- The `non-administrator physician` user will be able to perform the same functionalities as the `administrator physician` user, however it will only be able to add `patient` users instead of being able to add users of any type.
- All user types will also be able to edit and retrieve their password.

While the `administrator physician` user will be able to add/visualize/edit/delete data of any patient, the permissions of the other user types may vary. For instance, the `patient` user may only be able to visualize a subset of her personal data.

These functionalities resulted from the generalization of the union of the functionalities supported by the CDRs that we analysed. In particular, we sought to prioritize the key functionalities that a CDR usually has: add, visualize, edit and delete data. In addition, we enable to change the permissions of the user types over the data in order to give more flexibility.

3.3. User Interface

Among all the CDRs analysed, we have access to the full Umedicine user interface and only a few screenshots of Reuma.pt's user interface. This is enough to identify the widgets that are used in their user interfaces to enter data. Based on that information, we predefine the widgets that will be used in the user interface to enter data according to each type of attribute. For example, a text box will be used to enter the value of a string-type attribute. Regarding other aspects of the user interface, we follow the only CDR that we have full access: Umedicine.

4. CDRGen

In this section, we present a Clinical Data Registry Generator (CDRGen) that is able to generate a CDR with a minimum effort in terms of software design and development. The high-level architecture of CDRGen is illustrated in Figure 2.

In general terms, the process for generating a CDR starts with two JSON files: the JSON file

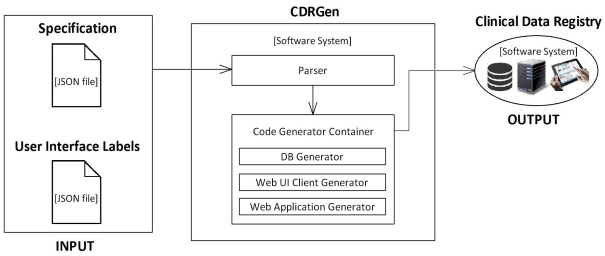

Figure 2: High-level architecture of CDRGen.

specification and the JSON file *user interface labels* (described in Sections 4.1 and 4.2, respectively). CDRGen reads these JSON files, parses them and generates code in order to produce a CDR (described in Section 4.3). The CDRGen's software modules that parse the input and that generate the CDR's code are described in Section 4.4. During the development of CDRGen, we use the Umedicine CDR as example.

4.1. CDRGen Input 1/2: JSON File Specification

We first designed a metamodel in order to describe the metadata of a CDR to be generated that is defined in the JSON file specification. This metamodel was designed based on the requirements analysis that we performed in Section 3, where we identified: the user types to be supported by a CDR to be generated; and the entity groups that form the basis of the data to be collected by a CDR. We must also bear in mind that a CDR to be generated must have a name. Given this, the basis of the metadata of a CDR to be generated consists of the CDR's name, the types of users to be supported, and the entity groups required to define the data to be collected.

Figure 3 represents the UML class diagram of the specification metamodel. In the specification, each entity group can optionally have a set of permissions and is composed of one or more entities. Each entity has a name, can optionally have a set of permissions and properties, and has one or more attributes. Each attribute has a name, a type and can optionally have a set of values, permissions and properties. In attributes:

- The *type* specifies the kind of values the attribute can hold (e.g., string, integer). String and numeric attributes can optionally have a minimum and a maximum number of characters/digits, respectively.
- The *set of values* is required for enumerated-type attributes. It shows the concrete values that can be hold by that attribute. For example, the gender can be an enumerated-type attribute whose values are male and female.

In entities and attributes, the *set of properties* specifies one or more properties. For example,

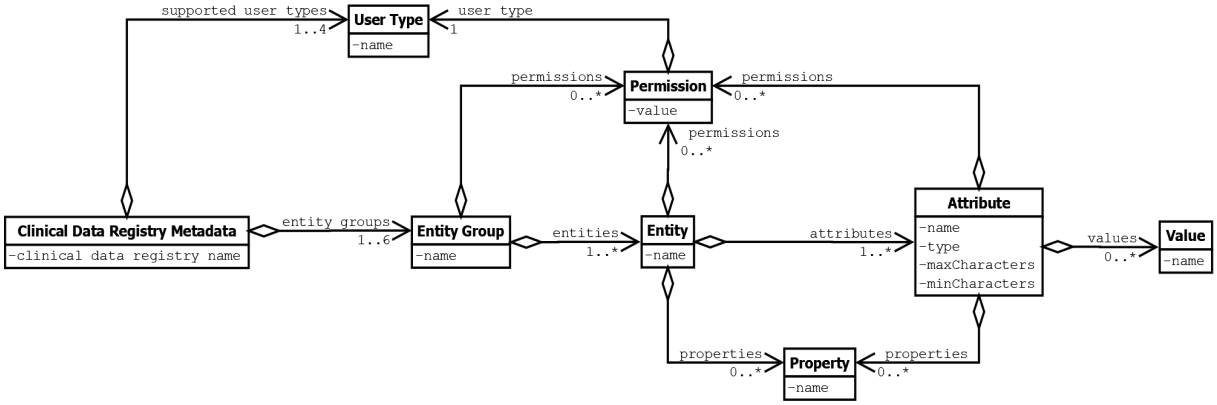

Figure 3: UML class diagram of the specification metamodel.

an entity can have the *historic* property[13], indicating that the entity can contain several instances that are identified by a date-type attribute. An attribute can have the *not null* property, which indicates that it cannot hold the null value. In entity groups, entities and attributes, the *set of permissions* specifies if a user type can read and/or write (i.e., permission's value) a particular entity group/entity/attribute.

Let us suppose that we want to generate a CDR according to the following example:

(1) **Example.** *Simple Registry* is a CDR that can be accessed by two types of users: `administrator physician` and `patient`. In *Simple Registry*, we want to collect:

- The patient's name which, in this case, identifies the patient
- The patient's gender (male or female)
- The date when a patient starts and ends a radiotherapy treatment.

In *Simple Registry*, a `patient` user can visualize and change her name and gender, but he/she can only visualize the start and end dates of radiotherapy treatments.

Figure 4 illustrates the JSON file specification that needs to be given as input to CDRGen in order to generate the CDR according to Example (1). This JSON file specification is described, in a structured form, by the following six aspects:

- *Clinical data registry name*: Simple Registry (specified in line 2).
- *Types of users supported*: `administrator physician` and `patient` (specified in line 3). Note that since the `administrator physician` user always exists for any CDR generated, it is not specified.

[13]We call the entities that have the *historic* property as historic-type entities, otherwise we call them non-historic entities, which are the entities that can only have one instance (the default).

```
1  {
2      "clinical data registry name": "Simple Registry",
3      "users": ["patient"],
4      "entity groups": {
5          "person": {
6              "entities": [
7                  {   "entity": [
8                          "Patient",
9                          {   "attribute": ["Name", "string"],
10                             "properties": ["identifier"],
11                             "permissions": { "patient": "RW" } },
12                         {   "attribute": ["Gender", "enum"],
13                             "values": ["male", "female"],
14                             "permissions": { "patient": "RW" } }
15                     ]
16                 }
17             ]
18         },
19         "treatment": {
20             "entities": [
21                 {   "entity": [
22                         "Radiotherapy",
23                         {   "attribute": ["Start date", "date"],
24                             "permissions": { "patient": "R" } },
25                         {   "attribute": ["End date", "date"],
26                             "permissions": { "patient": "R" } }
27                     ]
28                 }
29             ]
30         }
31     }
32 }
```

Figure 4: The resulting JSON file specification according to Example (1).

- *Entities*: `Patient` and `Radiotherapy` (specified in lines 7-16 and 21-28, respectively).
- *Attributes* of each *entity*: `Patient` entity has a name (specified in lines 9-11) that holds a string (the patient's identifier) and a gender (specified in lines 12-14), which has an enumerated type with "male" and "female" as possible values; `Radiotherapy` entity has a start and end dates (specified in lines 23-24 and 25-26, respectively), where both hold a date.
- The *permissions* of each *attribute* for the `patient` user: read (R) and write (W) the patient's name and the patient's gender (specified in lines 11 and 14, respectively); read (R) the radiotherapy's start date and radiotherapy's end date (specified in lines 24 and 26, respectively). We only need to define the `patient` user's permissions since the `admin-`

`istrator physician` user has full access to any generated CDR, i.e., he/she can read and write in all the entities' attributes.

- The *entities* that compose each *entity group*: `person` entity group (specified in lines 5-18) is composed of the `Patient` entity; `treatment` entity group (specified in lines 19-30) is composed of the `Radiotherapy` entity.

4.2. CDRGen Input 2/2: JSON File User Interface Labels

The JSON file user interface labels describes the labels of the user interface that are not related with the data to be collected by the CDR (for instance, labels of buttons, informative sentences). This JSON file enables to customize the user interface labels in order to match the same idiom (e.g., Portuguese, English, French) of the data to be collected by the CDR. This JSON file is only composed of an object that contains several key-value pairs. The keys are predefined English strings and the corresponding value is a string that represents the label that is given to an element of the user interface. For example, `Add` is one of the keys and its value represents the label that is given to the add buttons that appear in the user interface.

4.3. CDRGen Output: Clinical Data Registry

The architecture of the generated CDR is the same as Umedicine's architecture and is illustrated in Figure 5.

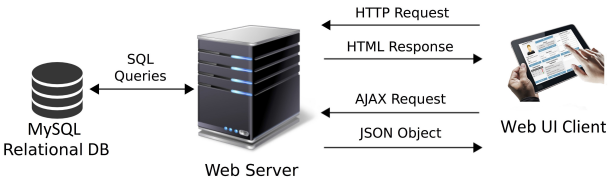

Figure 5: Architecture of the generated CDR. Figure edited from Lages *et al.* [3].

Basically, the generated CDR has a client-server architecture, which is the typical architecture of a system that queries a database whose access by users is achieved through a web browser. This client-server architecture is composed of three components: (i) a web user interface client, where the user interacts through a web browser; (ii) a web server; and (iii) a MySQL relational database.

The generated CDR will be accessible by the user types that were defined in the JSON file specification. Each user type may have different permissions over the data according to the given specification and it will be able to perform different functionalities as we stated in Section 3.2.

4.4. CDRGen: Parsing and Code Generation

CDRGen parses the two JSON files and generates the code for creating the CDR's database, web server and web user interface client. As shown in Figure 2 at the beginning of Section 4, CDRGen is composed of four modules as follows.

Parser. The parser is responsible for reading and parsing two JSON files: the JSON file *specification* and the JSON file *user interface labels*. Then, it sends the resulting data to the code generator modules (i.e., the database generator, the web user interface client generator and the web application generator). The data that results from the parsing of the JSON file *user interface labels* is sent only to the web user interface client generator module. On the other hand, the data that results from the parsing of the JSON file *specification* is sent to all code generator modules.

Database Generator. The database generator is responsible for creating a MySQL relational database to store the data that is entered by users through the user interface (i.e. the data collected by the CDR). The database to be created relies heavily on the specification. In general, each entity is represented by a table in the database and each attribute of that entity is represented by an attribute in the same table. In addition, each table that represents an entity has a relationship with the table that represents the patient (let us call it `patient` table). If a table represents a non-historic entity (i.e., an entity that can only have one instance), then there is a one-to-one relationship between this table and the `patient` table. Otherwise, if a table represents a historic-type entity (i.e., an entity that can have several instances), then there is a many-to-one relationship between this table and the `patient` table.

Web User Interface Client Generator. The web user interface client generator is responsible for generating the code that supports the interaction with the user. Specifically, this module creates the screens of the user interface. To that end, it generates Java Server Pages (JSP) files. The design of these screens is aesthetically based on the Umedicine user interface. These screens include the data entry forms and the screens for visualizing and navigating through the data. The data entry forms are the screens where the user enters data, i.e., are the screens where the user adds a new instance of an entity or edits an instance.

Web Application Generator. The web application generator is responsible for generating the code regarding the communication between the database and the user interface (i.e., the web server's role). As in Umedicine, the web server

must answer to requests sent by web clients, perform operations on the data, and store and retrieve data from the database [3]. The web server of the generated CDR has a structure equal to the Umedicine's web server: a Java application that uses the Spring Framework ecosystem[14]. The server component architecture is also the same as Umedicine's web server. It follows the Model-View-Controller design pattern, in which we have the model and the controller layers in the web server[15]. The model layer is composed of data access objects and services, while the controller layer is only composed of controllers. These three concepts are defined as follows:

- *Controllers* are responsible for the web client interactions, in which they receive web client requests, invoke services and send responses.
- *Services* are invoked by controllers or other services in order to process the data that is sent by clients or that is retrieved from the database. Basically, the services contain the business logic and process the data for the controllers use in their responses to clients.
- *Data Access Objects* are responsible for the interaction between the server and the database. It contains Structured Query Language (SQL) instructions that enable to store and retrieve data from the database. The data retrieved is converted into Java objects and then forwarded to the service layer.

In order to generate the web server of the CDR (i.e., a Java application), the web application generator generates Java classes that represent the controllers, the services, the data access objects and the entities that were specified. The Java classes that represent the specified entities are the classes used to convert the data retrieved from the database to Java objects.

5. Validation

In this section, we describe the assessment of CDRGen against two CDRs that were not used during its development. In concrete, we detail the testing of CDRGen against PRNC and its validation against PRECISE Stroke. We test CDRGen against PRNC in order to verify if our requirements analysis has been well performed, since PRNC was one of the CDRs used during that analysis. On the other hand, PRECISE Stroke was not used during the requirements analysis, so we use it to validate. We chose these two CDRs to assess CDRGen since we know what data is collected and the functionalities that they should provide. These CDRs cover two different medical specialties: Cardiology in the case of PRNC and Neurology in the case of PRECISE Stroke. Thus, we end up assessing CDRGen against two medical specialties besides the one that was used for testing CDRGen during its development, i.e., Urology.

In Section 5.1, we describe the evaluation metrics that we use and, in Section 5.2, we assess CDRGen against the concerned CDRs.

5.1. Metrics

To test and validate CDRGen, we measure the following metrics:

1. **Performance** of the generation process: *time* that the developer[16] spends writing the specification and time that CDRGen takes to generate the CDR.
2. **Quality** of the generated CDR, by measuring the *ratio* of:
   - The tables, their attributes and relationships that the generated CDR's database has in relation to what it is expected to have based on the data that is collected by the existing CDR.
   - The functionalities that we can perform through the generated CDR's user interface in relation to those provided by the existing CDR.

5.2. Testing with PRNC and Validation with PRECISE Stroke

For PRNC, the author (i.e., an expert) spent about 125 minutes (2 hours and 5 minutes) to write a specification that covered 5 entity groups, 12 entities and 243 attributes. Then, CDRGen took about 2 seconds to generate the CDR based on that specification. For PRECISE Stroke, the author spent about 284 minutes (4 hours and 44 minutes) to write a specification that covered 5 entity groups, 23 entities and 633 attributes. Then, CDRGen took about 4 seconds to generate the CDR based on that specification. Given this, we observe that the time for CDRGen to generate a CDR is irrelevant because it is in the order of magnitude of seconds. In addition, we observe that the time to write the specification based on the PRECISE Stroke's data was less compared to the time to write the specification based on the PRNC's data regarding the number of attributes specified (633 versus 243 attributes). This happened because many attributes of PRECISE Stroke are repeated in different entities, which simply led us to copy the definition of these attributes to the concerned entities.

[14] https://spring.io/

[15] The view layer is generated by the web user interface client generator module

[16] In this case we are referring to the author, i.e., an expert

For PRNC, CDRGen was able to generate a database that had all the tables and relationships between tables that it was supposed to have. Only 1 of the 244 attributes collected by the existing PRNC (i.e., approximately 100%) was not included in the generated database. These results shows that the requirements analysis performed regarding the data collected was almost excellent. On the other hand, we have not achieved as good results as these with PRECISE Stroke with respect to the generated database. Since PRECISE Stroke was not used in the requirements analysis, we were completely unaware of the data that it collected. Still, the database generated based on the data collected by the existing PRECISE Stroke had approximately 97% of the tables, 88% of the attributes and 91% of the relationships between tables that it was supposed to have.

These results with the existing PRECISE Stroke were not better since it collects data of types that CDRGen does not support, such as automatic computed fields[17], datetime, time, and image lists where each image has attributes that detail it. Finally, we also observed that the historic-type entities that exist in PRECISE Stroke are not properly supported by CDRGen.

Regarding the functionalities of the generated CDRs, we observed that the user is able to perform about 94% and 89% of the functionalities of the existing PRNC and PRECISE Stroke, respectively. These results were not better since CDRGen does not support the following three functionalities: (i) visualize statistical data; (ii) export all data of a set of patients; and (iii) delete all data of a patient.

Overall, as expected, we observe that we achieve a greater success with PRNC than with PRECISE Stroke, since PRNC was used in the requirements analysis. However, we also observe that we can generate a CDR that support much of the existing PRECISE Stroke that we barely knew. In addition, in both cases, we observe that the time required to generate a CDR when using CDRGen is on the order of magnitude of hours if the developer is an expert in the specification language. Possibly in the worst case this magnitude can go up to 2 or 3 days in the case that the developer does not know the specification language at all. Still, the time required to generate a CDR without using CDRGen (i.e., what currently happens) is much longer than this. In fact, the developer of the existing PRECISE Stroke took about 300 hours (12 days and 12 hours) of work to develop it, reaching the order of magnitude of weeks.

[17] Automatic computed fields are attributes that are calculated based on formulas that use the values of other attributes

6. Conclusions and Future Work

6.1. Conclusions

In this thesis, we presented a Clinical Data Registry Generator (CDRGen) software system to generate CDRs with a minimum effort in terms of software design and development. The CDR is generated from two JSON files: one that describes, in a high-level language, the characteristics of the data that needs to be collected by the CDR; and another that describes the user interface labels.

Throughout the development of CDRGen, we used the existing Umedicine CDR. Then, to assess CDRGen, we used the existing PRNC to test and the existing PRECISE Stroke to validate. As expected, we achieved a greater success testing with PRNC than validating with PRECISE Stroke, since PRNC was used in the requirements analysis. Although, we also generated a CDR that supported much of the existing PRECISE Stroke. Finally, during this assessment process, we observed that by using CDRGen, we can generate CDRs in just a few hours, as opposed to the standard procedure which can take weeks.

6.2. Limitations and Future Work

The limitations and the future work aspects of this thesis are separated in three groups. In the first group, we point out the aspects that ended up not being implemented in CDRGen as it was supposed to, namely:

- The attribute constraint that states that the value of an attribute must be one of the values that were already given to another attribute.
- The relationships between some specific entities (such as medicine and disease) as we identified in Umedicine during the requirements analysis.

The second group consists of the limitations that were identified during the evaluation of CDRGen. These limitations correspond to aspects that CDRGen does not support and, therefore, part of the future work would be to implement them.

Finally, in the third group, we consider other limitations, namely:

- The user interaction with the generated CDRs is always the same, which can be problematic if the user interface does not have good usability.
- The maintenance of the generated CDR, since over time it may be necessary to change the data to be collected, for example.
- The way the developer specifies the data to be collected by the CDR to be generated, which follows a verbose syntax that is described in

the JSON format. As such, part of the future work could be to develop a user interface that allows developers to specify the data to be collected visually.

References

[1] M. J. Santos, H. Canhão, A. F. Mourão, F. Oliveira Ramos, C. Ponte, C. Duarte, A. Barcelos, F. Martins and J. A. Melo Gomes. Reuma.pt contribution to the knowledge of immune-mediated systemic rheumatic diseases. *Acta Reumatologica Portuguesa*, 42:232–239, July 2017.

[2] Helena Canhão, Augusto Faustino, Fernando Martins, João Eurico Fonseca, Patrícia Nero and Jaime C. Branco. Reuma.pt - the rheumatic diseases portuguese register. *Acta Reumatologica Portuguesa*, 36:45–56, January 2011.

[3] Nuno F. Lages, Bernardo Caetano, Manuel J. Fonseca, João D. Pereira, Helena Galhardas and Rui Farinha. Umedicine: A System for Clinical Practice Support and Data Analysis. *Data Management and Analytics for Medicine and Healthcare (DMAH), VLDB Workshop*, 102–120, 2017.

Appendix A – List of Acronyms

CDR Clinical Data Registry

CDRGen Clinical Data Registry Generator

JSON JavaScript Object Notation

JSP Java Server Pages

LCDP Low-Code Development Platform

PMR Portuguese Myocarditis Registry

PRNC Portuguese Registry of Non-compaction Cardiomyopathy

ProACS Portuguese Registry of Acute Coronary Syndromes

PRo-HCM Portuguese Registry of Hypertrophic Cardiomyopathy

RNCI Portuguese Registry on Interventional Cardiology (*Registo Nacional de Cardiologia de Intervenção*, in Portuguese)

RNMAVD National Registry of Arrhythmogenic Right Ventricular Myocardiopathy (*Registo Nacional de Miocardiopatia Arritmogénica do Ventrículo Direito*, in Portuguese)

RON National Cancer Registry (*Registo Oncológico Nacional*, in Portuguese)

SQL Structured Query Language