BXE2E: A Bidirectional Transformation Approach for Medical Record Exchange

by

Jeremy Ho
B.Sc., University of Victoria, 2012

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Jeremy Ho, 2017
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

BXE2E: A Bidirectional Transformation Approach for Medical Record Exchange

by

Jeremy Ho
B.Sc., University of Victoria, 2012

Supervisory Committee

Dr. Jens Weber, Co-supervisor
(Department of Computer Science)

Dr. Morgan Price, Co-supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Jens Weber, Co-supervisor
(Department of Computer Science)

Dr. Morgan Price, Co-supervisor
(Department of Computer Science)

ABSTRACT

Modern health care systems are information dense and increasingly relying on computer-based information systems. Regrettably, many of these information systems behave only as an information repository, and the interoperability between different systems remains a challenge even with decades of investment in health information exchange standards. Medical records are complex data models and developing medical data import / export functions a is difficult, prone to error and hard to maintain process. Bidirectional transformations (bx) theories have been developed within the last decade in the fields of software engineering, programming languages and databases as a mechanism for relating different data models and keeping them consistent with each other. Current bx theories and tools have been applied to hand-picked, small-size problems outside of the health care sector. However, we believe that medical record exchange is a promising industrial application case for applying bx theories and may resolve some of the interoperability challenges in this domain. We introduce BXE2E, a proof-of-concept framework which frames the medical record interoperability challenge as a bx problem and provides a real world application of bx theories. During our experiments, BXE2E was able to reliably import / export medical records correctly and with reasonable performance. By applying bx theories to the medical document exchange problem, we are able to demonstrate a method of reducing the difficulty of creating and maintaining such a system as well as reducing the number of errors that may result. The fundamental BXE2E design allows it to be easily integrated to other data systems that could benefit from bx theories.

Contents

List of Tables

List of Figures

List of Algorithms

ACKNOWLEDGEMENTS

I would like to thank:

Jens Weber and Morgan Price, for mentoring, support, encouragement, and patience.

My parents, for supporting me through the best and worst of times.

My friends, for long, true and meaningful companionship.

By three methods we may learn wisdom: First, by reflection, which is noblest; Second, by imitation, which is easiest; and third by experience, which is the bitterest.
Confucius

DEDICATION

This thesis is dedicated to my parents.
For their endless love, support and encouragement.
waves

Chapter 1

Introduction

Medical practitioners are constantly working with large volumes of sensitive patient record data in order to perform their jobs. Over the last two decades, the introduction of Electronic Medical Record systems (EMRs) have changed the way how practitioners handle patient record data, replacing the old paper-only record systems in their practices. While there are the obvious benefits of saving resources such as paper, transitioning to EMRs also allow for more efficient record retrieval, enabling practitioners to spend more time on healthcare and less time on clerical work.

1.1 Terminology

Before we go any further, let us begin by first outlining the acronyms and terminology that are used throughout the thesis. A bidirectional transformation (bx) is a mechanism for maintaining consistency between two or more sources of information. [1] The many different approaches for applying bx theory is covered in Chapter 2. Our other non-bx related domain specific terminology can be grouped into either Healthcare or the Technology Stack.

1.1.1 Healthcare

The Health Level Seven (HL7) organization is a standards-developing organization for electronic health information. They are responsible for creating the Reference Information Model (RIM) and the Refined Message Information Model (RMIM) which are object models used in structured medical information exchange. A commonly used exchange format is the Clinical Document Architecture (CDA) which is a

flexible markup standard. It provides a standardized structure for certain types of medical records and provides a better way to exchange medical information between practitioners and patients.

In British Columbia, we had an initiative called the Physicians Information Technology Office (PITO). They were tasked with assisting BC physicians to transition to using EMR systems and achieve a level of "meaningful use" with them. Physicians were reimbursed some funds in order to offset EMR vendor service fees. One of the standards PITO introduced was the EMR-to-EMR Data Transfer & Conversion Standard (E2E-DTC or E2E) which is based off of the CDA. E2E was designed to support the standardized exchange of patient information between different EMR systems in BC with the aim to improve regional interoperability. The Open Source Clinical Application and Resource (OSCAR) is an EMR that is used by many practitioners in BC and is the main EMR platform used in this thesis.

1.1.2 Technology Stack

There are many software packages which may be required for certain programs to run correctly. Since OSCAR is written mainly in the Java language, many of the software dependencies are also Java based. Apache Velocity (Velocity) is a generalized template based engine which allows for transforming information. The MARC-HI Everest Framework (Everest) is an object-oriented framework which allows the manipulation of the CDA through Plain Old Java Objects (POJO). Everest is also capable of generating and parsing CDA documents which are normally found in the eXtensible Markup Language (XML) format.

The OSCAR EMR leverages a few frameworks and interfaces in order to store and manipulate health information. Historically OSCAR accessed the patient data in a MySQL database with the deprecated Java Database Connection (JDBC) method. Now OSCAR generally accesses the database through the Java Persistence API (JPA). JPA then translates the information into usable POJOs through Data Access Objects (DAO). Finally, the front-end interface is generally presented through Java Server Pages (JSP) which is rendered on the practitioner's browser. All of these frameworks must work together seamlessly in order to provide OSCAR users a smooth and safe EMR experience.

1.2 Electronic Medical Records

The number of Canadian practitioners using EMRs have drastically increased in recent years, with EMR adoption starting around 25% in 2007 and reaching up to 75% as of 2014 [2]. While the initial adoption of an EMR system does incur some training, workflow changes and monetary costs, the transition to an EMR has generally been positive. Many of the practitioners who have adopted EMRs have reported an improvement in their practice because it has increased their efficiency and reduced their practice costs [3].

Yet the introduction of EMRs into a medical workflow has not been the final and perfect solution to all the problems a medical practitioner may encounter. As there is no regulatory framework monitoring EMR system safety, we could encounter potential situations where EMR systems have programming errors or bugs, be dependent on unreliable hardware/software platforms, and introduce different clinical workflows which may lead to new failures [4]. Since practitioners handle sensitive patient data, patient safety is now dependent on an EMR's reliability and ability to mitigate potential failures before they happen.

Medical practitioners could use EMRs as just a digitized record storage system, but the true value of an EMR system would not be realized. An EMR system can only provide their true benefits when the information they store is standardized and structured so that it can help the practitioner make the best choices possible. Aside from just medical record storage, EMRs must also provide some form of "meaningful use", or the ability to generate recommendations, present context-sensitive information and transmit data to other EMR systems.

While EMR systems have made progress on recommendation systems and information presentation, transmitting and sharing data with other providers still remains a challenge. The lack of interoperability between differing EMR systems impedes with workflow tasks such as preparing lab reports, requesting lab orders, electronic prescription of medications, and requesting for consultation from a specialist. Studies have shown that high levels of EMR interoperability reduces time spent on those four tasks, improving the medical practitioner's efficiency [5].

1.2.1 Interoperability Problem

> "The absence of a robust set of standards to resolve data incompatibility issues is becoming increasingly costly to the U.S. healthcare delivery

system." [6]

The rapid adoption of EMRs in Canada has created an ecosystem of patchwork EMR systems across different regions and provinces. Similar to the U.S., the availability of a significant number of different EMR systems have led to a high degree of market fragmentation, and many of these systems lack any robust form of interoperability [7]. As a result, EMR usage is limited to mainly patient record storage and retrieval. While EMRs offer benefits over paper-only systems such as the ability to make practice related reflective exercises easier [8], the inability to digitally communicate and transfer medical record information to other EMR systems is currently a large roadblock for smooth patient care and advanced clinical decision support system development.

Medical practitioners will commonly need to perform a referral for consultation in order to acquire either a second opinion or more specialized advice. The referral will involve sharing some patient information with the specified consultant as they are also medical practitioners themselves. The consultant, usually a specialist, is expected to review the patient's information, schedule one or more encounters with them, and provide a consultation report back to the requesting practitioner with their second opinion or specialized advice.

The consultant will have problems acquiring the patient's health record from the practitioner if their EMR systems are not interoperable. In this situation, the only information that can easily be transmitted would be basic demographic and perhaps a few high level notes. As a result, the consultant will lack much of the pertinent data required in order to provide proper care. The consultant is forced to effectively rebuild a patient's health record locally, adding to their workload and reducing their ability to focus on providing effective healthcare. The final consultation report provided back to the originating practitioner will also be hard to process because the original practitioner will have to spend time manually recording the report.

Another use-case where EMR interoperability is important is when practitioners need to perform an EMR migration, or moving from one EMR system to another. This process requires the pertinent medical records in the old EMR system to be exported in some fashion and then imported into the new EMR system. Unfortunately, the fragmented Canadian EMR market will make this migration difficult due to a lack of interoperability. Unfortunately, EMR interoperability is a challenge, whether it be from lack of incentives to develop interoperability, technical variations between systems, or simply resistance from some EMR vendors due to political or business

conflicts [9].

Ultimately, these workflow examples illustrate the importance of EMRs having some form of medical record exchange mechanism. While practitioners can still perform their job even with the lack of interoperability, it does significantly hinder their efficiency and could allow for potential human errors to slip in. If EMR vendors had a standardized format to import and export patient records, it would go a long way to alleviating the medical record exchange issue practitioners are currently facing. As well, an import and export solution needs to have some form of consistency in order to ensure patient safety. Unfortunately, enforcing consistency is difficult to do without leveraging some form of bidirectional transformation technique.

1.3 Requirements

EMR interoperability is currently a challenge and needs to be addressed in order to facilitate smoother healthcare service. With the current medical record exchange problems in mind, an import / export solution needs to address three main factors in order to be successful.

1. The solution is correct with data presentation and semantics, minimizing any potential safety concerns.
2. The solution must be maintainable and easy to understand for new developers.
3. The solution should emphasize performance and scalability comparable with existing solutions.

Safety, maintainability and performance are factors that an interoperability solution must factor in when designed and implemented. Any solution that is unable to sufficiently satisfy all three factors mentioned above cannot be considered viable for live production use in the healthcare domain.

1.4 Thesis Statement

Current electronic medical record systems can import and export records with varying degrees of detail in some format. However, the import / export process is only as resilient as its ability to properly handle all the available data elements in the transfer

medium. By introducing bidirectional transformation techniques, we can apply certain guarantees to the import and export process and improve its reliability and utility.

The goal of this thesis is to answer the question: **To what extent is it possible to apply bidirectional transformation theory to a real-world medical record exchange problem and minimize its impact on performance?**

1.5 Thesis Outline

This thesis is broken down into seven chapters. In this chapter, we briefly introduced the interoperability problem, its applications from a clinical perspective and the terminology that is used throughout the thesis. We explained the desired features for a solution and the potential avenues to satisfy it. Chapter 2 reviews the current state of medical record transformations and provides an overview of the different bidirectional transformation theories. Chapter 3 explores the applications of bidirectional concepts in literature. Chapter 4 describes our solution from a high-level perspective. Chapter 5 outlines the rationale behind our implementation design decisions and provides a more in-depth perspective of our solution. Chapter 6 explains how our solution is evaluated and analyzes the results. Finally, Chapter 7 summarizes our findings and introduces future research avenues based on our solution.

Chapter 2

Foundations

In this chapter we begin with a brief overview of the OSCAR EMR ecosystem and how it relates to E2E, a type of CDA document. We also cover the basic concepts and fundamentals of Bidirectional Transformations (bx), including a brief survey of multiple approaches to implementing bx. The background and concepts explained in this chapter will lay the groundwork for understanding our BXE2E approach in later chapters.

2.1 EMRs and OSCAR

Electronic Medical Record (EMR) systems are responsible for storing and managing patient records specific to a single clinical practice. EMRs can store a wide range of health data including demographics, medications, allergies, immunizations, laboratory test results, measurements such as age and weight, vital signs, and billing information. EMRs accurately store the health state of a patient over time in one place, increasing its accessibility as compared to conventional paper records. As well, EMRs also support physicians with their delivery of care with things such as clinical decision support tools, alert reminders and billing systems.

The Open Source Clinical Application and Resource (OSCAR) is a web-based EMR system which has its origins from McMaster University. While it was originally designed mainly for academic primary care clinic use, it has since grown to become a multi-functional EMR and billing system for primary care physicians. OSCAR is a commonly used EMR system, holding a 15% market share of in British Columbia as of April 2015 [10] and 20% in Ontario as of August 2016 [11].

The usage of EMRs in BC rose over the last decade due to the initiatives set by the BC Physician Information Technology Office (PITO). PITO focused on encouraging physicians to not only transition to using EMRs in their practice by providing support and funding, but also to have "Meaningful Use" of said EMRs in their daily practice [12]. One of the ways an EMR system can assist with the meaningful use goal is by implementing the EMR-2-EMR (E2E) data transfer standard.

2.2 E2E

E2E is a BC implementation of the HL7v3 Clinical Document Architecture (CDA) R2 standard. CDA is a document markup standard which focuses on the exchange of health data between healthcare providers and patients [13]. The markup is implemented in the eXtensible Markup Language (XML) format which is designed to store and transport data as both a human-readable and machine-readable format as seen in Figure 2.1. As a CDA standard, E2E benefits from the structural organization of the Reference Information Model (RIM) [14], which provides models to represent health record content.

```
1   <recordTarget typeCode="RCT" contextControlCode="OP">
2     <patientRole classCode="PAT">
3       <id root="2.16.840.1.113883.4.50" extension="448000001" assigningAuthorityName="BC-PHN"/>
4       <addr use="H">
5         <delimiter>1234 Street</delimiter>
6         <state>BC</state>
7       </addr>
8       <telecom value="tel:2500000001" use="H"/>
9       <patient classCode="PSN" determinerCode="INSTANCE">
10        <name use="L">
11          <given>JOHN</given>
12          <family>CLEESE</family>
13        </name>
14        <administrativeGenderCode code="M" codeSystem="2.16.840.1.113883.5.1" codeSystemName="HL7 -
    ↪   Administrative Gender" displayName="Male"/>
15        <birthTime value="19400925"/>
16        <languageCommunication>
17          <languageCode code="EN"/>
18        </languageCommunication>
19      </patient>
20    </patientRole>
21  </recordTarget>
```

Figure 2.1: A chunk of an E2E record for a hypothetical patient named John Cleese

Although the CDA's XML structure ends up causing an E2E document to become quite verbose, patient information is encoded in such a way as to minimize any potential ambiguities and preserve the original intent of the record. Some of E2E's

use cases includes episodic document transfers, referral requests, data conversion and data transfer. Having E2E as a common data interchange format makes it easier for differing systems to properly access and handle medical data, improving the EMR's meaningful use.

2.3 OSCAR and E2E

OSCAR has a large marketshare in BC with an estimated 700 to 800 clinician providers as of 2015 [15]. Because of this, PITO wanted the E2E standard to also be supported in OSCAR to further promote provincial interoperability. Additionally, the SCOOP research network from the University of British Columbia was also interested in OSCAR having E2E support as it could facilitate standardized anonymous data collection and analysis [16]. This interest eventually led to the development of a functional E2E export in OSCAR.

OSCAR has three distinct generations of E2E export functionality. The first generation was designed using the Apache Velocity library, a Java based templating engine [17]. By creating a template which embeds both the the data fields and the logic required to create an E2E document in the right places, it was capable of satisfying the export requirements of PITO's E2E standard. Unfortunately, this approach rapidly became difficult to maintain and bug-fix since all the transformation logic resided on a single monolithic template.

	Strengths	**Weaknesses**
Velocity	Fast development Java/Template code separation	Difficult to maintain Cannot support imports
E2Everest	Java only code design Easy to unit-test	Confusing object inheritance Limited code re-usability
BXE2E	Enforces consistency Highly modular design	Requires Java 8 Many small class objects

Table 2.1: The major strengths and weaknesses in each E2E generation

Looking to improve OSCAR's E2E export maintainability, the second generation of OSCAR's E2E exporter, named E2Everest, swaps out Velocity and instead uses the Everest framework [18]. It used a deferred model-populator design, where the models transform the data elements, while the populators assemble the elements together. As

this was done completely in Java code, maintainability did significantly improve as compared to the Velocity template approach. The capability of adding unit tests to the exporter greatly improved its reliability.

While the Velocity and E2Everest are able to satisfy the export part of the E2E standard, neither of them were capable of safely importing E2E documents. Although Velocity does not have the ability to handle importing, E2Everest is technically capable of import transformations. However, even if E2Everest were to have an import function, it would be an independent transformation function with no explicit connections to its export equivalent. There would be no export and import function coupling to ensure data transformations in both directions would proceed correctly.

As patient record transformations are a safety critical process, it requires an equally rigorous design and implementation to ensure that the transformation preserves and maintains correctness. We would need some way of ensuring that both the export and import functions would follow the transformation mappings provided by the E2E specification. One way we can do this is by explicitly coupling the pair of transformations together through Bidirectional Transformations. This is done in BXE2E, the third generation exporter which will be discussed in more detail in Chapter 4.

2.4 Bidirectional Transformations

Before we can talk about Bidirectional Transformations, we must first define what a transformation is. We define a *transformation* as the finite set of operations which converts a set of data values from a source data model to the target data model. In effect, a single application of a transformation to a source dataset will change the data such that its representation conforms to the target data model. These transformations can be broken down into two main steps: the data mapping, and the code generation.

For any data transformation to occur, it requires an equally well-defined data mapping. A data mapping is a set of data element mappings between two differing data models. This mapping can also be considered a type of mathematical function because the mapping will define how the output target data will be based on the source input data. These data mappings are the core of the data transformation because they define where each discrete data element should go.

Depending on the behavior of our data mapping function, it can be classified as either injective, surjective, or bijective. An *injective* (one-to-one) function has every

element in the source domain mapping to at most one element in the target domain. A *surjective* (onto) function has every element in the source domain mapping to at least one element in the target domain. Finally, a bijective function is both injective and surjective. In essence, very element in the source domain will map to exactly one element in the target domain.

$$\forall x, x' \in X, f(x) = f(x') \Rightarrow x = x' \qquad \text{(Injection)}$$

$$\forall y \in Y, \exists x \in X \ s.t. \ y = f(x) \qquad \text{(Surjection)}$$

The best case scenario for a data mapping is if it is bijective. A bijective data mapping is easy to implement in both the forwards and backwards directions because each element has a clear and concise mapping. However, practical complex data models rarely allow for completely bijective mappings due to their differing ways of representing and grouping data. Although the two different models will have some common data elements, the rest of the elements will either be loosely associated on the other model, or not be represented at all.

This is one of the main issues when creating transformations between data models as inevitably some data elements are discarded. Adding a step further, when considering two-way functions, the dropped data needs to be reconstructed and restored in some way. A robust data mapping needs to define where data needs to go in a transformation as well as specify how missing data can safely be reconstructed and restored.

The second part of transformations involves the code execution. While the data mapping defines where things should go, code generation deals with how the data will be transformed. Code generation can either be done directly by the programmer, or it can be derived from a higher level abstraction of written code. The ultimate goal of this step is to perform the transformation, following the provided data mapping from the previous step. The final product of code generation is some form of an executable program where the transformation may be run from.

Data transformations are quite common, whether it be converting database data into a presentable front-end GUI, interfacing with other platforms, or even recording data inputs and saving them into some common format. A uni-directional transformation itself is not difficult for a developer to create and implement. However, it is difficult to create a proper transformation which can go not only forwards, but backwards as well. We can transform the data, but the challenge is guaranteeing consistency between the source and target data.

2.4.1 Properties

Bidirectional transformations (bx) are a mechanism for maintaining consistency between two or more related sources of information [1]. There is a large body of active research exploring how bx may be applied in many domains such as software engineering, programming languages and databases. Researchers are interested in bx because it provides a framework for defining and enforcing consistency between the source and target models.

There are two main schools of thought in bx: state based transformations and operation based transformations [1]. State based approaches solely rely on the source and target data structures in order to calculate the modifications. In this approach, the transformer must know how the data is structured in order to apply any modifications. For example, if the transformer needs to translate an update in the target back to the source, it would need both the original source and the updated target in order to calculate an updated source.

On the other hand, operation based approaches uses a transformation language which propagates changes in the data to both the source and target at the same time. Each operation propagates the updates to both sides simultaneously which ensures that both the source and target are always consistent with each other. While both approaches can generate consistent transformation results, the type of bx approach is usually determined by what the task requires and what is accessible to the program.

Consistency

Above anything else, we assert that the behavior of a transformation must be deterministic in order to be consistent. This is because a non-deterministic transformation cannot yield a consistent result, and thus cannot provide any reliable results. With this assertion, we can model our transformations as mathematical functions. To do this, we will set up some notation and terminology. Let S and T represent the sets of all source and target models, and R be the relationship between S and T. The source and target are considered to be in a relation if the relation $R \subseteq S \times T$ holds.

Given a pair of models where $s \in S$ and $t \in T$, $(s, t) \in R$ if and only if the pair of models s and t are considered consistent. We must also define the two directional transformations $\overrightarrow{R}$ and $\overleftarrow{R}$ which are a subset of R:

$$\overrightarrow{R} : S \times T \longrightarrow T \qquad \text{(Forward)}$$

$$\overleftarrow{R} : S \times T \longrightarrow S \qquad \text{(Backward)}$$

Given a pair of models (s, t), $\overrightarrow{R}$ will calculate how to modify t such that the relation R is enforced. $\overleftarrow{R}$ will also do something similar, but propagates the changes in the opposite direction to modify s. The end result of either transformation should should yield consistent models, i.e., $\forall s \in S, t \in T : (s, \overrightarrow{R}(s, t), (\overleftarrow{R}(s, t), t) \in R$. With this notation and terminology, we can begin to investigate the properties of bx which affect consistency.

In general, a properly implemented bx must at minimum obey the definitional properties of *correctness* and *hippocraticness* [1]. Transformations which follow those two properties can be considered *well-behaved*. Correctness in bx is defined such that each pair of transformations shall enforce the relationship between the source and target [19]. Essentially, the transformations $\overrightarrow{R}$ and $\overleftarrow{R}$ are to enforce the relation R, and we can posit that a transformation L (which enforces relation R) is *correct* if:

$$\forall s \in S,\ \forall t \in T\ :\ L(s, \overrightarrow{L}(s, t))$$

$$\forall s \in S,\ \forall t \in T\ :\ L(\overleftarrow{L}(s, t), t)$$

A transformation pair L is considered correct if it is capable of bringing the source and target into the specified relationship [19]. However, the notion of "correct" can have a somewhat subjective scope because it could either mean that there exists no invalid data elements, or it could mean that all parts of the data relations in a model make sense. Complete correctness is difficult to achieve simply because S and T contains an infinite number of potential states. For L to be proven completely correct, it must demonstrate that for all potential source and target pairs, it is either already in a relationship, or that the transformation is capable of applying the relationship on the data set.

One way we can address this is by limiting the scope to partial correctness. This makes it easier to achieve a correct transformation because we limit the scope of what the transformation L has to manage. For example, instead of assuming that L covers all potential S and T pairs, we can limit the scope by saying that L is correct for pairs (s, t) where s and t are considered valid. This makes it so that L does not need to worry about covering invalid model states.

Hippocraticness in bx is the property in which the transformation L avoids modifying any elements within the source and target which are already correct and covered within the specified relationship R [19]. Another way of putting this is that even if target models t_1 and t_2 are both related to s via R, it is not acceptable for $\overrightarrow{R}$ to return t_2 if the input pair was (s, t_1). Formally, the transformation L (which enforces relation R) is *hippocratic* if for all $s \in S$ and $t \in T$, we have

$$L(s,t) \implies \overrightarrow{L}(s,t) = t$$
$$L(s,t) \implies \overleftarrow{L}(s,t) = s$$

Drawing its name from Hippocrates quote of "First, do no harm", the property of hippocraticness ensures that we do not harm any existing relationships by modifying them. This is a useful property because it minimizes any unnecessary computation in the transformation. This property can be implemented in the transformer via a *check-then-enforce* pattern which ensures that any correct artifacts remains correct. The transformer then only enforces the specified relationship on artifacts which are not in the correct state.

Of course, this implies that should the relation R is not be bijective, then at least one of the transformation directions must look at both arguments. The transformation will behave differently depending on whether the target model t is supplied or is empty. This is expected because if you are supplied a target t, we can assume that the transformation does not break any pre-existing relationships that t may already have with s. Another way of viewing hippocraticness is that the transformation L will preserve any previously done work on the pair (s, t).

However, just the well-behaved properties of correctness and hippocraticness may not be sufficient to cover all potential transformation cases [19]. Suppose we let S be a tuple containing a name and an id, and let T be a tuple containing a name and a city. We let R be the relationship such that s and t are consistent iff the names appear on both sides, and let $\overrightarrow{R}(s,t)$ return t. This forward transformation will otherwise generate t' via the following steps: 1) deletes any tuples in t which do not have a corresponding name in s 2) creates new tuples in t for any names in s that did not exist in t 3) set the city value for *all* tuples to "Victoria". Conversely, we let $\overleftarrow{R}(s,t)$ return s if it is already consistent. Otherwise, we return a correct set of tuples with the correct names, and set *all* of the tuples' ids to a randomly generated value.

In this example, we can see that the transformation R is technically correct because

the relation only concerns itself with the name, not the id nor the city. It is also hippocratic because it by definition does not modify anything already within the relation. What we are left with are transformations which can maintain the relationship R, but do not properly preserve the peripheral id and city elements. Intuitively, when we think of a relationship, we expect it to be more vigilant when it comes to preserving data instead of overwriting any unmonitored field with either a preset or randomly generated value.

The well-behaved properties of correctness and hippocraticness are not the only properties that can help enforce consistent behavior. Other properties such as *undoability* [19], *history ignorance* [20], *invertibility* [21] and *incrementality* [22], can help enforce consistent transformation behavior. However, these other properties are not required like correctness and hippocraticness because they can impose restrictions which can severely limit the usefulness or usability of bx.

Undoability is the property where propagated state-based transformation changes can be fully undone and reverted back to its original state [19]. Intuitively, this implies that every modification applied to the source S which is propagated to target T can be rolled back by applying an equivalent modification which undoes the previous modification. The end result should be the original unchanged source and target models. Formally, we can say transformation L is *undoable* if:

$$\forall s, s' \in S \ : \ L(s,t) \implies \overrightarrow{L}(s, \overrightarrow{L}(s',t)) = t$$
$$\forall t, t' \in T \ : \ L(s,t) \implies \overleftarrow{L}(\overleftarrow{L}(s,t'),t) = s$$

History ignorance is the idea that the result of a transformation is independent of whether we have executed a transformation already or not [20]. For example, if we do a modification to s, then transform it to t, and then do another modification to s and transform it to t again, the result should be the same as if we were to do both modifications to s first and then transforming to t once. Formally, transformation L is *history ignorant* if:

$$\overrightarrow{L}(s, \overrightarrow{L}(s',t)) = \overrightarrow{L}(s,t)$$
$$\overleftarrow{L}(\overleftarrow{L}(s,t'),t) = \overleftarrow{L}(s,t)$$

The main problem with the undoability and history ignorance properties is that it may be too strong and restrictive of a definition for bx [19, 20]. Suppose we have a

transformation which involves deleting some information from s to yield s'. The delete operation would be propagated over to t, yielding the equivalent t'. However, our t' will lose the appropriate entry as well as any peripheral data which is not represented in s'. Suppose we then undo the delete operation by transforming s' back to s. While we can restore s, the propagation of this restore will not completely revert t' back to t because we may not be able to recreate all of the lost peripheral data originally in t.

In that example, although the transformation is correct and hippocratic, it is not undoable simply because we cannot guarantee all peripheral information to be restored. If we were to do the deletion and restore modification together first and then transformed, we would technically get the right t. However, the result would not be the same as when we treated the delete and restore modifications separately. As such, the transformation is not history ignorant simply because combining the modifications together does not yield the same result as keeping the modifications separate.

While undoability focuses on state-based transformations, invertibility focuses on operation-based transformations. An operation-based transformation is invertible if there exists a pair of operations which modifies the model state, but then returns back to the original model state [23]. Formally, for a relationship R, there exists an operation o which changes model s to s', of which $s, s' \in S$. To maintain relationship R, operation o would also similarly update model t to t', of which $t, t' \in T$. Operation o is considered *invertible* if there exists an associated operation o^{-1} where $o \times o^{-1} = 1_s$. 1_s represents the "do-nothing" operation to model s, and the application of o^{-1} to s' and t' should yield the original s and t back.

As with undoability, invertibility also faces a similar problem with data preservation. In the ideal situation of *strong invertibility*, the operations o and o^{-1} applied to both s and t would yield no change to their states at all [21]. However, the act of propagating operations between S and T is not clean because there can be information loss due to differing models, of which the inverse operation would not necessarily be able to fully restore. There is however a notion of *weak invertibility* where operations o and o^{-1} yield a relation equivalence between the original s and the final s [21]. Essentially, as long as the relation R is maintained, and the state of s before and after the operations is equivalent, regardless of whether or not the peripheral data loss is restored.

Incrementality is mainly used in the incremental view maintenance problem, where the act of propagating consistent model view changes is critical for preserving proper consistency [22]. Another way of viewing incrementality is by representing updates to the relations as deltas, or an ordered collection of incremental changes. Although

incrementality can influence how much a transformation can maintain consistency, they are not required in order for a bx to function because incrementality is classified as a quality property [1].

Data Loss

The multiple properties of bx act to preserve some degree of consistency between the source and target models. Unfortunately, it is inevitable that our transformations will encounter some form of data loss, even when the transformations are designed to be well-behaved. This is because source model S and target model T are inherently different models. Suppose we use our previous example where we let S be a tuple containing a name and an id, and let T be a tuple containing a name and a city. The relationship would only synchronize between the name from both S and T. This can be visualized with a simple venn-diagram.

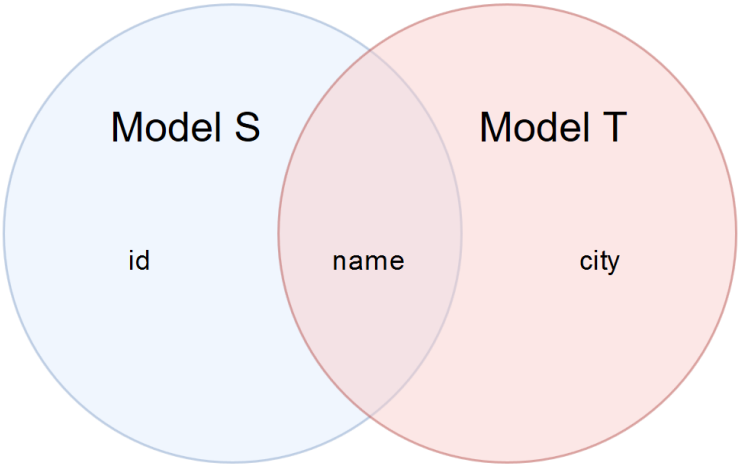


Figure 2.2: A venn diagram of the data set relationships between models S and T

As shown in Figure 2.2, the only element which is shared between both models is the name. The id and city elements can not be shared in between both models simply because the other model does not have an equivalent representation of the data element. This is an example of a symmetric transformation. A symmetric transformation will lose some information in both the forwards and backwards directions [24]. In contrast, an asymmetric transformation will lose some information in only one of the transformation directions.

In both the symmetric and asymmetric transformations, there will exist a common subset of data elements which are represented in both models S and T. In our

example, it is only the name element. We define this common subset of data as the *lowest common denominator* between S and T and write it as $lcd(S,T) = d$ such that $d \in S$ and $d \in T$. If relationship R only enforces data elements in d, we can guarantee a baseline level of data synchronization between S and T. However, the relationship would not guarantee data outside the lowest common denominator to remain synchronized.

With an operation-based bx approach, we may be able to update both models at the same time, but we will not have guarantees that data outside of d are updated accordingly. Unless there exists operations defining how the peripheral data is to be handled for all possible changes, at some point we will lose some information with the bx. With a state-based bx approach, we can only act on the state of the models. Because of this, the transformation process will simply lose the data that cannot fit in the other model.

There are multiple ways transformations can lose information, including deletions, syntax differences, and semantic differences [1]. A deletion is a transformation where there exists some data element from the source model does not appear in the target model. This is equivalent to discarding data when it is transformed into the target model. When performing the backwards transformation, a good transformation must reconstruct the discarded data in some way.

A syntax difference is more subtle than a deletion because it involves the way a data element is represented within the models [25]. Take for example the representation of a date. At first the concept of a date is straight-forward: it should contain the day, month, and year. However, we can quickly encounter the issue of how the date is represented - do we do month first, day first, or year first? They all represent the same date, but will appear differently depending on the model. Proper bx will handle these types of syntax differences as long as there are well defined rules mapping between the two forms.

Semantic differences are subtle nuances in data representation. While the piece of data is supposed to mean the same thing, the difference in representation can yield a slightly different meaning or lead to potential confusion. Suppose we have a statement "The patient was given pain medication" versus "The patient was given medication for pain". While they grammatically look the same, the subtle meaning between the two statements can imply different things [6]. The first asserts that pain medication was provided, whereas the second implies that medication was provided in order to address their pain. This type of potential meaning loss can be mitigated by consulting

with a domain expert.

One way we can preserve discarded or lost information is by using a *constant-complement* approach [26]. A *complement* is a data structure that preserves any source data that does not fit in the target model. Suppose we are to transform from model S to model T with relationship R. Forward transformation $\overrightarrow{R}$ will lose some information when converting s to t. A separate complement k can be generated along with the forward transformation which stores the data that does not fit in t. We can take advantage of k in the reverse transformation $\overleftarrow{R}$ by merging in the missing complement data to the output s.

In order to use complements in a function, we need to define a *function tuple*. Essentially a function tuple takes in one input, and yields two discrete outputs. Let two total functions $f \in S \to T$ and $g \in S \to K$. A tupled function $\langle f, g \rangle \in S \to (T, K)$ is defined as follows:

$$\langle f, g \rangle s = (f\ s, g\ s)$$

Given a source domain S, the tupled function $\langle f, g \rangle$ will return a pair of outputs. The tupled function does this by duplicating input s and passing one copy to f and the other to g [27]. The tuple then contains the results of f which is $t \in T$, and the other is the result of g which is a complement "residue" $k \in K$. Of course, this means that the constant-complement approach has to allocate memory for not only target t, but its associated complement k.

The complement information can be handled in multiple ways, depending on how the bx will behave. We could save the complement in the source system and then utilize it when the reverse transformation is invoked. Another approach could package both the complement and the target data together and send it to the target system. A reverse transformation would then require the target system to send back the associated complement and the target data. Yet another approach could be to skip generating the complement and instead have the reverse transformation require both the target data as well as its associated original source data.

2.5 BX Approaches

There exists a large body of active research exploring how bx can be applied in many domains such as software engineering, programming languages and relational

databases. Some practical applications of utilizing bx are maintaining Graphical User Interfaces [28], generating updateable database views [26, 29], data integration and exchange [30, 31], Domain-Specific Languages [32], and structure editing [33].

As we can see, the domain of bx research is vast and multi-disciplinary, with all bx approaches aiming to maintain consistency between two sources of information by handling any incurred data loss when transforming and respecting the properties of well-behavedness. However, with so many domains and research directions, we find that the terminology between differing domains may not match up even though they refer to the same concept [1]. While we cannot cover all aspects of current bx research here, we will present an introduction to the key concepts and approaches to bx in relational databases, Triple Graph Grammars, and Lenses.

2.5.1 Relational Databases

In database literature, the core unit of data transformation is called a query. Queries execute efficiently by leveraging a declarative syntax with clean semantics [1]. All database transactions, whether it be moving large quantities of data around or converting the data presentation, will involve some kind of query. Much of database bx research has been focused on constraining queries and investigating whether they can be "reversed" and yield some meaningful results [34, 35].

Bx in databases can be done in two main ways: either operationally, or by instance. Operational bx, also seen as the view-update problem, focuses on transforming the read and write operations on model S to yield equivalent results if also applied on model T. Let Q be a set of queries, S be the source model and T be the target model which is a set of views on source model S. Operational bx updates T by re-running Q on an updated S, regenerating an updated T as expected [1]. Most operational research investigates how the constraints on Q can facilitate this form of operational updates.

On the other hand, instance based bx focuses on the model state, determining how much of a model S can remain unaltered when transformed to model T and then reversed back to S. Instanced based bx is also known as data exchange in literature [36, 37]. The main goal of this approach is to transform instances of model S to instances of model T with a set of constraints $\sum_{ST}$. This creates a mapping tuple $(S, T, \sum_{ST})$. In order to create bidirectionality, another mapping tuple $(T, S, \sum_{TS})$ must be constructed such that composing both mappings creates an identity [35].

Information capacity can vary when using mappings since the composition of both mappings can only yield a partial identity, if at all [38]. Database schema evolution can be difficult because the act of evolving a model can impact services that relied on the old model [24]. The concept of co-evolution creates relationships between the source and target models because the evolution from S to T should have some notion of consistency. Tools supporting schema co-evolution need to make sure that the act of transforming still allows target T to be usable by both current and legacy services.

2.5.2 Triple Graph Grammars

The Graph Transformation community has a bx technique called Triple Graph Grammars (TGG). Drawing its influence from category theory, TGGs are a technique for defining correspondences between two differing models [39]. Originally introduced in 1994 by Andy Schürr, TGGs have become a declarative and practical method of applying transformations in both directions [40].

The main draw for TGGs is its ability to formally define a relation between two different models, and then provide a means of transforming a model from one type to the other. This is done by creating an explicit construct which defines how specific elements are related with each other, also known as a correspondence, between the two models [39]. Consistency between both models can be maintained by utilizing these correspondences and updates to either model can be propagated to the other model incrementally.

A TGG model consists of three sub-graphs consisting of the source model, the target model, and the correspondence model between the source and target. The correspondence graph describes the mappings and constraints that must be applied to the source and target in order to transform between the two models. Of course, a model alone does not create a bx. TGGs also rely on rules which define how a graph model will change.

Rules

A Triple Graph Grammar rule is a form of graph rewriting which does pattern matching and graph transformations. We define a graph G as $G = (V, E)$ where V is a finite set of vertices and E is a finite set of edges. Let S be the source graph, C be the correspondence graph and T be the target graph. Formally, we can define a TGG rule as a pair of morphisms between a source and target graph coordinated by the

correspondence graph. As such, let a TGG rule be $r = (S \leftarrow C \rightarrow T)$ where $C \rightarrow S$ is injective. $C \rightarrow S$ must be injective in order to allow for forward rule pattern matching.

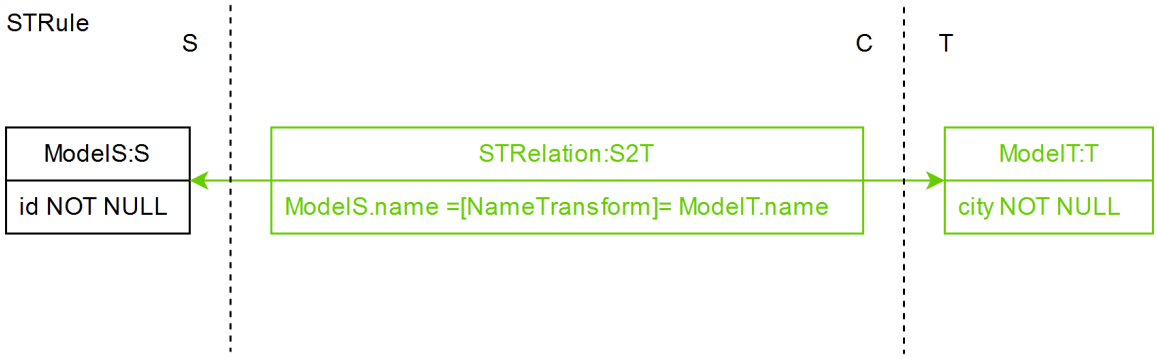


Figure 2.3: An example of a Triple Graph Grammar Rule

A rule is denoted with the two colors black and green. Black objects and links represent elements which occur on both sides of rule r, while green objects and links represent elements which only occur on the right hand side of the rule [39]. Another way of visualizing the rule is that the black objects are the parts we pattern match on, while the green elements specify how the transformation will proceed when there is a match. In Figure 2.3 we see an example shorthand TGG rule "STRule" for transforming between S and T. In the figure, the black object is "ModelS", meaning that this rule will match and execute for every ModelS object that is found.

Let R be a set of TGG rules, where $r \in R$. Each TGG rule defines three graph transformation rules: one forward rule, one backwards rule, and one synchronization rule. Ultimately, each rule in R will assure that a given source S and target T is able to consistently transform between each other. Each correspondence C tracks the differences between S and T on an element-to-element basis. The total sum of all the rules in R defines a language which describes the transformation which can occur between the source and target.

TGG rules can be classified into Islands, Extensions, and Bridges as seen in Figure 2.4 [41]. Islands are context-free rules which do not require any context. An island rule may create elements in just the source, the target, or both sides, yielding an "island" of new elements. Extension rules behave similar to island rules, but instead require a context. Extension rules extend the model by creating and directly attaching new elements to the models. Finally, bridge rules connect two differing island contexts together by creating new elements which form a "bridge" between them. Bridges can be generalized to connect multiple islands at once, although this is uncommon.

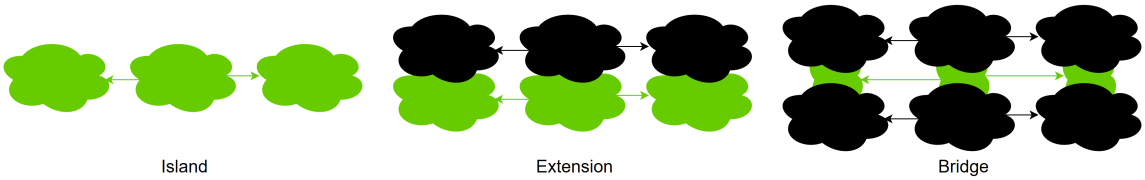


Figure 2.4: Three different kinds of TGG rules: Island, Extension and Bridge

In general, using extension rules is not recommended because these types of rules require a context and can end up becoming extremely complex and hard to comprehend. On the other hand, island and bridge rules are easier to compartmentalize and are easier to break down [41]. The ability to minimize the complexity of the rules is important because TGG rules are executed based off of pattern matching in the model graphs. Simpler and more concise pattern matching is easier to maintain and verify.

As TGG relies on pattern matching for rule execution, the order of rule execution is non-deterministic. Non-determinism is great for optimization because the rules can be matched and executed in parallel, quickly populating and transforming the graphs. However, this means that there are cases where the TGG will never terminate because there is always another pattern match. One of the ways we can address this non-termination issue is with *Negative Application Constraints* (NAC).

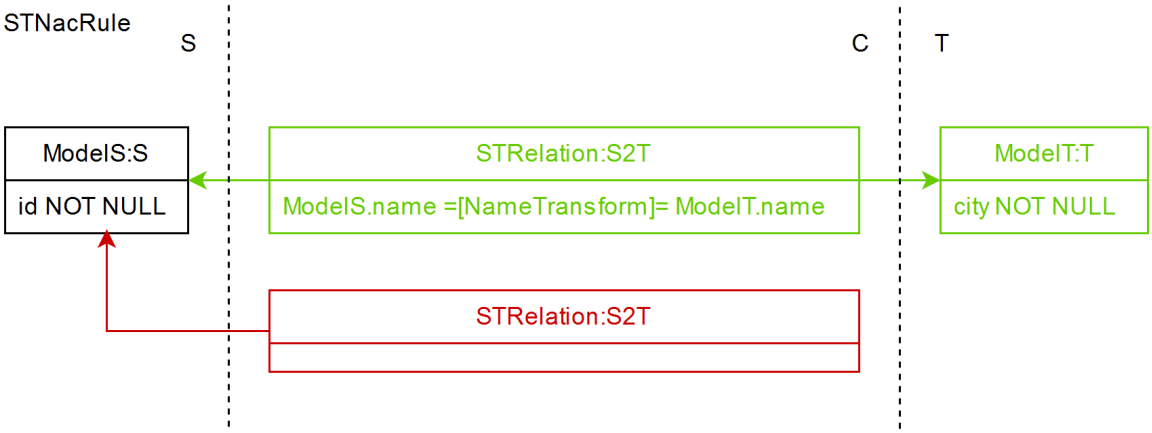


Figure 2.5: An example rule with a Negative Application Constraint

NACs assist with the TGG pattern matching by adding another stipulation saying that whatever element or link in the NAC must not already exist in order to satisfy the pattern match [39]. Well constructed NACs can restrict how many times a rule will be executed, such as ensuring that the rule does not re-execute on a section that has already been handled by the same rule previously. As shown in Figure 2.5, we

conventionally depict a NAC in red to signify that a pattern match shall not have those elements already there. In this figure, the NAC ensures that this rule only runs once for any "ModelS" that have not been paired with an "STRelation" object already.

Confluence

Another property we need to factor in with the non-deterministic order of rule execution is *confluence*. Confluence is the property where the order rules are applied will not affect the final graph state. As long as all the rules with pattern matches are executed, the final result will be the same, regardless of order. The local Church-Rosser theorem states that two rules can be executed in any order and will *converge* to the same end result [42].

Formally, if we have two parallel independent rules A and B, we can expect a model in state 0 to be transformed to the same final result when A and B are applied, regardless of order. Either A goes first (AB) or B goes first (BA), ultimately yielding a model in state A and B. If rules A and B are confluent, the results of AB and BA will converge to the same answer as seen in Figure 2.6.

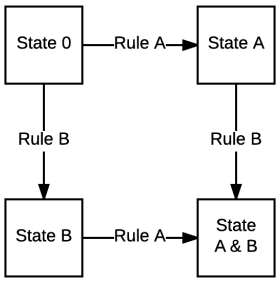


Figure 2.6: A visualization of how confluence uses the Church-Rosser theorem

However, the local Church-Rosser theorem does not fully cover the confluence property of our TGG rules. Suppose now that rule A deletes all X elements in the model and rule B generates a new X element in the model. As these two rules modify the same X in the model, they will conflict with each other. One way we can deal with this is with *critical pairs*. A critical pair is a set of two rules which do not have overlapping or conflicting contexts [43].

While a critical pair alone does not ensure confluence in a set of TGG rules, we can assert more if all pairings of rules in a finite set of rules are critical pairs. Formally, let

R be the set of rules defining a TGG, and let (r_0, r_1) be a critical pair between rules r_0 and r_1 where $r_0, r_1 \in R$. A set of rules is locally confluent if all of the potential combined pairings of rules in R form critical pairs. Essentially, $\forall r_n \in R$ where $n \in \mathbb{N}$ is an enumeration of all rules in R, $\exists\ (r_x, r_y)\ s.t.\ x \neq y$. Thus, a TGG is confluent if and only if all rules do not have overlapping or conflicting contexts [43].

2.5.3 Lenses

The Programming Languages community has a couple of approaches to handling bx. Bx programming languages can be classified according to two features: the semantic laws of the language itself, and the mechanisms which enforce bidirectionality [1]. We can further break down programming languages based on their semantics. Languages could be bijective, where the forward transformation is an injective function, and the backwards transformation is a direct inverse.

The other option for bx is to just have a bidirectional language, where the forward transformation is an arbitrary function. To compensate for potential data loss, the backwards transformation takes in two arguments: both the updated output and the original input [1]. This method usually requires the functions to obey some form of "round-tripping" laws to ensure that data loss is mitigated and restored in the opposite transformation direction.

With the bidirectional language method, we define a forward transformation as a *get* function, and a reverse transformation as a *put* function. A program can be considered bidirectional if a developer writes a get function in a pre-existing functional language and then applies a bidirectionalization technique to derive an associated put function [27]. Deriving the put can be done in two ways: utilizing an algorithm that parses the syntactic representation of the get, or by using higher-order and typed abstractions in the programming language itself for derivation [27].

One of the ways we can represent the get and put functions in an orderly manner is through a *lens*. Simply put, a lens is a bidirectional program consisting of at least a forward *get* and a reverse *put* computation [44]. Formally, let L be the set of lenses which enforces relationship R between source model S and target model T. A lens $l \in L$ will define the three functions *get*, *put* and *create* as follows:

$$l.get \in S \to T \quad \text{(Get)}$$

$$l.put \in T \times S \to S \quad \text{(Put)}$$

$$l.create \in T \to S \quad \text{(Create)}$$

The get function is a mapping from S to T, while the put and create functions are mappings from T to S. The difference between the put and create is that the put function takes in both the updated instance of T and the original instance of S, while the create only takes in the updated instance of T. As the create does not have the original S to draw from, it must rely on pre-defined default values in order to fill in any gaps between the model transformations.

When we consider model transformations, generally one of the two models will be considered the original source of information. Typically this model will also be the larger of the two models as well. We consider the model with the original source of information to be the *concrete* model, and the other typically smaller model to be the *abstract* model [44]. This difference is important because it affects whether a lens is considered symmetric or asymmetric. While an asymmetric lens transforms from a concrete model to an abstract model, a symmetric lens transforms between either two concrete models or two abstract models [44].

Asymmetric lenses generally lose information when transforming from the concrete model to the abstract model. This is why the put function requires the original source, while the create function generates pre-defined values in order to fill in the gap in information. However, symmetric lenses lose data in both transformation directions because both models either are unable to store or represent a certain subset of the other model's data [44], or both models can be considered original sources of information. As seen in Figure 2.7, the blue circle elements can be shared between the models. However, the information loss occurs with the orange diamond and green triangle elements which are not represented in the other model.

Intuitively, we want our bx to propagate any changes that appear in the abstract model back to the concrete model without losing information in the process. To do that, we need our lenses to be well-behaved, or abiding to the properties of correctness and hippocraticness. This is achieved for lenses via the following *round-tripping* laws for all $s \in S$ and $t \in T$:

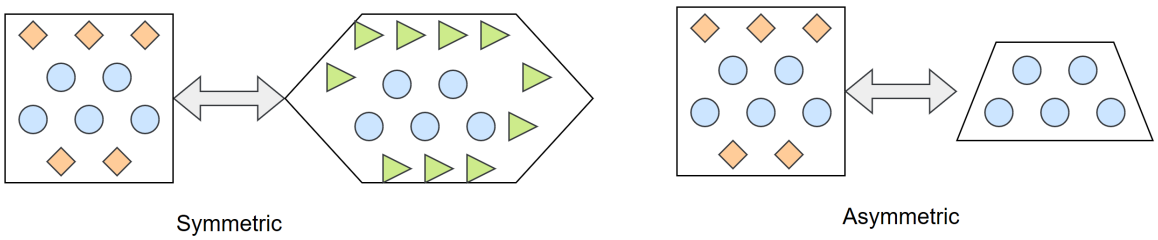


Figure 2.7: Difference between symmetric (left) and asymmetric (right) models

$$l.put\ (l.get\ s)\ s = s \qquad \text{(GetPut)}$$

$$l.get\ (l.put\ t\ s) = t \qquad \text{(PutGet)}$$

$$l.get\ (l.create\ t) = t \qquad \text{(CreateGet)}$$

Essentially, the round-tripping laws stipulate that information can pass through both transformation directions in such a way that information is still preserved. The GetPut law requires the put function to restore any data discarded by the get function. Laws PutGet and CreateGet force all propagated data to be reflected back onto the original concrete outputs. These two laws are important because they ensure that any updates to t will get back to s in the right state. These three laws lay the groundwork for tackling the classical view-update problem by reliably propagating any changes back to the source [45].

Composition

Of course, a single lens l will only go so far with complex model transformations. While it is possible to design a single lens which handles all aspects of the complex model transformations, it is in practice very difficult to maintain and verify correctness. Designing a bx as a monolithic lens will ultimately make the lens itself hard to debug and trace. Instead, if we consider functional composition, or the chaining of functions, we can make our lenses much simpler while maintaining overall transformation functionality [44].

In order to properly compose a lens, its get and put components each need to be composed together [44]. While chaining functions with one argument is a relatively straight-forward process, the difficulty lies in handling functions which require two arguments. Let lenses $l, k \in L$ and both lenses enforce relationship R between source model S and target model T. Suppose we want to create a new lens by composing

the first lens l with the second lens k, where $s \in S$ and $t \in T$. We can do this with the following compose function [46]:

$$compose(l,k) = \begin{cases} get(s) & = k.get(l.get(s)) \\ put(s,t) & = l.put(k.put(s,\ l.get(s)),\ t) \\ create(t) & = l.create(k.create(t)) \end{cases} \quad \text{(Compose)}$$

The composed get function is straightforward: call l's get function and then pass its result into k's get function. The composed put function is more complicated because it requires two arguments. We resolve this by calling l's get function to get an intermediate result that is compatible with k's put function [47]. Finally, the composed create function is straightforward like the composed get: call k's create function first and then pass its result into l's create function.

With the ability to compose lenses, we have the liberty of designing simple lenses which do one specific, well-defined transformation. A complex transformation will always have multiple steps that need to be performed. Instead of a monolithic lens function which handles all these steps, we can instead apply each step to a discrete lens. Chaining all these simple lenses together allows us to recreate the same complex transformation behavior as the monolithic lens. This addition of modularity also makes it easier to change or permute the overall transformation behavior by adding and subtracting lenses from the overall composition [44].

Chapter 3

Related Work

With the foundations covered in the previous chapter, we can now investigate the multiple approaches to bx in multiple domains, including Triple Graph Grammars, Programming Languages and Databases. This chapter will compare and contrast the work done by the bx community, exploring what aspects each of the approaches have in common and what their approaches mainly address. We also provide commentary on how viable these different approaches are for our medical record bx problem in OSCAR.

3.1 Bidirectional Transformations

Research on bx spans many disciplines including programming languages, database management, model-driven engineering, and graph transformations [1]. While each domain has the general notion of what a bx should do between two sources of information A and B, the details describing how each approach implements bx varies significantly. In many cases, one of the transformation directions such as A to B will dominate over the other direction B to A. Usually the dominant transformation will be considered the forward transformation while the other one is referred to as the backward or reverse transformation [1].

Much of current bx research focuses on implementing compatible forwards and backwards transformations within some form of unidirectional transformation language. However, there are approaches which try to specify both directions simultaneously. The eXtensible Markup Language (XML) has been a target for bx research because it is a good machine processable data exchange format while maintaining human readability.

Languages such as BiXid [48] and XSugar [49] consist of pairs of intertwined grammars, where the transformations are derived off the rules of one grammar and reformatted according to another set of rules [1].

BiXid uses a relation-based programming style with regular expression patterns in order to execute XML to XML transformations [48]. While regular expression patterns have also been used in the unidirectional XML transformation language XDuce [50], BiXid adds to that by using non-linear variables to make it easy to capture repeated substructures. The non-linear variables help address the issue when one of the models imposes a restriction on element ordering (i.e. folders must always be before elements) [48]. Effectively the regular expressions match and translate the local structures, while the non-linear variables allows for reordering.

XSugar is similar to BiXid as it also uses a relation-based programming style like BiXid. However, XSugar focuses on transformations between text and XML, specifically those with data structures of sufficient similarity [49]. By limiting the transformations to similar structures, it simplifies the grammars needed to parse the structures, and can guarantee that round-trip transformations can yield the original document. Unfortunately, this also means that this approach is limited because it is unable to deal with certain forms of ambiguity such as element reordering. XSugar leverages static analysis in order to guarantee the validity of the resulting XML.

BiFluX is inspired from a functional XML update language called Flux [51] and uses pattern matching. Their approach focuses on having developers write backward transformations specifying how a view can be used to update the source instead of writing a standard forward XML query [52]. By focusing on the put transformation first, it forces the developer to enumerate and ensure that view data can be transformed back to the source. Since there is a unique query from view to source, it offers bidirectionality by allowing view updates to propagate back to the XML database. However, this approach assumes an asymmetric transformation, or that that the view is always smaller than the source.

Since we wish to transform patient information from a database to E2E, a form of XML document, the above approaches at first glance appears promising. While BiXid, XSugar and BiFluX each have their own ways of approaching XML transformations, they ultimately do not cover the needs of our patient transformation workflow. This is because patient record data is relatively complicated and may contain multiple references which need to be preserved.

BiXid's non-linear variable approach can help with order independence, but their

approach is limited to XML to XML transformations. XSugar transforms from text to XML, but lacks the ability to reorder elements. As well, our database is not in text format, meaning we would require yet another transformation before XSugar could even begin to work. Finally, while BiFluX emphasizes the put transformation, we find that transforming between the database and E2E is a symmetric transformation due to patient record complexity.

3.1.1 Triple Graph Grammar Approaches

As previously discussed, Triple Graph Grammars were originally introduced by Andy Schürr [40]. TGG schemas and rules come together to specify how a transformation should behave when certain patterns are matched. TGGs are symmetric transformations because they affect the source, target and correspondence models of the TGG schema at the same time [53]. This sets up TGG as a promising approach to tackle our patient record transformation problem.

Current TGG research can be classified into five directions: expressiveness, concurrency, scalability, reliability and tolerance [53]. Expressiveness deals with the accuracy of capturing real-world consistency relationships. Some tools that deal with expressiveness are Negative Application Conditions (NACs), complex attribute conditions and multi-amalgamation. Concurrency handles the operational scenarios of TGG, focusing on maintaining consistency and synchronization while multiple TGG rules are being executed either simultaneously or in a batch.

Scalability is the measurement of the practical applicability of TGGs. A scalable TGG would have a polynomial run-time for its model size. Reliability focuses on the formal properties of bx which can be guaranteed such as transformation correctness and completeness. Finally, tolerance involves the degree of which TGGs can handle inconsistent model and/or delta inputs. A high tolerance TGG will be capable of handling unexpected inputs and yield a consistent transformation result.

NACs are well known from the graph transformation domain, and can be used to prevent rules from being applied twice to the same pattern. However, while NACs can increase a TGG's expressiveness, using NACs for more than just maintaining the schema can increase pattern matching complexity at the cost of the TGG's scalability [54]. Because of this, there have been some attempts to implement TGG with scalability in mind, omitting NAC support in the process [55]. Either way, NACs allow TGGs to have some control over rule application and may sometimes be worth

the cost in scalability.

While manipulating the TGG schema graph should be straightforward with rule application, the TGG must still ensure consistency of attribute values within the nodes of the graph in order to be practical. Algebraic graph transformations are better at structural manipulation than they are at attribute-relationship manipulations [56], which remains a long-standing problem for TGG research. Since patient record transformation focuses primarily on attributes and values instead of the structure, we need TGGs to have a reliable way to control how attributes are transformed.

One way attribute transformations can be done is with symbolic attribute handling. TGGs are extended to handle the relationships of attributes between the source and target with first-order functions [57]. This provides us with attribute labels which can uniquely identify if the attribute resides in the left-hand side (LHS) or right-hand side (RHS) of the rule [58]. While this approach accounts for attributes in a TGG rule, a more complex constraint solver is required to execute it. This approach can ensure the preservation of consistency, but does not guarantee completeness, or the idea that the best attribute assignment is being done.

Another approach for attribute transformations is by treating TGG rules as an "interface", where the attributes are handled with manually implemented operations inside the rule [55]. This idea can be extended further by treating these operations as a "black box". Inside the black box there are atomic constraint operations which can be mixed and combined to generate the complex constraint behavior desired for transformations [59]. The black box approach opens the door to combining the structural benefits of TGG transformation with other bx approaches with different sets of strengths.

One of the strengths of TGGs is the ability to simultaneously visualize the model triples. This property can be applied to creating generated test models which ensure correctness and can be inspected from a high-level [60]. A high-level software automation could leverage TGGs as a test case generator and a test oracle. However, while this is a feasible approach, they found that the correspondence link construction was a slower oracle than model differencing approximation oracles. As with any automation, it runs the risk of only covering certain aspects of a transformation and may miss some edge cases that need testing.

TGGs have also been used to generate a full triple from just a source model in the forward transformation, and the backward transformation has a similar derivation in reverse. This type of derivation is called a "batch" transformation because everything

other than the input is created from scratch [53]. While this TGG approach does have its uses, it is difficult to argue why this approach is better than a pair of unidirectional model transformations, especially when the latter is generally more efficient and easier to implement [61].

Ultimately, a framework's scalability with large datasets is what makes it practical or not. While graph pattern matching techniques are relatively efficient [62], the act of graph parsing is still exponential in complexity. Because of this, current TGGs are usually restricted in some way in order to minimize complexity and ensure some degree of scaling [63]. In practice, frameworks trade off some degree of expressiveness such as restricting the usage of NACs in order to maintain a polynomial runtime [53].

For TGGs to be considered reliable, the transformation must be considered correct and complete. A transformation is correct if it generates only consistent triples, and a transformation is complete if it is total on a precisely defined set of valid inputs [53]. These two properties tie in directly to the bx properties of well-behavedness because correctness ensures that the results stay consistent with each other, while completeness ensures that defined valid inputs can be transformed correctly. In practice, completeness makes the TGG framework throw an exception whenever it encounters an invalid input. This type of behavior makes the transformation reliable because we know if there is a resulting triple, it must be consistent by definition.

Of course, a hard-line view of reliability can be restrictive in real-world datasets. Some degree of tolerance should be allowed in order to bring slightly inconsistent models to alignment. However, tolerance treads a fine line because it may go against the well-behavedness property of hippocraticness. While it could improve the consistency of an input to a certain degree, extreme cases can end up deleting everything that is not considered to be a part of the consistent relation [64].

Tolerance is a new challenge in TGG research, and there does not exist a method of defining tolerance in current formal frameworks. This means that for now, tolerance must be dealt with on a case-to-case basis with domain experts specifying what is a safe and correct transformation. TGGs offer a concrete way of specifying how a bx transformation should proceed, and results in a triple graph structure which we know is consistent. While TGGs can do well on structure manipulation, attribute transformations still require more complicated constraint solvers. Since E2E is attribute and value centric, we still require an approach which focuses primarily on attribute transformations.

3.1.2 Programming Language Approaches

The programming languages research community classifies bx languages into two features: *semantic laws* and *mechanisms*. Bx languages can either follow semantic laws, or rules which define the permissible behavior of a program, or are built with mechanisms which facilitate programs to be bidirectional [1]. Based on this classification, a lens would be classified as a mechanism because lenses are constructs which contain both a get and put transformation.

Programming languages which focus on semantic laws can be further split into two types: bijective languages and bidirectional languages. Bijective languages ensure that every forward transformation is an injective function and the backwards transformation is an inverse to the corresponding forward transformation [1]. On the other hand, bidirectional languages can have an arbitrary forward transformation which will likely discard some information. The backwards transformation generally compensates for this by taking in both the updated output and the original input in order to restore the discarded information and merge in any updates.

While abstracting and decoupling datatype implementations from pattern matching interfaces increases the modularity of a program, it is difficult for developers to create and maintain the conversion functions between them. One bijective approach uses a design of views based off of a right-invertible language in Haskell [65]. Their approach uses a combinator-based language built from hard-wired primitives which have guarantees of consistency. Only one of the conversion functions needs to be written and maintained as their language guarantees a consistent, injective and inverse conversion function.

An alternative bijective language approach uses reversible models of computation, where every operation is invertible. Janus is a high-level imperative language which focuses on backwards and forwards determinism and reversible data updates [66]. The language uses a C-like syntax that requires all assignments and control constructs to be reversible, and the inverse semantics are accessed through "uncalling" procedures, or a procedure that executes in reverse. Janus' localised invertible operation approach avoids the conventional need for global analysis when reversing an entire program.

Unlike bijective languages, bidirectional languages focus less on generating injective functions and instead emphasizes ways to restore and manage discarded information. We can make any arbitrary function become injective if the discarded information from the forward transformation is included within its scope [1]. Typically this set of

discarded data is called the complement. The complement can be handled in many ways, whether it stays with the source, gets transmitted to the target along with the transformed content, or stored in a separate location elsewhere.

Complements also have applications in simple functional languages because they can be utilized to generate a corresponding backwards transformation for the forward transformation [67]. Given that a program is written in λ-calculus notation, injectivity analysis can be used to produce the backwards transformation. This approach can be extended with tupling transformations, a technique which allows the injectivity analysis to handle duplicate input parts and still yield an appropriate backwards transformation [68].

Higher-order functions can be used in order to yield a corresponding backward transformation result [69]. The polymorphic forward transformation does not require a special domain-specific language and is used as an argument for the higher-order function. The higher-order function is written in Haskell and relies purely on semantic values, using relational parametricity [70] and free theorems [71] for proving consistency. Unfortunately, while this approach is functional, its generated put functions were orders of magnitude slower than their equivalent hand-written put functions.

Another way of thinking about bx is through the use of lenses and combinators. The idea of lenses, or a pair of forwards and backwards transformations, were introduced by Nate Foster [47]. While just calling a pair of transformations a lens does not do much by itself, the ability for multiple lenses to be combined together allows for a wider range of transformation expressiveness [47]. Since lenses are "well-behaved" by nature, we have certain guarantees on consistency between the two models. The added ability to combine lenses together builds up more complex transformation behavior which is also "well-behaved".

One practical application of lens combinations is with the Boomerang language. Boomerang is a bx language with a focus on string lens combination [72]. While string transformations can use pattern matching through regular expressions, there may be instances where complex transformations are order dependent. Boomerang allows for execution order provenance, ensuring that certain operations are done in a certain order to preserve consistency.

While Boomerang is a good example of what a lens-based bx approach can be, it is limited to transforming strings and string types. Boomerang in general assumes everything is of the string type, which allows it to proceed, but as a result lacks the ability to handle multiple types of data that is commonly seen in mainstream

programming languages such as C and Java. As well, Boomerang is implemented as a non-deterministic finite automaton (NFA), meaning that while it has good enough performance in most realistic cases, it can get stuck in certain cases. Converting Boomerang to a deterministic finite automaton (DFA) can resolve this problem, but will result in the program taking orders of magnitude more memory.

While lenses themselves are an attractive method of implementing bx, generally the strong behavioral laws they offer can be a bit too strong. For example, unimportant details such as whitespace differences and indentation should not break a correspondence even when the pertinent data is shown to be consistent. The general theory of quotient lenses focuses on relaxing these strict definitions, instead focusing on canonical forms for transformation [44]. The quotient lens approach retains the consistency benefits of normal lenses by reducing models down to a canonical format before the transformation passes through the lens. Maintaining a bx on a canonical form ensures consistency while ignoring minor permutations which do not break the consistency of the data.

The key ideas we can take from the programming language approaches is the lens combinator approach. It provides a way to build small and modular transformation lenses that can be combined together to create more complex behavior. while TGGs are great for structure manipulation, lenses can handle the attributes within those structures. In a way, lenses could even be used in some of the TGG "black box" approaches as a way to address the medical record bx problem.

3.1.3 Database Approaches

The database research community also has a large amount of literature dealing with the bx problem. Since their core unit of data transformation revolves around queries, research has been focused constraining query design. There has been some discussion about what parts of updatable views should remain invariant, or not be changed even when there are incoming updates. There is a *constant-complement* property which states that any data which is not referenced by a view should remain constant or invariant [26]. As well, any base dataset should be considered unambiguously updatable in order to limit the potential for inconsistent information.

The classical view-update problem in database literature occurs when modifications to a database view may not uniquely correspond to the original database, eventually yielding inconsistencies. To achieve some degree of consistency, implementing some

form of "update policy" is required, whether it involves co-evolving schemas as the data changes, or specifying a formal method of updating both the original database and its associated view. There are two main methods of updating: operation based (done in the query operation itself) or instance based (a full evaluation of the database model for consistency).

One operational based approach involves using a relational lens query language where the forward query is treated as a view definition and the reverse query is treated as an update policy [29]. The language consists of primitives based off of standard relational operations such as select, join and project and type checking. Consistency is maintained with the usage of well-crafted functional dependencies. Unfortunately, this approach falls short on databases with foreign key dependencies or inclusion dependencies.

Transformations in databases can be classified into global as view (GAV), or local as view (LAV) depending on the view's scope [73]. A bx approach called both as view (BAV) integrates both GAV and LAV together to yield a unifying framework using reversible transformation sequences [74]. Just like relational lenses, BAV focuses on creating a set of atomic query transformations known to be bidirectional as a building block for more complex transformations. The BAV approach allows for scalable schema evolution by ensuring every operation is treated as a reversible delta change.

One instance based approach leverages the Microsoft Entity Framework (EF), an object-relational mapping tool, where a developer specifies a declarative mapping between an entity-relationship model and a relational database [75]. The specification then compiles into a pair of views (one for querying, one for updating) and yields database operations which correspond to the specified model operations. This approach has been used commercially, helping developers have a consistent view of the database and the operations they apply to it. The largest challenge with this approach was addressing its sensitivity to the underlying schema state and the chosen mapping language.

Another instance based approach is the GUi As View framework (Guava), a system which presents a user interface to an updatable view of a database model [76]. Guava uses a forms-based UI to create a conceptual model of a selected view, generating a query language based off of relational algebra. The resulting transformation channel can automatically transform actions from the user interface into usable database queries, and can also automatically generate software artifacts which can reduce the probability of software errors as compared to writing these artifacts manually.

Looking back on our medical record exchange problem, we do involve some form of relational database in our overall data flow. However, OSCAR already has software in place such as the JPA and Hibernate frameworks which manages the flow of data to and from the database in a predictable manner. While the bx database approaches do not directly satisfy any of our direct needs in our problem, they do illustrate how bx can be approached in multiple ways, whether it be between a visual interface, between differing views, or even between each query operation.

Chapter 4

BXE2E: An Overview

In this chapter we introduce BXE2E, a bidirectional approach for generating and consuming E2E documents for the OSCAR EMR environment. BXE2E is a proof-of-concept framework which incorporates bidirectional transformation principles and allows OSCAR to both import and export E2E document records with guarantees of well-behavedness. It refines the traditionally separate import and export processes by tightly binding them together with a modular design to emphasize discrete data element transformations.

BXE2E is designed to deliver the features and goals discussed in section 1.3 for data model transformations. Additionally, the BXE2E framework is built up from small modular components in order to make it easily extensible. Its novel design technique can be applied to other practical model transformations that can benefit from the well-behavedness properties.

4.1 Architecture Overview

The import and export process for exchanging patient records in an electronic medical record system is inherently complex. Relevant data constituting a singular patient record may come from multiple locations within the EMR and must first be aggregated before any transformation can occur. Additionally, the final exchange format for the import and export must be well defined with respect to the types of data that are represented in it. As a result, an import and export function can contain multiple intermediary transformation steps in order to achieve its end goal. These intermediary steps add to the difficulty of properly creating and maintaining transformation features.

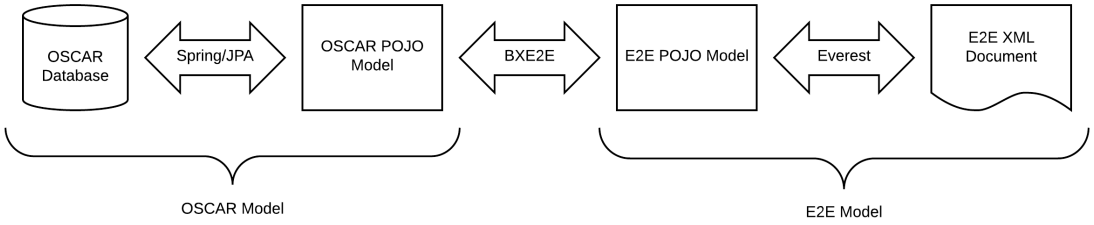


Figure 4.1: A high-level overview of the data flow steps in the import / export process

The OSCAR data model uses a relatively simple Plain Old Java Object (POJO) representation for its database tables. However, the E2E document model is handled by the MARC-HI Everest Framework library due to the complexity of HL7v3 data structures [18]. Everest provides a consistent framework to represent HL7v3 data structures such as E2E with native Java patterns, objects and paradigms. The Everest framework allows for a methodical and reliable construction and serialization of the E2E document.

BXE2E is a bidirectional transformation framework that relates the OSCAR data model and the E2E document model of a patient record together. The BXE2E framework is effectively a POJO model transformer since the OSCAR model and E2E model are both represented as POJOs. With both models being represented via POJOs, it makes transformations between the models much easier to perform. Of course, a full import and export function does not rely only on POJO transformations. As shown in Figure 4.1, there are sub-transformations that convert between the POJO models to their respective OSCAR data models and E2E XML models.

Although BXE2E is just one part in the import and export process chain, it plays a vital role in translating concepts between the two main differing model domains. The sub-transformations around BXE2E simplify and group relevant data elements together, making it easier to map between the two different patient record models. These sub-transformations effectively provide an alternate way of viewing their data structures, whereas BXE2E transforms by defining and enforcing a mapping between the two models.

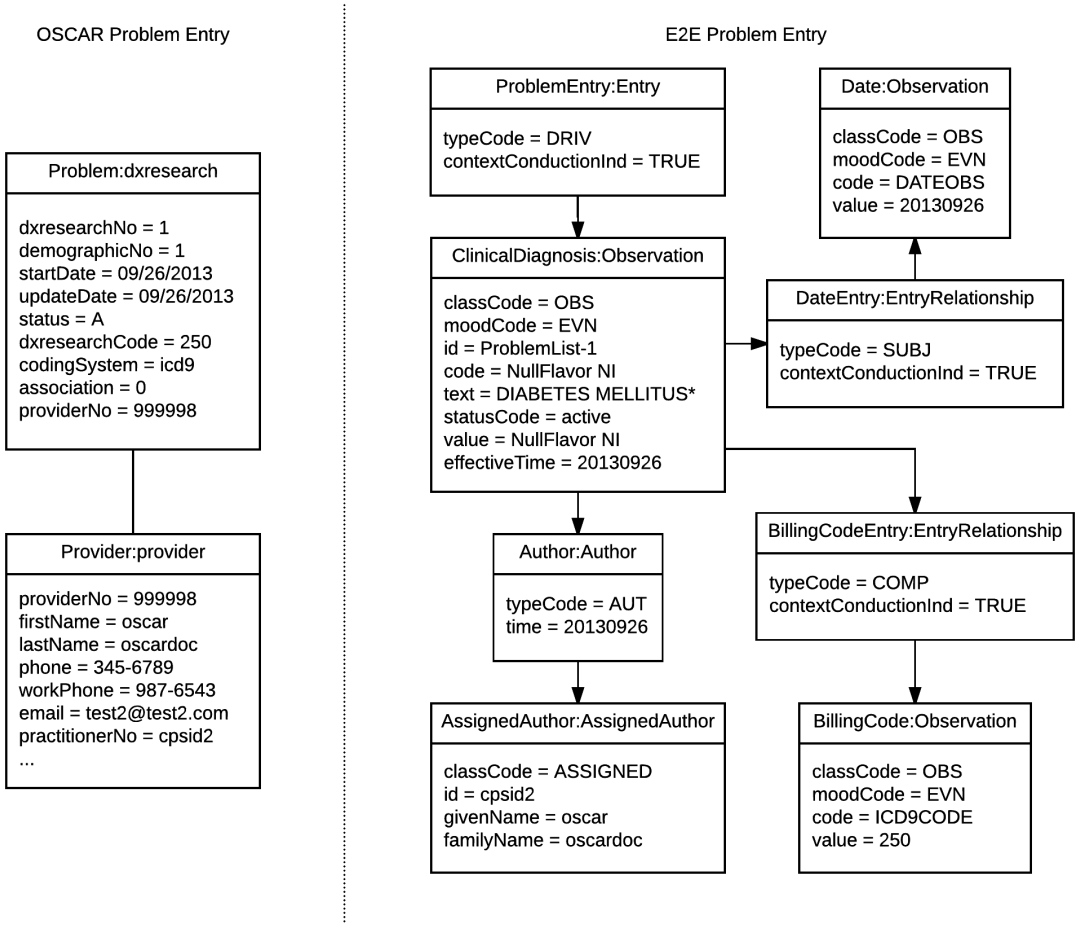


Figure 4.2: An example data representation of a Diabetes problem entry in both OSCAR and E2E models

4.1.1 BXE2E Design

Creating a mapping between two potentially very different data models is not a trivial task. Most of the effort is concentrated on understanding where the two data model elements either directly overlap, are related but require translation, or are unrepresentable in the other model. An understanding of how each data element fits within their larger model as well as the expected output and behavior between models is required before any proper transformation implementation can begin.

Suppose there is a patient record with a recorded problem or condition such as Diabetes. In the OSCAR EMR model, that information would be encoded as an ICD9 code of 250. That problem would be stored as an entry in its problem table, and it

would also contain other pieces of data such as when it was recorded, by whom, and its current status on the patient. On the E2E model, it contains a section called the Problem List which stores a list of observation entries. The data from OSCAR would need to be properly mapped over to the E2E observation entry in order to preserve not only the main coded element, but also any of the supporting data elements as shown in Figure 4.2.

Given an understanding of how the mapping between the two models should behave, we can implement a transformation function between the two models. Data model transformations are applied whenever data values need to be converted from their source format to the destination format. Creating a forward transformation in one direction is generally straight-forward. However, the process becomes significantly more difficult if there is a requirement to return from the destination back to the source, or a backwards transformation. The ability to tightly bind the behavior of both transformations becomes extremely valuable.

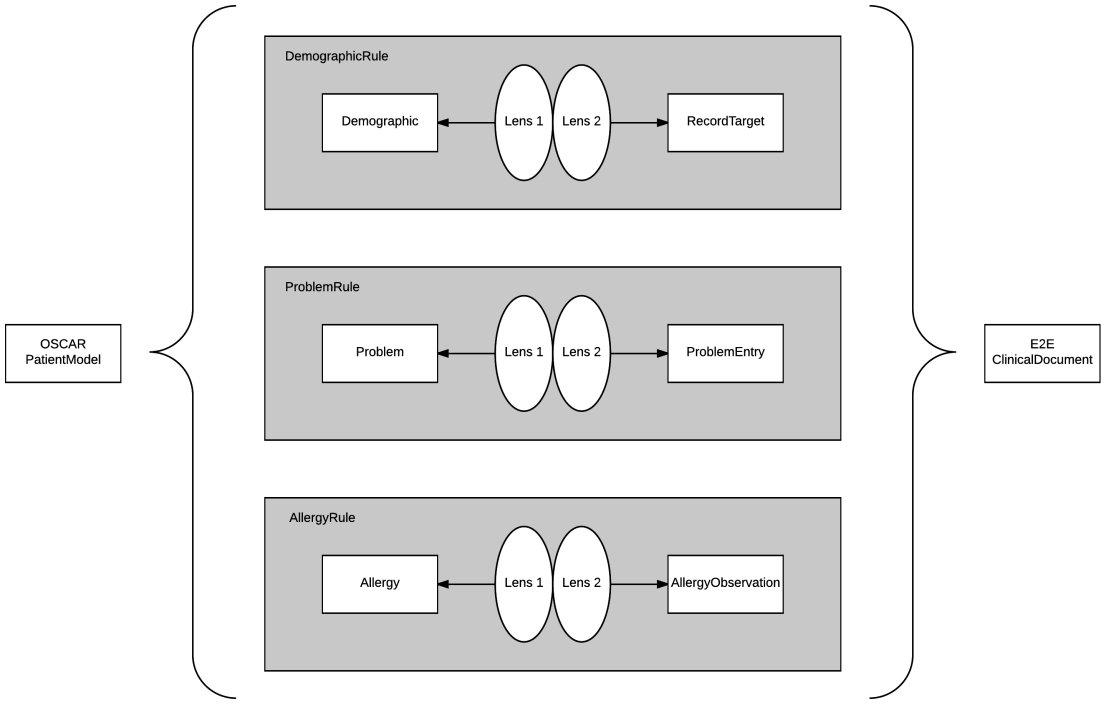


Figure 4.3: An overview of how TGG Rules and Lenses are organized in BXE2E

Taking inspiration from multiple facets of current bx theory, BXE2E utilizes a combination of Triple Graph Grammar Rules and Lens theory in order to enforce well-behavedness. Triple Graph Grammars have two main desirable properties, namely their

non-deterministic rule execution and their explicit representation of a correspondence between the source and target objects. Lens theory is desirable because a lens by definition is a pair of forwards and backwards transformations. As well, lens theory has an emphasis on functional programming concepts, allowing for modular lens composition.

BXE2E combines aspects of both bx theories and applies them to our design of the transformation functions as shown in Figure 4.3. Since a patient record is made up of multiple subcategories of information, we can define a TGG rule for each subcategory. This allows us to construct an explicit correspondence model and lets us take advantage of the non-deterministic execution property of TGGs. We then enforce the correspondence with lenses, making each rule have a tightly packaged forwards and backwards transformation. With modular lens composition, we can easily swap in and out different lenses based on our requirements.

4.1.2 Considerations

In order for BXE2E to be successful, it needs to address the three main factors discussed in Section 1.3. As with any health-related application, we must avoid any potential patient safety issue by ensuring that the data transformation is correct. BXE2E's code organization must be easy enough to follow that any developer can pick it up and understand how to use and extend the framework. Lastly, BXE2E must also be designed with scalability and performance in mind.

Correctness and Safety

In Panama City, there was an incident where twenty-three of the twenty-eight patients died from medical radiation overexposure [77]. The investigation afterwards discovered that patients received twice the dosage of radiation due to an error in dosage calculation. Although the calculation code was unit tested, the error came from an interaction between two different functions [78]. Had the code been integration tested properly, this error would have been caught before patient safety was compromised.

Since Electronic Medical Record systems are being used as data repositories for patient health, we have to make sure that the system is both storing and representing accurate data. Although there are many facets regarding how providers interact with the EMR and the potential safety hazards that may arise from said interaction, that aspect of safety is outside the scope of this thesis. Instead, our focus emphasizes

correctness, or the notion that any derivative data structure is properly representing the original source data. Since we are transforming this data, we must strongly emphasize the notion of preserving the original meaning of the data and minimize any potential avenues for data misinterpretation.

Of course, this is not an easy task. A piece of data can have certain degrees of correctness simply due to the way it is encoded. Falling back to our Diabetes example, we know that code 250 represents Diabetes, but ICD9 could encode more information such as 250.1 which means Diabetes with ketoacidosis. If not properly considered during all parts of the transformation, the 250.1 code encoding something more specific than just Diabetes could be omitted, resulting in a loss of data. Although it is still correct in representing Diabetes, ketoacidosis is no longer mentioned. This is an example of a capacity reducing transformation.

Each transformation has a certain degree of information capacity preservation. Depending on the transformation behavior, they can be categorized as either capacity preserving, capacity augmenting, or capacity reducing [79]. Capacity preserving transformations are reversible for both the forwards and inverse direction, capacity augmenting transformations are only reversible, while capacity reducing transformations are not reversible at all. When developers can understand how each transformation falls into one of the three categories, it makes them more aware about the potential safety implications that could arise.

Ultimately, any developer working on health record transformations will require a significant degree of domain knowledge about how the data is handled and represented in order to minimize the amount of correctness that could be lost. While the act of preserving correctness is very important in this domain, our thesis focuses more on the software design of the transformation with the assumption that the health domain knowledge is utilized to its fullest to minimize potential correctness loss.

Maintainability

As with any active software project, it needs to be maintained over time. The readability of code plays a major factor as to whether a software project can easily be understood and contributed to by new developers. One way of quantifying this is with Cyclomatic Complexity which counts the number of linearly independent paths the code may take [80]. Essentially, it encourages designing software such that each module or section does not have too many potential execution paths.

Applying both Lens theory and TGG rules can minimize the cyclomatic complexity of the codebase. Lenses inherently have low complexity and very predictable behavior because they come from the functional programming domain. They also have the added advantage of having co-located forwards and backwards transformation functions. Lens composition can combine simple lenses together for building up more complex transformation behavior without compromising readability. TGG rules aid in minimizing complexity by limiting each rule's context to one independent and well defined topic. By grouping only related data together, each rule has a well defined purpose and works independently of other rules.

Performance

Production EMRs will generally store thousands of patient records or more depending on the size of the practice. As a result, there will be use cases where providers may need to batch transform a large number of their patient records. In order for the BXE2E framework to be successful, it needs to accommodate use cases which require a high degree of scalability. Once again leveraging the modular properties of lenses and non-deterministic execution property of TGG rules, we can tightly bound the units of transformation work into independent parts.

This high degree of independence allows the framework to work efficiently and allow for concurrent operation. Although BXE2E is designed to be easily parallelizable, this is outside the thesis scope because true concurrency is difficult to measure and quantify without explicitly designing a scheduler or manager to handle the fork-join process manually. As well, the inherent nature of thread scheduling is not always deterministic, which can make it significantly more difficult to measure the framework's impact.

This framework also natively uses Java as the programming language in part because the OSCAR EMR we are modelling after is written in that language. Although other languages were considered, ultimately there is no equivalent HL7 toolset similar to the Everest framework that could fit well in OSCAR. Using Java as the programming language to apply bx theory only works on the JavaSE 1.8 API because of the introduction of lambda functions which are heavily utilized in the BXE2E framework.

One of the big departures from current implementations of lens theory is that this framework uses objects as the transformation medium instead of strings [44]. Although it is possible to leverage existing string bx frameworks by serializing and coercing objects to strings and recasting them afterwards, this obviously adds unnecessary

overhead to a performance-sensitive environment. However, the techniques applied to string-based bx transformations can be extended upon to yield similar well-behavedness properties as we will demonstrate in BXE2E.

Chapter 5

BXE2E: Design and Implementation

The previous chapter introduced the BXE2E framework and how it addresses the three requirements discussed in Section 1.3. In this chapter we examine BXE2E in greater detail. We first cover algorithms of each component of BXE2E, including discussion on extensibility and alternative design approaches. We then look at the software environment that BXE2E is built in, and provide context to how it influences BXE2E's architecture. Finally, we explore the implementation details by providing a software walkthrough and extension example for the BXE2E framework.

5.1 Algorithm Design

As was shown in Figure 4.3, BXE2E consists of one Transformer (represented as the giant brackets), and multiple independent rules, of which each rule encapsulates a set of rule-specific lenses. This provides us with two degrees of permutation: one at the rule level, and one at the rule execution level via the defined lenses. The two levels of permutation allow the developer to quickly swap modular features around to meet requirements.

Although multiple transformers can exist in our framework, each transformer is custom tailored to satisfy a specific transformation requirement. In our example proof-of-concept code, we implemented E2EConversionTransformer in order to handle E2E's EMR Conversion use case. Another transformer could be written to handle a different requirement such as E2E's Chart Transferal use case.

5.1.1 Transformer

The BXE2E transformer is the first entry point of our framework, serving as the first of the two main permutation points of our architecture. The transformer's main job is to facilitate the map and reduce of all the rules needed in the transformation of a patient record. All transformers follow the general pattern described in Algorithm 5.1. After determining whether the initial patient or clinical document is the original source, it starts the map step which parses the source for sections such as Demographics, Author, and Problems.

Algorithm 5.1 General Transformer Map-Reduce

```
1:  procedure TRANSFORMER(Patient, ClinicalDocument)
2:      Determine the original source
3:      for all Sections not assigned a rule do
4:          Assign rule to section
5:          Add rule to map list
6:      end for
7:      Map and execute all rules
8:      for all Rules in map list result do
9:          Retrieve transformed component from rule
10:         Reduce component into final result target
11:     end for
12:     return
13: end procedure
```

For each section that can be acted on, an appropriate rule is instantiated and added to the map's list of rules. This step continues until all of the sections have been assigned into a rule. Although the algorithm represents the map as a loop, in reality this step is more deterministic and linear because we can anticipate what types of sections will be present in a medical record and where to look for them.

Once all sections are assigned a rule, the map executes all rules in the list as seen in Figure 5.6. This step may be done in parallel. Afterwards, the reduce loop inspects each rule and retrieves the transformed components. Each component is then applied into the final result target location. This step repeats until all rules have been inspected and the final transformation target is complete. At this point, the transformer returns containing a synchronized source and target medical record.

5.1.2 Rule

We use Triple Graph Grammars in our design to leverage its property of non-deterministic rule execution. This allows us to not worry about when a certain section entry is transformed at all, as long as it is done at some point. However, this is only useful if the final overall transformation result still yields a deterministic output. In the Triple Graph Grammar domain, there is the notion of *confluence*, or the property where the rule application order does not affect the final output state [43]. We provide more background about the properties of confluence in Section 2.5.2.

Each of our TGG rules are specifically designed to handle one top-level topic or entry per rule. The main reason we use TGGs in our design is because they are a great construct for containing context. Rules offer a succinct way of representing a direct relationship between a source and target model object. By dividing up our entire patient record transformation into sensible rules, we bound the complexity of the transformation into independent and easy to process chunks.

A *critical pair* is a pairing of two potentially conflicting rules that could fail to converge. When the results of a critical pair are combined and they yield the same answer regardless of the order of execution, they are said to *converge*. When we test for all possible critical pairs in our framework for convergence, we will find that they all do. This is because all of our rules are designed to operate independently. Since none of the rules in the framework interfere with each other, we have a confluent rule set. Knowing this, we can take advantage of the map-reduce design because as long as all the rules in our set execute at some point, they can all be joined back together without altering the final result.

The rule is the second entry point of our framework, serving as the second of the two main permutation points of our architecture. Lenses may be added or removed here depending on requirements. Each rule is context-specific to the section it manages, and contains a set of context-specific lenses that will perform the bidirectional transformation. The rule execution algorithm is quite simple as seen in Algorithm 5.2.

The rule creates an ImmutablePair of the Source and Target, and then defines the lenses that will compose together into the bidirectional transformation. When the rule is executed from the map operation, the rule will determine whether the source or target is the original, and either apply the lens get or lens put operation appropriately. After the rule execution, the rule will have an ImmutablePair containing the original and the final transformed result. In this state, the contents of that pair represent the

Algorithm 5.2 General Rule Execution

```
1: procedure RULE(Source, Target)
2:     Generate and populate ImmutablePair
3:     Define and compose all lenses
4:     if Original is Source then
5:         Get new ImmutablePair from lens
6:     else if Original is Target then
7:         Put new ImmutablePair from lens
8:     end if
9:     return
10: end procedure
```

same information. The transformer can parse this pair in order to acquire the result component.

5.1.3 Lens

Although Map-Reduce manages what needs to be transformed, and the TGG rules handle the transformation context, the Lens is the workhorse of the transformation. A lens by definition is a pair of forwards and backwards transformations. We leverage the following three key principles of functional programming for creating and utilizing bidirectional lenses:

1. **Higher Order Functions** - Drawing from the lambda calculus domain, HOFs are capable of taking in one or more functions as arguments and returning functions as results.

2. **Pure Functions** - Functions are considered pure if and only if they have no side effects on memory or I/O. Another way to put this is that pure functions are considered stateless.

3. **Immutable Data Structures** - Immutable data structures are more thread-safe than mutable data structures. This is a critical principle that can yield more performance via parallelization.

Higher Order functions are useful in many ways. An example of a HOF is the *map* function we used higher up the BXE2E chain. In a map, we can apply a function f to each element of a list, and get a resulting list of elements with f applied. However, the properties of HOF are not strictly limited to just maps. We utilize HOFs in our

lenses by *composing* them together in order to create more complex transformation behavior.

Functional composition is useful because it enables the programmer to write simple, easier to understand functions. Instead of being forced to have a single lens which defines complex behavior, it is possible to break the process down to multiple simple lenses. By deliberately creating a lens for only one data element, it makes the transformation easy to comprehend and yields predictable behavior. As shown in 5.8, we can compose these simple lenses together for more complex transformation behavior without compromising readability.

Lenses need to be built off of pure functions because any form of side-effects will compromise its ability to yield predictable behavior. Composing lenses with side effects will likely lead to unexpected results, and the errors that appear will be difficult to debug. To that end, we avoid that problem by ensuring all our lens functions are stateless. Instead, all of our lenses specify exactly what form of output should appear given an expected input. The lens functions themselves are pure and do not know the current state of the transformation.

Each lens defines and enforces the transformation of specific data elements by utilizing a lambda function for each transformation direction. As shown in Figure 5.10, our lens behaviors for both the get and put are defined in the constructor. All of our lenses extend the AbstractLens class, which contains the composition function and other helper functions for lenses.

Algorithm 5.3 General Lens Function Structure

```
1: procedure LENS(Source, Target)
2:     Find the data element(s) to transform
3:     if Destination element(s) do not exist then
4:         Do element transformation
5:     end if
6:     Assign transformed data element to destination
7:     return New ImmutablePair
8: end procedure
```

Each transformation function needs to follow a general format in order to prevent potential errors. As shown in Algorithm 5.3, the lambda function takes in the source and target as parameters and immediately finds the relevant data elements from the data structure. Then there is a general check on the source and target to prevent overwriting existing data in the destination. The transformation then occurs, and

then the transformed element is applied to the destination data structure. Finally, a new ImmutablePair is generated and returned containing the changes.

5.1.4 Extending BXE2E

Extending the BXE2E framework is a simple process. For example, if a new developer wishes to add Alerts section support, the developer would first identify what data elements are part of the alerts mapping in both the source and target models. They would represent it as a TGG such as the one shown in Figure 5.1. The developer would then write up a new AlertsRule by emulating the existing design of other sections. With the mapping and TGG correspondences on hand, the developer would then write a lens and unit test for each of the six data elements that needs to be transformed in alerts.

All six of these lenses would then be composed with a general AlertsLens definition and applied as AlertsRule's correspondence similar to other existing rules. With a working AlertsRule in place, the developer would need to modify the map and reduce in the transformer to handle the new AlertsRule by wiring the source and target model parts into the rule. Once that is done, effectively the framework now supports the Alerts section and the rule will execute during the map stage.

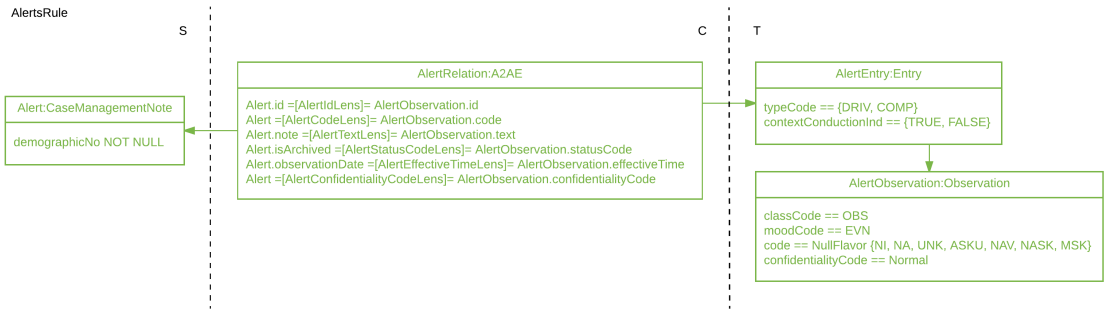


Figure 5.1: The TGG Rule for transforming an Alert Entry

Should a developer need a modified version of Alerts at some point, this is easy to implement as well. Say for example the Alerts section now requires an extra time element to be added to the transformation. To implement this, the developer would just need to write a new lens which exclusively handles the new time element, and then compose this new lens with the existing lens definition in AlertsRule. This modifies the AlertsRule transformation to handle the new time element in addition to the

existing transformation behavior.

5.1.5 Alternative Designs

BXE2E's design uses Map-Reduce, TGGs and Lens theory to implement bidirectional transformations. However, this design is a product of exploring existing techniques and discovering the shortcomings of using just one bx theory. The first option we explored was using a full lens approach. Full lens transformations have been done before, albeit only with strings [44]. We learned very quickly that since Java is a strongly-typed language, our biggest problem would be finding a common ground object type that could elegantly handle different model object types.

One option for full lenses was to coerce all POJO objects into a string representation and manipulate them as a string. However, we discarded this idea because the approach would not be easy to follow for developers. As well, not all POJO objects in our models would have a clear string representation, adding to the complexity and potential for error. The second option we considered was casting all our lens input / output types to be a generic Java Object. Although this would allow us to easily compose all our lenses together, we quickly ran into the issue of requiring a significant amount of boilerplate code checking for the object type and casting in many places. In effect, we were attempting to circumvent the strongly-typed nature of Java and ended up with code that was liable to crash or misbehave.

We learned from the two potential options that previous work was all based on strings, something we could not do in our situation. Since our data models were highly nested in nature, attempting to flatten out the nesting by casting to a similar data type would be error prone. Instead, we realized that we would have to track the context, or the how deep into the nesting the transformation would manage.

We then thought about the merits of a full TGG design. With well constructed rules, we could manage the context by making sure the rule only matched a specific pattern before executing. Of course, a full TGG approach with pattern matching would work, but with drawbacks in efficiency because it would have to re-parse the model for matches until everything was covered. We know the data structure of both the source and target, so the pattern-matching approach of TGG was discarded in favor of a more deterministic map-reduce.

Though we could handle the context with well defined TGG rules, we didn't have a well-formed method of enforcing the correspondences. Since TGGs handled the

context, and lenses were great at transformation operations, we decided to couple the two methods together, with the lenses acting as the correspondence for each TGG rule. By utilizing the strengths of both approaches and bringing them together, we had a solution to the problems we were facing.

The typing issue remained however, and we had to figure out a way for the lenses to be composable in a limited context. Since TGGs operate on an entire graph, or in our case both the source and target simultaneously, we needed our lenses to also do the same. By introducing a standard construct that could handle multiple types and couple them into a pair, we could solve our typing issue found in lenses as well as let our TGGs operate on both the source and target simultaneously. We eventually chose to use the Apache ImmutablePair to meet this need.

5.2 Environment Design

Fundamentally, BXE2E is designed to be an import / export function for the OSCAR EMR by leveraging the Everest framework. BXE2E is designed to be one of the many modules that co-exist in the OSCAR architecture. Since its use cases are straightforward and do not require the involvement of other OSCAR modules, BXE2E is effectively an independent transformation tool. BXE2E only requires patient data stored on the database and whatever data structure layers are involved in acquiring that patient data.

OSCAR itself is a full-fledged large scale software project. Though BXE2E is mainly designed not to be reliant on other OSCAR modules, the OSCAR environment is complex and contains many other modules. Other OSCAR modules could significantly impact the performance measurements of BXE2E should they run concurrently with BXE2E. In order to minimize the potential of unexpected behavior caused by other modules, we built up BXE2E in a mock OSCAR environment.

5.2.1 OSCAR

OSCAR is an Electronic Medical Record system which uses a traditional Client-Server architecture. Clients access the OSCAR server through the use of a web browser, typically with a computer in the clinic. There is usually one server that services all the clients in a clinic. The server is either physical and is located in the clinic, or is virtual and hosted / managed by a third party.

The client part of OSCAR utilizes Java Server Pages (JSP) to generate and render the front-end user interface for the client. These pages utilize a mixture of Java, Javascript and HTML to generate the client view. The client communicates with the server via an HTTP or HTTPS connection depending on how the server is configured. Although any modern web browser can handle the client side of OSCAR, Firefox is usually recommended because OSCAR has been tested mainly with Firefox.

The server is a Java EE application which runs on the Apache Tomcat platform. OSCAR utilizes a MySQL database on the server for data record persistence. The application connects to the MySQL database with a Java Database Connectivity (JDBC) connection. OSCAR manages this connection with the Spring and Hibernate frameworks. Within the application, each data table entry on the database is represented as a POJO facilitated by the Java Persistence API (JPA). These POJOs are retrieved and commited to the database through a Data Access Object (DAO) layer.

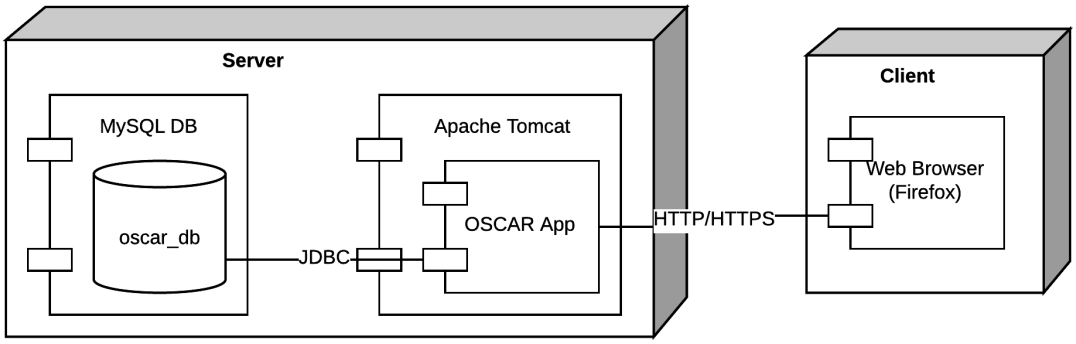


Figure 5.2: A high level overview of the Client Server model of OSCAR

The database schema for OSCAR contains few foreign key constraints and other schema-level enforcement. Instead, the enforcement is all done within the OSCAR web application. This means that OSCAR's DAO layer controls the behavior of the database table entries. The lack of schema enforcement at the database level makes it easier to generate and parse records because we can directly access the required tables. However, ensuring data correctness lies solely on how the transformation is implemented within the constraints of OSCAR's DAOs.

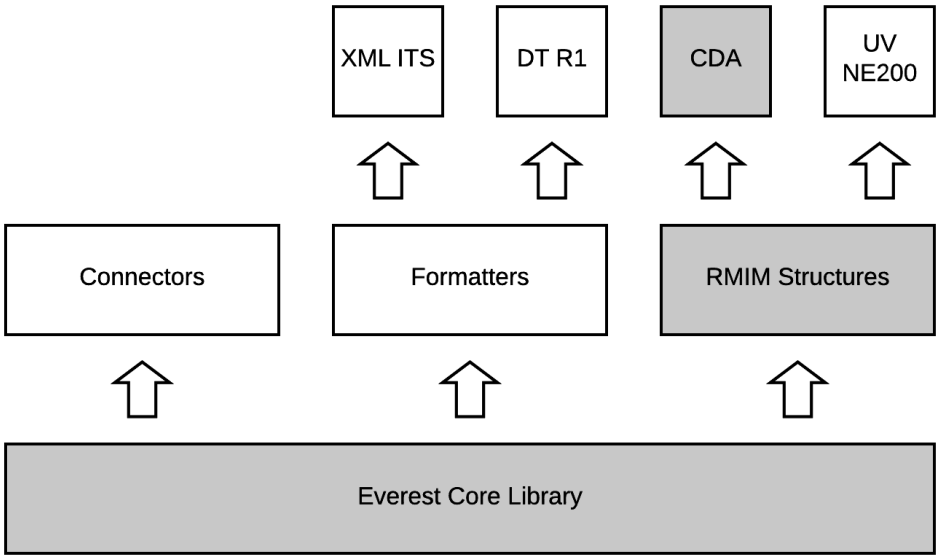


Figure 5.3: A high level overview of the Everest Framework stack [18]

5.2.2 Everest

The Everest Framework focuses on making it easier to create, format, and transmit HL7v3 data structures. HL7v3 is a large and complex standard, and arguably Everest's greatest strength is its ability to present and enforce these standards in a relatively intuitive manner to developers. The framework also emphasizes performance and standards flexibility with HL7v3 standards, making it a suitable choice for creating and parsing E2E records.

A *canonical* model is a format usually designed to answer the business needs of its environment, but may not necessarily be designed to be interoperable with other systems. Everest was designed with the medical record exchange workflow in mind. By creating a mapping from the canonical OSCAR model to a RIM graph, Everest can serialize the RIM graph and generate an appropriate Refined Message Information Model (RMIM) XML data exchange document. Since E2E is a form of CDA which is a type of RMIM, Everest supports the ability to handle E2E documents.

Everest was chosen as the main framework for generating and parsing HL7v3 XML documents because it is a mature framework with active development. Although Model Driven Health Tools (MDHT) is also a prominent model driven CDA implementation library [81], it approaches the problem by auto-generating CDA Java classes. Since

we need both transformation directions, we require a more hands-on approach to managing CDA structures. Everest's ability satisfies that because it encapsulates and represents the complex structures present in HL7v3 in a simple POJO format.

5.2.3 Environment Sandboxing

One of the main reasons we decided to pursue the sandbox route is because OSCAR is currently written on the JavaSE 7 API. Although Java is mostly backwards compatible and allows running older API code on newer versions, there are key language features introduced in the JavaSE 8 API such as Lambda Functions and Streams that the BXE2E Framework requires in order to function. Although we could upgrade OSCAR's API level to 8, this would entail a non-trivial amount of development time as there exists a large amount of legacy code and dependencies that may not necessarily work as intended with the newer version of Java.

Since our BXE2E Module only requires access the patient records in the database and a way to handle E2E XML documents, we know we can build a bare-bones OSCAR sandbox that can fulfill BXE2E's requirements. To construct an OSCAR sandbox, we would require Spring and Hibernate to interface with our database. Since we do not require the heavy-weight features that MySQL offers, we elected to use an HSQL in-memory database instead. Java's JPA is capable of interfacing with any type of SQL type database since it abstracts the technical details of connecting to a database.

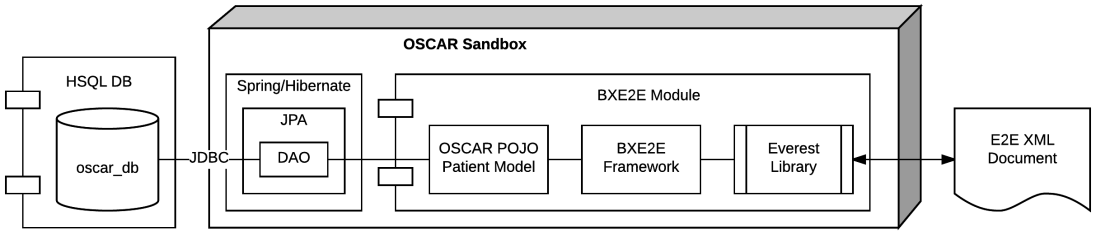


Figure 5.4: An overview of the BXE2E Module and its sandbox environment

We would also require the DAOs present in OSCAR for all the tables we need access to, as well as any POJO model definitions required to store the table entries. Everest can be implemented within our BXE2E Module in order to manage the creation and consumption of the E2E XML documents. Since Everest is self-contained, there are no other prerequisites in order for our sandboxed environment to function.

Another advantage for using a sandbox approach instead of building directly into OSCAR is that we are able to have faster, more agile development. OSCAR is a large-scale project, and it takes a considerable amount of time to compile and test. Using a sandbox eliminates the need to constantly recompile other modules. Since we do not utilize the code base of the other modules, we do not need to worry about recompiling and testing those unused parts. Since BXE2E is also designed to be an independent OSCAR module, we have the capability of "transplanting" BXE2E from the sandbox back into the actual OSCAR environment and can expect to have it work with minimal modifications.

5.3 Implementation Details

BXE2E is a framework that resides within the OSCAR ecosystem. We have built our proof-of-concept framework within an OSCAR sandbox to have more agile development as well as limit external factors on performance. Expanding on our introduction of the BXE2E Design in section 4.1.1, we will first look at the technical details and then the rationale behind the design.

5.3.1 Programming Language

Any software tool is only as strong as the language it is built on. As BXE2E is meant to be a part of OSCAR, this limits our language choice to only those which utilize the Java Virtual Machine (JVM). Though there are languages such as Clojure, Groovy and Scala which all can use the JVM, ultimately the act of introducing another language into the OSCAR ecosystem would be detrimental as doing so would make OSCAR less accessible to new programmers. As shown in Table 5.1, 86.8% of its codebase written in Java, with most of the remaining codebase being Javascript, CSS and HTML for the front end user interface.

Clojure, a functional Lisp dialect language, would lend itself well to lens theory since they are both functional in nature. Even though Clojure is able to invoke and manipulate Java objects with Java Interop, the different languages must be compiled to JAR files in the right order before they can be run together [83, 84]. This would easily add complexity to the already lengthy OSCAR compilation process. Clojure also generally runs slower than Java because of its startup and interpretation overhead [85] Ultimately, the features that are available in other JVM languages does not outweigh

Language	Percent
Java	86.8%
JavaScript	10.6%
CSS	1.3%
HTML	0.8%
SQLPL	0.2%
XSLT	0.2%
Other	0.1%

Table 5.1: OSCAR's codebase language distribution [82]

the cost of the developer overhead incurred to learn another language.

Lambda Functions

Picking Java as our language, we must also investigate what tools the language provides us with. The introduction of the JavaSE 8 API brings with it Lambda Functions and Streams. Both features are used in BXE2E's design. Previous Java versions did not have true functional programming constructs which would have impeded the creation of lenses. Although we could technically work around the lack of functional programming through other design approaches, it would not take advantage the functional aspects of lens theory.

In the domain of lambda calculus, all functions are anonymous, or not bound to some form of identifier. As such, lambda expressions are also known as anonymous functions. Although these anonymous functions provide a more concise way of expressing operations, our model transformations still require high performance. Oracle claimed that using Java 8's lambda expressions would yield more efficient code [86], and the claim was verified by Ward and Deugo [87]. Since lambda functions are actually efficient in practice, we could leverage them in BXE2E's design.

Streams

Java streams are also another desirable feature because they make it easier to apply multiple operations to a sequence of objects. Although the notation style used in streams does increase readability when used correctly, the main desirable feature of streams is its ability to easily make their operations all run in parallel. Instead of requiring an explicitly written fork-join model for generating and managing threads, streams are able to do all that under the hood.

The key idea with streams is that the sequence of objects enter what is known as a stream pipeline. This pipeline includes a source (such as an input list), zero or more intermediate operators that act on the stream, and a final terminating operator which yields a side-effect or result. What makes streams also functional in nature is how intermediate operators are constrained. They must, in most cases, be stateless (not affected by the execution state of the stream pipeline) and non-interfering (does not modify the stream source) [88]. These intermediate operator constraints makes stream pipelines significantly easier to parallelize because there are few side-effects to worry about by design.

Streams are only useful if they can offer performance comparable to more traditional for loops. Even though the more traditional for-each loops perform better than streams, a properly implemented stream can still perform comparably [89]. Where streams outshine the traditional for-each implementation is in parallelization. Even though a regular stream is marginally slower than a for-each equivalent, a parallelized stream can increase performance by simply making it easier to implement concurrency. The performance impact of streams was considered when we were designing BXE2E.

5.3.2 Code Organization

BXE2E's design draws inspiration from TGGs, Lenses and Map-Reduce. In order to make the source code coherent and traversible, we organized the code into multiple discrete java packages as seen in Figure 5.5. The *constant* package handles all the static strings and values that make up the majority of the boilerplate data structures found in E2E. The *lens* package (in blue) contains all the infrastructure required to define lenses for each subsection. The *model* package represents the metamodel used to represent an OSCAR patient, while the *rule* package (in green) contains the TGG rules which bind the source and target models together. The *transformer* package (in yellow) performs the map-reduce operation in our transformation, and the *util* package contains frequently used utility functions that assist the transformation such as converting a ClinicalDocument into a string representation.

Our TGGs embed the composed lenses in the rule's constructor, and the transformer organizes and determines which rules need to be run for the transformation. They are separated into different packages intentionally in order to make it clear what each part of our code is doing. Although we did consider grouping related rules and lenses together into packages, we found in practice this made it harder to trace the code.

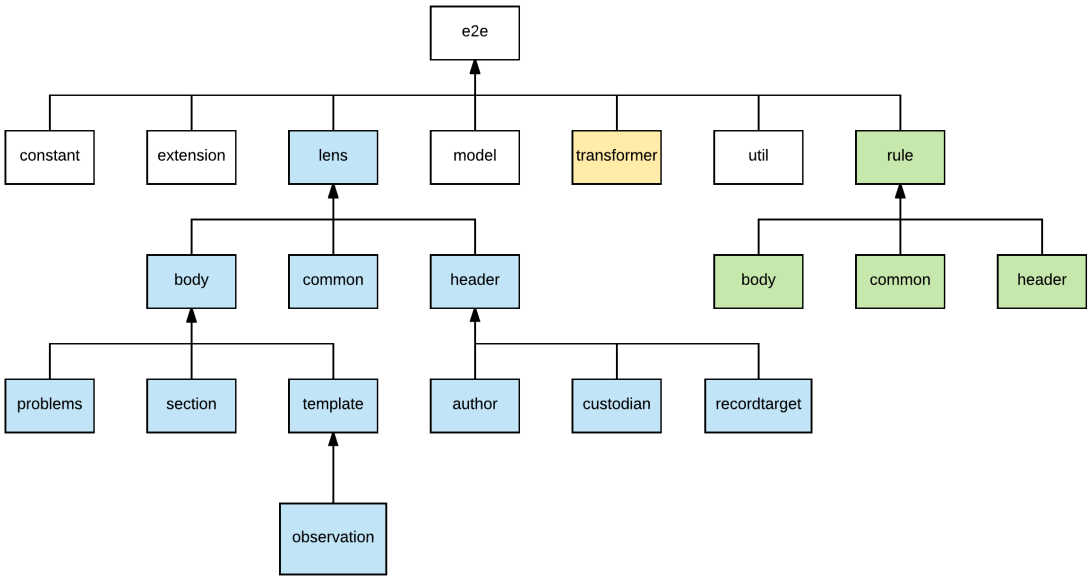


Figure 5.5: A view of the package organization tree in BXE2E

However, we organized related lenses and rules with a similar subpackage name and hierarchy in order to maintain clarity as well as indicate the context relations.

5.3.3 Map-Reduce

A map-reduce design works by splitting the incoming data-set into independent chunks that can be processed in parallel, and then recombined into a final result via reduction. Map-reduce makes sense for BXE2E because the transformation workflow is repetitive in nature and allows us to leverage the functional programming features in Java 8. Transforming between the OSCAR and E2E models involves a set of specific transformations for each topic such as Demographics, Problems and Medications. Though each section has its own transformation requirements, certain sections may have multiple entries that need to be transformed. As each section entry is independent and does not influence the state of other entries, this makes our transformation problem easy to parallelize.

We should note that TGGs are rule based, meaning that they execute when they find and match certain patterns. TGG engines have some form of rule engine that searches for pattern matches and then act on the graph model. Although pattern matching would work for our medical record transformation work, it introduces

some unnecessary overhead. This is because medical records and their subsections are predictable. Instead of non-deterministically searching for and matching object patterns, we can instead use our understanding of how a medical record is structured to remove the pattern matching step altogether. Our map-reduce looks at what is contained in the source and generates all the required rule objects.

```
return rules.parallelStream()
    .map(IRule::execute)
    .sequential()
    .collect(Collectors.toConcurrentMap(IRule::getName, IRule::getPair));
```

Figure 5.6: A code snippet of the map operation in BXE2E

One of the advantages of the map-reduce design is its code simplicity. The map operation utilizes the Java Stream API in order to succinctly perform the transformations. As shown in 5.6, the mapping operation short and simple. It takes the list of rules named *rules* and makes each rule element execute. Its terminating operation is to collect the results into a new map with the rule's name as the key and the resulting model pair as the value. We also have the added ability to parallelize the rule executions to improve performance when we explicitly manipulate the stream to be parallel for the map operation and returning to sequential for the collect operation.

The reduce operation of BXE2E is also quite straightforward. After our map operation generates the key-value pairs for each executed rule, the reduction takes all of the generated results and assigns them into the appropriate part of the target model. In order to let the reduce operation know where each output needs to go, it looks up and matches the key name in the new map. For example, a rule name with a value of "ProblemsRule-64377d7e" tells the reducer that the value belongs in the problems section. We avoid key collision in our map by generating and appending a unique hashcode to the key name for every rule that has a multiplicity larger than one.

5.3.4 Triple Graph Grammar Components

BXE2E's rules consist of two main components: the IRule interface, and the AbstractRule class. The IRule interface specifies methods to access its name, execution state, an execution function, and getters for the source and target. The base AbstractRule class implements the IRule interface and contains a few components

tracking its own state. The AbstractRule tracks whether the source or the target model is the origin, generates and stores a uniquely identifiable name on instantiation, and tracks its own execution state to prevent double-running. Finally, it stores the source and target model objects to be transformed.

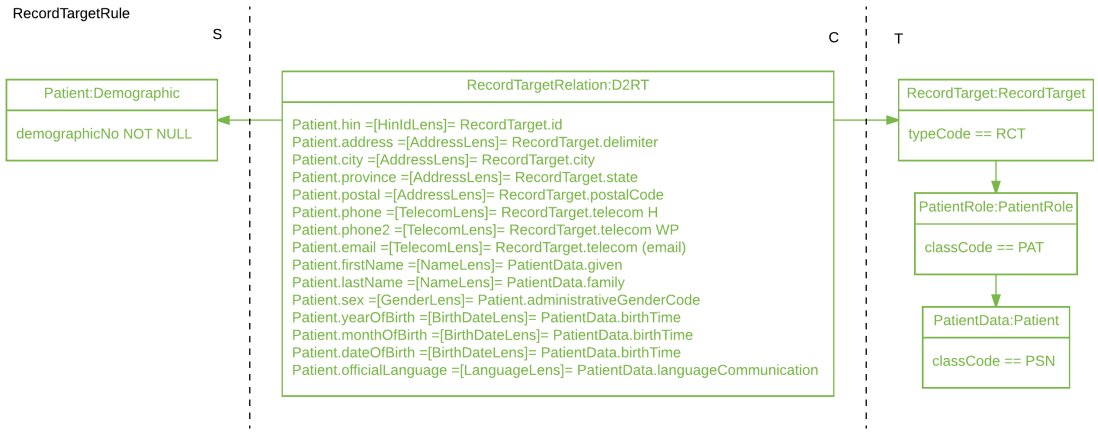


Figure 5.7: The TGG Rule for transforming between a Demographic and RecordTarget

Every TGG rule consists of three parts: a left side, a right side, and a correspondence between the two. For example, Figure 5.7 represents a TGG rule for the transformation between a Demographic and RecordTarget. Each side represents how the model objects should appear, while the correspondence in the middle defines how the different model objects are related to each other. The correspondence also defines which element attributes are correlated with each other via the Lens functions specified in the brackets.

An AbstractRule contains everything needed in order to function as a rule except the correspondence. For an actual rule implementation of a section such as RecordTarget, the RecordTargetRule class would extend the base AbstractRule, and then specify a single lens which acts as the rule's correspondence. The lens defines how the rule will behave when it is executed. Each rule implementation will contain its own unique lens definition specifying the behavior of its execute function.

Each rule only requires one lens definition because a lens will have both transformation directions defined. As well, lenses are able to be composed, building up from multiple simpler lenses in order to define a more complex transformation behavior. As shown in 5.8, the RecordTargetRule is defined by a RecordTargetLens and a composition of other lenses such as the id, address and telecom. Overall, BXE2E

```
@Override
protected AbstractLens<Pair<Demographic, RecordTarget>, Pair<Demographic,
 ↪  RecordTarget>> defineLens() {
    return new RecordTargetLens()
        .compose(new HinIdLens())
        .compose(new AddressLens())
        .compose(new TelecomLens())
        .compose(new NameLens())
        .compose(new GenderLens())
        .compose(new BirthDateLens())
        .compose(new LanguageLens());
}
```

Figure 5.8: A snippet of the RecordTargetRule lens composition definition

rules are simple in design because it only contains a lens definition for transformation, the model object pair to be transformed, and a few unique identifiers. After the rule is executed, both the model and target objects will represent the same data.

5.3.5 Lens Components

All of our lenses build up from the AbstractLens class. Our AbstractLens implements the functional interfaces IGet and IPut which are derived off of the Get and Put functions respectively as described in Section 2.5.3. We also overload our Put function with Create functionality by passing in an empty value in lieu of the original source. Each lens contains a list of strings called lensTypes. This variable tracks what type of lens it is by recording all of the sublenses that it was composed from. We also directly use Java 8's Function API to define the get and put functions for the lens. These functions are stored as part of the lens definition as variables. Lastly, our lens defines the compose function which allows us to chain together different lenses together and yield a more complex lens.

We implement a Java lens compose function which leverages the functional programming properties of higher order functions and statelessness as shown in Figure 5.9. This implementation is based off of the Compose function as covered in Section 2.5.3. As all of our lenses extend from our AbstractLens class, we have the groundwork implementation for lens theory in a Java framework. Using a lens is as simple as creating a new instance of the desired lens, and calling either a get or put and passing in the object to be transformed. This can be done in a single line, making our lens transformations a very concise operation.

```
public <U> AbstractLens<S, U> compose(AbstractLens<T, U> innerLens) {
    ...
    AbstractLens<S, U> newLens = new AbstractLens<>();
    newLens.lensTypes = this.lensTypes;
    newLens.lensTypes.add(innerLens.getClass().getSimpleName());
    newLens.get = this.get.andThen(innerLens.get);
    newLens.put = (s, u) -> this.put.apply(s,
    ↪  innerLens.put.apply(this.get.apply(s), u));
    return newLens;
}
```

Figure 5.9: A snippet of the Java lens compose derived from the Compose function

Our lenses are designed so that all lens implementations consist of a single constructor defining the behavior of the get and put functions, with Java Templating specifying what type of source / target pair the lens will be handling. This provides us with a very succinct way to define the transformations as seen in Figure 5.10. Having both the get and put functions appearing in the same source file is beneficial because 1) code maintainability is increased and 2) round-trip testing is significantly easier to do. In essence, keeping the get and put functions together makes the code becomes easier to read and easier to verify for correctness.

```
public SomeLens() {
    get = source -> {
        // Do Get Transformation
        return new ImmutablePair<>(source.getLeft(), source.getRight());
    };

    put = (source, target) -> {
        // Do Put Transformation
        return new ImmutablePair<>(target.getLeft(), target.getRight());
    };
}
```

Figure 5.10: The basic layout of a lens constructor for SomeLens

With a get and put together on the same source file, the developer will have an easier time writing unit tests for the lens because they can test whether the well-behaved round-tripping properties explained in Section 2.5.3 are in effect. Being able to prove that the transformations have well-behaved round-tripping properties improves the medical record transformation's reliability.

The Java Function API does not explicitly prohibit the creation and usage of impure functions because ultimately Java is still a strongly-typed imperative language

[90]. In order to ensure that our lens functions are as pure as possible, we explicitly use immutable data structures. We do this by using the ImmutablePair in the Apache Commons library [91]. The ImmutablePair was chosen because it provides a simple way of pairing related source and target model objects together, and it satisfies our need for an immutable data structure. It also has the added benefit of providing a common object type to pass around BXE2E, ultimately increasing its readability.

Immutable results are important because functional lens composition relies on immutable inputs and outputs. Our transformation functions always return a new ImmutablePair object containing our results in the get and put functions. This ensures that our lenses behave as pure functions by satisfying the immutable data structure aspect of functional programming. ImmutablePairs also have the added effect of making it easier to unit test because the results are more predictable.

Chapter 6

Evaluation and Analysis

This chapter focuses on our evaluation and analysis of the BXE2E framework and compares BXE2E to the older Velocity and E2Everest framework implementations. We evaluated BXE2E with respect to the requirements listed in Section 1.3. Each aspect will be addressed individually and the results will be discussed. As with any software engineering research, we will check if the framework meets specifications (verification), and ensure it fulfills its intended purpose (validation).

6.1 Evaluation Metrics

In order to have a comprehensive verification and validation of the BXE2E framework, we need to first enumerate and address the requirements of Section 1.3. In essence, BXE2E needs to be correct, maintainable, and performant. Evaluating for correctness will mostly be a verification effort, maintainability will mostly be a validation effort, and performance will be a mix of verification and validation. This is because correctness is primarily influenced by how conformant the transformations are to the E2E specification, whereas maintainability factors in the somewhat subjective measurement of readability.

Even though we can categorize certain aspects into verification and validation specific tasks, we still need to do a holistic analysis of BXE2E's overall software design impact. One consideration is the potential trade-off between maintainability and performance. An extreme example is to write an extremely convoluted and hard to read method that would yield high performance. The opposite extreme is to write a very naïve method that functions, but has terrible upper-bound runtime performance.

Of course this does not mean that performance and maintainability are mutually exclusive, as it is possible to have highly maintainable and performant code with proper engineering [92].

Although we were able to successfully apply bx theories and techniques into the medical record exchange problem with BXE2E, we have to ascertain whether this approach is viable for production use. Ultimately we have to carefully consider how our decisions on the BXE2E design influence correctness, maintainability and performance on both the large and small scales. There are multiple ways of quantifying the goodness of our BXE2E framework which we will cover in relation to the categories laid out in Section 1.3.

6.2 Correctness

Medical record exchange involves the safety-critical act of transferring patient record data. Software is considered safety-critical if its failure could lead to the harm of human life. Because of this, we need to reduce the risk of software faults by deliberately designing software that focuses on increasing its own reliability. For a patient record transformation to be correct, the transformer must not only satisfy specification requirements, but also be fault tolerant and capable of handling malformed data gracefully.

One reliable method of verifying the correctness of the BXE2E framework is by creating rigorous unit tests that cover all potential code paths. In Figure 6.1, there are a total of 8 unique paths. With tests that can ensure that BXE2E is yielding expected values on all paths, users can have higher confidence that the framework is reliable. This is a form of conformance testing, which is influenced heavily by the specifications of the record transformation.

Medical systems are complex enough that unit tests alone are not sufficient for software assurance [93]. However, there are other available methods to ensure correctness. Since our transformations are primarily defined by mappings between the source and target, we can also factor in expected bx properties that should be present in our framework. Although basic unit tests can ensure correct forwards and reverse transformations, we can also go one step further and utilize the laws of round-tripping.

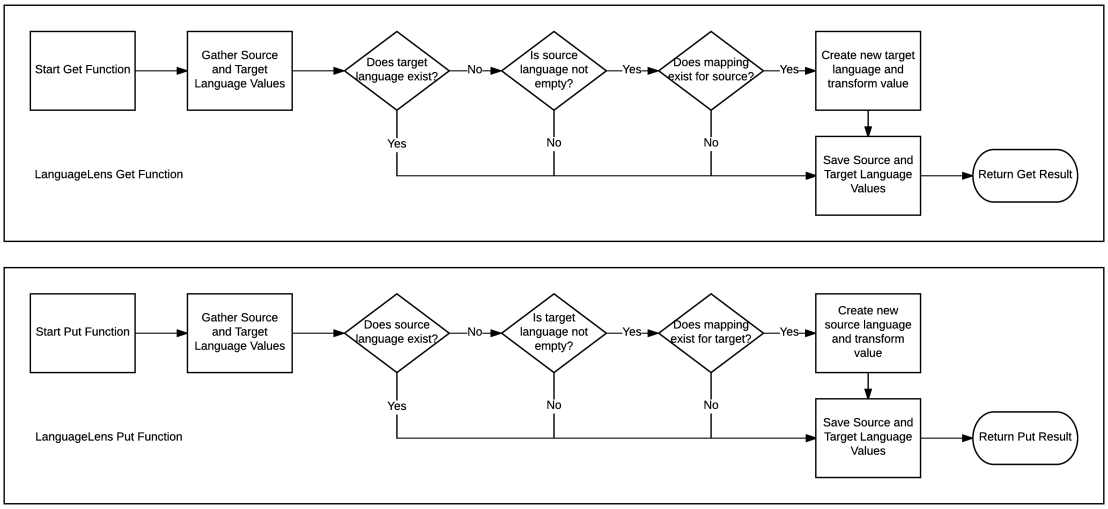


Figure 6.1: A flow diagram of the LanguageLens showing all 8 potential code paths that need to be unit-tested

6.2.1 Well-behavedness

One of the key benefits of using bx theory is the ability to perform round-trip testing. With round-trip testing, we can ensure information that passes through both the forwards and reverse transformations are properly handled and correct. This is because bidirectional transformations have three properties that are demonstrated in round-tripping: correctness, hippocraticness, and undoability. A transformation is coherent and well-behaved if all three of those properties are satisfiable [19].

For a well-behaved bx, correctness is the ability for transformations to create and enforce the relationship between the source and target [19]. A transformation that encounters source and target models that are not synchronized has to modify the models so that they become synchronized. As long as the transformation can modify the models so that they are synchronized, it is considered correct.

Hippocraticness is the property which ensures that the transformation functions do not modify the source and target if they are already synchronized [19]. This "check-then-enforce" behavior ensures that transformations only bring non-synchronized models to the correct state. An already synchronized source and model do not require further modifications to stay synchronized. Hippocraticness prevents the possibility of a transformation from making synchronized models no longer synchronized.

Undoability is the property which allows the propagation of reversion changes to be reflected properly in both the source and target [19]. For example, suppose there is

a synchronized source and target pair S and T. If S is modified to become S', the transformation should propagate the change and yield T'. Undoability allows the modification from S to S' to be rolled back such that when the source is returned back to the original S, we can expect the transformation to propagate and yield the original T.

If both transformation directions satisfy those three properties, we know that the bidirectional transformation is well-behaved. Well-behavedness can be demonstrated by applying the three laws of round-tripping (GetPut, PutGet, CreateGet) as discussed in Section 2.5.3. A bx which is able to satisfy the round-tripping laws is considered well-behaved because it is able to guarantee consistent and correct transformations between the two differing models.

6.2.2 Confluence

Another aspect of correctness takes into account the order of operations during a transformation. Triple Graph Grammar rules should be confluent, meaning that the order rules are executed does not affect the final result. We provide more background about the properties of confluence in Section 2.5.2. For our BXE2E architecture to be correct, its rules must be order of execution independent. This means that a developer working with BXE2E needs to ensure that every rule definition is an independent transformation, lacking the ability to influence the results of other rules.

One aspect of confluence leverages the local Church-Rosser theorem, which states that two rules can be executed in any order and does not make a difference to the end result [42]. Essentially if our rules are constructed to follow the Church-Rosser theorem, we can be assured that rule application order does not affect the outcome of the transformation. We can do this by applying the property of critical pairs, which is a set of two rules which do not have overlapping or conflicting contexts [43]. A system of rules is locally confluent if all potential pairings of rules are critical pairs.

In order to show that BXE2E's rule framework is confluent, all possible critical pairs in the framework must demonstrate convergence and thus be locally confluent. We do this pragmatically by ensuring that each rule only covers one unique type of data and that each rule cannot be nested into another rule's definition. For example, we can create separate rules for Problem and Alert sections because they deal with different data, but cannot create a rule for a sub-alert because a higher level rule already handles that data. Because the context of our rule sections deliberately do

not overlap, we know the results of each rule can only be influenced by one rule.

Medical record data consists of many different but interconnected parts. Although many different sections come together to represent the whole status of a patient, the sections themselves are differentiated. All a developer needs to do in order to use and extend the BXE2E framework is to make sure that each rule handles one and only one type of section data. This ensures that any critical pair in the framework will converge because the results of a rule simply do not influence any other rule's results. Because all possible critical pairs are locally confluent, the framework's rules are confluent.

6.2.3 Safety

Yet another aspect of correctness deals with software safety. Software is considered safety-critical if its failure could cause harm to human life, other living things, or the environment [94]. Electronic Medical Record systems are safety-critical because they handle, process and convey patient record data. We want to avoid situations such as the Panama City medical radiation exposure incident [77]. User interface consistency, effective information presentation, error prevention and the naturalness of clinical workflows are some of the aspects that can affect EMR usability and safety [95].

Although the topic of software safety encompasses numerous domains, we can address the aspects of error prevention, consistency, and reliability in software safety. The main way our BXE2E framework addresses software safety is by ensuring data correctness. Given a correct patient record state, BXE2E will ensure that the results of its transformation will also yield a correct patient record. While our BXE2E rules define "what" is to be transformed, our lenses specify "how" data will be handled.

We can ensure that BXE2E's transformations are consistent and reliable because our lenses take advantage of functional programming properties. BXE2E is specifically designed to minimize the risk of incorrect transformations by ensuring that each data element in the lens transformation has a well defined behavior and outcome. The associated unit tests can assure that the lens transformations behave as intended by testing with both correct and incorrect inputs. We can bolster our framework's safety and reliability by demonstrating what the framework will do in both cases.

6.3 Maintainability

One aspect of software maintenance is its readability. Software readability is a subjective human judgement regarding the ease of understanding code. Software that is not written for humans is hard to understand, handle and ultimately maintain. Code maintenance typically takes over 70% of a software project's lifecycle cost [96]. Therefore it is important to minimize the burden of maintenance by making the code easy to understand and act upon.

There are a couple measurements such as McCabe's Cyclomatic Complexity [80] and Halstead's metrics [97] which can quantify the complexity of code. However, it is important to note that readability is not synonymous with complexity. Brooks claims that software complexity is "essential" and results from satisfying system requirements [98]. In his model, readability is considered "accidental" because software engineers can more easily affect coincidental readability than intrinsic software complexity.

We can look into aspects of maintainability and break it down into four categories: complexity, conformity, changeability and invisibility. Complexity in software appears because no two subroutines are absolutely identical in implementation. If they were, those subroutines could be merged and reduced to one subroutine. The inherent uniqueness of each subroutine makes it difficult to reduce and refactor the code into something less complex without altering its expected behavior and output.

Conformity deals with the unwritten standards of written code. Well conformant code is usually familiar to other developers because its written conventions such as indentation and formatting are expected and predictable. Even following common indentation practices can make code significantly more readable as seen in Figure 6.2. Maintaining conformant code is easier to do because there are implicit expectations that exist within the codebase. Changeability quantifies the degree that a code base is capable of handling new functions and extensions. Its ability to evolve with changing requirements and platforms is strongly connected to how successful the software package is.

Finally, invisibility deals with the difficulty of diagramming software structure. Although graphs representing things such as control flow, data flow, and dependencies can be made, they do not provide the full picture of the software organization. As such, it is important to ensure that code is well organized and traceable. Maintaining a codebase is easier when it is packaged and organized into a sensible format such as a hierarchy.

Without Indentation

```
602 <!-- About Modal -->
603 <div class="modal fade hidden-print" id="about-modal">
604 <div class="modal-dialog"><div class="modal-content"><div class="modal-header">
605 <h3 class="modal-title">About Template Toolkit v1.0.2</h3></div>
606 <div class="modal-body"><h4 class="text-warning">Use This Tool At Your Own Risk!</h4>
607 <p>Ideally, users of this tool will have attended the lecture on template toolkits.<br />
608 Please notify the authors of any errors you find.<br />
609 Usage of this template constitutes your acceptance of these conditions.</p>
610 <h4 class="text-info">Authors</h4>
611 <ul><li>Content: John Cleese</li><li>Layout Design: Mary Jane</li></ul>
612 <h4 class="text-info">License</h4>
613 The <strong>Template Toolkit v1.0.2</strong> is licensed under a <a rel="license" href="http://
    creativecommons.org/licenses/by-sa/3.0/deed.en_US">Creative Commons Attribution-ShareAlike 3.0 Unported License</
    a>. <img src="http://i.creativecommons.org/l/by-sa/3.0/80x15.png" /><br />
614 Permissions beyond the scope of this license may be available at <a href="http://www.example.com">http://
    www.example.com</a>.</div>
615 <div class="modal-footer"><input type="button" class="btn btn-primary" data-dismiss="modal" value="Close" /></div
    ></div></div></div>
```

With Indentation

```
602                 <!-- About Modal -->
603                 <div class="modal fade hidden-print" id="about-modal">
604                     <div class="modal-dialog">
605                         <div class="modal-content">
606                             <div class="modal-header">
607                                 <h3 class="modal-title">About Template Toolkit v1.0.2</h3>
608                             </div>
609                             <div class="modal-body">
610                                 <h4 class="text-warning">Use This Tool At Your Own Risk!</h4>
611                                 <p>Ideally, users of this tool will have attended the lecture on template
                                    toolkits.<br />
612                                 Please notify the authors of any errors you find.<br />
613                                 Usage of this template constitutes your acceptance of these conditions.</p>
614                                 <h4 class="text-info">Authors</h4>
615                                 <ul>
616                                     <li>Content: John Cleese</li>
617                                     <li>Layout Design: Mary Jane</li>
618                                 </ul>
619                                 <h4 class="text-info">License</h4>
620                                 The <strong>Template Toolkit v1.0.2</strong> is licensed under a <a rel="license"
                                     href="http://creativecommons.org/licenses/by-sa/3.0/deed.en_US">Creative
                                    Commons Attribution-ShareAlike 3.0 Unported License</a>. <img src="http://
                                    i.creativecommons.org/l/by-sa/3.0/80x15.png" /><br />
621                                 Permissions beyond the scope of this license may be available at <a href="http://
                                    www.example.com">http://www.example.com</a>.
622                             </div>
623                             <div class="modal-footer">
624                                 <input type="button" class="btn btn-primary" data-dismiss="modal" value="Close"
                                    />
625                             </div>
626                         </div>
627                     </div>
628                 </div>
```

Figure 6.2: A comparison of how proper indentation on the same code significantly improves readability

Software maintainability is also very dependent on the platforms it is built upon. For comparison, OSCAR has multiple versions of E2E export software. The first version was built using the Apache Velocity framework, while the second version utilized the Everest framework, but was implemented with a model-populator design. Just by comparing between those two frameworks, it becomes apparent that their different approaches can achieve the same goal, but with different costs to code maintainability, testability and performance.

6.3.1 Testability

As with any successful software package, it needs to be well tested. Software testing is essential because it provides an automated way of verifying that its behavior satisfies the requirements. Well documented tests also make maintenance easier by providing practical examples of how the software is to be used. Tested code is also more reliable because well written tests can cover both normal and edge-case scenarios. Proper testing simply reduces the chance of the software yielding unexpected behavior.

As Bertolino puts it, software testing does dynamic verification of the behavior of a program on a finite set of test cases [99]. Proper selection of these finite test cases help ensure that the software yields expected behavior. As the potential problem space of software can have an infinite number of possibilities, it is simply not feasible to test every single one of the potential outcomes. However, this can be accounted for when developers know which parts of the codebase affect the majority of the common problem space.

One major goal of software testing is to detect software faults early in development. These faults may exist in code without ever being discovered simply because the preconditions for triggering the fault may have never been encountered [99]. However, these faults can lead to errors, or an unstable program state. Should errors become an observable output, they lead to failures, whether it be through testing or normal operation conditions. Testing can expose these failures, but the originating faults can only be found via offline analysis by developers.

The other goal of software testing is to evaluate software quality, measuring things such as reliability, usability and performance. Ultimately, software testing should yield a certain degree of confidence in the code because the tests can objectively measure to what extent its behavior is valid. Deliberately selecting what parts of the code to test as well as the method is not a simple task, as it requires the developer to have domain knowledge of how the software should behave. However, having a good testing procedure is a major factor in how maintainable a codebase is.

6.3.2 Software Design

Even the design of the software can have a large impact on how unit tests are designed. Designing code into small blocks allows the creation of smaller and more concise unit tests. Smaller code blocks typically have fewer discrete code paths, and creating complete unit tests is easier when there are less potential paths to consider. Smaller

code blocks have the added benefit of preventing potential maintenance errors because the code will be easier for a developer to understand and properly modify.

A good case example comparing the difference between code block size can be found in the historical E2E export implementations in OSCAR. The first generation exporter used the Apache Velocity library, which is a Java based template engine [17]. While this approach satisfies the requirements for creating an export document, it is not well suited for document importation. A template engine is able to integrate components together into an eventual output, but lacks the parsing functionality required for consuming documents.

Since Velocity is a template engine, nearly all of the transformation logic has to reside on the template itself. This creates a monolithic template which is not only hard to maintain, but also relatively difficult to properly test. As shown in Figure 6.3, both the E2E XML and the Velocity logic code are mixed together in the template. A new developer making sense of this template would need a significant amount of time to understand what the template is doing. Finding and fixing any bugs in the template will also require a substantial amount of effort.

```
1  #*##foreach ( $med in $patient.getMedications() )
2  #*    *##if( $stringUtils.isNullOrEmpty($med.regionalIdentifier) )
3  #*        *##set( $currentDin = "null" )
4  #*    *##else
5  #*        *##set( $currentDin = $med.regionalIdentifier )
6  #*    *##end
7  #*    *##if( !$previousDin.equals("0") && ( !$previousDin.equals($currentDin) ||
   ↪  $currentDin.equals("null") ) )
8                </substanceAdministration>
9            </entry>
10 #*    *##end
11 #*    *##if( !$previousDin.equals($currentDin) || $currentDin.equals("null") )
12           <entry typeCode="DRIV" contextConductionInd="true">
13             <templateId root="2.16.840.1.113883.3.1818.10.3.18"/>
14             <substanceAdministration classCode="SBADM" moodCode="EVN">
15               <id root="2.16.840.1.113883.3.1818.10.10.3" extension="Medications-$med.id"
   ↪  assigningAuthorityName="OSCAR EMR"/>
16               <code code="DRUG" displayName="Drug Therapy" codeSystem="2.16.840.1.113883.5.4"
   ↪  codeSystemName="HL7 ActCode"/>
17 #*        *##if ( $med.longTerm || $patient.isActiveDrug($med.getEndDate()) )
18               <statusCode code="active"/>
19 #*        *##else
20               <statusCode code="completed"/>
21 #* *##end
```

Figure 6.3: A snippet of the E2E Velocity template code

The second generation OSCAR exporter starts using Everest in the transformation workflow and uses a deferred model-populator design. The deferred model-populator design uses models to transform elements, and the populators to assemble the elements

together. Comparing between Everest and Velocity, it is immediately apparent that the transformation code is easier to maintain simply because the business logic is in Java. While Velocity needed its logic embedded into the transformation template, Everest allows the transformation to be broken down into small units, leading to easier unit test creation and overall maintenance.

The big shortcoming of the second generation E2E exporter is its inability to import E2E documents with any guarantees of correctness. While the deferred model-populator design could be applied to the import transformation, the design is not conducive of tightly binding discrete elements together into one bidirectional transformation. Both the import and export would effectively be independent functions, lacking any construct to influence each other's behavior. This is one of the main motivators for creating the third generation BXE2E framework for OSCAR.

6.4 Performance

Any software project worth using will require a certain degree of performance and scalability. In the context of medical record exchange, a patient record needs to be transformed correctly and quickly without impacting the rest of the medical record system. The transformation must also be capable of handling batch transformation operations, meaning that hundreds or thousands of records may require transformation within a short notice. The transformation engine should be efficient with memory, especially since the software may be running on a platform with limited resources.

For a benchmarking methodology to be successful, it needs to be taking representative measurements, be robust, portable, and transparent [100]. We can ensure a fair comparison between architectures by benchmarking the right metrics. Having a robust benchmark means that our results should not be sensitive to external factors. A portable and transparent benchmark methodology is achieved by ensuring our process is repeatable on various platforms and easy for other people to follow and replicate [101].

Software performance is typically measured in terms of speed, and resource utilization. This correlates directly with CPU utilization and its memory usage footprint respectively. By benchmarking our different export implementations under the same use case and measuring their results, we can begin to understand the benefits and drawbacks of each approach. Ultimately, the best performing architecture is the one which uses the least amount of CPU and memory per transformed patient.

Measuring raw CPU usage does not provide the full picture of how efficient the transformation operation is. Much of the performance gains in modern computing environments take advantage of concurrency [102]. Since most contemporary CPUs contain more than one logical core, we must also factor in multi-threaded performance. Assuming the transformation is designed properly, it is possible to take advantage of multi-threading in batch operations. Although multi-threading raises overall CPU usage as compared to single-threading, the overall batch operation duration would be significantly shorter.

Another factor to be considered with benchmarking is the language platform itself. As Java runs on top of the Java Virtual Machine (JVM), we must also acknowledge that the JVM will affect the transformation software's performance. Aside from executing the program in question, the JVM also performs tasks such as Just-In-Time (JIT) compilation, garbage collection, thread scheduling and memory allocation. These tasks serve as sources of non-determinism when it comes to reliably benchmarking the performance of our transformers [103].

Even the act of benchmarking a program can alter its results. In order to measure the memory and performance impact the transformer has on the JVM, the benchmarking program must run at the same time, and thus taking up another non-insignificant chunk of resources. Although it is impossible to benchmark a program without affecting its performance and behavior in some way, we can minimize its impact by accounting for it and providing a detailed benchmarking methodology [103]. This allows the benchmarking methodology to be more robust and portable by being easier to replicate and confirm for other parties.

6.4.1 Benchmark Configuration

Even though OSCAR is designed to have many independent modules, we are unable to reliably control when these other modules will run. This makes it exceedingly difficult to analyze only a specific module's performance because at any time another module may begin to run and cause unexpected performance results. Ultimately, our main goal in performance evaluation is to compare the differences between the three main approaches to medical record transformation: Velocity, E2Everest, and our BXE2E framework. As all three transformation approaches are built to satisfy the same use case, we can perform a direct performance comparison between them.

One way we can address OSCAR's multi-module performance issue is by isolating

Component	Specification
Operating System	Microsoft Windows 10 Home
System Type	x64-based PC
Processor	Intel® Core i7-4770k CPU @ 3.50GHz
CPU Cores	4 Cores, 8 Logical Processors
Memory	16 GB @ 1866MHz
Storage	Samsung Evo 840 SSD @ 500GB

Table 6.1: The machine specifications used for performance benchmarking

the frameworks into an OSCAR sandbox environment. By building only the bare essentials required for the frameworks to run on, we can ensure that only the target code is running and get more reliable results. This approach minimizes the impact of any other concurrent running tasks. The sandbox also provides a consistent environment to test within by ensuring we do the transformation operations on the same set of data and assumptions.

For a standardized dataset, we use a hypothetical patient record named John Cleese. All sections of data (i.e. demographics, problems, etc) that may be connected to the E2E patient record transformation workflow are populated fully in order to exercise all potential data paths. This patient record is designed to represent an average patient record which could exist in the OSCAR EMR. The John Cleese patient is inserted into our sandbox SQL database via an initialization script. Our database uses an in-memory representation with HyperSQL as its backend.

For our benchmarking tests, we used a computer with the specifications as listed in Table 6.1. All three E2E record transformation implementations were tested on the Java™ 8 SE Runtime Environment on build 1.8.0_101-b13. The transformation code is invoked by Eclipse Neon Release 4.6.0 with the org.oscarehr.e2e.Main class as the Main class. The Java Virtual Machine runs with the -Dfile.encoding=UTF-8 argument, initial heap size of 256MB and maximum heap size of 3.99GB. All non-essential programs and services on the test computer were terminated before benchmarking in order to minimize potential interference.

6.4.2 Benchmark Procedure

Our benchmarking consists of two main parts. The first part involves exporting John Cleese 10,000 times on each of the three frameworks and then comparing their performance results. A single patient record transformation normally does not take

any longer than a fraction of a second to perform. While there is no major significance with the number 10,000, the number of repeat exports must be large enough to let the benchmark period last a couple of minutes. This allows us to increase our sampling time window as well as allow for any potential variances in transformation to be averaged out.

Each iteration of the export will explicitly require a newly parsed patient from the database and a new set of transformation tools to execute. Each patient export must be independent to avoid any potential patient data caching that could significantly alter the overall execution time and performance. Since we know the resulting transformation will yield the same E2E patient record document for all 10,000 iterations, we skip printing the results to console. We do not need to manually inspect all of the output, and printing to console is a task that could significantly alter the overall runtime of the benchmark.

The second part of benchmarking involves exporting and importing the John Cleese patient record 10,000 times in a full round-trip on the BXE2E framework only. Both Velocity and E2Everest lack the import functionality so those two frameworks are not tested in this part. Once again for the same reasons as part one, we keep each iteration independent of each other, and iterate 10,000 times in order to smooth out potential variances and get a sizeable sampling time window.

On each iteration, the round trip exports the E2E patient record for John Cleese, and then that result is then imported back to the OSCAR data model. This benchmark is done for two reasons, one of which is to demonstrate a practical and complete bx round-trip transformation. The second reason is to compare the execution times between exporting and importing. Ideally both transformation directions should have a similar execution time because they are handling the same volume and type of data.

6.4.3 Profiler Tools

Although overall execution time can be indicative of an algorithm's performance, it does not delve into why a certain framework behaves the way it does. To address this, we require benchmarking tools which can break down how the program is spending its time and resources while transforming our patient records. While there are various tools available for profiling Java applications, we were specifically looking for ones which were free to use and had minimal impact to the performance of the program.

Drawing from the survey of Java profilers done by Flaig and his colleagues, we

determined that VisualVM and Java Mission Control were the best candidates [104]. These two tools were picked because they come with the Oracle JDK packages, had free usage licenses, provided useful runtime statistics in an easy to use interface, and had a minimal footprint on the executing program. While no single tool can cover all aspects of profiling, we were able to gather useful runtime metrics from the two tools.

VisualVM

VisualVM is a profiling tool developed by Oracle. Its main focus is to provide lightweight profiling capabilities and to present the results visually in a simple and clear GUI [105]. VisualVM is integrated as part of the Java JDK releases, which makes this tool very accessible. VisualVM is also open source with a GPLv2+CE license, and it collects profiling data through the jvmstat and Java Management Extensions (JMX) API. Because of this, there is a bit of overhead in performance, but its impact to program execution is minimal.

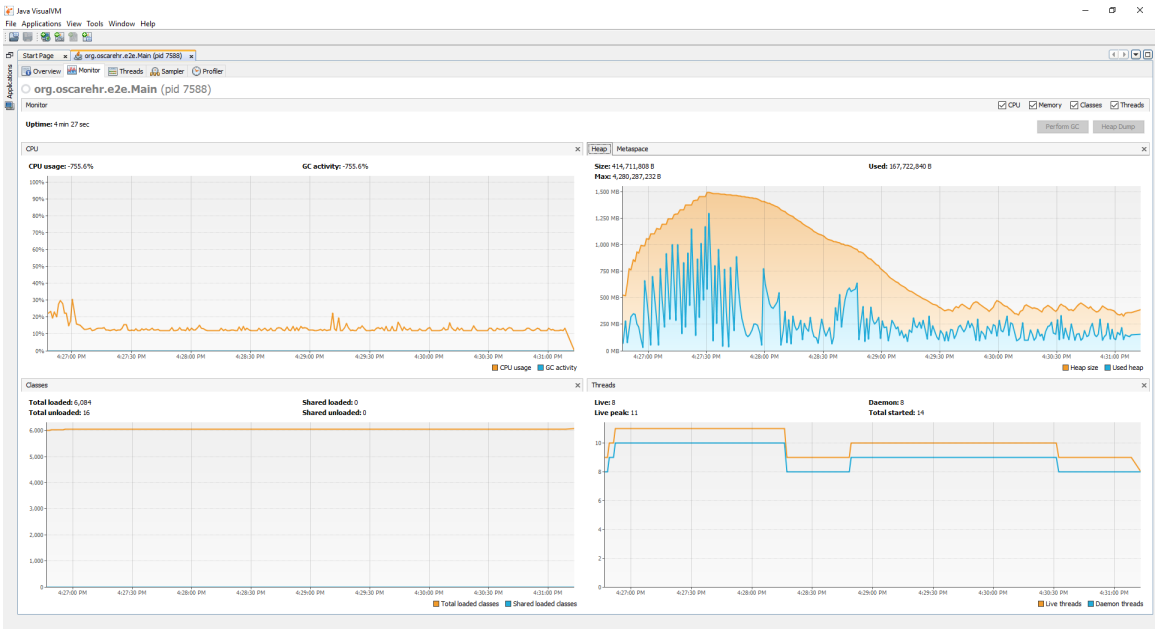


Figure 6.4: A screenshot of the main VisualVM interface

We chose to use VisualVM because it is great at providing high level overview graphs of a program's state. It also provides information about code hotspots, letting us know which parts of the codebase are taking up the most amount of CPU time. Code hotspots are essential in sampling because they let you understand which parts of the codebase are either doing the most amount of work, or to identify potential

bottlenecks in performance. We also have access to the overall memory footprint usage over time, which lets us know how efficiently our programs are using memory.

While VisualVM provides a great overview of profiling sessions, it is not without faults. Since VisualVM is designed to be a lightweight profiler, it does not provide in depth statistics about the JVM state. Most of the code hotspot identification is based off of CPU sampling. As opposed to true profiling which intercepts and records the entry and exit of a method, sampling periodically polls the CPU in order to determine which method is currently executing. While you can mitigate this by increasing the number of samples taken either by frequency or duration, ultimately code hotspot detection will only be an approximation.

Java Mission Control

To cover VisualVM's sampling gap, we utilize Java Mission Control (JMC). JMC is a free, closed-source profiling tool suite designed by Oracle [106]. The main tool we use from this suite is the Java Flight Recorder (JFR). One key difference between VisualVM and JFR is that JFR collects data directly from the JVM. This approach drastically minimizes performance overhead, with reports of its impact being less than one percent [107]. Unlike VisualVM, JFR is a true profiler because it collects high resolution data as the JVM executes it instead of sampling the CPU to determine what method is currently executing.

Since JMC and JFR are also a part of the Java JDK release environment, we already have these tools at our disposal. While JFR may not have a GUI as simple and clear as VisualVM, we gain access to more detailed statistics such as garbage collection, memory allocation and utilization, and method call counts. While VisualVM allows us to look at overall general trends and patterns, JFR provides us with in-depth runtime details that are hard to enumerate without direct JVM access.

One of the major drawbacks of using JMC is its need to tamper with JVM arguments. Since flight recording depends on Oracle's commercial features, specific flags need to be passed into a running JVM for JFR to profile the running application. Another drawback is that the JFR can only hook into an already running JVM instance and cannot be started exactly with the start of the application. As such, only VisualVM is capable of measuring how long an application executes from start to finish. Either way, JFR brings to the table a large amount of detailed profiling information which is beneficial for our analysis of framework comparisons.

Figure 6.5: A screenshot of the main Java Mission Control interface

6.5 Benchmarking and Analysis

In this section we explore the results of the various tests we performed between Velocity, E2Everest and BXE2E frameworks. The first test involves exporting a single patient in a single-thread multiple times. The second test involves the export and import of that single patient in a single thread multiple times. The third test is similar to our first test, but is performed with multiple-threads instead. The final test performs an export and import of that single patient with multiple-threads.

6.5.1 Single-Thread Export

Our first cross-framework evaluation involves the head-on performance comparison of exporting the John Cleese patient 10,000 times. In all three tests, we elected to use a single-threaded loop approach in order to allow any inefficiencies in processing to more easily appear in the overall execution time. In all three frameworks, output verification is an optional feature. Since verification significantly alters the behavior of all three frameworks, we analyze the single-threaded export operation in two parts: with and without verification.

Without Verification

The key results of our benchmark test without verification are aggregated in Table 6.2. We find that out of the three frameworks, BXE2E is the fastest of the three at exporting 10,000 patients, clocking in at 57 seconds. However, the most memory efficient framework is E2Everest with the lowest average memory use of 298 MB. The framework with the smallest garbage collection pause time is Velocity with 4.223 ms. Finally, Velocity also comes in with the least amount of average CPU usage at 22.5%.

	Velocity	**E2Everest**	**BXE2E**
Execution Time	79 s	68 s	57 s
Avg Memory Use	458 MB	298 MB	379 MB
Max Memory Use	1.28 GB	1.01 GB	1.25 GB
Avg GC Time	4.223 ms	5.601 ms	9.399 ms
Max GC Time	6.987 ms	10.609 ms	17.686 ms
Avg CPU Use	22.5 %	34.8 %	34.9 %
Max CPU Use	66.1 %	71.7 %	69.5 %

Table 6.2: Benchmark results of the single-threaded 10,000 patient export without verification

While BXE2E is the fastest at completing the export task, it trails behind the other two frameworks on the other measurements. However, measurements of the raw CPU, memory usage and garbage collection efficiency only scratch the surface when it comes to understanding how and why a particular framework behaves the way it does. For example, even though Velocity has the lowest average CPU usage time, it has the longest execution time and has the highest average memory usage out of the three frameworks. Its garbage collection time is the fastest out of the three however.

When we take a closer look at Velocity's hotspots, or methods and packages which take up a significant amount of time, we find that the org.apache.velocity.runtime.parser package takes up 50.46% of the overall execution time. Out of the parser package, the Velocity framework spends the most amount of time on the jj_34_24(), FillBuff() and jjMoveNfa_5() functions. This makes sense as Velocity's framework is designed around Apache Velocity for its transformation logic.

With the logic embedded into the template, the parser must spend a notable amount of time and resources interpreting the template before any inline transformation occurs. This could be one of the factors as to why we have on average lower CPU usage, shorter GC times and higher memory usage. Since it takes time to interpret a template,

more memory will be needed per export. However, garbage collection can be shorter because there are effectively less objects to clean up. Finally, since parsing is a more memory intensive process, it could explain the lower CPU usage.

When we look at E2Everest, we find that it is the second fastest in execution time, but uses the least amount of memory. However, it does have an increased garbage collection time and cpu usage as compared to Velocity. Looking at E2Everest's hotspots, we find a more even distribution of package with java.util at 19.66%, java.lang at 17.35% and org.hsqldb at 11.11%. Looking at time spent in functions, we find E2Everest spending most of its time on getSetterMethod() from Everest, getConnectionFromDriverManager() from Spring, and copyAdjustArray() from hsqldb. Since E2Everest is mainly an object oriented transformation approach, we can see that there is a larger distribution of objects taking up processing time.

Going one step further, we can investigate the Call Tree to see what high level tasks take up the most amount of time. E2Everest spends 54.29% of its time gathering the patient information into a usable OSCAR model from the database, and spends 40.25% of its time graphing, or converting from objects to an XML format. The actual data transformation of E2Everest only occupies around 3.61% of the time which could be a factor in E2Everest's average memory usage being the lowest of the three.

Finally, BXE2E's hotspots are similar to E2Everest's with java.util at 16.93%, java.lang at 14.90% and org.hsqldb at 12.33%. BXE2E also spends a significant amount of its time on the getSetterMethod() from Everest, getConnectionFromDriverManager() from Spring, and copyAdjustArray() from hsqldb. BXE2E has a similar footprint as E2Everest because both architectures utilize the Everest library as the main transformation engine. However, this alone does not provide a potential explanation to the differences we saw in Table 6.2.

Investigating BXE2E's Call Tree, we find that it spends 60.58% of its time also gathering patient information into a usable OSCAR model from the database, 36.25% of its time on graphing to XML, and 4.32% of its time performing the data model transformation. Once again, we find that BXE2E has a similar footprint to E2Everest. However, we find that BXE2E's garbage collection time and average memory usage is significantly higher than E2Everest, yet maintains a similar CPU utilization. With such similar footprints, this difference boils down to BXE2E's architecture being class heavy.

Unlike E2Everest which generally contains a Populator class and an associated Model class pairing for each section, BXE2E contains a Rule class for each section, as

well as a multitude of Lens classes which must be composed together using lambda functions. The higher number of classes means there is a higher amount of instantiation and cleanup occurring, which could explain why we are observing higher memory usage and garbage collection. However, BXE2E's execution time is noticeably shorter than the other two frameworks, justifying the tradeoff in memory utilization.

With Verification

We then ran the same tests as before but with the added verification step. The key results of our benchmark test with verification are aggregated in Table 6.3. Out of the three frameworks, BXE2E is still the fastest of the three at exporting 10,000 patients, clocking in at 237 seconds. This time, the most memory efficient framework is BXE2E with the lowest average memory use of 221 MB. The framework with the smallest garbage collection pause time is Everest with 2.864 ms. Finally, Velocity also comes in with the least amount of average CPU usage at 21.0%.

	Velocity	**E2Everest**	**BXE2E**
Execution Time	267 s	253 s	237 s
Avg Memory Use	527 MB	237 MB	221 MB
Max Memory Use	1.33 GB	686 MB	807 MB
Avg GC Time	3.886 ms	2.864 ms	2.949 ms
Max GC Time	8.305 ms	6.167 ms	5.421 ms
Avg CPU Use	21.0 %	34.4 %	32.8 %
Max CPU Use	65.5 %	73.4 %	71.9 %

Table 6.3: Benchmark results of the single-threaded 10,000 patient export with verification

Verification contains two main stages, XSD verification, and Everest verification. On the Velocity framework, it only performs XSD verification since it does not use the Everest framework. E2Everest and BXE2E will have both XSD and Everest verification. The key thing to note with Everest verification is that it will always run regardless of whether verification is requested or not because it is done during the graphing to XML stage. The main cost of verification should effectively come from the XSD verification step.

Our previous analysis in the previous section should still hold up here because verification is an extra step that is performed after the final XML document is generated. Since verification is always done after the generation of the E2E XML

document, our frameworks now spend time alternating between transforming and verificating. Since XSD verification involves parsing quite a few XSD documents for conformance specifications as well as parsing the generated XML output, we can expect a significant increase in execution time in our results.

In general, the verification step adds around 3 minutes to the overall execution time. As with before, we find that Velocity has a similar behavior profile as before. It is still the slowest of the three frameworks in execution time, the highest in average memory usage, and the lowest in average CPU utilization. However, we find that its average garbage collection time has now become the slowest out of the three.

With the introduction of verification, we find the org.apache.velocity.runtime.parser package dropping to 18.21% of the time, and many com.sun.org.apache.xerces.internal sub-packages rising up to a total of 47.47% of total execution time. As expected, verification functions such as isValidXML() from Velocity's architecture now take up the lion's share of time.

E2Everest maintains its position in the middle of the pack with execution time. However, its average memory usage is actually lower than without verification, and we find average garbage collection lower than without verification. E2Everest's hotspots also has a similar shift to com.sun.org.apache.xerces.internal sub-packages taking up 52.21% of execution time and the other three main packages java.util, java.lang and org.hsqldb taking up a total of 25.85% time together. The isValidXML() function takes up a large chunk of processing time similar to Velocity.

Finally BXE2E is still the fastest in execution time of the three frameworks. An interesting result is that with verification, BXE2E is now the most memory efficient with the lowest average memory use, and a substantially lower garbage collection time as compared to without verification. Unsurprisingly, BXE2E also has a similar hotspot shift to com.sun.org.apache.xerces.internal sub-packages taking up 57.17% of the time and the other three main packages java.util, java.lang and org.hsqldb taking up a total of 24.96% time together. The isValidXML() function exhibits the same rise in processing time behavior as the other two frameworks.

Overall, the addition of verification yields a couple of interesting results. While overall execution time comparisons maintain their same rankings, we find that memory use and garbage collection mainly dictated by the verification step instead of the transformation framework. What this shows is that ultimately the minor memory utilization differences we saw in the without verification test are easily eclipsed by the verification routine.

That is not to say the non-verification results are not useful, as those results provide us insight as to how effective a framework is at using resources. However, in perspective of using this on the production level, it is safe to say that if the previous framework implementations were producing acceptable results and performance, our BXE2E framework will also fit within that performance profile. What we gain by using BXE2E is mainly execution speed with a minor change in memory utilization.

6.5.2 Single-Thread Round Trip

In order to demonstrate that BXE2E is a true bidirectional transformation framework, we need to perform a round-trip test. We set up our test case to export the John Cleese patient, and then import the generated E2E XML. This is repeated 10,000 times in a single-threaded loop in order to better evaluate how the framework spends its time. Each loop will export and then import the patient record in that order without overlapping.

From the previous section, we know that verification adds a constant amount of work to the overall execution time, so our round-trip test will be skipping verification. While we should be including the verification step in practice, its inclusion would only skew the results to be verification centric and dilute any potential insights that could be acquired about the framework itself. Standard output is suppressed because we want to minimize any potential result skewing that may be caused by console output lag.

	Export Only	**Round Trip**
Execution Time	57 s	108 s
Avg Memory Use	379 MB	516 MB
Max Memory Use	1.25 GB	1.34 GB
Avg GC Time	9.399 ms	9.321 ms
Max GC Time	17.686 ms	14.483 ms
Avg CPU Use	34.9 %	24.0 %
Max CPU Use	69.5 %	59.1 %

Table 6.4: Comparison of BXE2E's 10,000 single-threaded patient export vs round trip without verification

The results of our benchmark are compiled in Table 6.4 find that the total execution time is 108 seconds. This is roughly twice as long as just the export case which comes in at 57 seconds. The round trip execution time falls in line with our expectations

since a round trip must both export and import the patient record and thus should take twice as long to complete.

An interesting outcome however is that the round trip operation uses roughly 10% less CPU than just the export case. The results from Section 6.5.1 indicate that the transformation process is not CPU bound. One potential explanation for the lower CPU utilization could be that the round-trip emphasizes more on memory manipulation and the CPU utilization is lower because it is waiting on the data in memory to be moved around more in the round-trip case. We actually find that the round-trip case has a higher average memory usage at 516 MB as compared to the export only case of 379 MB.

While memory utilization on average is 37% higher in the round-trip case than the export only case, we find that the garbage collection behavior to be the same in both cases. Since we are using the same framework in both scenarios, we can expect memory cleanup patterns to be similar. Investigating BXE2E's round-trip hotspots we find java.util at 31.74%, java.lang at 11.59% and java.util.regex at 11.06%. It spends a decent amount of time on the getSetterMethod() from the Everest library which is used in generating XML.

Finally, when we look into the round-trip's call tree, we find that it spends 33.74% of its time gathering patient information into an OSCAR model, 20.37% of its time exporting, and 43.54% of its time importing. What this tells us is that the overall export process including both the gathering and exporting step takes 54.11% of its time, meaning it takes more time to do an export than it takes to do an import. Overall, the round trip has a relatively even balance of execution time between both transformation directions, suggesting that our BXE2E framework is viable for production use in terms of single-threaded performance.

6.5.3 Multi-Threaded Export

For our second cross-framework evaluation, we repeat the benchmark procedure as described in Section 6.5.1 which involves exporting the John Cleese patient 10,000 times. However, we modify the single-threaded loop to be parallel instead. As with before, we have the option to include or exclude verification. As we saw from previous tests, the act of verification effectively adds a constant amount of computation time to the overall execution time. We will nonetheless analyze the frameworks both without and with verification.

While there are many approaches to multi-threading, we elected to leverage the Stream library's parallelStream function. Parallelstream uses the common fork-join pool that the JVM provides, and schedules tasks such that each stream is assigned its own thread in the fork-join pool. Parallelstream provides us with a quick and easy way of converting without having to worry about manually handling the thread scheduling work ourselves.

Without Verification

The key results of our benchmark test without verification are aggregated in Table 6.5. Unsurprisingly, we find that BXE2E is still the fastest of the three frameworks at 24 seconds. E2Everest still maintains the most efficient average memory usage out of the three at 622 MB. Velocity is the best on garbage collection, with the lowest average time of 7.448 ms. Maximum memory usage and CPU usage in this multi-threaded test are effectively equivalent.

Much of our single-threaded analysis from Section 6.5.1 also applies here, including the way the frameworks spend their time on functions, call trees, and the code hotspots. Our single-threaded benchmark exposes the details of how the framework spends its time, whereas the current multi-threaded benchmark informs us to what degree each framework is capable of working under parallel workloads.

Since the act of transforming a patient record does not affect the state of other records, we can make our frameworks take advantage of concurrency. All three frameworks were designed to be thread-safe from the beginning, which allows us to take advantage of the availability of multi-core processors. On our test environment, we can see that the fork-join pool is actively dispatching and efficiently processing our patient records as shown in Figure 6.6. Since BXE2E is able to process records at or faster than the other previous competing frameworks, we can confidently say that BXE2E is capable of handling production use cases.

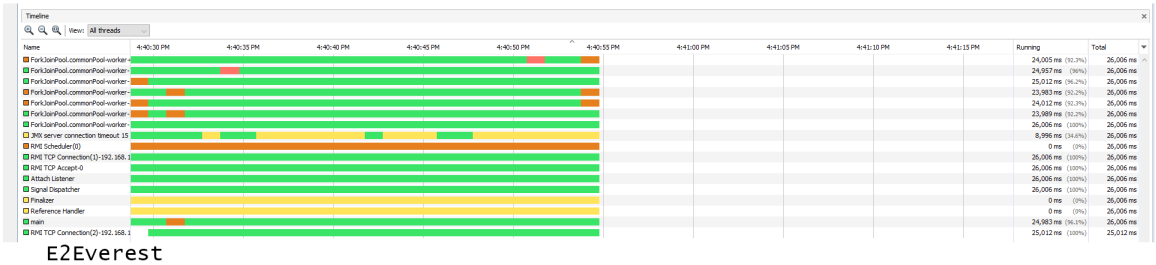
With Verification

The key results of our benchmark test with verification are aggregated in Table 6.6. Following the patterns seen from other tests, BXE2E is still the fastest of the three frameworks at 73 seconds. E2Everest has the best average memory usage at 688 MB. However, E2Everest now has the best average garbage collection times at 9.780 ms. Maximum memory usage and CPU utilization are effectively equivalent.

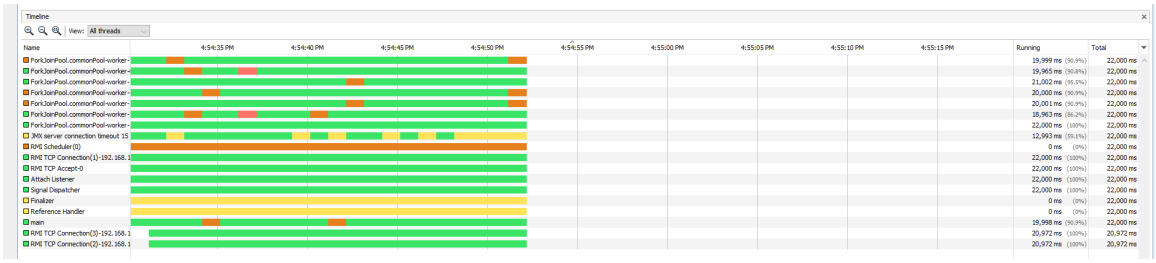
	Velocity	E2Everest	BXE2E
Execution Time	28 s	25 s	24 s
Avg Memory Use	710 MB	622 MB	671 MB
Max Memory Use	1.35 GB	1.34 GB	1.34 GB
Avg GC Time	7.448 ms	11.275 ms	23.889 ms
Max GC Time	16.747 ms	14.951 ms	38.081 ms
Avg CPU Use	99.1 %	98.7 %	96.6 %
Max CPU Use	100.0 %	100.0 %	99.6 %

Table 6.5: Benchmark results of the multi-threaded 10,000 patient export without verification

Velocity



E2Everest



BXE2E

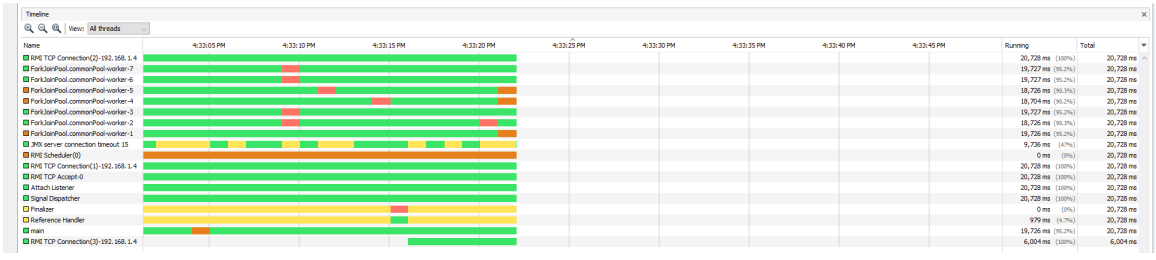


Figure 6.6: Overall thread utilization of the three frameworks without verification

Once again, our previous analysis in Section 6.5.1 continues to apply here. The distribution of time spent on function, call trees and code hotspots in this multi-threaded test remain very similar to their respective single-threaded counterparts. Based off of the single-threaded results, we expected verification to add a constant amount of computation time to the transformations. We found this behavior in the execution time results, where on average the verification step triples the overall execution time.

This test is more representative of the maximal use case of batch patient exporting because it is verifying every exported document in the multi-threaded operation. As shown in Figure 6.7, we see the fork-join pool threads also being utilized very well. The only exception is with the Velocity framework at the last couple of seconds where we see a noticeable number of fork-join threads going into park mode. This does cause Velocity's execution time to be a few seconds longer, but does not detract from the overall efficacy of the framework. Since BXE2E is processing the patient records faster than the other two competing frameworks and is effectively utilizing all available fork-join threads, we know our BXE2E architecture production ready.

6.5.4 Multi-Threaded Round Trip

As we saw in our evaluation of the single-threaded round trip in Section 6.5.2, BXE2E is capable of being a true bidirectional transformation network. We perform the same 10,000 export and import of the John Cleese patient the same way we did in the single-threaded round trip test. However, we alter the loop to utilize the parallelStream() construct to enable multi-threading. We forego the verification step because it only adds a constant amount of work to the overall execution. As well, standard output is suppressed to minimize any potential console lag that may skew the results.

We compiled our benchmark results in Table 6.7. We found the total round-trip execution time to be 37 seconds, whereas the equivalent export only case clocks in at 24 seconds. The round-trip case takes about 1.5 times longer to complete than just the export only case, meaning that the overall workload distribution of the round-trip is asymmetric. We will need to investigate the call tree to determine whether this is the case.

As is expected of a multi-threaded test, the average and maximum CPU usage is nearly maxed out in both the export only and round-trip cases. The average memory footprint is also relatively similar, with the average memory usage within 35

	Velocity	E2Everest	BXE2E
Execution Time	90 s	82 s	73 s
Avg Memory Use	698 MB	688 MB	736 MB
Max Memory Use	1.33 GB	1.35 GB	1.39 GB
Avg GC Time	10.143 ms	9.780 ms	13.260 ms
Max GC Time	18.521 ms	17.482 ms	20.591 ms
Avg CPU Use	99.7 %	99.7 %	99.6 %
Max CPU Use	100.0 %	100.0 %	100.0 %

Table 6.6: Benchmark results of the multi-threaded 10,000 patient export with verification

Figure 6.7: Overall thread utilization of the three frameworks with verification

	Export Only	**Round Trip**
Execution Time	24 s	37 s
Avg Memory Use	671 MB	706 MB
Max Memory Use	1.34 GB	1.38 GB
Avg GC Time	23.889 ms	12.152 ms
Max GC Time	38.810 ms	19.507 ms
Avg CPU Use	96.6 %	98.6 %
Max CPU Use	99.6 %	100.0 %

Table 6.7: Comparison of BXE2E's 10,000 multi-threaded patient export vs round trip without verification

MB of each other. While we saw a larger memory difference in the single-threaded comparison, our multi-threaded test yields very little difference in memory footprint. Since we are processing up to 8 patient records at the same time, each patient would require a certain amount of memory. Because we were using a common fork-join pool, this could also have an effect on the overall utilization of memory since each thread could be allocated a certain amount of memory.

One notable difference in our results is the garbage collection, with the multi-threaded export-only case taking twice as long to garbage collect on average than the round-trip case. The main likely cause of this could be the faster turnover of each patient record because the export-only case takes less time to process each record as compared to the round-trip case. Because of this, every time the JVM needs to garbage collect, there would be more memory objects needing to be cleaned out. With a shorter window of time, there would also be fewer garbage collections, meaning that there is simply more memory that needs to be reclaimed in less time.

Inspecting the BXE2E's round-trip hotspots, we find java.util at 34.05%, java.lang at 13.35% and java.util.regex at 6.65%. It spends a decent amount of time on the parseXSITypeNameInternal() and getSetterMethod() functions, both of which come from the Everest Library used in generating XML. We find the call-tree of our round-trip test spending 40.78% of its time on importing, 24.21% on gathering patient information into an OSCAR model, and 16.4% of its time exporting. The rest of the time is spent in the overhead of managing the fork-join pool. Combining the gathering and exporting stages, we find it uses 40.61% of overall execution time which is equivalent to the import part.

With both the effective import and export work about balanced with each other,

this does not explain the difference in times we are observing in overall execution time. However, we should note that the export part is done in both the export-only and the round-trip cases, meaning that there could exist a bottleneck somewhere in that part. One plausible explanation for the only slightly slower execution time of the round-trip could be the patient gathering stage, because it relies on talking to a database. In the export-only case, we are hitting the database almost 1.5 times harder than the round-trip case, and the database may not be able to keep up with the load as compared to the round-trip case.

6.5.5 Benchmarking Conclusion

In our benchmarks and tests, we compared BXE2E's raw performance with the existing Velocity and E2Everest frameworks, analyzing its behaviors, code hotspots, and usage profiles. We also investigated round-trip transformations to evaluate BXE2E's ability to be a viable bidirectional transformation framework. In general, BXE2E is a more efficient implementation for transformation in terms of execution time than Velocity and E2Everest.

However, BXE2E pays for a shorter execution time with generally more time spent on garbage collection. Unlike the other two frameworks, BXE2E generates many small classes such as the lenses and rules. These classes are then manipulated and composed together to yield the transformation we want. Because of this step, there will be more objects that need to be cleaned up by the JVM, leading to longer average garbage collection windows.

We also learned that verification adds a constant amount of time to the patient record processing time. In fact, verification itself generally takes more time to execute than the transformation itself because XSD verification involves parsing the output XML and then making sense of the structure. Ultimately, we found that BXE2E was performant under the multi-threaded tests and was capable of processing more patient records in less time than the competing frameworks. This means that our bx approach to record transformation is viable and worthwhile to pursue.

6.6 Requirement Evaluation

Referring back to the requirements outlined in Section 1.3, we need BXE2E to be correct, maintainable, and have good performance. In order to satisfy correctness,

our framework leverages bx round-tripping and well-behavedness laws, as well as confluence with the transformation rulesets. Finally, BXE2E ensures safety in the context of error prevention, consistency and reliability. These aspects all help ensure that our framework is safe to use and that the data maintains the correct meaning after transforming.

For maintainability, BXE2E has a heavy focus on software readability, testability and modular software design. BXE2E is designed to be easy to understand, contains a high degree of code test coverage, and is built up of small, easy to modify code segments. By applying a disciplined approach to the development of BXE2E, we can ensure that our BXE2E framework is maintainable, adding to the value of utilizing this as the transformation framework of choice.

Finally, with our battery of benchmarks to investigate how BXE2E behaves as compared to the Velocity and E2Everest frameworks, we have empirical evidence that BXE2E's architecture is superior to its competitors. In all tests, we found that BXE2E had the fastest execution times and reasonable memory usage footprints. While BXE2E's garbage collection is higher than its competitors as a side effect of how the software is designed, it does not impede in its ability to transform records quickly and efficiently.

By combining the best aspects of bidirectional theory, we were able to merge Triple Graph Grammars and Lenses together into a practical and functional framework. We were able to demonstrate a working example of how bx theory can be used in real-world scenarios by directly implementing it in a commonly used language such as Java. Since BXE2E is capable of addressing the requirements of correctness, maintainability and performance, we have a high degree of confidence that the BXE2E framework will meet the needs of medical practitioners needing to transform patient records.

Chapter 7

Conclusion and Future Work

Over the course of this thesis, we have evaluated whether bidirectional transformation techniques could be applied to a real-world medical record exchange with a minimal impact on performance. This chapter will outline future work that can be done on BXE2E and shed light on what aspects of bx theory and software design could be further investigated. We also summarize our findings and provide remarks to where else our BXE2E approach can benefit.

7.1 Future Work

While BXE2E provides a solid platform for bidirectional medical record exchange, there are still aspects of its design and implementation which have room for improvement. Since the healthcare industry is always rapidly changing, BXE2E must also be flexible enough to adapt to the ever changing needs of physicians in order to maintain its utility. Some notable areas for further research and development include concurrency, framework coverage, and memory usage improvements.

As BXE2E is a proof-of-concept framework, its codebase only contains support for E2E's header information, advance directives, and problems sections. A complete E2E document contains other sections such as allergies, medications and laboratory results. Although these other sections are not included in BXE2E in its current state, these other sections mirror the design of the implemented sections in E2E such as the problems section. Since these other sections are intended to have the same code structure of rules and lenses, they should maintain and preserve the benefits of the BXE2E framework's design.

One of the drawbacks of having a highly modular lens design is the large volume of stand-alone lens class objects in the codebase. While this design approach neatly packages the bx operations together, a BXE2E transformer with full coverage of an E2E document will contain hundreds of small lens classes. It may be worth investigating if lenses which work on immediately related content could be refactored to reduce the sheer number of classes and improve the readability of the codebase.

Another improvement that can be considered is to apply more caching to the overall data flow. While we already have some memory caching in place in order to reduce the number of database calls needed to export a patient, there may be some nominal performance increases possible with more aggressive caching. For example, since medical record structure are relatively repetitive, it may be possible to cache some of these redundant structures in order to reduce execution time.

BXE2E is currently capable of executing under a concurrent workflow because its design is highly modular and independent. As a result, the framework scales decently with the number of available CPU cores. However, there may be some room for improvement by reorganizing the way these independent work units are executed. While a standard Java fork-join pool is sufficient for concurrency, more performance could potentially be achieved by implementing our own thread scheduler instead.

Finally, while BXE2E has the fastest transformation times out of the three E2E frameworks, it is by no means the most memory efficient. While all three of the frameworks have large memory usage simply because it is using the JVM, there may be ways to reduce the amount of memory used within the framework by consolidating some chunks of data together. A rigorous analysis of memory artifacts that may be inadvertently generated during runtime can provide some insight into memory optimization.

7.2 Summary

BXE2E has demonstrated its capacity to meet the requirements of correctness, maintainability and performance as outlined in Section 1.3. Our results from Chapter 6 have shown that BXE2E is capable of meeting the needs of medical practitioners who need to perform patient record transfers. This brings the framework one step closer to actually being used in practice, where it can have a positive impact for medical practitioners.

When comparing between the three generations of E2E exporters, we find that

BXE2E provides the most versatility. BXE2E is able to deterministically enforce transformation correctness by leveraging the bx properties of well-behavedness. The property of confluence in BXE2E's TGGs ensure that rule execution is order independent and easy to make parallel. Record safety is preserved by ensuring that any transformation BXE2E performs does not create an invalid result.

With three generations of OSCAR E2E exports, the overall software maintainability has improved in leaps and bounds. The original Velocity exporter performed its job well enough but is a maintenance nightmare as shown in Figure 6.3. When comparing to BXE2E, we find that its architecture facilitates easy maintainability by keeping chunks of code small, co-located, and easy to test. While E2Everest had a similar principle of software design, it does not have constructs available to enforce consistent transformations in both directions.

In our performance evaluation, we found BXE2E's execution time to be the fast of the three architectures with relatively comparable memory usage and CPU utilization. The usage of lenses and TGG bx theories leads to at least an equivalent or slightly improved transformation performance as compared to the other exporter frameworks. As well, BXE2E has a stable and scalable multi-threaded performance, allowing for efficient batch transformations of large volumes of patient data.

7.2.1 Applications

BXE2E enables clinicians to import and export patient record information with other clinicians, significantly reducing the barrier to a smooth electronic transfer of patient data. If the different clinicians use EMR systems which support E2E as an exchange medium, BXE2E ensures that the represented patient data is properly processed and integrated into the EMR's system. With BXE2E, clinicians can spend less time worrying about acquiring the patient information they need, and spend more time providing quality healthcare.

Many programs traditionally stick with either a functional programming or an object oriented programming paradigm, leaving many programs unable to leverage the inherent strengths of other programming paradigms. Since BXE2E heavily leverages the functional features of Java 8, our proof-of-concept implementation demonstrates the viability of a hybrid design with lambda function components within an object oriented environment. We show that hybridizing both approaches together is effective when they are carefully encapsulated into limited contexts.

After developing and applying the BXE2E approach to OSCAR's import and export, we find ourselves with an interesting and useful fusion of two bx theories: Lenses and TGGs. By using both theory frameworks, we are able to take the best aspects from both of them to yield a reliable bx approach. TGGs help define and encapsulate the context of the content, while the lenses enforce the one-to-one transformations for each relevant data element.

We believe the BXE2E approach can be applied to any tree-like structured data source which requires the consistency guarantees of bx. Tree-like structures have the benefit of grouping related data elements together into co-located branches, thus allowing for an easy mapping to discrete TGG rules. The lenses inside the TGG rules are modular, allowing a decent degree of modularity and feature-swapping on both the lens level and the rule level, depending on the requirements.

However, it is not to say that BXE2E cannot work on any cyclic data structures. As TGG rules are a form of graph transformations, BXE2E would just need to focus the transformation effort more onto TGG rules and less on lens transformations. Effectively we would see an increase in the number of TGG rules and fewer composed lenses in order to handle highly cyclic data structures.

While we have outlined some potential applications BXE2E can fill, it is by no means an exhaustive list. The hybrid bx approach can lend itself to many other problems which require consistency guarantees because it offers a way to break the problem down into separate rule contexts and then apply well-behaved lenses onto the contexts. This type of transformation design combines the best of bx theory together in order to ultimately create consistent transformations, and can lead to improving the quality of life for many people.

Appendix A

Additional Information

The full BXE2E source code is found at https://github.com/jujaga/bxe2e [108].

A.1 Complete Problem Section Example

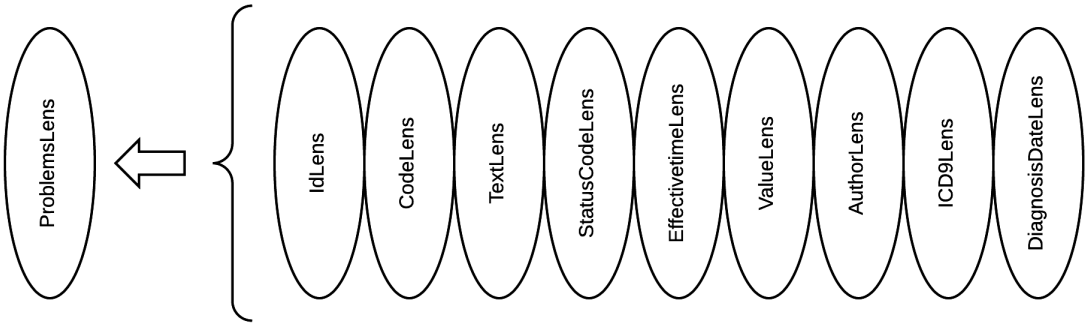


Figure A.1: The nine distinct lenses which are composed together to create the ProblemsLens bx definition.

```
1  package org.oscarehr.e2e.lens.body.problems;
2
3  import ...
4
5  public class ProblemsStatusCodeLens extends AbstractLens<Pair<Dxresearch, Entry>, Pair<Dxresearch,
   ↪  Entry>> {
6     public ProblemsStatusCodeLens() {
7        get = source -> {
8            Character status = source.getLeft().getStatus();
9            CS<ActStatus> statusCode =
   ↪  source.getRight().getClinicalStatementIfObservation().getStatusCode();
10
11           if(statusCode == null) {
12               if(status != null && status.equals('A')) {
13                   statusCode = new CS<>(ActStatus.Active);
14               } else if (status != null && status.equals('C')) {
15                   statusCode = new CS<>(ActStatus.Completed);
16               }
17           }
18
19           source.getRight().getClinicalStatementIfObservation().setStatusCode(statusCode);
20           return new ImmutablePair<>(source.getLeft(), source.getRight());
21       };
22
23       put = (source, target) -> {
24           Character status = target.getLeft().getStatus();
25           CS<ActStatus> statusCode =
   ↪  target.getRight().getClinicalStatementIfObservation().getStatusCode();
26
27           if(status == null) {
28               if(statusCode != null) {
29                   if(statusCode.getCode() == ActStatus.Active) {
30                       status = 'A';
31                   } else if (statusCode.getCode() == ActStatus.Completed) {
32                       status = 'C';
33                   }
34               } else {
35                   status = 'D';
36               }
37           }
38
39           target.getLeft().setStatus(status);
40           return new ImmutablePair<>(target.getLeft(), target.getRight());
41       };
42    }
43 }
```

Figure A.2: The implementation of the ProblemsStatusCodeLens lens. Both the get and put functions are co-located and declared upon class instantiation.

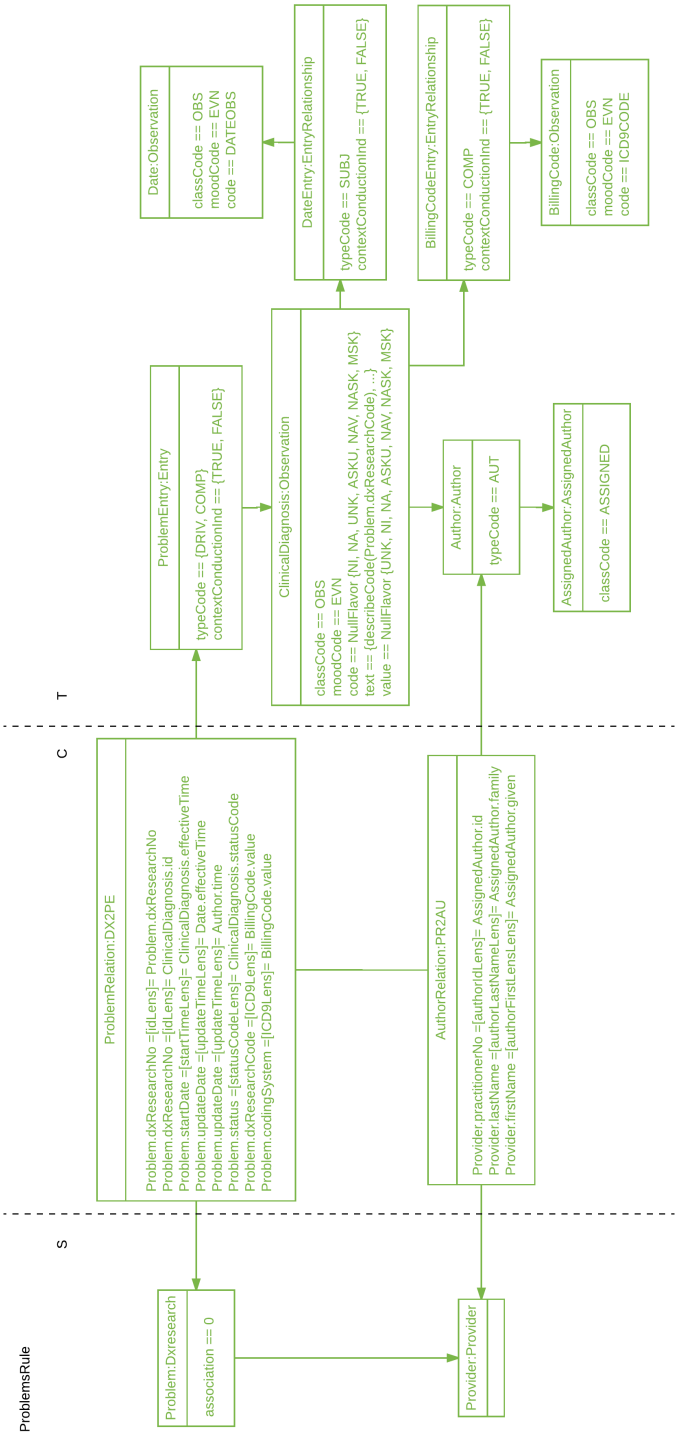


Figure A.3: The TGG rule of a problem entry. The lenses are defined inside the =[]= equivalences in the correspondence subgraph.

```java
package org.oscarehr.e2e.rule.body;

import ...

public class ProblemsRule extends AbstractRule<Dxresearch, Entry> {
    public ProblemsRule(Dxresearch source, Entry target) {
        super(source, target);
        this.ruleName = ruleName.concat("-").concat(Integer.toHexString(
 ↪   System.identityHashCode(this)));

        if(this.pair.getLeft() == null) {
            pair = new ImmutablePair<>(new Dxresearch(), pair.getRight());
        }
        if(this.pair.getRight() == null) {
            pair = new ImmutablePair<>(pair.getLeft(), new Entry());
        }
    }

    @Override
    protected AbstractLens<Pair<Dxresearch, Entry>, Pair<Dxresearch, Entry>> defineLens() {
        return new ProblemsLens()
                .compose(new ProblemsIdLens())
                .compose(new ProblemsCodeLens())
                .compose(new ProblemsTextLens())
                .compose(new ProblemsStatusCodeLens())
                .compose(new ProblemsEffectiveTimeLens())
                .compose(new ProblemsValueLens())
                .compose(new ProblemsAuthorLens())
                .compose(new ProblemsICD9Lens())
                .compose(new ProblemsDiagnosisDateLens());
    }
}
```

Figure A.4: The implementation of the problem TGG rule. Note that the ProblemsRule class itself is the correspondence object linking the source Dxresearch and target Entry objects.

A.2 Complete E2E Document Example

```
<?xml version="1.0" encoding="UTF-8"?>
<ClinicalDocument xmlns="urn:hl7-org:v3" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:hl7="urn:hl7-org:v3" xmlns:e2e="http://standards.pito.bc.ca/E2E-DTC/cda"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="urn:hl7-org:v3
    Schemas/CDA-PITO-E2E.xsd" classCode="DOCCLIN" moodCode="EVN">
  <realmCode code="CA-BC"/>
  <typeId root="2.16.840.1.113883.1.3" extension="POCD_HD000040"/>
  <templateId root="2.16.840.1.113883.3.1818.10.7.1"/>
  <templateId root="2.16.840.1.113883.3.1818.10.1.1"/>
  <id root="9F865A61-1C6C-4BF1-91D6-DD31D6FF4004" extension="1"/>
  <code code="11503-0" codeSystem="2.16.840.1.113883.6.1" codeSystemName="LOINC"
    codeSystemVersion="2.44"/>
  <title language="en-US">E2E-DTC Record of John Cleese</title>
  <effectiveTime value="201702111744-0800"/>
  <confidentialityCode code="N" codeSystem="2.16.840.1.113883.5.25"
    codeSystemName="x_BasicConfidentialityKind"/>
  <languageCode code="en-CA"/>
  <recordTarget typeCode="RCT" contextControlCode="OP">
    <patientRole classCode="PAT">
      <id root="2.16.840.1.113883.4.50" extension="448000001" assigningAuthorityName="BC Patient
    Health Number"/>
      <addr use="H">
        <delimiter>1234 Street</delimiter>
        <city>city</city>
        <state>BC</state>
        <postalCode>V8T1D6</postalCode>
      </addr>
      <telecom value="tel:1234567" use="H"/>
      <telecom value="tel:7654321" use="WP"/>
      <telecom value="mailto:test@test.com" use="H"/>
      <patient classCode="PSN" determinerCode="INSTANCE">
        <name use="L">
          <given>John</given>
          <family>Cleese</family>
        </name>
        <administrativeGenderCode code="M" codeSystem="2.16.840.1.113883.5.1"
    codeSystemName="AdministrativeGender" displayName="Male"/>
        <birthTime value="19400925"/>
        <languageCommunication>
          <languageCode code="EN"/>
        </languageCommunication>
      </patient>
    </patientRole>
  </recordTarget>
  <author typeCode="AUT" contextControlCode="OP">
    <time value="20170211"/>
    <assignedAuthor classCode="ASSIGNED">
```

Figure A.5: Part 1/4 of an E2E document example for a test patient.

```
41        <id root="2.16.840.1.113883.3.40.2.11" extension="cpsid" assigningAuthorityName="BC MSP Provider
      ↪   License Number"/>
42        <telecom value="tel:3456789" use="H"/>
43        <telecom value="tel:9876543" use="WP"/>
44        <telecom value="mailto:test2@test2.com" use="H"/>
45        <assignedPerson classCode="PSN" determinerCode="INSTANCE">
46          <name use="OR">
47            <given>oscar</given>
48            <family>oscardoc</family>
49          </name>
50        </assignedPerson>
51      </assignedAuthor>
52    </author>
53    <author typeCode="AUT" contextControlCode="OP">
54      <time value="20170211"/>
55      <assignedAuthor classCode="ASSIGNED">
56        <id nullFlavor="NI"/>
57        <assignedAuthoringDevice classCode="DEV" determinerCode="INSTANCE">
58          <softwareName language="en-US">OSCAR EMR</softwareName>
59        </assignedAuthoringDevice>
60      </assignedAuthor>
61    </author>
62    <custodian typeCode="CST">
63      <assignedCustodian classCode="ASSIGNED">
64        <representedCustodianOrganization classCode="ORG" determinerCode="INSTANCE">
65          <id root="2.16.840.1.113883.3.3331" extension="123456" assigningAuthorityName="OSCAR EMR"/>
66          <name>McMaster Hospital</name>
67        </representedCustodianOrganization>
68      </assignedCustodian>
69    </custodian>
70    <informationRecipient typeCode="PRCP">
71      <intendedRecipient nullFlavor="NI"/>
72    </informationRecipient>
73    <component typeCode="COMP" contextConductionInd="true">
74      <structuredBody classCode="DOCBODY" moodCode="EVN">
75        <confidentialityCode code="N" codeSystem="2.16.840.1.113883.5.25"
      ↪   codeSystemName="x_BasicConfidentialityKind"/>
76        <languageCode code="en-CA"/>
77        <component typeCode="COMP" contextConductionInd="true">
78          <section classCode="DOCSECT" moodCode="EVN">
79            <templateId root="2.16.840.1.113883.3.1818.10.2.2"/>
80            <code code="42348-3" codeSystem="2.16.840.1.113883.6.1" codeSystemName="LOINC"/>
81            <title language="en-US">Advance Directives Section [without entries]</title>
82            <text mediaType="text/x-hl7-text+xml">This section is not supported by the Originating
      ↪   Application</text>
83          </section>
84        </component>
85        <component typeCode="COMP" contextConductionInd="true">
86          <section classCode="DOCSECT" moodCode="EVN">
87            <templateId root="2.16.840.1.113883.3.1818.10.2.21.1"/>
88            <code code="11450-4" codeSystem="2.16.840.1.113883.6.1" codeSystemName="LOINC"/>
```

Figure A.6: Part 2/4 of an E2E document example for a test patient.

```
89            <title language="en-US">Problems and Conditions - Problem List [with entries]</title>
90            <text mediaType="text/x-hl7-text+xml">
91              <list>
92                <item>ICD9: 428 - HEART FAILURE*</item>
93                <item>ICD9: 401 - ESSENTIAL HYPERTENSION*</item>
94              </list>
95            </text>
96            <confidentialityCode code="N" codeSystem="2.16.840.1.113883.5.25"
    ↪   codeSystemName="x_BasicConfidentialityKind"/>
97            <entry typeCode="DRIV" contextConductionInd="true">
98              <templateId root="2.16.840.1.113883.3.1818.10.3.15"/>
99              <observation classCode="OBS" moodCode="EVN">
100               <id root="2.16.840.1.113883.3.3331" extension="ProblemList-2"
    ↪   assigningAuthorityName="OSCAR EMR"/>
101               <code nullFlavor="NI"/>
102               <text representation="TXT" language="en-CA">ESSENTIAL HYPERTENSION*</text>
103               <statusCode code="completed"/>
104               <effectiveTime>
105                 <low value="20130926"/>
106               </effectiveTime>
107               <value xsi:type="CD" nullFlavor="NI"/>
108               <author typeCode="AUT" contextControlCode="OP">
109                 <templateId root="2.16.840.1.113883.3.1818.10.4.2"/>
110                 <time value="20130926"/>
111                 <assignedAuthor classCode="ASSIGNED">
112                   <id root="2.16.840.1.113883.3.40.2.11" extension="cpsid" assigningAuthorityName="BC
    ↪   MSP Provider License Number"/>
113                   <assignedPerson classCode="PSN" determinerCode="INSTANCE">
114                     <name use="OR">
115                       <given>oscar</given>
116                       <family>oscardoc</family>
117                     </name>
118                   </assignedPerson>
119                 </assignedAuthor>
120               </author>
121               <entryRelationship typeCode="COMP" contextConductionInd="true">
122                 <templateId root="2.16.840.1.113883.3.1818.10.4.28"/>
123                 <observation classCode="OBS" moodCode="EVN">
124                   <code code="ICD9CODE" codeSystemName="ObservationType-CA-Pending"
    ↪   codeSystem="2.16.840.1.113883.3.3068.10.6.3"/>
125                   <value xsi:type="CD" code="401" codeSystem="2.16.840.1.113883.6.42"
    ↪   codeSystemName="ICD9" displayName="ESSENTIAL HYPERTENSION*"/>
126                 </observation>
127               </entryRelationship>
128               <entryRelationship typeCode="SUBJ" contextConductionInd="true">
129                 <templateId root="2.16.840.1.113883.3.1818.10.4.4"/>
130                 <observation classCode="OBS" moodCode="EVN">
131                   <code code="DATEOBS" codeSystemName="ObservationType-CA-Pending"
    ↪   codeSystem="2.16.840.1.113883.3.3068.10.6.3"/>
132                   <effectiveTime>
133                     <low value="20130926"/>
134                   </effectiveTime>
135                 </observation>
136               </entryRelationship>
```

Figure A.7: Part 3/4 of an E2E document example for a test patient.

```
137             </observation>
138           </entry>
139           <entry typeCode="DRIV" contextConductionInd="true">
140             <templateId root="2.16.840.1.113883.3.1818.10.3.15"/>
141             <observation classCode="OBS" moodCode="EVN">
142               <id root="2.16.840.1.113883.3.3331" extension="ProblemList-1"
    ↪   assigningAuthorityName="OSCAR EMR"/>
143               <code nullFlavor="NI"/>
144               <text representation="TXT" language="en-CA">HEART FAILURE*</text>
145               <statusCode code="active"/>
146               <effectiveTime>
147                 <low value="20130926"/>
148               </effectiveTime>
149               <value xsi:type="CD" nullFlavor="NI"/>
150               <author typeCode="AUT" contextControlCode="OP">
151                 <templateId root="2.16.840.1.113883.3.1818.10.4.2"/>
152                 <time value="20130926"/>
153                 <assignedAuthor classCode="ASSIGNED">
154                   <id root="2.16.840.1.113883.3.40.2.11" extension="cpsid" assigningAuthorityName="BC
    ↪   MSP Provider License Number"/>
155                   <assignedPerson classCode="PSN" determinerCode="INSTANCE">
156                     <name use="OR">
157                       <given>oscar</given>
158                       <family>oscardoc</family>
159                     </name>
160                   </assignedPerson>
161                 </assignedAuthor>
162               </author>
163               <entryRelationship typeCode="COMP" contextConductionInd="true">
164                 <templateId root="2.16.840.1.113883.3.1818.10.4.28"/>
165                 <observation classCode="OBS" moodCode="EVN">
166                   <code code="ICD9CODE" codeSystemName="ObservationType-CA-Pending"
    ↪   codeSystem="2.16.840.1.113883.3.3068.10.6.3"/>
167                   <value xsi:type="CD" code="428" codeSystem="2.16.840.1.113883.6.42"
    ↪   codeSystemName="ICD9" displayName="HEART FAILURE*"/>
168                 </observation>
169               </entryRelationship>
170               <entryRelationship typeCode="SUBJ" contextConductionInd="true">
171                 <templateId root="2.16.840.1.113883.3.1818.10.4.4"/>
172                 <observation classCode="OBS" moodCode="EVN">
173                   <code code="DATEOBS" codeSystemName="ObservationType-CA-Pending"
    ↪   codeSystem="2.16.840.1.113883.3.3068.10.6.3"/>
174                   <effectiveTime>
175                     <low value="20130926"/>
176                   </effectiveTime>
177                 </observation>
178               </entryRelationship>
179             </observation>
180           </entry>
181         </section>
182       </component>
183     </structuredBody>
184   </component>
185 </ClinicalDocument>
```

Figure A.8: Part 4/4 of an E2E document example for a test patient.

Bibliography

[1] Krzysztof Czarnecki et al. "Bidirectional Transformations: A Cross-Discipline Perspective". In: *Theory and Practice of Model Transformations*. Ed. by Richard F. Paige. Lecture Notes in Computer Science 5563. Springer Berlin Heidelberg, 2009, pp. 260–283. URL: http://link.springer.com/chapter/10.1007/978-3-642-02408-5_19.

[2] Roger Collier. "National Physician Survey: EMR use at 75%". en. In: *Canadian Medical Association Journal* 187.1 (Jan. 2015), E17–E18. ISSN: 0820-3946, 1488-2329. DOI: 10.1503/cmaj.109-4957. URL: http://www.cmaj.ca/cgi/doi/10.1503/cmaj.109-4957.

[3] R. Hillestad et al. "Can Electronic Medical Record Systems Transform Health Care? Potential Health Benefits, Savings, And Costs". en. In: *Health Affairs* 24.5 (Sept. 2005), pp. 1103–1117. ISSN: 0278-2715, 1544-5208. DOI: 10.1377/hlthaff.24.5.1103. URL: http://content.healthaffairs.org/cgi/doi/10.1377/hlthaff.24.5.1103.

[4] Sue Bowman. "Impact of Electronic Health Record Systems on Information Integrity: Quality and Safety Implications". In: *Perspectives in Health Information Management* 10.Fall (Oct. 2013). ISSN: 1559-4122. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3797550/.

[5] Yuan Zhou et al. "The impact of interoperability of electronic health records on ambulatory physician practices: a discrete-event simulation study". In: *Informatics in Primary Care* 21.1 (Feb. 2014), pp. 21–29. ISSN: 1475-9985, 1476-0320. DOI: 10.14236/jhi.v21i1.36. URL: http://hijournal.bcs.org/index.php/jhi/article/view/36.

[6] Charles N. Mead et al. "Data interchange standards in healthcare it-computable semantic interoperability: Now possible but still difficult. do we really need a better mousetrap?" In: *Journal of Healthcare Information*

Management 20.1 (2006), p. 71. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.8758&rep=rep1&type=pdf.

[7] Feng Chang and Nishi Gupta. "Progress in electronic medical record adoption in Canada". In: *Canadian Family Physician* 61.12 (Dec. 2015), pp. 1076–1084. ISSN: 0008-350X. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4677946.

[8] Morgan Price, Francis Lau, and James Lai. "Measuring EMR Adoption: A Framework and Case Study ElectronicHealthcare Vol. 10 No. 1 2011". In: *Longwoods Publication* (2011). URL: http://www.longwoods.com/content/22354.

[9] Gaby Loria. *Are Patients Ready for EHR Interoperability?* Sept. 2015. URL: http://www.softwareadvice.com/resources/address-ehr-interoperability-concerns/.

[10] *EMR FAQ for PSP RSTs Version 1.0 Nov 2015.pdf*. URL: https://divisionsbc.ca/CMSMedia/WebPageRevisions/PageRev-9977/EMR%20FAQ%20for%20PSP%20RSTs%20Version%201.0%20Nov%202015.pdf.

[11] *Certified EMRs*. URL: https://www.ontariomd.ca/portal/server.pt/community/certified_emrs/vendor_market_share/.

[12] Alan Brookstone. *CanadianEMR: EMR Comparisons, Ratings, Directory Services and Expert Discussions: Significant Changes for BC - PITO Program*. URL: http://blog.canadianemr.ca/canadianemr/2014/03/significant-changes-for-bc-pito-program.html.

[13] *HL7 Standards Product Brief - CDA® Release 2*. URL: http://www.hl7.org/implement/standards/product_brief.cfm?product_id=7.

[14] *HL7 Reference Information Model*. URL: http://www.hl7.org/implement/standards/rim.cfm.

[15] *ABOUT OSCAR — OSCAR Canada Users Society*. URL: http://oscarcanada.org/about-oscar/brief-overview/index_html#user-base.

[16] Morgan Price. *SCOOP Overview — SCOOP - UBC Primary Care Research Network*. URL: http://scoop.leadlab.ca/scoop-overview.

[17] *The Apache Velocity Project*. URL: https://velocity.apache.org/.

[18] Justin Fyfe. *MARC-HI Everest Framework for Java - Home*. CodePlex. URL: http://jeverest.codeplex.com/.

[19] Perdita Stevens. "Bidirectional model transformations in QVT: semantic issues and open questions". In: *Software & Systems Modeling* 9.1 (2010), pp. 7–20. URL: http://link.springer.com/article/10.1007/s10270-008-0109-9.

[20] Yingfei Xiong et al. "Synchronizing concurrent model updates based on bidirectional transformation". en. In: *Software & Systems Modeling* 12.1 (Feb. 2013), pp. 89–104. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-010-0187-3. URL: http://link.springer.com/10.1007/s10270-010-0187-3.

[21] Zinovy Diskin et al. "From state-to delta-based bidirectional model transformations: The symmetric case". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 304–318. URL: http://link.springer.com/chapter/10.1007/978-3-642-24485-8_22.

[22] Todd J. Green, Zachary G. Ives, and Val Tannen. "Reconcilable Differences". en. In: *Theory of Computing Systems* 49.2 (Aug. 2011), pp. 460–488. ISSN: 1432-4350, 1433-0490. DOI: 10.1007/s00224-011-9323-x. URL: http://link.springer.com/10.1007/s00224-011-9323-x.

[23] Zinovy Diskin. "Algebraic models for bidirectional model synchronization". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 21–36. URL: http://link.springer.com/chapter/10.1007/978-3-540-87875-9_2.

[24] James F. Terwilliger, Anthony Cleve, and Carlo A. Curino. "How Clean Is Your Sandbox?" In: *Theory and Practice of Model Transformations*. Ed. by Zhenjiang Hu and Juan de Lara. Lecture Notes in Computer Science 7307. Springer Berlin Heidelberg, 2012, pp. 1–23. URL: http://link.springer.com/chapter/10.1007/978-3-642-30476-7_1.

[25] Zhenjiang Hu et al. "Bidirectional Transformation" bx"(Dagstuhl Seminar 11031)." In: *Dagstuhl Reports* 1.1 (2011), pp. 42–67. URL: http://drops.dagstuhl.de/volltexte/2011/3144/.

[26] François Bancilhon and Nicolas Spyratos. "Update semantics of relational views". In: *ACM Transactions on Database Systems (TODS)* 6.4 (1981), pp. 557–575. URL: http://dl.acm.org/citation.cfm?id=319634.

[27] Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. "Three complementary approaches to bidirectional programming". In: *Generic and Indexed Programming*. Springer, 2012, pp. 1–46. URL: http://link.springer.com/chapter/10.1007/978-3-642-32202-0_1.

[28] Lambert Meertens. *Designing constraint maintainers for user interaction*. Citeseer, 1998. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.3250&rep=rep1&type=pdf.

[29] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. "Relational lenses: a language for updatable views". In: *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2006, pp. 338–347. URL: http://dl.acm.org/citation.cfm?id=1142399.

[30] Todd J. Green et al. "Update exchange with mappings and provenance". In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 675–686. URL: http://dl.acm.org/citation.cfm?id=1325929.

[31] Alon Y. Halevy et al. "Schema mediation in peer data management systems". In: *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 2003, pp. 505–516. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1260817.

[32] Nick Benton. "Embedded interpreters". In: *Journal of functional programming* 15.04 (2005), pp. 503–542. URL: http://journals.cambridge.org/abstract_S0956796804005398.

[33] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. "A programmable editor for developing structured documents based on bidirectional transformations". In: *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 2004, pp. 178–189. URL: http://dl.acm.org/citation.cfm?id=1014025.

[34] Umeshwar Dayal and Philip A. Bernstein. "On the correct translation of update operations on relational views". In: *ACM Transactions on Database Systems (TODS)* 7.3 (1982), pp. 381–416. URL: http://dl.acm.org/citation.cfm?id=319740.

[35] Ronald Fagin. "Inverting schema mappings". en. In: *ACM Transactions on Database Systems* 32.4 (Nov. 2007), 25–es. ISSN: 03625915. DOI: 10.1145/1292609.1292615. URL: http://portal.acm.org/citation.cfm?doid=1292609.1292615.

[36] N. C. Shu et al. "EXPRESS: A Data EXtraction, Processing, and Restructuring System". In: *ACM Trans. Database Syst.* 2.2 (June 1977), pp. 134–174. ISSN: 0362-5915. DOI: 10.1145/320544.320549. URL: http://doi.acm.org.ezproxy.library.uvic.ca/10.1145/320544.320549.

[37] Ariel Fuxman et al. "Peer data exchange". In: *ACM Transactions on Database Systems (TODS)* 31.4 (2006), pp. 1454–1498. URL: http://dl.acm.org/citation.cfm?id=1189778.

[38] Marcelo Arenas, Jorge Pérez, and Cristian Riveros. "The recovery of a schema mapping: bringing exchanged data back". In: *ACM Transactions on Database Systems (TODS)* 34.4 (2009), p. 22. URL: http://dl.acm.org/citation.cfm?id=1620589.

[39] Ekkart Kindler and Robert Wagner. *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios*. University of Paderborn, June 2007.

[40] Andy Schürr. "Specification of graph translators with triple graph grammars". In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 1994, pp. 151–163. URL: http://link.springer.com/chapter/10.1007/3-540-59071-4_45.

[41] Anthony Anjorin et al. "A Systematic Approach and Guidelines to Developing a Triple Graph Grammar". In: *Bidirectional Transformations* (2015), p. 81. URL: http://ceur-ws.org/Vol-1396/bx2015.pdf#page=85.

[42] Hartmut Ehrig. "Introduction to the algebraic theory of graph grammars (a survey)". In: *International Workshop on Graph Grammars and Their Application to Computer Science*. Springer, 1978, pp. 1–69. URL: http://link.springer.com/chapter/10.1007/BFb0025714.

[43] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. "Confluence of typed attributed graph transformation systems". In: *International Conference on Graph Transformation*. Springer, 2002, pp. 161–176. URL: http://link.springer.com/chapter/10.1007/3-540-45832-8_14.

[44] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. "Quotient Lenses". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. New York, NY, USA: ACM, 2008, pp. 383–396. ISBN: 978-1-59593-919-7. DOI: 10.1145/1411204.1411257. URL: http://doi.acm.org/10.1145/1411204.1411257.

[45] J. Nathan Foster et al. "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem". en. In: *ACM Transactions on Programming Languages and Systems* 29.3 (May 2007), 17–es. ISSN: 01640925. DOI: 10.1145/1232420.1232424. URL: http://portal.acm.org/citation.cfm?doid=1232420.1232424.

[46] Arif Wider. "Towards combinators for bidirectional model transformations in scala". In: *Software Language Engineering*. Springer, 2012, pp. 367–377. URL: http://link.springer.com/chapter/10.1007/978-3-642-28830-2_21.

[47] J. Nathan Foster et al. "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem". In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 233–246. URL: http://dl.acm.org/citation.cfm?id=1040325.

[48] Shinya Kawanaka and Haruo Hosoya. "biXid: A Bidirectional Transformation Language for XML". In: *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '06. New York, NY, USA: ACM, 2006, pp. 201–214. ISBN: 1-59593-309-3. DOI: 10.1145/1159803.1159830. URL: http://doi.acm.org/10.1145/1159803.1159830.

[49] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. "Dual Syntax for XML Languages". In: *Inf. Syst.* 33.4-5 (June 2008), pp. 385–406. ISSN: 0306-4379. DOI: 10.1016/j.is.2008.01.006. URL: http://dx.doi.org/10.1016/j.is.2008.01.006.

[50] Haruo Hosoya and Benjamin C. Pierce. "XDuce: A Typed XML Processing Language (Preliminary Report)". In: *The World Wide Web and Databases: Third International Workshop WebDB 2000 Dallas, TX, USA, May 18–19,2000 Selected Papers*. Ed. by Gerhard Goos et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 226–244. ISBN: 978-3-540-45271-3. URL: http://dx.doi.org/10.1007/3-540-45271-0_15.

[51] James Cheney. "FLUX: functional updates for XML". In: *ACM Sigplan Notices*. Vol. 43. ACM, 2008, pp. 3–14. URL: http://dl.acm.org/citation.cfm?id=1411209.

[52] Hugo Pacheco, Tao Zan, and Zhenjiang Hu. "BiFluX: A Bidirectional Functional Update Language for XML". In: *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. PPDP '14. New York, NY, USA: ACM, 2014, pp. 147–158. ISBN: 978-1-4503-2947-7. DOI: 10.1145/2643135.2643141. URL: http://doi.acm.org/10.1145/2643135.2643141.

[53] Anthony Anjorin, Erhan Leblebici, and Andy Schürr. "20 Years of Triple Graph Grammars: A Roadmap for Future Research". In: *Electronic Communications of the EASST* 73 (2016). URL: http://journal.ub.tu-berlin.de/eceasst/article/view/1031.

[54] Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. "Construction of integrity preserving triple graph grammars". In: *International Conference on Graph Transformation*. Springer, 2012, pp. 356–370. URL: http://link.springer.com/chapter/10.1007/978-3-642-33654-6_24.

[55] Holger Giese, Stephan Hildebrandt, and Leen Lambers. "Bridging the gap between formal semantics and implementation of triple graph grammars: Ensuring conformance of relational model transformation specifications and implementations". en. In: *Software & Systems Modeling* 13.1 (Feb. 2014), pp. 273–299. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-012-0247-y. URL: http://link.springer.com/10.1007/s10270-012-0247-y.

[56] Arend Rensink. "The edge of graph transformation—graphs for behavioural specification". In: *Graph transformations and model-driven engineering*. Springer, 2010, pp. 6–32. URL: http://link.springer.com/chapter/10.1007/978-3-642-17322-6_2.

[57] Fernando Orejas and Leen Lambers. "Symbolic attributed graphs for attributed graph transformation". In: *Electronic Communications of the EASST* 30 (2010). URL: http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/405.

[58] Leen Lambers et al. "Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case". In: *Electronic Communications of the EASST* 49 (2012). ISSN: 1863-2122. DOI: 10.14279/tuj.eceasst.49.706.714.

[59] Anthony Anjorin, Gergely Varró, and Andy Schürr. "Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques". In: *Electronic Communications of the EASST* 49 (2012). URL: https://pdfs.semanticscholar.org/b500/bf90a2d00040894da69c876eafe64ed20602.pdf.

[60] Martin Wieber, Anthony Anjorin, and Andy Schürr. "On the Usage of TGGs for Automated Model Transformation Testing." In: *ICMT* 8568 (2014), pp. 1–16. URL: http://link.springer.com/content/pdf/10.1007/978-3-319-08789-4.pdf#page=14.

[61] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. "Developing eMoflon with eMoflon". In: *International Conference on Theory and Practice of Model Transformations*. Springer, 2014, pp. 138–145. URL: http://link.springer.com/chapter/10.1007/978-3-319-08789-4_10.

[62] Stephan Hildebrandt et al. "A survey of Triple Graph Grammar Tools". In: *Electronic Communications of the EASST* 57 (2013). URL: http://jgreen.de/wp-content/documents/2013/bx13.pdf.

[63] Anthony Anjorin et al. "A Static Analysis of Non-confluent Triple Graph Grammars for Efficient Model Transformation". In: *Graph Transformation*. Springer, 2014, pp. 130–145. URL: http://link.springer.com/chapter/10.1007/978-3-319-09108-2_9.

[64] Perdita Stevens. "Bidirectionally Tolerating Inconsistency: Partial Transformations". In: *International Conference on Fundamental Approaches to Software Engineering*. Ed. by Stefania Gnesi and Arend Rensink. Springer, 2014, pp. 32–46. ISBN: 978-3-642-54804-8. DOI: 10.1007/978-3-642-54804-8_3. URL: http://link.springer.com/chapter/10.1007/978-3-642-54804-8_3.

[65] Meng Wang et al. "Translucent Abstraction: Safe Views through Invertible Programming". In: (2010). URL: https://www.cs.ox.ac.uk/files/2280/total.pdf.

[66] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. "Principles of a reversible programming language". In: *Proceedings of the 5th conference on Computing frontiers*. ACM, 2008, pp. 43–54. URL: http://dl.acm.org/citation.cfm?id=1366239.

[67] Kazutaka Matsuda et al. "Bidirectionalization transformation based on automatic derivation of view complement functions". In: *ACM SIGPLAN Notices*. Vol. 42. ACM, 2007, pp. 47–58. URL: http://dl.acm.org/citation.cfm?id=1291162.

[68] Kazutaka MATSUDA et al. "Bidirectionalizing Programs with Duplication through Complementary Function Derivation". In: *Computer Software* 26.2 (2009), 2_56–2_75. DOI: 10.11309/jssst.26.2_56.

[69] Janis Voigtländer. "Bidirectionalization for free! (Pearl)". In: *ACM SIGPLAN Notices*. Vol. 44. ACM, 2009, pp. 165–176. URL: http://dl.acm.org/citation.cfm?id=1480904.

[70] John C. Reynolds. "Types, Abstraction and Parametric Polymorphism". In: *IFIP Congress*. 1983, pp. 513–523. URL: ftp://ftp.cs.cmu.edu/user/jcr/typesabpara.pdf.

[71] Philip Wadler. "Theorems for Free!" In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. New York, NY, USA: ACM, 1989, pp. 347–359. ISBN: 0-89791-328-0. DOI: 10.1145/99370.99404. URL: http://doi.acm.org/10.1145/99370.99404.

[72] Aaron Bohannon et al. "Boomerang: resourceful lenses for string data". In: *ACM SIGPLAN Notices*. Vol. 43. ACM, 2008, pp. 407–419. URL: http://dl.acm.org/citation.cfm?id=1328487.

[73] Alon Y. Levy. "Logic-based techniques in data integration". In: *Logic-based artificial intelligence*. Ed. by Jack Minker. Norwell, MA, USA: Kluwer Academic Publishers, 2000, pp. 575–595. ISBN: 0-7923-7224-7. URL: http://dl.acm.org/citation.cfm?id=566344.566383.

[74] P. McBrien and A. Poulovassilis. "Data integration by bi-directional schema transformation rules". In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. Mar. 2003, pp. 227–238. DOI: 10.1109/ICDE.2003.1260795.

[75] Sergey Melnik, Atul Adya, and Philip A. Bernstein. "Compiling mappings to bridge applications and databases". en. In: *ACM Transactions on Database Systems* 33.4 (Nov. 2008), pp. 1–50. ISSN: 03625915. DOI: 10.1145/1412331.1412334. URL: http://portal.acm.org/citation.cfm?doid=1412331.1412334.

[76] James F. Terwilliger, Lois M.L. Delcambre, and Judith Logan. "Querying through a user interface". en. In: *Data & Knowledge Engineering* 63.3 (Dec. 2007), pp. 774–794. ISSN: 0169023X. DOI: 10.1016/j.datak.2007.04.007. URL: http://linkinghub.elsevier.com/retrieve/pii/S0169023X07000730.

[77] International Atomic Energy Agency, ed. *Investigation of an accidental exposure of radiotherapy patients in Panama: report of a team of experts, 26 May-1 June 2001*. Vienna: International Atomic Energy Agency, 2001. ISBN: 978-92-0-101701-7.

[78] Youngsul Shin, Yunja Choi, and Woo Jin Lee. "Integration testing through reusing representative unit test cases for high-confidence medical software". en. In: *Computers in Biology and Medicine* 43.5 (June 2013), pp. 434–443. ISSN: 00104825. DOI: 10.1016/j.compbiomed.2013.01.024. URL: http://linkinghub.elsevier.com/retrieve/pii/S0010482513000450.

[79] Renée J. Miller, Yannis E. Ioannidis, and Raghu Ramakrishnan. "The Use of Information Capacity in Schema Integration and Translation". In: *Proceedings of the 19th International Conference on Very Large Data Bases*. VLDB '93. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 120–133. ISBN: 1-55860-152-X. URL: http://dl.acm.org/citation.cfm?id=645919.672515.

[80] Thomas McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering*. Vol. SE-2. 4. IEEE, 1976, pp. 308–320.

[81] *Model Driven Health Tools — projects.eclipse.org*. URL: https://projects.eclipse.org/projects/modeling.mdht.

[82] *scoophealth/oscar: OSCAR EHR Mirror*. URL: https://github.com/scoophealth/oscar.

[83] *leiningen/MIXED_PROJECTS.md at master · technomancy/leiningen*. URL: https://github.com/technomancy/leiningen/blob/master/doc/MIXED_PROJECTS.md.

[84] *polymatheia - Advanced Clojure and Java Mixing with Leiningen*. URL: http://hypirion.com/musings/advanced-intermixing-java-clj.

[85] *Clojure vs Java (64-bit Ubuntu quad core) — Computer Language Benchmarks Game*. URL: https://benchmarksgame.alioth.debian.org/u64q/clojure.html.

[86] Raymond Gallardo. *JDK 8 Documentation - Developer Preview Release (The Java Tutorials Blog)*. Sept. 2013. URL: https://blogs.oracle.com/thejavatutorials/entry/jdk_8_documentation_developer_preview.

[87] A. Ward and D. Deugo. "Performance of Lambda Expressions in Java 8". In: *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015, p. 119. URL: http://search.proquest.com/openview/5c4c7b777820896b4455b620338e91a2/1.pdf?pq-origsite=gscholar&cbl=1976341.

[88] *Stream (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.

[89] Alex Zhitnitsky. *How Java 8 Lambdas and Streams Can Make Your Code 5 Times Slower — Takipi Blog*. Nov. 2015. URL: http://blog.takipi.com/benchmark-how-java-8-lambdas-and-streams-can-make-your-code-5-times-slower/.

[90] Ben Evans. *How Functional is Java 8?* May 2014. URL: https://www.infoq.com/articles/How-Functional-is-Java-8.

[91] *ImmutablePair (Apache Commons Lang 3.5-SNAPSHOT API)*. URL: https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/tuple/ImmutablePair.html.

[92] Connie U. Smith and Lloyd G. Williams. "Software performance engineering: A case study including performance comparison with design alternatives". In: *IEEE Transactions on software engineering* 19.7 (1993), p. 720. URL: http://search.proquest.com/openview/a2159ef335d73720af98129ebe298545/1?pq-origsite=gscholar.

[93] Insup Lee et al. "High-confidence medical device software and systems". In: (2006). URL: https://works.bepress.com/george_pappas/291/.

[94] Pierre Bourque, R. E Fairley, and IEEE Computer Society. *Guide to the software engineering body of knowledge*. English. OCLC: 926093687. 2014. URL: http://www.computer.org/portal/web/swebok/swebokv3.

[95] Maryam Zahabi, David B. Kaber, and Manida Swangnetr. "Usability and Safety in Electronic Medical Records Interface Design A Review of Recent Literature and Guideline Formulation". In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 57.5 (2015), pp. 805–834. URL: http://hfs.sagepub.com/content/57/5/805.short.

[96] Barry Boehm and Victor R. Basili. "Software defect reduction top 10 list". In: *Computer* 34.1 (2001), pp. 135–137. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=962984.

[97] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN: 0444002057.

[98] Frederick P. Brooks Jr. "No Silver Bullet Essence and Accidents of Software Engineering". In: *Computer* 20.4 (Apr. 1987), pp. 10–19. ISSN: 0018-9162. DOI: 10.1109/MC.1987.1663532. URL: http://dx.doi.org/10.1109/MC.1987.1663532.

[99] Antonia Bertolino. "Software Testing Research and Practice". In: *Proceedings of the Abstract State Machines 10th International Conference on Advances in Theory and Practice*. ASM'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 1–21. ISBN: 3-540-00624-9. URL: http://dl.acm.org/citation.cfm?id=1754749.1754751.

[100] Mark Bull et al. “A benchmark suite for high performance Java”. In: *Concurrency - Practice and Experience* 12.6 (2000), pp. 375–388. URL: https://pdfs.semanticscholar.org/f981/f644d7f3125604775b0854da3b044b741b1f.pdf.

[101] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. “Using benchmarking to advance research: A challenge to software engineering”. In: *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 74–83. URL: http://dl.acm.org/citation.cfm?id=776826.

[102] Stephan Rehfeld, Henrik Tramberend, and Marc Erich Latoschik. “Profiling and benchmarking event- and message-passing-based asynchronous realtime interactive systems”. en. In: ACM Press, 2014, pp. 151–159. ISBN: 978-1-4503-3253-8. DOI: 10.1145/2671015.2671031. URL: http://dl.acm.org/citation.cfm?doid=2671015.2671031.

[103] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically rigorous java performance evaluation”. In: *ACM SIGPLAN Notices* 42.10 (2007), pp. 57–76. URL: http://dl.acm.org/citation.cfm?id=1297033.

[104] Albert Flaig, Daniel Hertl, and Florian Krüger. “Evaluation von Java-Profiler-Werkzeugen”. In: *05 Fakultät Informatik, Elektrotechnik und Informationstechnik* (2013). DOI: 10.18419/opus-3222. URL: http://elib.uni-stuttgart.de/handle/11682/3239.

[105] *VisualVM - Home*. URL: https://visualvm.java.net/.

[106] *Oracle Java Mission Control*. URL: http://download.oracle.com/technology/products/missioncontrol/updatesites/base/5.2.0/eclipse/.

[107] *Oracle - About Java Flight Recorder*. URL: https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm.

[108] Jeremy Ho. *jujaga/bxe2e*. GitHub. URL: https://github.com/jujaga/bxe2e.