Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Master's Thesis Nr. 478

Systems Group, Department of Computer Science, ETH Zurich

Brel - A python library for XBRL

by

Robin Schmidiger

Supervised by

Prof. Gustavo Alonso, Dr. Ghislain Fourny

September 2023 – March 2024

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

**Authored by** (in block letters):
*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| | |
| | |
| | |
| | |
| | |

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| | |
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*

# Contents

# Chapter 1

# Preface

This thesis was written as part of Robin Schmidiger's master's degree in computer science at ETH Zurich. It was supervised by Prof. Gustavo Alonso and Dr. Ghislain Fourny.
Both the thesis and the source code of the library are available on GitHub[26]
The contents of this thesis are based heavily on both the XBRL standard[9] created by Charles Hoffman and "The XBRL Book"[7] by Ghislain Fourny.

# Abstract

TODO: Write the abstract

# Chapter 2

# Introduction

## 2.1 Introduction

In the era of data-driven decision making, the ability to efficiently interpret and analyze business reports is becoming increasingly important. Approximately 20 years ago, the predominant medium for publishing business reports was paper. This format posed challenges for automated processing of reports by computers. Nevertheless, the advent of the eXtensible Business Reporting Language (XBRL) in 2000 marked a significant shift in this domain[22]. XBRL is a standardized format for business reports that is both machine-readable and can produce human-readable reports. Initially conceptualized for financial reporting, XBRL is now used in a wide variety of business reports.[18] While early iterations of XBRL were exclusively based on XML, subsequent developments have enabled its compatibility with additional formats such as JSON and CSV.

Previously, XBRL was a specialized technology utilized by a select group of companies. Presently, XBRL is witnessing increased adoption across both public and private sectors. Regulatory bodies such as the US Securities and Exchange Commission (SEC)[32] and the European Banking Authority (EBA)[2] are progressively mandating the submission of reports in XBRL format. In the corporate domain, entities like JP Morgan Chase, Microsoft, and Hitachi are leveraging XBRL to streamline their financial reporting mechanisms.[4]

Nonetheless, XBRL encompasses certain complexities, many of which are stemming from historical design choices. Moreover, despite XBRL's evolution beyond its initial XML foundations, the association with XML format persists.

Given XBRL's primary audience of non-technical users, its accessibility is crucial.

However, the current state of XBRL does not entirely meet this requirement. This situation has led to the emergence of a varied range of XBRL tools, most of which are proprietary and not freely available. Although there are some open-source alternatives, they are relatively few in number and often limited in scope. One significant exception is Arelle[23], a well-known tool in the XBRL field.

In 2021, XBRL International published a new specification termed Open Information Model (OIM)[19]. The OIM is a logical data model for XBRL reports that is independent from the XBRL syntax. One objective of the OIM is to make XBRL more approachable for both developers and non-technical users by iterating on XBRL's design. The OIM is not yet finalized and does not cover all aspects of XBRL.

## 2.2 Goals of this thesis

The goal of this thesis is to create a new open source XBRL library that is based on the XBRL standard, notably the OIM. The library is called `Brel` (short for Business Reporting Extensible Library) and is written in the Python programming language. It should be easy to use and well documented. Brel should provide a simple python API that allows developers to easily read XBRL reports and extract information from them. Fundamentally, the library should act as a pythonic wrapper around all elements of an XBRL report. Lastly, the library should support XBRL reports in XML, but its design should be extensible to support other formats in the future. The research questions that this thesis aims to answer are:

- **RQ1** How can the OIM be translated into an easy-to-use python API?
- **RQ2** How can the non-OIM sections of XBRL be converted into an easy-to-use python API that is consistent with the OIM?
- **RQ3** How can the library be designed to support multiple formats in the future?

## 2.3 Limitations of this thesis

The XBRL standard has grown in size and complexity since its inception in 2000. In its current form, implementing a complete XBRL library is not feasible within the scope of a single thesis. Therefore, Brel will have to make some compromises.
The first limitation of this thesis is that the library will only support XBRL reports in XML format. Secondly, the library will only support reading XBRL reports, not creating or modifying them. Third, Brel will not semantically validate XBRL reports.
While the initial two limitations are straightforward, the third limitation necessitates further clarification. XBRL reports can be interpreted in terms of syntax and semantics, similar to the source code of a program. An XBRL report that is syntactically accurate adheres to the XBRL specifications, yet it may not always be logically coherent [1]. A semantically correct report is both syntactically correct and logically coherent. This thesis will not address the semantics of XBRL reports, as they are rarely detailed in the XBRL specification, but are instead outlined in supporting documents.
Nonetheless, semantic validation of XBRL reports can be achieved through the use of the Brel API.

## 2.4 Structure of this thesis

Grasping the underlying XBRL standard is essential to understand how Brel provides a pythonic API for XBRL reports. Hence, Chapter 4 will offer a concise introduction to XBRL. The chapter will present the fundamental concepts of XBRL within the framework of the OIM, followed by an exploration of the non-OIM aspects of XBRL. This chapter will focus on the concepts relevant to this thesis, rather than delving deep into the technical specifics of the XBRL standard.
Subsequently, Chapter 5 will introduce the API of Brel. This chapter, which forms part of the thesis results, is positioned prior to the implementation of the API. The reason for this deviation from the conventional structure is that it is more logical to

[1] The XBRL specification does sometimes branch out into the realm of semantics. Brel ignores these parts of the specification.

introduce the API before its implementation. The API chapter will answer research question 2.2 and 2.2.
Chapter 6 details the implementation of the Brel API, linking the API discussed in Chapter 5 with the XBRL standards explored in Chapter 4. While aiming for a comprehensive overview, the chapter will particularly emphasize the more involved aspects of the implementation. Additionally, the chapter will explore the design of the library and its capability to accommodate different formats in future iterations, thereby addressing Research Question 2.2.
Chapter 7, known as the Results chapter, evaluates the library's effectiveness in meeting the goals set out in the introduction. This chapter offers an in-depth examination of Brel's alignment with different XBRL conformance suites. Additionally, it showcases real-world uses of the library, highlighting how Brel can be employed for reading and verifying XBRL reports.
Chapter 8, the concluding chapter, will reiterate the main discoveries of this thesis. It will also offer insights into possible avenues for further research and development related to Brel.

# Chapter 3

# Related work

The goal of this thesis is to create a new open-source XBRL library, primarily based on the OIM. In the context of Brel, three main areas of interest exist. The first area involves examining the XBRL specification and its interpretations. The second encompasses the examination of other XBRL libraries and platforms similar to Brel, including public databases of XBRL reports. The third area covers reviewing the requirements set by authorities and other organizations that XBRL processors must fulfill.

## 3.1 XBRL Specification

As indicated in Section 2, this thesis builds upon the XBRL standard originally created by Charles Hoffman[9]. The XBRL standard comprises numerous components, and this thesis specifically focuses on the following components: the Open Information Model (OIM)[19], the XBRL 2.1 specification[12], the extension for dimensional reporting[13], and the specification for generic links[14]. Chapter 4 will delve into these components in detail. In addition to the XBRL standard, Charles Hoffman manages a personal website[8], which offers information about XBRL and good practices for its utilization.

## 3.2 The XBRL Book

Understanding the XBRL specification requires a good grasp of both XML and XBRL. To help newcomers, Dr. Ghislain Fourny has authored "The XBRL Book" [7], which serves as a comprehensive guide to XBRL. This book covers all important aspects of the XBRL standard, including the relatively recent OIM, making it an invaluable resource for those looking to learn about XBRL.

## 3.3 Arelle

Arelle[23] is an open-source XBRL platform. As of the current writing, Arelle holds the distinction of being the most comprehensive open-source platform in its category. It provides support for all features found in the XBRL 2.1 specification and the OIM. Similar to Brel, Arelle is implemented in Python and is available as open-source software. However, it's important to note that Arelle is a complete XBRL platform, in contrast to Brel, which is primarily a Python library. The reader can think of Arelle as the "Excel for XBRL."

## 3.4 Xule

Xule[34] serves as a rule language tailored for XBRL. This declarative language empowers users to create rules for the validation of XBRL reports. The Arelle project employs Xule to validate reports. While Xule is not directly incorporated into this thesis, it is worth noting that Brel has the potential to offer support for Xule in the future.

## 3.5 EDGAR

The U.S. Securities and Exchange Commission, known as the SEC [1], operates a system referred to as EDGAR (Electronic Data Gathering, Analysis, and Retrieval) [30]. This EDGAR system serves as a public repository for XBRL reports submitted to the SEC[2].

## 3.6 ESEF filings

The European counterpart to the SEC is ESMA, the European Securities and Markets Authority, which operates a database containing ESEF filings [27]. ESEF stands for European Single Electronic Format, a standardized format for XBRL reports within the European Union.
Much like EDGAR, the ESEF database is accessible to the public[3]. An interesting aspect of this database is its hosting by XBRL International, the organization responsible for maintaining the XBRL standard.

## 3.7 SEC - Interactive Data Public Test Suite

In order to verify the compliance of XBRL processors utilized by companies for generating XBRL reports, the SEC established the Interactive Data Public Test Suite [29]. This comprehensive test suite comprises a vast assortment of XBRL reports designed to assess the performance of XBRL processors. It is noteworthy that the SEC offers this test suite at no cost, and it can be accessed on their official website.
Brel does not employ the aforementioned test suite to evaluate its XBRL processor. The majority of the test cases within the suite focus on functionalities that are unique to EDGAR or on features that Brel presently does not support.

## 3.8 ESMA Conformance Suite

ESMA, the European Securities and Markets Authority, administers a test suite designed for XBRL processors, as well [28]. This test suite shares similarities with the SEC's Interactive Data Public Test Suite, albeit being under the administration of a different regulatory authority. One notable divergence is that the ESMA Conformance Suite is tailored to facilitate automated testing of xHTML reports.
As of the present, Brel does not possess the capability to support xHTML reports, rendering the ESMA Conformance Suite irrelevant to the scope of this thesis. Nonetheless, it is important to acknowledge that there are intentions for Brel to incorporate support for xHTML reports in the future.

[1] not to be confused with the U.S.SEC, which stands for U.S. Soybean Export Council
[2] `https://www.sec.gov/edgar/search/`
[3] `https://filings.xbrl.org/`

# Chapter 4

# XBRL

## 4.1 Overview

The content of this thesis is largely based on the XBRL standard[9] created by Charles Hoffman and Dr. Ghislain Fourny's interpretation of it in *the XBRL Book*[7]. Since the thesis builds on the foundation laid by the two, it is important to understand the ground work that they have done. This chapter will give a brief introduction to XBRL.

In essence, XBRL is a standardized format for representing reports [1]. After all, XBRL stands for **eXtensible Business Reporting Language**.[9]

As Ghislain Fourny has put it in *the XBRL Book* [7]:

> "If XBRL could be summarized in one single definition, it would be this:
> XBRL is about reporting facts."

Keeping this in mind, the subsequent sections will first introduce the basic concepts of XBRL, namely facts, concepts and QNames. Subsequently, the discussion will shift to more involved concepts that put facts and other elements into relation with each other, specifically through roles, networks, and report elements. The initial segment aligns with the OIM framework, whereas the latter part delves into aspects of XBRL not covered by the OIM.

Armed with this fundamental knowledge about XBRL, you will then be able to understand how Brel implements the core parts of the standard and how it hides a lot of the complexity of XBRL behind a Python API.

This chapter will not cover the XBRL specification in its entirety. It will also gloss over a lot of the details of the specification. It is more focused on giving the reader a high-level overview of the contents of the XBRL specification.

Moreover, understanding many XBRL concepts requires familiarity with other XBRL concepts. Circular dependencies among these concepts complicate their explanation in a sequential manner. Therefore, this chapter will revisit certain concepts that it has already introduced, aiming to progressively acclimate the reader to the more intricate aspects of XBRL.

[1] Both the term "report" and "filing" are used to describe the documents that are represented in XBRL and are used interchangeably in this thesis.

## 4.2 Facts

A fact is the smallest unit of information in an XBRL report. The word "Fact" is a term used to describe an individual piece of financial of business information within an XBRL instance document. This section aims to represent facts and its supporting concepts in a way that is in line with the OIM.
Let's examine a simplified case focusing on the financial report of Microsoft Corporation for the fiscal year 2022. Microsoft's annual report is accessible on the company's investors page[2]. This document provides extensive details about Microsoft's financial health and its business operations. In this scenario, we will solely focus on the company's revenue for the fiscal year 2022, as reported by Microsoft in the following manner:

Microsoft | Annual Report 2022

SUMMARY RESULTS OF OPERATIONS

| (In millions, except percentages and per share amounts) | | 2022 | | 2021 |
|---|---|---|---|---|
| Revenue | $ | 198,270 | $ | 168,088 |
| Gross margin | | 135,620 | | 115,856 |
| Operating income | | 83,383 | | 69,916 |
| Net income | | 72,738 | | 61,271 |
| Diluted earnings per share | | 9.65 | | 8.05 |
| Adjusted net income (non-GAAP) | | 69,447 | | 60,651 |
| Adjusted diluted earnings per share (non-GAAP) | | 9.21 | | 7.97 |

Figure 4.1: Microsoft's summary results of operations for the fiscal year 2021 and 2022

[6]

This table displays multiple facts about Microsoft for the fiscal years 2021 and 2022, as indicated by the horizontal axis. The vertical axis outlines the subjects of these facts, with the term **concept** used to describe what each fact reports. Values for concepts such as "Revenue", "Gross Margin", "Operating Income", etc., are presented for both fiscal years. To summarize, the table showcases 14 facts across 7 concepts for two fiscal years.
For the moment, our attention will be on the top left fact, which details the company's revenue for the fiscal year 2022. In XBRL terminology, this specific fact would be represented in the following way:

- **Concept:** Revenue
- **Entity:** Microsoft Corporation
- **Period:** from 2022-04-01 to 2023-03-31 [3]
- **Unit:** USD
- **Value:** 198'270'000 [4]

In this example:

- The **concept** refers *what* is being reported. In this case, "Revenue" signifies the fact pertains to the company's revenue figures.

[2]https://www.microsoft.com/investor/reports/ar22/index.html
[3]Refers to the fiscal year 2022, which starts on April 1, 2022 and ends on March 31, 2023
[4]https://www.microsoft.com/investor/reports/ar22/index.html

- The **entity** points to *who* is reporting. For our example, the entity is "Microsoft Corporation". Though implicitly understood from Microsoft's annual report, the entity of a fact must be explicitly mentioned in an XBRL report.
- The **period** specifies *when* the information is being reported, defined here as the fiscal year 2022, as shown by the "2022" column heading.
- The **unit** clarifies *how* the information is presented, with "USD" indicating the figures are in US dollars, marked by the dollar symbol $ in the table.
- The **value** reveals *how much* is being reported, with Microsoft's 2022 annual report stating the company's revenue for the fiscal year 2022 as approximately 198 billion US dollars.

The concept, entity, period, and unit associated with a fact are referred to as its **aspects**. If necessary, facts can be further detailed through **dimensions**, which are additional aspects that will be elaborated on in section 4.8. Aspects that are not dimensions are known as **core aspects**. Contrary to what the term might imply, not all core aspects are compulsory for a fact to have. The only mandatory core aspect is the concept.

## 4.3 Concepts

In section 4.2, we learned that a fact is the smallest unit of information in an XBRL report. The central aspect of a fact is its concept, which details the subject matter of the reported information. For instance, if a fact conveys data regarding a company's revenue, then "Revenue" is the concept associated with this fact. This section aims to delve deeper into concepts and their specification within XBRL. Concepts are essential components of XBRL, outlined within what is known as the **taxonomy**.

### 4.3.1 Taxonomy

In essence, the XBRL taxonomy is a collection of concepts and the relationships between them. It differs from the XBRL instance document, which holds the report's actual data. Each XBRL report outlines its taxonomy within a taxonomy schema file, referred to as the **extension taxonomy**.
This extension taxonomy contains references to other taxonomies, which, in turn, may link to additional taxonomies. Hence, when a report and its extension taxonomy are loaded into memory, the entire span of referenced taxonomies is also loaded. The transitive closure of all these references is called the **DTS** (short for **D**iscoverable **T**axonomy **S**et).
It's important to note that most taxonomies within the DTS are not stored locally with the report. Rather, they are hosted online and fetched as required.
Commonly encountered taxonomies in a report's DTS include `us-gaap`[5], which contains concepts for US Generally Accepted Accounting Principles (GAAP), `dei`[6], which contains concepts for the SEC's Document and Entity Information (DEI) requirements, `iso4217`[7], which contains concepts for currency codes, and many more.
Given that many DTSs from various reports share numerous identical taxonomies, it is practical to store these taxonomies locally, rather than re-downloading them each time they are required.

[5] https://xbrl.us/us-gaap/
[6] https://www.sec.gov/info/edgar/dei-2019xbrl-taxonomy
[7] https://www.iso.org/iso-4217-currency-codes.html

### 4.3.2 Concepts

Within the DTS, every concept is designated by a **QName**. The complexities of QNames will be explored in section 4.4. For the moment, the reader can think of them as a unique identifier for each concept. Typically, the QName of a concept is designed to be both human-readable and self-explanatory. Nevertheless, it's common for accountants and business analysts to employ elaborate naming conventions. Here are some examples of QNames for concepts:

- `us-gaap:Assets`
- `ko:IncrementalTaxAndInterestLiability`
- `dei:EntityCommonStockSharesOutstanding`
- `us-gaap:ElementNameAndStandardLabelInMaturityNumericLowerEndTo-NumericHigherEndDateMeasureMemberOrMaturityGreaterThanLowEnd-NumericValueDateMeasureMemberOrMaturityLessThanHighEndNumeric-ValueDateMeasureMemberFormatsGuidance`

Concepts in XBRL extend beyond merely possessing a QName. They also impose restrictions on certain aspects and values of the facts that reference these concepts. Using our ongoing example from section 4.2, the concept `us-gaap:Revenue` sets forth specific constraints.
It restricts the value to a `monetaryItemType`, which mandates that the fact's value must be numerical, as opposed to any arbitrary string.[8] It further limits the unit to currencies recognized by the ISO 4217 standard.[1] Moreover, monetary facts must be identified as either "debit" or "credit" through the `balance` attribute. In the context of `us-gaap:Revenue`, this stipulation means the fact should reflect a "debit" balance, attributing revenue as an asset.
The concept `us-gaap:Revenue` also specifies that the fact's period should be of the "duration" type. This indicates that the period must span a certain time frame, such as a fiscal year or quarter, or it could be of the "instant" type, referring to a specific moment in time.
This concludes our exploration of concepts in XBRL. The forthcoming section will delve into QNames and their role within XBRL, complementing our understanding of concepts and facts to solidify our grasp of XBRL's fundamental components.

## 4.4 QNames

Even though Brel, the XBRL processor, aims to simplify the user's interaction by abstracting away the complexity of XML, it retains a crucial element of XML within its API: QNames.
QNames serve as unique identifiers for XML elements or attributes, comprising three components: a namespace prefix, a namespace URI, and a local name. The prefix is a shorthand representation of the namespace URI, which we will refer to as a namespace binding.
For instance, the QName `us-gaap:Assets` signifies the element `Assets` within the `us-gaap` namespace. In this example, the namespace prefix `us-gaap` stands in for the namespace URI `https://xbrl.fasb.org/us-gaap/2022/elts/us-gaap-2022.xsd`,

[8]The `monetaryItemType` encompasses additional restrictions beyond being numerical, but these will be set aside for now.

Figure 4.2: The us-gaap:Assets QName

- Namespace prefix: `us-gaap`
- Namespace URI: `https://xbrl.fasb.org/us-gaap/2022/elts/us-gaap-2022.xsd`
- Local name: `Assets`

Since QNames offer a robust and straightforward method for identifying elements, Brel employs them in its API. However, there is one important difference between QNames in Brel and QNames in XBRL: Currently, most XBRL filings are based on XML, where namespace bindings are defined on a per-element basis. Therefore, the namespace bindings form a hierarchical structure, where the namespace binding of a QName depends on the location of the QName.
Brel takes a different approach, employing a fixed, global mapping from namespace prefixes to namespace URIs. This decision was made to simplify the API. Further details about this mapping will be provided in section 6.3.

## 4.5 Transition to Advanced XBRL Concepts

With the completion of our discussion on QNames, we have set the groundwork necessary for generating functional, albeit limited, XBRL reports.
A notable limitation at this stage is the lack of structure among the facts in a report, resulting in a collection of facts without any inherent organization. Since facts are not interrelated, it is impossible to verify if the values within the report are consistent.
Furthermore, the current state of XBRL is not particularly user-friendly. For instance, the use of QNames for naming concepts, while generally human-readable, results in verbosity. Additionally, the predominantly English nature of QNames poses challenges for non-English speakers in understanding the report.
The upcoming sections aim to address these challenges by introducing the more sophisticated aspects of XBRL, with a focus on networks. These advanced topics extend beyond the Open Information Model (OIM) and delve into the more traditional, XML-based aspects of XBRL.

## 4.6 Networks

Networks in XBRL are used to represent these relationships between concepts, facts and other elements of an XBRL filing.
For instance, the concepts `Assets` and `Liabilities` are interconnected, as both are part to the `Balance Sheet`. Moreover, the `Assets` concept is subdividable into `Current Assets` and `Non-Current Assets`. Such relationships can be depicted through a directed graph, as illustrated in figure 4.3.

Figure 4.3: Example of a relations between concepts in XBRL

```
Balance Sheet
├── Assets
│   ├── Current Assets
│   └── Non-Current Assets
└── Liabilities
```

A reader with a foundational knowledge of mathematics will identify the aforementioned example as a directed acyclic graph (DAG), more specifically, a tree. Graphs are a widespread method for illustrating relationships among entities. XBRL often uses the terms `Extended Link` or `Link` when discussing networks or their components, whereas Brel will consistently use the term `network` to avoid ambiguity. We exclusively use the term `Link` to refer to a specific element within the XBRL specification, XBRL groups multiple links into entities known as `linkbases`.

### 4.6.1 Types of Networks

The XBRL 2.1 specification defines six built in types of networks[17][9]:

- `link:presentationLink`: This network depicts the hierarchy of concepts. Figure 4.3 illustrates this.
- `link:calculationLink`: This network shows the calculation relationships between concepts. For instance, as shown in Figure 4.3, the `Assets` concept results from adding `Current Assets` to `Non-Current Assets`.
- `link:definitionLink`: This network outlines connections beyond those in other networks. The `definitionLink` is detailed in Section 4.6.5.
- `link:labelLink`: A network that associates report elements with human-readable labels.
- `link:referenceLink`: This network ties report elements to external references. For instance, the concept `Total Shareholder Return Amount` might have an official definition in the SEC's Code of Federal Regulations (CFR). The reference network would link the concept to the subparagraph `17 CFR 229.402(v)(2)(iv)`[33].
- `link:footnoteLink`: This network associates report elements, facts, and other components with explanatory footnotes.

XBRL refers to built-in networks as `standard extended links`. If necessary, XBRL permits the creation of user-defined networks, known as `custom extended`

[9] The XBRL 2.1 specification is inconsistent about `link:footNotelink`. Section 1.4 does not list it as a standard extended link, section 3.5.2.4 does. I will assume that it is a standard extended link.

`links`[17]. XBRL does permit both directed and undirected cycles within networks. Yet, networks within XBRL predominantly take the form of directed acyclic graphs (DAGs).
Although labels will receive a more thorough examination in Section 4.6.6, they will be used throughout this chapter for the sake of readability. Labels assign human-readable descriptions to report elements. For instance, the concept `us-gaap:CurrentAssets` may be labeled as "Current Assets" in English. The following sections will delve into the conceptual implementation of networks within XBRL.

### 4.6.2 presentationLink

The `link:presentationLink` network represents concept hierarchies within a report. I will offer a more detailed exploration of presentationLinks than other networks, as the implementation of all other network types mirrors that of presentationLinks.
XBRL structures its networks through a sequence of directed edges, referred to as `arcs`. Each arc possesses a source and a target, with duplicate arcs being prohibited. Referring to the illustration in figure 4.3, the presentationLink network would manifest as the following list of edges:

Figure 4.4: Example of a presentationLink network in edge list format

```
Balance Sheet -> Assets
Assets -> Current Assets
Assets -> Non-Current Assets
Balance Sheet -> Liabilities
```

In Figure 4.4, every arc is denoted by a `link:presentationArc` element within XBRL. PresentationLinks, in addition to presentationArcs, include "locators" (`link:loc`) indicating network nodes. For presentation networks, locators point to XBRL taxonomy concepts. Referring to the example in Figure 4.4, the arc from `Balance Sheet` to `Assets` is represented in XML as follows:

```xml
<link:loc
    xlink:type="locator"
    xlink:href="file_1.xsd#BalanceSheet"
    xlink:label="BalanceSheet_loc"
/>
<link:loc
    xlink:type="locator"
    xlink:href="file_1.xsd#Assets"
    xlink:label="Assets_loc"
/>
<link:presentationArc
    xlink:type="arc"
    xlink:arcrole="http://www.xbrl.org/2003/arcrole/parent-child"
    xlink:from="BalanceSheet_loc"
    xlink:to="Assets_loc"
    order="1"
/>
```

Figure 4.5: Example of a presentationArc in XML syntax

The XML snippet in Figure 4.5 displays two locators and one arc, where the locators represent the nodes linked by the arc, corresponding to the `BalanceSheet` and `Assets` concepts. The arc signifies the connection between these two nodes.
The breakdown of the XML snippet 4.5 is as follows:

- **Type**: The attribute `xlink:type` defines the roles of locators and the arc, labeling locators as `locator` and the arc as `arc`.
- **Connecting nodes to edges**: Each locator has an `xlink:label` attribute for identification, while the arc uses `xlink:from` and `xlink:to` attributes to connect the locators.
- **Ordering edges**: The `order` attribute on the arc dictates the sequence of outgoing edges from a node.
- **Arcrole**: The arc's `xlink:arcrole` attribute clarifies the relationship between the arc's source and target, set to `parent-child` for presentationLinks.

In XBRL, locators and arcs are fundamental to constructing networks, especially presentationLinks. For brevity, only the XML syntax for the first arc in the network is provided. A presentationLink is merely a collection of locators and arcs. To craft a comprehensive presentation network like the one in figure 4.3, additional locators and arcs are incorporated into the presentationLink.

### 4.6.3 Motivation for Report Elements

Until now, the focus has been on the implementation of presentation networks in XBRL and their role in structuring a hierarchy of concepts. However, this explanation is not completely accurate.
Concepts have been defined as the "what" aspect of a fact. For instance, when the company Foo declares a revenue of 1000 USD for the year 2019, the "what" part is represented by the concept `Revenue`.
Yet, not every element within the presentation network, as seen in figure 4.3, is capable of being linked to a fact. The `BalanceSheet` concept serves as an example. In XBRL terminology, concepts that cannot be directly associated with facts are termed `Abstract`.
XBRL categorizes both abstracts and concepts under the collective term "report element". These elements can appear within a report, with concepts being directly linked to facts and abstracts outlining the report's structure. In total, XBRL recognizes six types of report elements, which will be detailed in subsequent sections.
With the clarification of report elements, the understanding of presentation networks slightly shifts. Instead of merely organizing concepts, presentation networks arrange report elements into a hierarchy. Nonetheless, within the context of a fact, a concept is still required, not just any report element.

### 4.6.4 calculationLink

The `link:calculationLink` network delineates how concepts derive from the sum of other concepts. Although built similarly to presentationLinks, calculationLinks exhibit notable differences:

1. Arcs are termed `link:calculationArc`.
2. Links are designated as `link:calculationLink`.
3. The `xlink:arcrole` attribute for `link:calculationArc` is set to `summation-item`.

4. An extra attribute, `weight`, is part of the `link:calculationArc`.

5. All locators within the link refer exclusively to concepts.

While the initial distinctions are direct and bear no semantic impact, the addition of the `weight` attribute and the locator's exclusive reference to concepts are pivotal aspects that warrant further explanation.
Calculation networks in XBRL are designed to enable processors to calculate or verify the correctness and consistency of a concept's value. For the XBRL processor Brel, the emphasis is on verification. "The XBRL Book" delves into various consistency checks in Chapter 6.4 [7].
In calculation networks, facts linked to a concept are calculated as the weighted sum of their child concepts. The `weight` attribute in the `link:calculationArc` element determines the child's contribution to this sum. Moreover, facts are associated exclusively with concepts, not any report element, hence all locators in a calculation network reference concepts.
The concept's `balance` attribute, introduced in Section 4.3, indicates whether it is a debit or credit. XBRL 2.1 specification imposes rules on the interaction between a concept's `balance` attribute and the `weight` attribute of a `link:calculationArc` [15]. A negative `weight` is required if one concept is a debit and the other is a credit. Conversely, a positive `weight` is necessary if both concepts share the same `balance`.

| **Concept 1** | **Concept 2** | **Connecting edge weight** |
|---|---|---|
| Debit | Credit | $\leq 0$ |
| Credit | Debit | $\leq 0$ |
| Debit | Debit | $\geq 0$ |
| Credit | Credit | $\geq 0$ |

Figure 4.6: Balance and weight constraints in calculation networks

A network that adheres to the specified balance and weight rules is recognized as a `balance consistent network` [7].
Balance consistency represents just one form of consistency applicable to calculation networks. Another form is `roll-up consistency`, which is categorized into: `simple roll-up consistency` and `nested roll-up consistency`.
`Simple roll-up consistency` pertains to networks without nested concepts, limiting the network to a depth of 1.
`Nested roll-up consistency` involves networks with nested concepts, allowing for a network depth greater than 1.
`Roll-up consistency` examines both calculation and presentation networks to ensure their structures align. For instance, if a calculation network includes the arc `Assets -> Savings Accounts`, the presentation network must also feature the arc `Assets [Abstract] -> Savings Accounts`.
It's important to note that calculation networks consist solely of concepts, excluding abstracts. Thus, the report element `Assets` in the calculation network differs from `Assets [Abstract]` in the presentation network. The relationship between these two elements is established through the presentation network, which includes the arc `Assets [Abstract] -> Assets`.
Figure 4.7 will illustrate this concept by displaying the calculation and presentation networks side by side, demonstrating their roll-up consistency. The example is further detailed with concepts such as `UBS Savings Account`, `Raiffeisen Savings Account`, and `Liabilities`. Note, the calculation network includes arc weights, unlike the presentation network.

Figure 4.7: Example of nested roll-up consistency

```
Assets (weight: 1)
├── Savings Accounts (weight: 1)
│   ├── UBS Savings Account (weight: 1)
│   └── Raiffeisen Savings Account (weight: 1)
└── Liabilities (weight: -1)
```

Figure 4.8: Calculation network

```
Assets [Abstract]
├── Savings Accounts [Abstract]
│   ├── UBS Savings Account
│   ├── Raiffeisen Savings Account
│   └── Savings Accounts
├── Assets
└── Liabilities
```

Figure 4.9: Presentation network

In Figure 4.7, the presentation network is roll-up consistent with the calculation network. It includes two abstracts, `Assets [Abstract]` and `Savings Accounts [Abstract]`, which do not appear in the calculation network. These abstracts act as placeholders for the concepts `Assets` and `Savings Accounts` in the calculation network.

Beyond roll-up and balance consistency, there exists a third form of consistency, unnamed by XBRL, which I will term `aggregation consistency`. This type of consistency ensures that within a calculation network, the sum of child concepts accurately contributes to the value of their parent concept. Referring to our example in Figure 4.7, the values for `UBS Savings Account` and `Raiffeisen Savings Account` should aggregate to match the `Savings Accounts` concept's total value. Aggregation consistency relies on the weighted sums of child concepts, with each weight determined by the arc's weight.

For aggregation consistency evaluation, it's essential to have the values of facts linked to the concepts, implying that this form of consistency compares a list of facts against a calculation network.

A practical demonstration of aggregation consistency can be seen in the list of facts presented in Figure 4.10, relevant to the calculation network in Figure 4.7:

Figure 4.10: Alice's savings accounts and liabilities in CHF

| **Concept** | **2022** | **2023** |
|---:|---:|---:|
| UBS Savings Account | 1000 | 1000 |
| Raiffeisen Savings Account | 2000 | 3000 |
| Savings Accounts | 3000 | 4000 |
| Liability | 500 | 500 |
| Assets | 2500 | 3499 |

As demonstrated, each concept has two reported facts, one for 2022 and another for 2023.

When multiple facts are reported for a concept, the process involves iterating through these facts and "pinning" all aspects except for the concept itself. Subsequently, a search through all facts is conducted to identify those with identical pinned aspects. Using this refined list of facts and the calculation network, aggregation consistency is then assessed.

In the given example, the assessment would commence with all facts for 2022, followed by those for 2023. For 2022, the list of facts is deemed aggregation consistent as the sums of the UBS Savings Account and the Raiffeisen Savings Account align with the total value of the Savings Accounts concept. Additionally, subtracting the value of the Liabilities concept from the Savings Accounts concept yields the value of the Assets concept, affirming consistency.

However, for the year 2023, the facts list does not maintain aggregation consistency. The discrepancy arises because the combined value of the Savings Accounts, after adjusting for the Liabilities concept's value, fails to match the declared value of the Assets concept.

### 4.6.5 definitionLink

The `link:definitionLink` network represents various relationships between report elements not encapsulated by presentation or calculation networks. In terms of implementation, definitionLinks operate similarly to presentationLinks and calculationLinks, with a few syntactic distinctions:

1. Arcs are denoted as `link:definitionArc`
2. The network itself is referred to as `link:definitionLink`
3. The `xlink:arcrole` attribute for `link:definitionArc` encompasses a range of values, varying across the network

The naming conventions for links and arcs are straightforward and carry no inherent semantic significance. The `xlink:arcrole` attribute specifies the nature of the relationships between arc sources and targets.

For example, the `essence-alias` arcrole indicates that one report element serves as an alias for another, while `hypercube-dimension` signifies the association of a "dimension" report element with a "hypercube" report element. Further details on hypercubes and dimensions will be explored in Section 4.8.

Definition networks extend beyond these roles, aiming to document relationships unaddressed by other network types. They are instrumental in clarifying the classification of report elements, with hypercubes, discussed further in Section 4.8, relying heavily on definition networks.

### 4.6.6 labelLink

The `link:labelLink` network serves to connect report elements with labels that are readable and understandable by humans. Instead of presenting users with the QName of report elements, applications like Arelle display friendly, human-readable labels.

For example, in the Coca Cola Company's condensed consolidated statement of income for Q2 2019, the concept identified as `us-gaap:Revenues` is shown in Arelle as "Net Operating Revenues", thanks to the `link:labelLink` network included in the XBRL report.

| Concept | 2023-06-30 |
|---|---|
| 0000002 - Statement - CONDENSED CONSOLIDATED STATEMENTS OF INCOME | |
| Income Statement [Abstract] | |
| Statement [Table] | |
| Statement | |
| Net Operating Revenues | 22,952,000,000 |
| Cost of goods sold | 9,229,000,000 |
| Gross Profit | 13,723,000,000 |
| Selling, general and administrative expenses | 6,506,000,000 |
| Other operating charges | 1,338,000,000 |
| Operating Income | 5,768,000,000 |
| Interest income | 392,000,000 |
| Interest expense | 374,000,000 |
| Equity income (loss) — net | 813,000,000 |
| Other income (loss) — net | 91,000,000 |
| Income Before Income Taxes | 2,880,000,000 |
| Income taxes | 359,000,000 |
| Consolidated Net Income | 5,634,000,000 |
| Net Income (Loss) Attributable to Noncontrolling Interest | -20,000,000 |
| Net Income Attributable to Shareowners of The Coca-Cola Company | 5,654,000,000 |
| Basic Net Income Per Share1 | 1.31 |
| Diluted Net Income Per Share1 | 1.30 |
| Average Shares Outstanding — Basic | 4,325,000,000 |
| Effect of dilutive securities | 18,000,000 |
| Average Shares Outstanding — Diluted | 4,343,000,000 |

Figure 4.11: Statement of income of the Coca Cola Company of Q2 2019

LabelLinks, similar to other extended links, bind report elements to textual strings. While they share the implementation basis with presentationLinks and calculationLinks, labelLinks uniquely incorporate `link:label` elements alongside arcs and locators.
Labels, from a semantic viewpoint, differ from other report elements like concepts and abstracts, being classified under `resource`, which acts as metadata for report elements, facts, and other XBRL report components.
Taking the earlier example, the definition of the `us-gaap:Revenues` concept happens independently of any linked labels, which are subsequently linked via the `link:labelLink` network. A single report element may be linked to various labels, available in different languages or levels of detail. To manage these labels, XBRL employs the concept of `roles`, further discussed later in this section. Similar to the `xlink:arcrole` attribute seen in `link:presentationArc`, labels' roles function comparably.
In XBRL, the `link:labelLink` network mirrors the `link:presentationLink` network's setup, with the addition of the `link:label` element to denote a label. Label elements contain several pieces of information:
In the `link:labelLink` network, labels are detailed through specific attributes in their XML representation:

1. The `xlink:label` attribute acts as a unique identifier for the label, linking it to its corresponding arc. This is distinct from the label's textual content.

2. The `xlink:role` attribute defines the label's role, which will be explained in further detail subsequently.

3. The `xml:lang` attribute indicates the language in which the label is written.

4. The `xlink:type` is invariably set to `resource`, categorizing the element as a resource.

5. Lastly, the label's text is the human-readable string, as seen in Arelle for the "Net Operating Revenues" in figure 4.11.

For the ”Net Operating Revenues” label depicted in figure 4.11, the following XML snippet illustrates how the label is represented:

```xml
<link:label
    id="1234"
    xlink:label="lab_us-gaap_Revenues"
    xlink:role="http://www.xbrl.org/2003/role/terseLabel"
    xlink:type="resource"
    xml:lang="en-US">
    Net Operating Revenues
</link:label>
```

Figure 4.12: Example of a label in XML syntax

Note that we have omitted both the connecting arc and the locator in this example, as they are implemented similarly to other networks.

The label in figure 4.12 has the role `http://www.xbrl.org/2003/role/terseLabel`, which signifies that the label text is brief and succinct. XBRL defines several other roles, with the most significant ones including:

| **Role** | **Description** |
|---|---|
| `http://www.xbrl.org/2003/role/label` | The default label role. |
| `http://www.xbrl.org/2003/role/terseLabel` | A short, human-readable label. |
| `http://www.xbrl.org/2003/role/verboseLabel` | A long, human-readable label. |
| `http://www.xbrl.org/2003/role/positiveLabel` | A label for positive values. |
| `http://www.xbrl.org/2003/role/negativeLabel` | A label for negative values. |
| `http://www.xbrl.org/2003/role/zeroLabel` | A label for zero values. |
| `http://www.xbrl.org/2003/role/documentation` | A label for documentation. |

Figure 4.13: Important label roles

The roles listed in Figure 4.13 are not exhaustive, as numerous other label roles are commonly employed. Users can even define their own label roles. For a complete list of standard label roles, refer to the XBRL 2.1 specification[16].

### 4.6.7 referenceLink

The `link:referenceLink` network facilitates the connection of report elements to external resources, serving a role akin to citations in academic literature. For instance, the `us-gaap:Revenues` concept might be linked to its official definition within the SEC’s Code of Federal Regulations (CFR), creating a bridge between the report element and an authoritative external source.

In terms of structure, referenceLinks mirror the setup of LabelLinks, incorporating arcs, locators, and resources. However, the distinction lies in the nature of the resources: referenceLinks point to external resources rather than textual labels. While labels typically consist of text, references are structured as a dictionary. Figure 4.14 presents an example of a reference in XML syntax, excluding the accompanying arc and locator for brevity.

```xml
<link:reference
    xlink:type="resource"
    xlink:label="SECRegulationS-K229402v2vi"
    xlink:role="http://www.xbrl.org/2003/role/presentationRef"
>
    <ref:Publisher>SEC</ref:Publisher>
    <ref:Name>Regulation S-K</ref:Name>
    <ref:Number>229</ref:Number>
    <ref:Section>402</ref:Section>
    <ref:Subsection>v</ref:Subsection>
    <ref:Paragraph>2</ref:Paragraph>
    <ref:Subparagraph>vi</ref:Subparagraph>
</link:reference>
```

Figure 4.14: Example of a reference for the concept `edc:CoSelectedMeasureName`

The `link:reference` element's child elements collectively construct a dictionary detailing the external resource referenced. In the example provided (4.14), the reference directs to `17 CFR 229.402(v)(2)(vi)`[31], which stands for a specific section within the Code of Federal Regulations (CFR). However, references within XBRL reports are not limited to the CFR; they can link to a variety of external resources, including other XBRL reports, PDF documents, websites, and more.
Typically, an XBRL report will include a single `link:referenceLink` to connect the concepts it contains with the relevant regulatory codes or documentation.

### 4.6.8 footnoteLink

The `link:footnoteLink` network functions similarly to the `link:referenceLink` and `link:labelLink` networks, facilitating the association of report elements with additional resources. The primary distinction lies in the nature of these resources: footnotes rather than labels or external references. Another notable difference is that within the `link:footnoteLink` network, locators can reference not only report elements but also facts. Apart from these variations, footnoteLinks are structured in much the same manner as the previously discussed networks.

## 4.7 Roles

Even though the networks introduced in the previous section 4.6 provide a good foundation for structuring XBRL reports, they are not sufficient to create a comprehensive overview of the report whole report, only individual sections of it. Moreover, the roll-up consistency of calculation networks4.6.4 introduced the notion of having networks related to each other. With our current understanding of XBRL, there is no way to express this relationship. This is precisely the function of `Roles` in XBRL.
Roles serve to group networks, akin to chapters within a report, by assigning each network set a unique URI and, optionally, a label. This structuring allows for a segmented yet unified report presentation, where each section, such as the cover page, balance sheet, and income statement, is encapsulated within a specific role. Typically, a role encompasses a presentation network, a calculation network, and a definition network, with these networks forming the core components of a report section. Other network types, such as label and reference networks, are usually associated with the report as a whole rather than being confined to specific roles.

The balance sheet section, for example, might solely consist of a presentation network, whereas the income statement could integrate a presentation network, a calculation network, and possibly a definition network, reflecting the complexity and requirements of each report section.

The implementation of roles within the XBRL XML syntax adopts a straightforward approach, which will be elucidated through the forthcoming illustration of a balance sheet role.

Figure 4.15: Example of the role "Balance Sheet" expressed in XBRL XML syntax

```
<link:roleType
  id="BalanceSheet"
  roleURI="http://www.foocompany.com/role/BalanceSheet"
>
    <link:definition>Foo balance</link:definition>
    <link:usedOn>link:presentationLink</link:usedOn>
    <link:usedOn>link:calculationLink</link:usedOn>
</link:roleType>
```

Figure 4.15 showcases a role with certain attributes:

- `roleURI` (required): The unique URI for the role. Other elements within the XBRL taxonomy utilize this URI to link to the role. It serves as the role's primary identifier.
- `definition` (optional): An accessible explanation of the role's purpose.
- `usedOn`: Specifies the types of links the role is applicable for.

Associations between networks and the role do not reside within the role itself. Instead, link elements employing the role must specify it in the `role` attribute to establish the connection.

A link element that includes a role necessitates the role's `usedOn` attribute to list the link's type. Referring again to figure 4.15, a compliant XBRL processor would throw an error if a definition network attempted to reference the balance sheet role. This error occurs because the balance sheet role lacks a `definitionLink` designation in its `usedOn` list.

## 4.8 Hypercubes

A notable insight from examining XBRL report facts is their resemblance to a hypercube structure. The characteristics of a fact act as the hypercube's dimensions, with the fact's value representing the hypercube's value at those specific dimensions. Contrary to networks, hypercubes are included within the OIM [10].

[10] This inclusion of hypercubes extends the OIM's scope beyond the XBRL 2.1 core specification to encompass the XBRL Dimensions 1.0 specification as well.

Figure 4.16: Example of a hypercube

| Period | Entity | Concept | Value |
|---|---|---|---|
| 2020 | Foo | Sales | 100$ |
| 2020 | Foo | Costs | 50$ |
| 2020 | Bar | Sales | 200$ |
| 2020 | Bar | Costs | 100$ |
| 2021 | Foo | Sales | 150$ |
| 2021 | Foo | Costs | 75$ |
| 2021 | Bar | Sales | 250$ |
| 2021 | Bar | Costs | 125$ |

TODO: 3D image of hypercube

Data structuring through hypercubes has become widely adopted in recent times. However, when XBRL was initially developed, hypercubes were not as commonly used. The early versions of XBRL lacked hypercube support, which was later integrated into the XBRL framework in 2006.[13].

### 4.8.1 Dimensions

Considering facts as components of a hypercube introduces four inherent dimensions. These dimensions align with the primary aspects of a fact: `Period`, `Entity`, `Concept`, and `Unit`. XBRL facilitates adding custom dimensions, categorized as either explicit or typed.
The terminology "dimension" is used in XBRL to denote both the hypercube's dimensions and the two types of custom dimensions.

### 4.8.2 Explicit dimensions

Explicit dimensions specify dimensions with a set range of possible values. For instance, consider the Foo Company operates two branches: Foo United States and Foo Europe. The company could establish a `Subsidiary` dimension featuring `Foo United States` and `Foo Europe` as possible values. These predefined values are referred to as `members`.
Both dimensions and members are designated through report elements, akin to concepts and abstracts. A member is depicted as part of a dimension by positioning it as a dimension's child within the definition network.
Moreover, members within a dimension may possess their own subordinate members. As an illustration, Foo Europe could encompass two subsidiaries: Foo Switzerland and Foo EU.

Figure 4.17: Visualizations of the explicit dimension "Subsidiary"

```
[Dimension] Subsidiary
├── [Member] Foo United States
└── [Member] Foo Europe
    ├── [Member] Foo Switzerland
    └── [Member] Foo EU
```

### 4.8.3 Typed dimensions

Typed dimensions differ from explicit dimensions by not having a pre-established set of possible values. Instead, the scope of values for a typed dimension is defined

by a specific data type. For instance, a dimension may be restricted to accept only `xs:integer` type values.
Just like explicit dimensions, typed dimensions are also delineated through report elements. However, unlike explicit dimensions, typed dimensions do not include members but consist only of the dimension report element, which stipulates the dimension's data type.

### 4.8.4 Line Items and Hypercubes

Given our understanding of hypercubes, it's apparent that the entire report could be viewed as a large hypercube. Particularly with the addition of extra dimensions, it's likely that most facts will engage just a fraction of the available dimensions. This scenario results in a highly dimensional hypercube, with many dimensions remaining unutilized. Analyzing the report based on this large hypercube would be inefficient.
XBRL addresses this issue by introducing the `hypercube` report element. A hypercube is conceptually a smaller sub-cube of the overarching report hypercube. Hypercube report elements are typically defined on a role-specific basis within a definition network. They select a subset of the dimensions from the entire report hypercube, determined within the definition network, where "hypercube-dimension" arcs indicate the included dimensions.
XBRL further introduces the `lineItems` report element. Line items pinpoint which concepts belong to the hypercube. Although a report might detail tens of thousands of concepts, only a select number are relevant for a given role. The line items report element identifies these relevant concepts by listing them as children within the definition network.
The reader may find it helpful to think of the relationship between line items and concepts as akin to that between dimensions and members.

## 4.9 XBRL Summary

As demonstrated in this chapter, XBRL is a multifaceted standard with numerous components, which has been refined over a period of more than 20 years.
The initial portion of this chapter explained that an XBRL report fundamentally comprises a set of facts. Subsequently, the latter section illustrated how these facts can be organized into networks, roles, and hypercubes. While this chapter was not exhaustive, it concentrated on the aspects most relevant to this thesis.
The succeeding chapter will introduce the API of Brel, illustrating how Brel interprets the XBRL standard. The API chapter precedes the implementation chapter, which will explain how Brel implements the XBRL standard. Despite being a result of this thesis, it is more logical to introduce the API before its implementation.

# Chapter 5

# API

This chapter provides an overview of the Brel API, illustrating its capabilities and how it simplifies and abstracts the XBRL standard, while still offering comprehensive access to XBRL's functionality as required.
The content here does not serve as an exhaustive guide. Brel includes a variety of helper functions and classes not detailed in this section. For a thorough reference, consult the Brel API documentation[25]. The elements outlined here represent the essential functionality required to access all Brel features. [1] The Brel API is segmented into various components, each discussed in this section:

1. **Core** - Introduces Brel's core components, including Filings, Facts, Components, and QNames.
2. **Characteristics** - Explores Brel's characteristics, such as Concepts, Entities, Periods, Units, and Dimensions.
3. **Report Elements** - Discusses Report Elements, specifically Concepts, Members, Dimensions, Line Items, Hypercubes, and Abstracts.
4. **Resources** - Details Resources like Labels, References, and Footnotes.
5. **Networks** - Describes the representation of networks and nodes in Brel.

These components should be familiar, as they were thoroughly examined in chapter 4. While the preceding chapter minimized details on XML implementation, this API abstracts the XML structure entirely. [2] This section addresses research questions 2.2 and 2.2.
Note that Brel's current version is not finalized. It does not support the creation of new filings, facts, components, etc., focusing instead on accessing and analyzing existing filings. Brel also does not interpret the semantics of the reports it analyzes.

## 5.1 Core

The foundation of Brel is built upon filings, facts, components, and QNames, where each component is encapsulated within one or more classes. At its core, every filing is comprised of numerous facts and components. QNames are extensively employed throughout Brel, highlighting their importance as a core element. The UML diagram provided below illustrates the core components of Brel.

[1] Some helper functions directly interact with the XBRL standard, bypassing the API to reduce overhead.
[2] The `QName` class is an exception, closely mirroring the XML QName type.

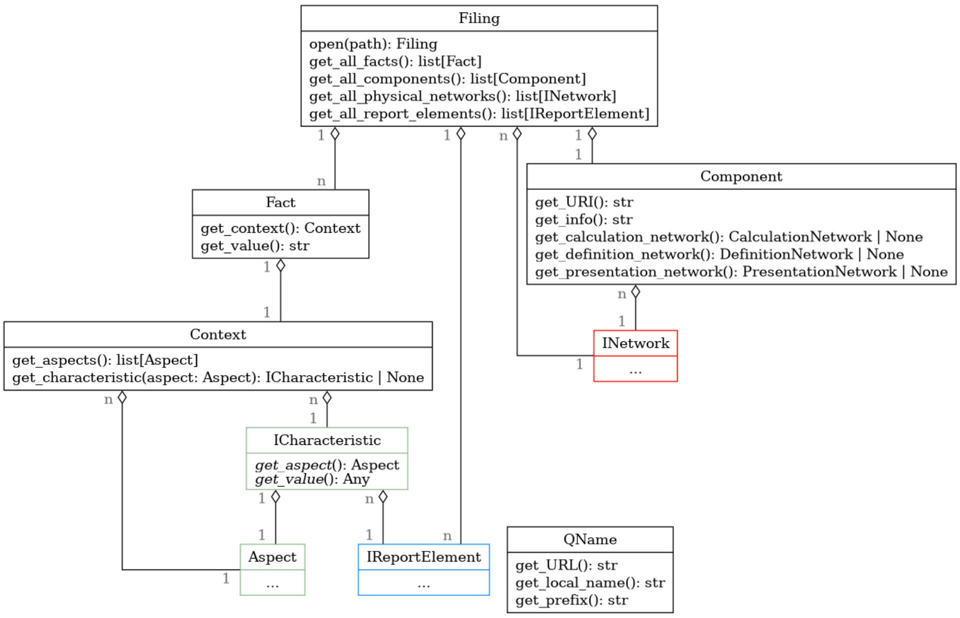



Figure 5.1: UML diagram of the core of Brel

### 5.1.1 Filing

The `Filing` class is designed to represent an individual XBRL filing. A key functionality of this class is the `Filing.open` method, which facilitates the loading of a filing from various sources such as a file, directory, or URL. This method accepts a single parameter that could be a path to a file or directory, or a URL, and intelligently determines the type of input to process the filing. It also ensures the rejection of any inputs that are not valid.

Following this, the methods `Filing.get_all_facts` and `Filing.get_all_components` are available to retrieve all facts and components of a filing, respectively. These are returned as lists of `Fact` and `Component` objects, though the order of the facts and components is arbitrary. While Brel offers helper methods for accessing specific facts and components, the aforementioned methods can be used to achieve similar outcomes.

Moreover, there are certain networks, termed `physical networks`, which do not belong to any Component. Direct access to these networks through Components is not possible, hence the `Filing` class provides a `Filing.get_all_physical_networks` method for retrieving all physical networks.

Additionally, the `Filing` class caters to report elements that are neither associated with any network nor fact. The `Filing.get_all_report_elements` method returns all such report elements within a filing, including those not referenced by facts or networks.

The essential classes tied to a filing are the `Fact` and `Component` classes, each addressing different fundamental aspects of XBRL. The `Fact` class and its related classes are governed by the Open Information Model (OIM)[19], addressing research question 2.2. Conversely, the `Component` class and its related entities delve into areas beyond the OIM's scope, thus addressing research question 2.2.

The discussion will proceed with an examination of the `QName` class, given its utility across numerous other classes. Subsequent sections will explore the `Fact` class along with its related classes. The final segments will focus on the `Component` class and its associated classes.

### 5.1.2 QName

The `QName` class signifies a qualified name, merging a namespace with a local name. This element is the singular link to XBRL's XML background found in the Brel API. Its effectiveness in identifying elements across various namespaces led to its inclusion in the API.

As outlined in chapter 4, a QName consists of a namespace URL, a prefix, and a local name. Therefore, the `QName` class provides methods to retrieve these components: `QName.get_URL` for the namespace URL, `QName.get_prefix` for the prefix, and `QName.get_local_name` for the local name.

### 5.1.3 Fact

The `Fact` class symbolizes a single XBRL fact, encapsulating a value along with various characteristics that give context to the value. The method `Fact.get_value` returns the value as a string. Characteristics defining a fact are accessible through the `Fact.get_context` method, which consists of a set of characteristics that describe the fact. Essentially, each fact is situated within a multi-dimensional space, with each characteristic pinpointing one dimension's coordinate.

At first glance, the choice to return fact values as strings via `Fact.get_value` might raise questions, given that XBRL values encompass a range of types, including integers, decimals, dates, etc. This approach stems from the understanding that facts possess a unit characteristic, which categorizes the value's type. Units in XBRL thus act as a contextual dimension for a fact. The string format for all values ensures a universal representation, considering strings can encapsulate any XBRL fact value.

Nevertheless, Brel facilitates type-appropriate conversions of these values through helper methods. These methods assess the fact's unit to convert the string value into a more suitable data type, aligning with the specific nature of the value.

### 5.1.4 Context

Contexts, as outlined in section 5.1.3, constitute sets of attributes that describe what a fact represents. The `Context` class embodies an individual context, with each fact possessing a unique context, and every context being linked to a singular fact. [3]

Contexts offer two methods for characteristic retrieval. The `Context.get_aspects` method yields a list of all aspects associated with characteristics within the context. For instance, most contexts in XBRL have the aspects entity, period, unit and concept. Given an aspect, the `Context.get_characteristic` method returns the corresponding characteristic. For example, the unit aspect of a context might return the characteristic "USD". The API of both Aspects and Characteristics will be covered in section 5.3.

To elaborate, a characteristic denotes a coordinate along a singular dimension, and aggregating multiple characteristics constructs a coordinate within a multi-dimensional framework.

One such point might be "Foo Inc.'s net income for the year 2020 in USD". Another point might be "Bar Corp.'s net income for the year 2021 in CHF". Each represents a position within a four-dimensional hypercube [4].

[3] Although there may be multiple contexts with the same characteristics, they are treated as distinct entities.

[4] The dimensions in these instances are entity, period, unit, and concept.

### 5.1.5 Report Elements

Report elements form the foundational aspects of XBRL filings, with concepts being among the most crucial, as initially discussed in section 4.3. In figure 5.1, the `IReportElement` interface is depicted as representing all types of report elements, not solely concepts.
Characteristics often utilize report elements to denote a fact's position within a dimension. Referring back to the example in 5.1.4, the "net income" concept is employed as a characteristic to define the fact's location along the concept dimension. Given the diversity of report elements, the `IReportElement` interface includes a method for getting the type of the report element. A more detailed discussion on report elements is scheduled in section 5.2, to be addressed later in this chapter.

### 5.1.6 Component

Transitioning to the `Component` class, as illustrated on the opposite end of figure 5.1, it symbolizes the chapters within a filing and represents a structure not covered by the OIM.
Each component is characterized by a series of networks and an identifier. Components can contain no more than one network of each type. The types of networks include calculation, presentation, and definition networks. A component may also feature an optional, human-readable description elucidating its purpose.
The methods `Component.get_calculation_network`, `Component.get_presentation_network`, and `Component.get_definition_network` respectively retrieve the component's calculation, presentation, and definition networks. Each method may return `None` if the component lacks a network of the specified type.
The `Component.get_uri` method yields the component's identifier, a URI that distinctively identifies the component within the filing.
Moreover, `Component.get_info` delivers the component's human-readable description when available, or an empty string if absent.
The networks within a component implement the `INetwork` interface, set to be discussed in the latter half of this chapter.

## 5.2 Report Elements

Given that characteristics incorporate report elements, we first discuss report elements. They were introduced previously in chapter 4. Brel categorizes several types of report elements, each represented by distinct classes but unified under the `IReportElement` interface. We categorize six principal types of report elements as follows:

1. **Concept** - Defines the nature of the information a fact represents.
2. **Abstract** - Utilized to group other report elements for organizational purposes.
3. **Dimension** - Describes a custom axis to position a fact within a specific context.
4. **Member** - Identifies a specific location along a dimension where a fact is positioned.
5. **LineItems** - Organizes concepts along an axis, analogous to the grouping function of dimensions for members.
6. **Hypercube -** Specifies a subset of the filing's global hypercube structure.

From a technical standpoint, concepts, members, and dimensions directly relate to the OIM, while the rest do not. Nevertheless, for clarity and coherence, all types are discussed collectively. Brel's approach to implementing these elements is visualized in figure 5.2, presenting a UML diagram of the report element classes. Aiming for simplicity in Brel's design, the inheritance hierarchy is intentionally kept flat. Following the introduction of the common interface, detailed discussions on each report element type are presented in subsequent sections.

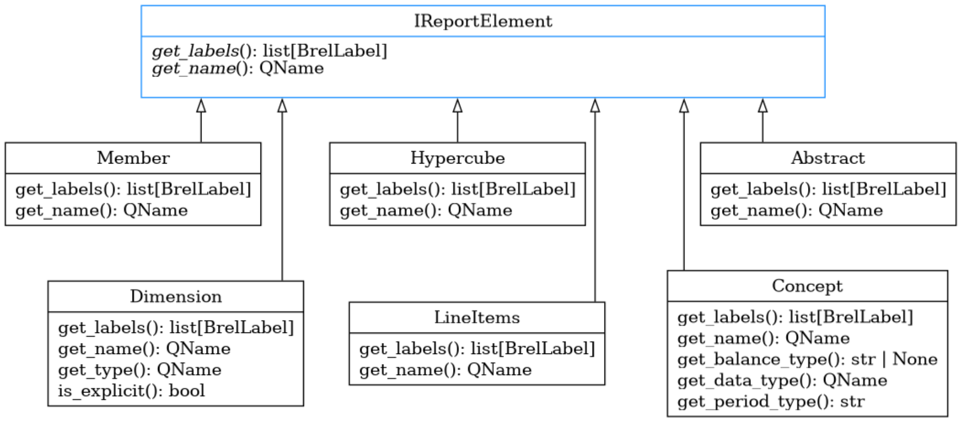



Figure 5.2: UML diagram of the report element classes in Brel

### 5.2.1 IReportElement

The `IReportElement` interface defines the common interface that all report elements share. In essence, a report element is an just a name,

Given that QNames may not be entirely intuitive for human reading and lack multilingual support, Brel supplements this with a method to obtain a report element's label(s), although labels will be more thoroughly explained in section 4.6.6. A report element may have multiple labels, which can be retrieved using the `get_labels` method.

### 5.2.2 Concept

The `Concept` element holds a central role within the XBRL framework, essential for every fact recorded. It specifies the nature of the data each fact represents.

Beyond the capabilities provided by the `IReportElement` interface, the `Concept` also provides information about its associated facts. As detailed in section 4.3, concepts have the authority to impose restrictions on certain properties of the facts they are linked to. The methods `get_balance_type`, `get_data_type` and `get_period_type` return the balance, data type and period type of the concept, respectively.

The balance type applies solely to monetary concepts, highlighting if the concept is categorized as an asset or a liability. This classification influences the weighting within calculation networks, as outlined in section 4.6.4. A concept's data type specifies the value's data type for a given fact. Further, a concept's period dictates if a fact's period is a "duration" or an "instant".

### 5.2.3 Dimension

Dimensions define custom axes, along which facts are positioned. There are two types of dimensions: explicit and typed. In addition to inheriting methods from the `IReportElement` interface, the `Dimension` class offers two specific methods. The `is_explicit` method returns a boolean indicating the nature of the dimension—whether it is explicit. The `get_type` method discloses the dimension's type for typed dimensions and triggers an exception for explicit ones.

Merging typed and explicit dimensions into a singular class reflects their semantic alignment, despite minor operational variances. These differences are largely inconsequential from the user standpoint.

### 5.2.4 Abstract, Hypercube, LineItems, and Member

The classes `Abstract`, `Hypercube`, `LineItems`, and `Member` bear a strong resemblance to each other. Aside from their names, they offer identical functionalities and properties. The decision to segregate them into separate classes is based on their distinct semantic roles, despite their functional similarities.

## 5.3 Characteristics

Characteristics serve to pinpoint a fact's location within a dimension. They utilize report elements for their descriptions or introduce novel, separate characteristics. All characteristics are unified under the `ICharacteristic` interface. Every characteristic functions as an aspect-value duo, with the aspect defining the dimension's axis and the value specifying the fact's location on that axis. The dynamic between aspects and characteristics is depicted in figure 5.3.

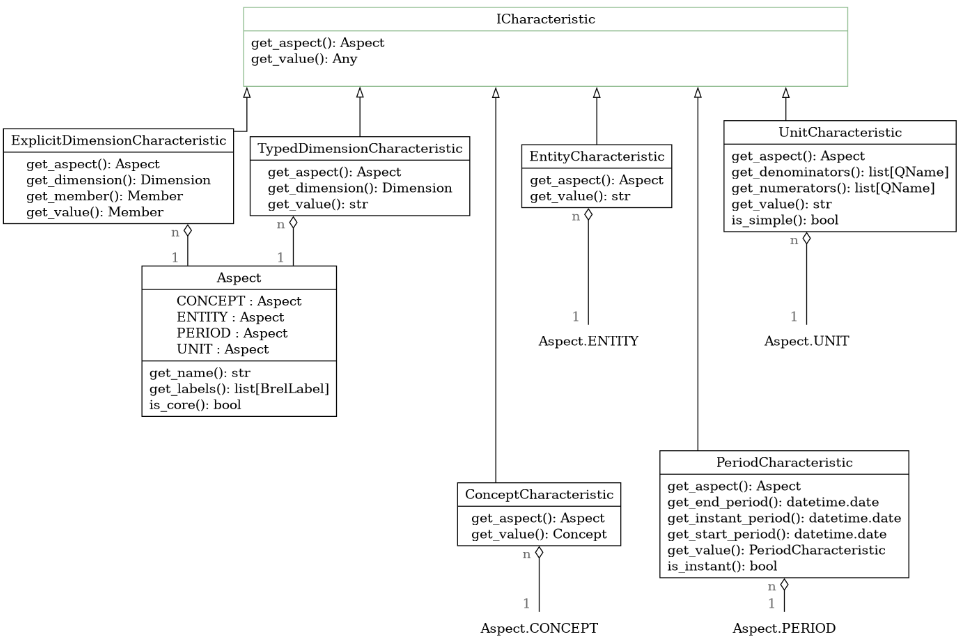



Figure 5.3: The interplay between aspects and characteristics

### 5.3.1 ICharacteristic

As mentioned earlier, characteristics describe a fact's dimensional location through an aspect-value combination. Here, the aspect describes the dimension's axis, and the value conveys the fact's precise location on the axis.
Directly aligning with this concept, the `ICharacteristic` interface offers the methods `get_aspect` and `get_value`.

### 5.3.2 Aspect

Aspects define the axes of dimensions. Each `Aspect` class instance corresponds to a unique aspect. The primary aspects - Concept, Entity, Period, and Unit - are represented as instances of the `Aspect` class. Moreover, these core aspects are accessible as public constants within the `Aspect` class, specifically through the fields `Aspect.CONCEPT`, `Aspect.ENTITY`, `Aspect.PERIOD`, and `Aspect.UNIT`.
The `Aspect` class includes a method named `get_name` for retrieving an aspect's name, which returns a string.
Just like the `IReportElement` interface, the `Aspect` class incorporates a method to obtain the label of the aspect. The method `get_labels` produces a collection of labels because an aspect might possess several labels.
Moreover, the `is_core` function determines if an aspect belongs to the category of core aspects.

### 5.3.3 Concept Characteristic

The `ConceptCharacteristic` specifies the concept employed by a fact. In the context of hypercubes, `Aspect.CONCEPT` serves as a dimension encompassing various concepts, while the specific `Concept` report element represents a point within this dimension. It is essential for every context to include this dimension characteristic.
As expected, the `ConceptCharacteristic` adheres to the `ICharacteristic` interface. `get_aspect` yields `Aspect.CONCEPT`, and `get_value` returns the `Concept` instance.

### 5.3.4 Entity Characteristic

The `EntityCharacteristic` identifies the entity associated with a fact. An entity, such as a corporation, is distinguished by a tag and a scheme[5] serving as the tag's namespace. These identifiers are merged into a singular string format `{scheme}tag`, [6]
Implementing the `ICharacteristic` framework, `get_aspect` returns `Aspect.ENTITY` and `get_value` provides the entity's string representation as previously mentioned.

### 5.3.5 Period Characteristic

The `PeriodCharacteristic` specifies the time frame of a fact. Periods can be either instant or duration, which can be checked using the `is_instant` method.
The methods `get_start_date` and `get_end_date` return the start and end date of a duration period respectively. Should the period be an instant, these methods will trigger an exception, highlighting that instant periods lack defined start or end points. Conversely, the method `get_instant` returns the instant of the period. If the period is duration, the method raises an exception, since duration periods do not

[5]Schemes are typically URLs.
[6]This format mirrors the Clark notation used for QNames.[36]

have an instant. Each of the three methods yields a date of the type `datetime.date`, which is a conventional Python class used for date representation.
Additionally, period characteristics conform to the `ICharacteristic` interface. `get_aspect` emits `Aspect.PERIOD` and `get_value` returns the period characteristic itself. The justification for `get_value` delivering the characteristic itself stems from the lack of a primitive type in Python to represent XBRL periods. The Python `datetime` module, regarded as the standard for date depiction in Python, does not offer a class that represents both instant and duration periods together.

### 5.3.6 Unit Characteristic

The `UnitCharacteristic` specifies the unit of a fact and adheres to the `ICharacteristic` interface. Semantically, the unit characteristic further determines the fact's value type. A fact assigned with a `USD` unit is of the `decimal` type, whereas a fact with a `date` unit signifies a date value.
Units are categorized into two types: simple and complex. Simple units are indivisible, exemplified by `USD` or `shares`. Complex units consist of multiple simple units combined, such as `USD per share`. Each complex unit is constructed by dividing one or more simple units by another set of simple units.

Figure 5.4: Schematic of composition of complex units

$$\frac{num\_unit_1 \cdot num\_unit_2 \cdot ...}{1 \cdot denom\_unit_1 \cdot denom\_unit_2 \cdot ...}$$

Brel depicts the complex unit in figure 5.4 through a pair of lists containing simple units. The function `get_numerators` delivers the numerator's list of simple units, while `get_denominators` yields the list of denominators [7].
Similar to the `PeriodCharacteristic`, the method `get_value` of the `UnitCharacteristic` returns the characteristic itself.

### 5.3.7 Dimension Characteristics

XBRL differentiates between two types of dimension characteristics: typed and explicit. The two types are represented by the `TypedDimensionCharacteristic` and `ExplicitDimensionCharacteristic` classes respectively. Like every other characteristic, both implement the `ICharacteristic` interface.
As discussed in section 5.2, Brel models custom dimensions as `Dimension` report elements. Accordingly, the aspect of a dimension characteristic should represent a `Dimension` report element. Therefore, the `get_aspect` method of both classes returns the QName of the `Dimension` instance as a string.
Dimension characteristics facilitate direct access to the `Dimension` object itself through the `get_dimension` method.
As the name suggests, the value of a typed dimension characteristic pertains to a specific type. The `get_value` method should reflect this accordingly. It should return the value in a type that encompasses all possible values of the dimension. The most general type of any value in XBRL is a string.
The actual type of the value is determined by the `get_type` method of the `Dimension` element. [8] Brel includes auxiliary methods designed to convert the value into the

[7] The list of denominators provided excludes the implicit denominator of 1.
[8] The `Dimension` object returned by `get_dimension` is guaranteed to be a typed dimension with `is_explicit` returning `False`.

format that best reflects its intended representation. While these helper methods are not included within the minimal API outlined in this chapter, they are part to the comprehensive API.
Explicit dimensions are the second category of custom characteristic. They are extremely similar to typed dimensions, but they do not have a type. Instead of a type, they have a set of possible values.
The `ExplicitDimensionCharacteristic` class is almost identical to the `TypedDimensionCharacteristic` class. The main difference between the two is that `get_value` returns a `Member` object instead of a string.

## 5.4 Answering Research Question 1

The Open Information Model (OIM) is a conceptual model for XBRL.[19] Unlike the XBRL specification, the OIM is not a standard. Chapter 4 already gave an intuition of the OIM. The chapter only diverged from the OIM once it reached parts of XBRL that are not yet covered by the OIM.
Given that the OIM is already organized systematically, the Brel API aligns closely with its structure. Similar to the OIM, the Brel API remains agnostic to the specific format of its underlying XBRL reports. It encompasses Reports, Facts, and the Concept-, Entity-, Period-, Unit-, and Dimension characteristics, all of which are part of the OIM. [9]

- **Report** - Represented by the `Filing` class, it encapsulates a single XBRL report. Mirroring the OIM, it functions as a container for facts. Beyond facts, a report comprises a taxonomy, a collection of report elements, accessible through the `Filing` class.
- **Fact** - The `Fact` and `Context` classes represent a single XBRL fact. Aligning with the OIM, a fact includes a value and various characteristics that define the value's meaning.
- **Characteristics** - Brel materializes all characteristics listed in the OIM into classes - concepts, entities, periods, units, explicit, and typed dimensions. [10]

Therefore, the initial section of this chapter provides an answer to research question 2.2 by offering a Python API grounded in the OIM.

> **RQ1:** How can the OIM be translated into an easy-to-use python API?

Brel's distinction from the OIM lies in the introduction of report elements. While the OIM defines concepts, dimensions, and members, it lacks a collective term for them. [11] In the context of the OIM alone, these terms denote unrelated concepts. Yet, Brel also addresses XBRL aspects not covered by the OIM, including networks, components, and resources. Networks demand a uniform approach to handling elements such as concepts, dimensions, and members. The logic for this will be detailed in the latter half of this chapter, which aims to answer research question 2.2.

[9]The term "characteristic" is not used by the OIM. Brel adopts this terminology to prevent confusion with report elements that have similar names.
[10]While the OIM mentions language and Note ID core dimensions, Brel has not incorporated these yet but allows for their simulation through the typed dimension characteristic. Their rare usage justifies their current exclusion.
[11]The OIM might categorize concepts and members under "dimension"; however, the term "dimension" is so broadly used it often lacks clear meaning.

## 5.5 Resources

Prior to delving into networks, it is essential to discuss Brel's method of handling resources. Resources serve as the mechanism within XBRL for depicting metadata, which is connected with other components of an XBRL report through networks.
Similar to report elements and characteristics, resources share a common interface. XBRL distinguishes three types of resources: label, reference, and definition. Brel represents each resource type with a specific class, and the class diagram displayed in figure 5.5 depicts how these resource classes interrelate.

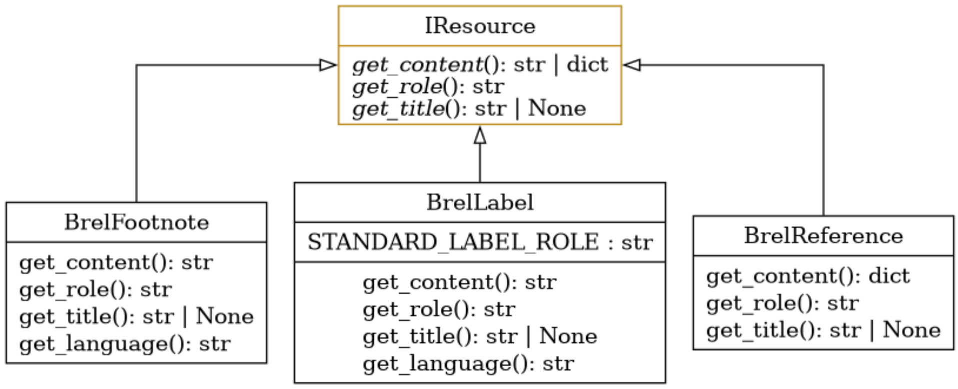



Figure 5.5: UML diagram of the resource classes in Brel

### 5.5.1 IResource

Resources are comprised of three elements: a role, a title, and content.
The role functions as an identifier for the type of resource. For instance, while every XBRL label is represented by a `BrelLabel` object, labels are differentiated by their role. The role essentially categorizes the label types, and the `get_role` method retrieves the resource's role.
Following this, the resource's content is accessible via the `get_content` method. Typically, the content is textual. However, for references, the content encompasses embedded XML.
Besides the content, the resource includes a human-readable description, or title. The `get_title` method retrieves the resource's title.

### 5.5.2 Labels and Footnote

Footnotes and labels represent two distinct resource types utilized to establish connections with other components within an XBRL report. Despite each having its dedicated class, they are functionally similar from an API perspective.
Both implement the `IResource` interface, with the `get_role`, `get_title`, and `get_content` methods operating in a similar manner across both. Diverging from their shared interface, labels and footnotes introduce an additional method: `get_language`, which returns the resource's language.

### 5.5.3 References

References constitute the third resource category within XBRL and links XBRL reports to external resources. Like labels and footnotes, references implement the `IResource` interface. However, they lack a `get_language` method, unlike labels and footnotes. Moreover, the content of a reference is a dictionary rather than a string.

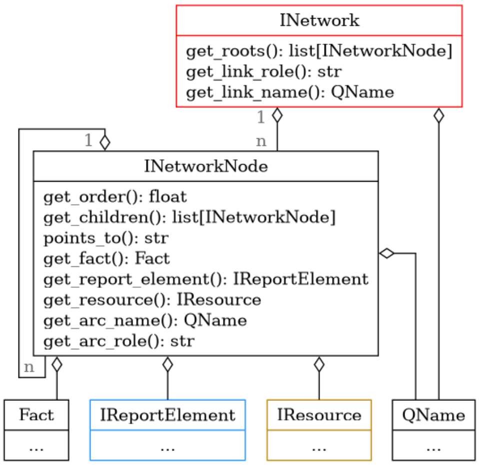
## 5.6 Networks

Networks represent the final component essential to the Brel API, completing the framework alongside resources and components. These elements constitute parts of XBRL not yet encompassed by the OIM, but Brel seeks to integrate them with the same level of detail and organization.

At their core, networks are a collection of nodes. Each node points to either a fact, a resource or a report element. Nodes can have at most one parent node and an arbitrary number of child nodes. The whole network can be represented simply by a list of root nodes. These root have children, which have children, and so on. Thus, the whole is accessible through the roots.

Consistent with previous Brel parts of the API, networks implement a common interface, in this case, two interfaces: `INetwork` and `INetworkNode`.

Figure 5.6: UML diagram of the `INetwork` and `INetworkNode` interfaces




### 5.6.1 INetwork and INetworkNode

The `INetwork` interface functions as a wrapper around a list of root nodes. It provides a method for getting all root nodes, named `get_roots`.

In addition to roots, `INetworkNode` provides access to both the link role and the link name of the underlying network with `get_link_role` and `get_link_name`. These two methods expose the underlying XML structure of XBRL, raising the question of their inclusion in the Brel API.

The justification lies in their utility for debugging. Errors in networks are common in XBRL submissions. The link role and link name function as verification tools for both filers and analysts. Although their presence in the API might be temporary, they currently serve a vital role in debugging.

Furthermore, the `INetworkNode` interface offers a way to obtain a node's child nodes via `get_children`. While direct access to a parent node is not available, one can identify a node's parent by navigating from the graph's roots, provided by the `INetwork` interface.

These functionalities suffice for navigating through a network. The subsequent methods address the retrieval of elements that a node points to.

Given the variety of elements a node can reference, the `INetworkNode` interface includes the `points_to` method. This method returns a string that indicates the type of element that the node points to. The possible return values are `fact`, `resource` and `report element`.

Additionally, the interface defines the `get_fact`, `get_resource`, and `get_report_element` getters. If the node does not point to the requested element, the methods raises an exception.

The methods `get_arc_role` and `get_arc_name` disclose diagnostic information about the network's underlying XML structure.

### 5.6.2 Network Types

As outlined in section 4.6, XBRL features six different network types. Each type is represented by its own Network and Node classes in Brel, named to reflect the network type they represent, with the suffix `Network` or `NetworkNode`. The table 5.1 lists the different network types and their corresponding classes.

Table 5.1: Network types and their corresponding classes

| **Network type** | **Network class** | **Node class** |
|---|---|---|
| Presentation | `PresentationNetwork` | `PresentationNetworkNode` |
| Calculation | `CalculationNetwork` | `CalculationNetworkNode` |
| Definition | `DefinitionNetwork` | `DefinitionNetworkNode` |
| Label | `LabelNetwork` | `LabelNetworkNode` |
| Reference | `ReferenceNetwork` | `ReferenceNetworkNode` |
| Footnote | `FootnoteNetwork` | `FootnoteNetworkNode` |

All of these classes implement the `INetwork` and `INetworkNode` interfaces without any modifications. Given the interfaces' simplicity and comprehensive access to network information, alterations are unnecessary for each network type. The unique semantic meanings of the networks are addressed through helper functions, which this chapter does not cover. For completeness, the following diagrams show the inheritance structure of the network- and node-classes.

Figure 5.7: UML diagram of the network classes

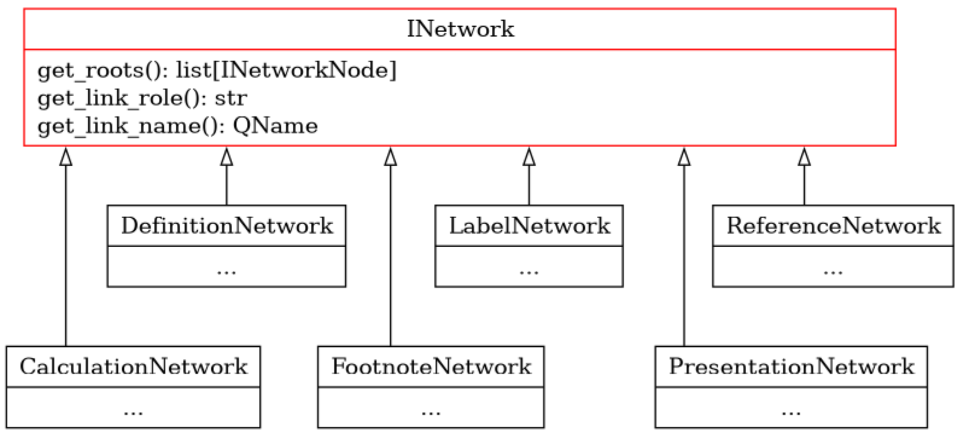
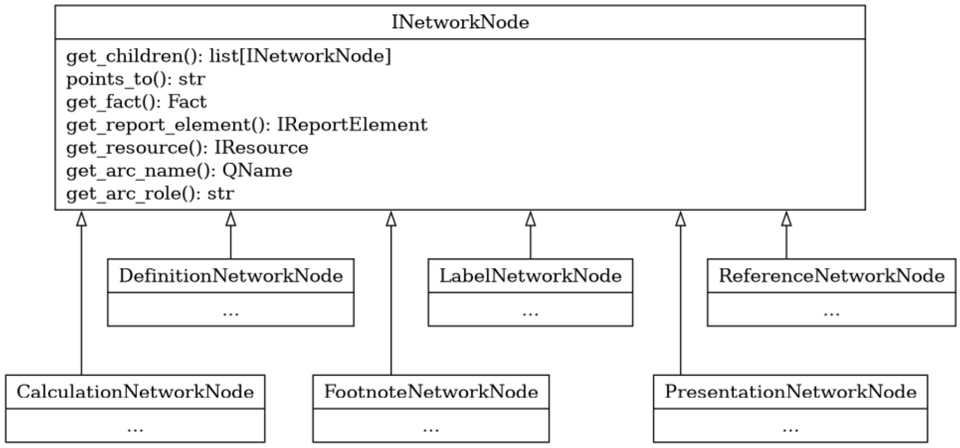


Figure 5.8: UML diagram of the node classes




By addressing network- and node classes, Brel has successfully covered all aspects of XBRL it aimed to encompass. What remains concerning the API is to explore how the latter section of this chapter addresses research question 2.2.

## 5.7 Answering Research Question 2

The latter portion of this chapter presents Brel's approach to depicting networks and resources, elements not currently addressed by the OIM. Consequently, many design choices discussed in this section do not draw from OIM guidelines. This segment aims to respond to research question 2.2.

> **RQ2:** How can the non-OIM sections of XBRL be converted into an easy-to-use python API that is consistent with the OIM?

Research question 2.2 is twofold. First, the question asks how the non-OIM parts of XBRL can be converted into a python API. Secondly, it seeks to understand how this API can align with the OIM.
The answer to the first part is detailed constructively in the second half of this chapter. Here, an API for networks and resources is presented, which effectively deconstructs the non-OIM components of XBRL into their fundamental elements.
Addressing the second part requires a more conceptual approach. Essentially, Brel integrates the OIM with the non-OIM elements of XBRL through the introduction of report elements and characteristics.
Earlier in this chapter, a Python API based on the OIM was introduced. The primary aim of the OIM is to facilitate the reporting of facts, each possessing characteristics like concepts, explicit dimensions, entities, etc.
The latter part of this chapter focuses primarily on networks and resources, where networks are linked to report elements, among other things.
While report elements are part of the OIM framework, they are not strictly essential for reporting facts. Aside from concepts, dimensions, and members, the OIM does not refer to any other report elements.
The link between these two segments of the chapter is established through characteristics and report elements, specifically three types of characteristics that essentially serve as wrappers for report elements. These are the concept characteristic, the explicit dimension characteristic, and the typed dimension characteristic. The interaction between characteristics and report elements is depicted in figure 5.9.

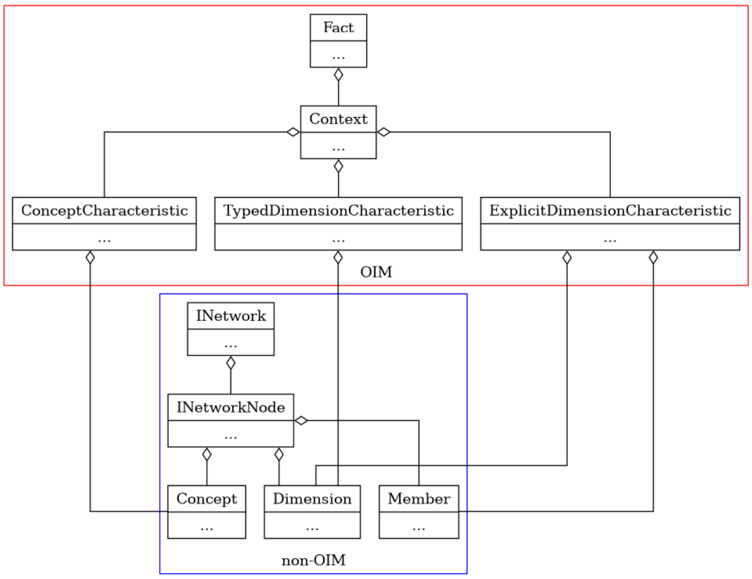



Figure 5.9: The Interaction Between Characteristics and Report Elements

In this setup, while the OIM utilizes characteristics to describe facts, networks employ the report elements within these characteristics. For instance, the `Concept` class could have implemented both the `IReportElement` and `ICharacteristic` interfaces. However, given that concepts in facts and networks serve different purposes, they should not be interchangeable. Consequently, Brel employs different classes for the two distinct use cases.

## 5.8 API Summary

This concludes the chapter on the Brel API. It addressed both the OIM and non-OIM elements of XBRL. Additionally, it detailed how these components are integrated into a singular API, thereby addressing research questions 2.2 and 2.2. Following this comprehensive coverage of the API and the foundational XBRL standard, the subsequent chapter will focus on the implementation of the Brel API.

# Chapter 6

# Implementation

The development of the API aligns closely with the API's design presented in Chapter 5. Brel is created using the Python programming language. Python, a high-level language, is among the most favored programming languages globally. It also enjoys popularity beyond the realms of computer science and software engineering. The 2020 Stack Overflow Developer Survey ranks Python as the fourth most favored programming language among all participants, not limited to professional developers [24].
Readers can interpret Brel's implementation as converting XBRL reports into Python objects. The majority of these conversions from design to implementation are direct and will be briefly discussed in the initial section of this chapter.
Brel's implementation diverges from the standard XBRL in three main aspects: DTS caching, namespace normalization, and networks. Each aspect is detailed in its own section within this chapter.
This chapter is exclusively concerned with XBRL reports in the XML format.

## 6.1 General Implementation

Brel systematically processes XBRL reports using an eager bottom-up strategy. The process begins with the fundamental units of XBRL reports - the report elements. Once every report element is parsed, Brel progresses to interpreting facts and their related characteristics. Subsequently, Brel examines all networks along with their connected resources. Finally, Brel analyzes the components of the report.
The rationale for this bottom-up method is the reliance of both networks and facts on report elements. Networks require report elements as their nodes may link to these elements. Similarly, facts need report elements because their attributes can be associated with concepts, dimensions, and members. It is common for both networks and facts to reference identical report elements. Hence, their corresponding Python classes should utilize the same instances of report elements. Adopting a bottom-up approach guarantees that all report elements are fully parsed prior to their utilization in networks and facts. The subsequent four sections of this chapter will briefly discuss each phase of Brel's bottom-up parsing approach.

### 6.1.1 Parsing Report Elements

Report elements represent the most fundamental components of XBRL reports. As such, they do not rely on other XBRL elements and are parsed first. These elements are specified in the XBRL report's taxonomy set, which consists of a series of `.xsd` files in XML format. For now the reader can assume that all files are

stored locally on the user's computer. While XBRL does not mandate local storage of the taxonomy set, Brel requires it. However, Brel is designed to automatically download the taxonomy set from the internet if it is not already available locally. Details about this downloading process are in section 6.4.
Taxonomies comprise three element types: linkbases, roles, and report elements. Linkbases are covered in section 6.2. Roles are addressed in section 6.1.3. This section focuses on report elements.
In XBRL, a taxonomy can reference other taxonomies and assign them a namespace prefix. For the moment, it is assumed that different taxonomies concur on the namespace prefix and URI for a given taxonomy. Brel validates this assumption through a procedure known as namespace normalization, discussed in section 6.3. When there is consensus on a prefix and URI for a taxonomy, all report elements defined within inherit the same prefix and URI as part of their QName.
Within a taxonomy, report elements are arranged as a flat list of XML elements. Each XML element is uniquely identified by a name attribute, which denotes the local name of the report element's QName. Given that the Brel API identifies six distinct types of report elements, Brel must determine the specific type for each XML element. This decision is not based on a single attribute in the XML element but rather on a combination of various attributes. The methodology used to ascertain the type of each report element is detailed in the following table:

| **Report element type** | **XML abstract attribute** | **XML substitutionGroup attribute** | **XML type attribute** |
|---|---|---|---|
| Member | | "xbrli:item" | "domainItemType" |
| Concept | "false" | | |
| Hypercube | "true" | "xbrldt:hypercubeItem" | |
| Dimension | "true" | "xbrldt:dimensionItem" | |
| Abstract | "true" | "xbrli:item" | |

Table 6.1: Determining the type of report element

Brel implements the procedure outlined in table 6.1 to identify the type of each report element. It examines the table from the top to the bottom, choosing the first row that fulfills all the specified conditions. If a cell in the table is blank, Brel disregards that particular condition.
The table does not include a row for the "LineItems" type. This is because line items and abstracts are indistinguishable based on their XML attributes alone. They can be differentiated only through their placement within a definition network. As a result, Brel initially categorizes both line items and abstracts as abstracts. Later, within the context of the definition network, Brel determines which abstracts are actually line items and adjusts their types accordingly. This procedure is further elaborated in section 6.2.
After parsing all report elements, Brel establishes a lookup table for these elements. This table, when provided with a QName, returns the corresponding instance of the report element. Brel utilizes this lookup table extensively in the subsequent stages of the parsing process.

### 6.1.2 Parsing Facts

Brel processes facts immediately following the parsing of report elements. Facts are analyzed prior to networks because footnote networks may reference facts.
Facts are solely defined in the instance document of the XBRL report. This document is an XML file containing a straightforward list of facts, syntactic contexts, and units, represented as XML elements. It might also include a list of footnotes, which are detailed in section 6.2.

**Fact** XML elements hold the fact's value and references to both the syntactic context and unit. The XML element's tag represents the QName of the concept associated with the fact.
**Syntactic context** XML elements outline a part of a fact's characteristics. They differ from `Contexts` as defined in the Brel API. A `Context` in Brel encompasses all characteristics of a fact, while a syntactic context includes only the period, entity, and dimensions of a fact. During parsing, Brel initially uses syntactic contexts to create `Context` instances. Subsequently, it supplements the `Context` with the remaining characteristics - the concept and the unit.
**Unit** XML elements, as their name implies, define the unit of a fact.
The rationale for XBRL segregating facts, syntactic contexts, and units into distinct XML elements is to minimize redundancy. Multiple facts can share the same syntactic context and unit.
Brel parses all facts by identifying all fact XML elements and resolving their links to syntactic contexts and units. It reutilizes units, entities, and dimensions across various facts.

### 6.1.3 Parsing Components

Components represent the final aspect of the XBRL report that Brel parses. By this stage, Brel has already processed all report elements, facts, and networks. This chapter has not yet delved into networks due to their complexity, which is addressed in a separate section, section 6.2. For the moment, the reader can assume that Brel has successfully parsed all networks and a network lookup table is in place.
Components, akin to report elements, are specified in the XBRL report's taxonomy set. In XBRL terminology, these are referred to as "roleTypes" rather than "components". To parse all components, Brel examines every taxonomy file for roleType XML elements. These roleType XML elements encompass three elements: a role URI, an optional description, and a list of used-on elements. `Components` in Brel directly extract both the role URI and the description from the roleType XML element. To identify the networks associated with a component, Brel searches the network lookup table using the role URI.
The used-on elements denote a list of network types authorized to utilize the component. For instance, if the network lookup yields a `PresentationNetwork` instance, the roleType XML element must include "presentationLink" in its used-on elements list.
This segment concludes the discussion on Brel's general implementation. Excluding networks, this section has encompassed every aspect of XBRL and Brel's method of parsing it. The ensuing section will delve into the intricacies of network parsing.

## 6.2 Implementation of Networks

In Chapter 4, the concept of networks was introduced, which was then further explored in Brel's context in Chapter 5. In Brel, a network consists of two distinct classes: `INetwork` and `INetworkNode`.
A Brel network is structured as a directed acyclic graph. [1] Each node within this graph maintains an ordered list of children and can have, at most, one parent. It is not mandatory for the network to be connected; hence, it may contain several disjoint subgraphs.
Technically, a network in Brel is a collection of root nodes. These root nodes are linked to their respective children, who then connect to their own children, and so

[1] While the XBRL specification permits cycles within networks, Brel does not support this feature.

forth. Thus, to navigate through a network, knowledge of its root nodes is sufficient. The `INetwork` class offers a method to access all the network's root nodes, and the `INetworkNode` class provides a method to retrieve the children of a node, enabling the traversal of the network.
An important aspect of Brel's network implementation is that networks cannot be devoid of nodes. They must contain at least one node.

### 6.2.1 Transforming Links into Networks

Section 4.6 outlined various network types, illustrating how each consists of a collection of arcs, locators, and resources. It also clarified how locators and resources symbolize nodes, and arcs represent edges within a network. Thus, converting a link into a network essentially involves translating a list of nodes and edges into a graph. Brel follows a four-step algorithm to parse links:

1. Initially, it examines all elements within the link. For elements identified as locators or resources, Brel generates an `INetworkNode` class instance. In the case of an arc element, Brel notes the arc's from and to attributes in an edge list.

2. In the second step, with all nodes already instantiated, Brel iterates over the edge list, appending the to-node as a child of the from-node.

3. Subsequently, Brel reviews each node, adding those without a parent to the network's root list. This root list is encapsulated in an `INetwork` instance.

4. Finally, Brel applies any overarching implications of the network to the report. For instance, if a label network assigns a label to a concept, Brel incorporates this label into the report's concept. The specific implications vary based on the network type.

Chapter 5 introduced the diverse network types. For each network variant, Brel provides corresponding node and network classes, all derived from the `INetwork` and `INetworkNode` interfaces. Brel employs the factory pattern to generate appropriate network and node instances for a given link. Each network type has its dedicated factory, which is utilized in the algorithm 6.2.1 to create the relevant network and node instances.

### 6.2.2 Parsing Locators

As indicated in section 4.6, locators serve to reference report elements or facts. This section details the method Brel uses to interpret locators.
XBRL locators utilize XPointer[35] expressions for referencing other XML elements, potentially from different XML documents. These XPointers in XBRL take the form `filename#id`, where `filename` denotes the XML document's URI and `id` is the id of the XML element. To interpret a locator, Brel first identifies the targeted XML element. Subsequently, it translates this XML element into a report element or a fact.
Brel accomplishes this by tracking the id of each fact and report element it parses. It constructs a lookup table mapping ids to their corresponding facts and report elements. Whenever Brel encounters a locator during parsing, it interprets the locator by consulting the lookup table with the locator's id.

### 6.2.3 Parsing Resources

Resources, the alternate type of element referable by arcs, do not point to other elements within the report. Instead, they directly encapsulate the value of the element they signify.
The current XBRL 2.1 specification outlines three inherent resource types: label, reference, and footnote, though custom resources are feasible.
Resources comprise three components: a role, a label, and a value.
The role is a URI defining the resource's type. For instance, the `terseLabel` role[2] denotes a label resource offering a concise label for a concept, while `footnote` role[3] indicates a footnote resource associated with a concept.
The label functions as an identifier for the resource, utilized by arcs to reference the resource. This label should not be mistaken for a concept's human-readable label.
The resource's value embodies the actual resource content. For labels, this means the label text itself. For references, it is typically a dictionary pointing to an external resource, like an article in the SEC's Code of Federal Regulations.
Given that resources contain all necessary information for parsing, Brel does not need to resolve external references to analyze them. Hence, their parsing is straightforward.

### 6.2.4 Consequences of Networks

As previously noted in section 6.2.1, networks can influence the entire report. Two widespread outcomes associated with all networks are labels and line items promotion.
Labels serve to assign human-readable titles to report elements, generated via the label network `link:labelLink`. The intricacies of label links are elaborated in section 4.6.6. Brel processes report elements before networks because many networks include locators pointing to report elements. Thus, when Brel interprets a label network, it is already aware of all report elements referenced in the network.
After parsing the label network, Brel examines each label within it. If the network contains an edge between a label and a report element, Brel adds the label to the report element.
The other implication of networks is the promotion of line items. Report elements, defined in the taxonomy, come in six varieties: concepts, abstracts, line items, members, dimensions, and hypercubes. Determining the exact type of a report element from its XML element in the taxonomy is not always straightforward. Abstracts and line items are represented by structurally similar XML elements. Two methods are employed to distinguish them:

1. The first method involves examining the QName of the element. "LineItems" often appears within the QName of line items. However, this is a convention rather than a rule. Thus, it is not guaranteed that every line item's QName will contain "LineItems".

2. The second method assesses the element's role in networks, particularly in definition networks that outline relationships between report elements. For instance, the arc role `hypercube-dimension` defines the link between a hypercube report element and a dimension report element. Likewise, the arc role `all` denotes the connection between a hypercube report element and a line items report element.

[2]`http://www.xbrl.org/2003/role/terseLabel`
[3]`http://www.xbrl.org/2003/role/footnote`

Brel adopts the second strategy to differentiate line items from abstracts. Initially, it treats all report elements as abstracts. Then, during the parsing of a definition network, it considers the arc roles within the network. If an arc with the role `all` connects a hypercube to an abstract, Brel classifies the abstract as a lineitem.

This section concludes the discussion on the implementation of networks. Combined with the previous segments of this chapter, it encompasses the entirety of XBRL and Brel's parsing methodology. However, section 6.1.1 made an assumption about taxonomies that is not always accurate. Each taxonomy can incorporate other taxonomies under a specific prefix-URI pair. The presumption was that there is a universal agreement on the prefix-URI pair for each taxonomy. This presumption does not always hold true. The upcoming section will detail Brel's approach in handling such scenarios.

## 6.3 Namespace Normalization

Both chapters 4 and 5 reveal that Brel utilizes QNames to identify various elements within the XBRL report. QNames are a fundamental concept in XML and XML-based languages, like XBRL. As such, for most QNames in Brel, the necessary information is directly retrievable from the corresponding XML elements in both the XBRL taxonomy and the XBRL filing. However, a key distinction exists between QNames in XML and those in Brel, particularly in terms of Namespace bindings.

Namespace bindings represent the associations between prefixes and namespace URIs. In XBRL, a URI typically links to a taxonomy file. The prefix is employed to succinctly reference the namespace URI within the XBRL filing. For instance, the prefix `us-gaap` might be bound to the namespace URI `http://fasb.org/us-gaap/2023`.

In XML documents, these namespace bindings can be specified for individual elements. Child elements inherit their parent elements' namespace bindings, except when they establish their own. This flexibility allows the creation of intricate namespace hierarchies, where each element can possess unique namespace bindings. Conversely, Brel's approach to namespace bindings is more simplified, maintaining a flat and globally defined structure.

This section is devoted to discussing the implementation of QNames in Brel, focusing particularly on namespace bindings. Given that XML documents include information irrelevant to namespace bindings, the figures in this section omit any extraneous information that is not relevant to namespace bindings and their hierarchical structure. An example figure is provided below.

Figure 6.1: Example of namespace bindings defined on a per-level basis

```
root
├─ element1 foo = "http://foo.com"
│  └─ element2 bar = "http://bar.com"
│     └─ element3
└─ element4 baz = "http://baz.com", foo = "http://other-foo.com"
```

The term **namespace normalization** refers to the process of converting a hierarchical structure such as 6.1 into a flat structure. This process not only simplifies the namespace hierarchy but also addresses potential conflicts in namespace bindings that may arise during the simplification process. The rationale behind adopting a flat structure for namespace bindings in Brel is to reduce complexity for the user.

### 6.3.1 Flattening Namespace Bindings

Flattening a tree structure into a flat one is a common challenge in computer science. A popular solution is the use of a depth-first search algorithm, which is the method employed in Brel to flatten the XBRL taxonomy's namespace hierarchy.
It is important to remember that in XML, child elements inherit their parent elements' namespace bindings. Consequently, when flattening the namespace hierarchy, it is crucial to ensure that all parent namespace bindings are also present in the children, except where the children define their own namespace bindings.
To illustrate this process, the following figure depicts a flattening of the namespace hierarchy shown previously in figure 6.1:

Figure 6.2: Flattened version of the XML snippet using our custom notation

```
root
├─element1 foo = "http://foo.com"
├─element2 foo = "http://foo.com", bar = "http://bar.com"
├─element3 foo = "http://foo.com", bar = "http://bar.com"
└─element4 baz = "http://baz.com", foo = "http://other-foo.com"
```

In this representation, the namespace hierarchy is transformed into a flat structure [4]. All elements are positioned on the same level, and the sequence of elements is determined by the depth-first search algorithm.
Each child element inherits the namespace bindings from its parent. Therefore, `element2` and `element3` inherit the namespace bindings from `element1`.
To extract the namespace bindings from this flat structure, one can simply iterate over the elements and record the namespace bindings of each. For the example provided, the extracted list of namespace bindings would be as follows:

Figure 6.3: Extracted namespace bindings from the flattened hierarchy

```
foo = "http://foo.com"
bar = "http://bar.com"
baz = "http://baz.com"
foo = "http://other-foo.com"
```

### 6.3.2 Handling Namespace Binding Collisions

It may have been noted by the attentive reader that the list of namespace bindings from the previous section includes two bindings for the `foo` prefix. The first binding is `foo = "http://foo.com"`, while the second is `foo = "http://other-foo.com"`. Such a scenario is referred to as a collision. While Brel generally prohibits and resolves most collisions in namespace bindings, there are exceptions. The subsequent section details the various types of collisions and Brel's approach to managing them.
In Brel, three kinds of namespace collisions can occur:

- **Version Collision**: This occurs when two namespace bindings share the same prefix and namespace URI, differing only in the version specified within the URI. The version is identified by the numbers and dashes in the namespace URI, indicating its relative recency.

  Example: `foo = "http://foo.com/2022"` and `foo = "http://foo.com/2023"`

[4] Technically, the namespace hierarchy is not entirely flat due to the presence of the root element. However, since the root element does not contain any namespace bindings, it has no impact on the namespace hierarchy.

- **Prefix Collision**: This type of collision happens when two namespace bindings share the same prefix but point to different *unversioned* namespace URIs. An unversioned namespace URI is one without any version-related details.

  Example: `foo = "http://foo.com"` and `foo = "http://other-foo.com"`

- **Namespace URI Collision**: This collision occurs when two namespace bindings have identical *unversioned* namespace URIs but utilize different prefixes.

  Example: `foo = "http://foo.com"` and `bar = "http://foo.com"`

### 6.3.3 Resolving Version Collisions

Version collisions arise when two namespace bindings share the same prefix and namespace URI, but differ in their respective versions.
Consider an XBRL filing with the following namespace bindings, which exemplifies a version collision:

Figure 6.4: Illustration of a Version Collision

```
root
├─element1 foo = "http://foo.com/01-01-2022"
│  └─foo:bar
└─element2 foo = "http://foo.com/01-01-2023"
   └─foo:bar
```

The example above demonstrates a version collision. In Brel, version collisions are permissible, as different versions of the same namespace URI often coexist within a single XBRL filing. If a user seeks a QName `foo:bar`, Brel will automatically search the `bar` element under both versions of the namespace URI.

### 6.3.4 Resolving Prefix Collisions

A prefix collision arises when two namespace bindings use the same prefix but are linked to different *unversioned* namespace URIs. An example of a prefix collision is depicted in the following figure:

Figure 6.5: Illustration of a Prefix Collision

```
root
├─element1 foo = "http://foo.com"
│  └─foo:bar
└─element2 foo = "http://other-foo.com"
   └─foo:bar
```

In Brel, prefix collisions are not permitted since they can lead to ambiguity. For example, two separate taxonomies might both include the report element `bar`, defined in one as a concept and in the other as a hypercube. Should the filing employ the identical prefix `foo` for these taxonomies, Brel would face challenges in differentiating between the two distinct report elements.
To resolve such a collision, Brel will rename one of the conflicting prefixes. For instance, in the example above, Brel will change `element2`'s binding from `foo = "http://other-foo.com"` to `foo1 = "http://other-foo.com"` and update all relevant QNames with the new prefix. Brel will also indicate that the binding `foo = "http://other-foo.com"` has been modified to `foo1`.

Figure 6.6 depicts the figure 6.5 after resolving the prefix collision.

Figure 6.6: Representation of a Resolved Prefix Collision

```
root
├─element1 foo = "http://foo.com"
│ └─foo:bar
└─element2 foo1 = "http://other-foo.com"
  └─foo1:bar
```

### 6.3.5 Resolving Namespace URI Collisions

A namespace URI collision occurs when two namespace bindings share the same *unversioned* namespace URI but have different prefixes. The figure below exemplifies a namespace URI collision:

Figure 6.7: Illustration of a Namespace URI Collision

```
root
├─element1 foo = "http://foo.com"
│ └─foo:bar
└─element2 bar = "http://foo.com"
  └─bar:baz
```

In Brel, namespace URI collisions are not permitted because they can cause errors where elements are not found. For instance, consider the scenario where a user searches for the QName `foo:baz` in the previously mentioned example. Brel would fail to locate it. However, since both `foo` and `bar` are linked to the identical namespace URI, Brel should be capable of finding the QName `bar:baz`. This rationale underpins the prohibition of namespace URI collisions in Brel.

To resolve such collisions, Brel selects one prefix as the preferred option and renames the other to eliminate the conflict Brel opts for the shorter prefix as the preferred one. If both prefixes are of equal length, the choice is based on alphabetical precedence.

In our example, `bar` is chosen as the preferred prefix. Consequently, Brel will rename the `foo` prefix, along with all its occurrences, to `bar`.

Figure 6.8 depicts the figure 6.7 after resolving the namespace URI collision.

Figure 6.8: Representation of a Resolved Namespace URI Collision

```
root
├─element1 bar = "http://foo.com"
│ └─bar:bar
└─element2 bar = "http://foo.com"
  └─bar:baz
```

Certain prefixes are deemed special and are always chosen as the preferred prefix, regardless of their length or alphabetical order. These special prefixes need not be explicitly defined in the XBRL taxonomy. If a namespace binding corresponds to the same namespace URI as a special prefix, that special prefix will automatically be selected as the preferred one.

The special prefixes include:

| | | | | |
|---|---|---|---|---|
| xml | xlink | xs | xsi | xbrli |
| xbrldt | link | xl | iso4217 | utr |
| nonnum | num | enum | enum2 | formula |
| gen | table | cf | df | ef |
| pf | uf | ix | ixt | entities |

Figure 6.9: Table containing all special prefixes

Each prefix in figure 6.9 is associated with a specific namespace URI. For example, the prefix `xsi` corresponds to the namespace URI `http://www.w3.org/2001/XMLSchema-instance`. Should an XBRL filing include a namespace binding like `foo = "http://www.w3.org/2001/XMLSchema-inst` then `xsi` will be selected as the preferred prefix. Consequently, all instances of `foo` will be renamed to `xsi`.
Having grasped the concept of namespace normalization, we have now addressed one assumption highlighted in section 6.1. Yet, there remains another assumption that needs to be tackled.

## 6.4 Discoverable Taxonomy Set (DTS) Caching

The second significant assumption made in section 6.1 is that all files pertaining to both the taxonomy set and the XBRL report are available locally on the user's computer. However, this is often not the case. Typically, only the XBRL report itself is stored locally. The taxonomy files referenced within the XBRL report usually point to additional taxonomy files that are not locally stored. As discussed in section 4.3, taxonomies may include references to other taxonomies. To successfully parse the XBRL report, Brel must identify and download the complete set of all linked taxonomy references, a process known as DTS caching.

### 6.4.1 Discovery Process in DTS Caching

The 'D' in DTS caching represents "discoverable," implying that Brel's initial step is to identify all taxonomy files referenced by the XBRL report. Brel commences this process by parsing the taxonomy file included in the XBRL report and extracting all its taxonomy references. Taxonomy files may reference other taxonomy files in several ways:

- **schemaRef** - The most prevalent method is through the `schemaRef` element. This element contains a `href` attribute with a URL pointing to another taxonomy file.
- **linkbaseRef** - Another way is using the `linkbaseRef` element. Similar to `schemaRef`, this element also includes a `href` attribute.
- **import** - Taxonomy files might reference others using the `import` element, which has a `schemaLocation` attribute specifying a URI leading to another taxonomy file.
- **include** - Similarly, the `include` element, containing a `schemaLocation` attribute, can also reference additional taxonomy files.

When Brel parses a taxonomy file, it identifies all the taxonomy references within and adds them to a list of references to be processed. After parsing a given taxonomy file, Brel selects the first reference from this list and repeats the process of parsing and extracting references. This approach resembles a breadth-first search through the taxonomy reference graph. If Brel has already processed a particular taxonomy file, it does not parse it again.

### 6.4.2 Downloading Taxonomies

Retrieving the taxonomy file from a given URI is straightforward for most URIs. However, certain URIs are relative, meaning the URI specifies the location of another taxonomy file in relation to the current one[3]. Brel deduces the domain of relative URIs by recalling the domain from which the relative URI was referenced. It then merges the domain of the current taxonomy file with the relative URI to create a complete URI. When storing taxonomy files, Brel names them based on their absolute URIs, ensuring each file has a unique name. Since URI-based file names are not always valid, Brel removes any illegal characters to form a valid file name.

By employing the discovery and downloading mechanisms, Brel successfully retrieves all taxonomy files referenced by the XBRL report. Brel then saves these files in the `dts_cache` directory, addressing the second crucial assumption outlined in section 6.1.

## 6.5 Addressing Research Question 3

Now that Brel's implementation for the XBRL XML syntax is complete, we can address research question 2.2:

> **RQ3:** How can the library be designed to accommodate multiple formats in the future?

To make Brel compatible with multiple formats, both the design and implementation of the Brel API need to be format-agnostic. The design of the Brel API is largely format-independent. Its first segment is grounded in the OIM, which is a logical data model and thus inherently format-agnostic. The latter half of the Brel API, while based on the XBRL XML syntax, largely abstracts away the specifics of this format.

The only exceptions are the `get_link_role`, `get_link_name`, `get_arc_role`, and `get_arc_name` methods. These methods return attributes bearing the same names in XML, primarily serving debugging purposes. They are not essential to the API's functionality. Therefore, the second half of the Brel API is almost entirely format-agnostic, with the exception of debugging methods.

The primary aspect where Brel relies on the XBRL XML syntax is the `QName` class. This class mirrors the QName structure in XML, comprising a prefix, a namespace URI, and a local name. However, even though QNames originate from XML, they have been adopted in other XBRL specifications. Notably, both the JSON[21] and CSV[20] specifications of XBRL adopt QNames in a similar structure to the XML specification, rendering the `QName` class format-neutral.

Given the API's format-agnostic design, the aspect of Brel that relies on the XBRL XML syntax is exclusively the parser. Brel's parser is encapsulated in a distinct module, named `brel.parser.XML`. To support different formats, only this parser module needs modification, allowing the rest of Brel to remain as is.

## 6.6 API Summary

This marks the end of the implementation chapter, which focused on the development of Brel. Drawing on insights from chapter 4 and chapter 5, it detailed how Brel transforms XBRL reports in XBRL XML syntax into Python objects that conform to the Brel API. Additionally, it outlined Brel's response to research question 2.2. The following chapter will assess Brel, evaluating both its accuracy and performance.

# Chapter 7

# Results

Following the discussion on XBRL, the Brel API, and its implementation, we are now in a position to assess Brel in light of the objectives outlined in section 2.2. The evaluation will prioritize correctness and performance, in that order. Additionally, this chapter will qualitatively discuss Brel's robustness and usability.
The usability of Brel will be assessed through the development of a simple CLI tool for viewing XBRL reports. This tool will demonstrate the capabilities of the Brel API and act as a practical example of its application.
Although this thesis does not explicitly list Brel's performance as a requirement, it remains a crucial aspect of the evaluation. Assessing Brel's performance will provide a benchmark for future versions of the software, facilitating performance comparisons over time.
The assessment of Brel's correctness will involve the use of XBRL conformance suites. Furthermore, we will conduct an analysis of a selected component of an XBRL report, comparing the structure Brel extracts with that produced by the XBRL viewer Arelle. This comparison will offer a qualitative evaluation of Brel's accuracy.
Robustness, being a qualitative metric, presents challenges in measurement. However, by loading the 10K and 10Q reports of the 50 largest US companies by market capitalization as of the date of this analysis, we can gain valuable insights into Brel's practical robustness.

## 7.1 Usability

One of the goals of this thesis is to create a usable API for working with XBRL reports. To evaluate the usability of the Brel API, we will implement a simple CLI XBRL report viewer. This viewer will cover every feature of the Brel API and serve as a proof of concept for the Brel API.
The CLI XBRL report viewer will be implemented in Python and will use the Brel API to load and display XBRL reports. It will be able to load XBRL reports from local files and from URLs.
The viewer will be able to display the following information about an XBRL report:

- The facts in the report, which can be filtered by concept and a dimension. [1] For each fact, the viewer should display all characteristics of the fact in a easy-to-read manner.

[1] Filters for entity, period, and unit are not implemented since most reports have very few entities, periods, and units.

- The components of a report together with their networks. The viewer should be able to display the relationships between report elements in the networks in a human-readable manner.

Before implementing the viewer, we will first install Brel and its dependencies. Since Brel is published on the Python Package Index (PyPI), we can install it using pip.

```
pip install brel-xbrl
```

After installing Brel, we can start implementing the viewer. The viewer will be implemented in a single file called `viewer.py`. The first part of the implementation is shown in figure 7.1. It shows the imports and the command-line interface (CLI) definition of the viewer.

Figure 7.1: The imports and the CLI argument definition of the XBRL report viewer

```python
import argparse, brel

# Parse the command line arguments
parser = argparse.ArgumentParser()
parser.add_argument("file", nargs="?")
parser.add_argument("--facts", default=None)
parser.add_argument("--components", default=None)
args = parser.parse_args()
```

Listing 7.1: The implementation of the CLI XBRL report viewer

The viewer is implemented as a command-line interface (CLI) using the `argparse` module. It has two subcommands: `facts` and `components`. The `facts` subcommand is used to display the facts in an XBRL report, and the `components` subcommand is used to display the components of a report together with their networks. Both subcommands take a single argument, which acts as a filter for the facts and components that are displayed.
The facts filter is used to filter the facts in the report by concept or dimension. If a fact has the concept or dimension that matches the filter, it will be displayed. The components filter is used to filter the components in the report by URI. If a component has a URI that contains the filter as a substring, it will be displayed [2]. First of all, the viewer uses `Filing.open` on line 11 to open the XBRL report. The argument of this method can be a local file path or a URI. Since Brel potentially needs to download the report from the internet, the call to `Filing.open` can take a few seconds to complete. The figure 7.2 shows the second part of the implementation of the viewer responsible for loading the report.

Figure 7.2: The implementation of the XBRL report viewer responsible for loading the report

```python
report = brel.Filing.open(args.file)
```

Listing 7.2: The implementation of the CLI XBRL report viewer

The next section of the implementation should deal with the facts portion of the viewer. First, it should get all facts in the report. Then, it should filter the facts that either have the concept or dimension that matches the filter. Next, it should pretty-print the facts in a human-readable manner. We can get all facts

[2]We use a substring since typing out the full URI of a component can be cumbersome.

in the report using the `report.get_all_facts` method. The `fact.get_concept` and `fact.get_aspects` methods can be used to get the concept and aspects of each fact. Since dimensions are just aspects, we can check all aspects of a fact to see if one of them matches the filter. Finally, we can use the `utils.pprint` method to pretty-print the facts in a human-readable manner. `utils.pprint` is a convenience method that, given a list of facts, it calls both `fact.get_aspects` and `fact.get_characteristic` for each fact and pretty-prints the result. The figure 7.3 shows the third part of the implementation of the viewer responsible for displaying the facts in the report.

Figure 7.3: The implementation of the XBRL report viewer responsible for displaying the facts in the report

```python
if args.facts:
    # Get the facts, filter them and print them.
    facts = [
        fact
        for fact in report.get_all_facts()  # gets all facts from the report
        if args.facts == str(fact.get_concept())  # filter by concept or
        or args.facts in map(str, fact.get_aspects())  # filter by any aspect
    ]
    brel.utils.pprint(facts)
```

Listing 7.3: The implementation of the CLI XBRL report viewer

The components portion of the viewer should be implemented in a similar manner. First, it should get all components in the report using `report.get_all_components`. Then, it should filter the components that have a URI that contains the filter as a substring. This is done using the `component.get_uri` method. Finally, it should pretty-print the components in a human-readable manner using `utils.pprint`. The method `utils.pprint` not only works for lists of facts, but also for a list components. For each network in the component, the `utils.pprint` uses a DFS algorithm on both the `network.get_roots` and `network_node.get_children` methods to pretty-print the network in a human-readable manner. It also automatically uses the labels of report elements if they are available. The figure 7.4 shows the fourth and final part of the implementation of the viewer responsible for displaying the components in the report.

Figure 7.4: The implementation of the XBRL report viewer responsible for displaying the components in the report

```python
elif args.components:
    # Get the components that match the filter and print them.
    components = [
        component
        for component in report.get_all_components()  # get all components
        if args.components in component.get_URI()  # filter by URI
    ]
    brel.utils.pprint(components)
```

Listing 7.4: The implementation of the CLI XBRL report viewer

The implementation of the viewer is now complete. The viewer can be used to display the facts and components of an XBRL report using only a few lines of code and using python's built-in list comprehension.
The following two examples illustrate how the viewer can be used to display the facts and components of an XBRL report. Each example will show both the command that is used to display the information and the output of the command. For both

examples, we will use the Q3 2023 10-Q report of Apple Inc. (AAPL) that is available on the SEC's website[11]. To keep the commands simple, we will use the `report.zip` file that contains the report. However, the viewer can also load reports from URLs.

Figure 7.5: Assets in the Q3 2023 10-Q report of Apple Inc.

```
python viewer.py report.zip --facts us-gaap:Assets
```

```
-------------------------------------------------------------------------------------------+
|                                        Facts Table                                         |
+----------------+---------------+------------------------------------+------+--------------+
|        concept |        period |                             entity | unit |        value |
+----------------+---------------+------------------------------------+------+--------------+
| us-gaap:Assets | on 2023-07-01 | {http://www.sec.gov/CIK}0000320193 |  usd | 335038000000 |
| us-gaap:Assets | on 2022-09-24 | {http://www.sec.gov/CIK}0000320193 |  usd | 352755000000 |
+----------------+---------------+------------------------------------+------+--------------+
```

As shown in figure 7.5, the command returns the facts in the report that have the concept `us-gaap:Assets`. It clearly shows the concept, period, entity, unit, and value for both facts. Neither fact has additional dimensions, so they are not displayed.

Figure 7.6: Insider trading arrangements in the Q3 2023 10-Q report of Apple Inc. The output is truncated.

```
python viewer.py report.zip --components InsiderTradingArrangements
```

```
Component: http://xbrl.sec.gov/ecd/role/InsiderTradingArrangements
Info: 995445 - Disclosure - Insider Trading Arrangements
Networks:
link role: .../InsiderTradingArrangements, link name: link:presentationLink
arc roles: ['.../parent-child'], arc name: link:presentationArc
└──[LINE ITEMS] Insider Trading Arrangements [Line Items]
    ├──[HYPERCUBE] Trading Arrangements, by Individual [Table]
    │  ├──[DIMENSION] Trading Arrangement [Axis]
    │  │  └──[MEMBER] All Trading Arrangements [Member]
    │  └──[DIMENSION] Individual [Axis]
    │     └──[MEMBER] All Individuals [Member]
    ├──[CONCEPT] Material Terms of Trading Arrangement [Text Block]
    │   ...
    └──[CONCEPT] Trading Arrangement, Securities Aggregate Available Amount
```

As shown in figure 7.6, the command returns the component identified with URI `http://xbrl.sec.gov/ecd/role/InsiderTradingArrangements`. It clearly shows the URI, label, and networks for the component. The networks are pretty-printed in a human-readable manner and show the relationships between report elements in the networks. Additionally, the network shows the link/arc roles and names for each network, which can be useful for debugging.

This example shows that the Brel API is easy to use and can be used to implement a simple CLI XBRL report viewer. The viewer is able to display the facts and components using only a few lines of code and using python's built-in list comprehension. Of course, the viewer is not perfect and can be improved in many ways. The viewer also does not cover every feature of the Brel API, but it serves as a proof of concept for the Brel API. More examples are available both in the Brel documentation[25] and in the Brel repository[26].

## 7.2 Correctness

To evaluate Brel's correctness, we will use Charles Hoffman's XBRL conformance suite [10]. The conformance suite contains a set of XBRL reports that are known to be correct. It checks for conformance to the XBRL specification from both a syntactic and a logical point of view. Brel will ignore the logical checks, as it only deals with the syntactic part of the XBRL specification.
We will not cover either the SEC's or the ESMA's conformance suites, as they contain many test cases that are not relevant to Brel in its current state.
Besides the conformance suite, we will also look at a hand-picked XBRL report. Since the source files of an XBRL report are not human-readable, we will use the XBRL platform Arelle[23] to visualize the structures that Brel extracts from the reports. Arelle closely follows the XBRL specification and can therefore be considered a reliable source of truth. We will compare the structures that Brel extracts from the reports against the structures that Arelle extracts from the same reports. For the sake of brevity, we will only look at a single network within the report. Even though this evaluation only covers a single component within the report, it will give us a more intuitive understanding of Brel's correctness compared to the conformance suite. Brel also contains its own test suite, which is not covered in this thesis.

### 7.2.1 Conformance suite

Charles Hoffman maintains a conformance suite for XBRL [10]. It covers both the syntactic and the semantic aspects of the XBRL specification. The conformance suite contains a set of XBRL reports that are known to be correct or incorrect. It checks for conformance to the XBRL specification from both a syntactic and a logical point of view.
Since Brel only deals with the syntactic part of the XBRL specification, it will be unable to check the logical part of the conformance suite. Therefore reports that are syntactically correct, but semantically incorrect, will be considered correct by Brel.
Figure 7.7 shows the results of running the conformance suite on Brel.

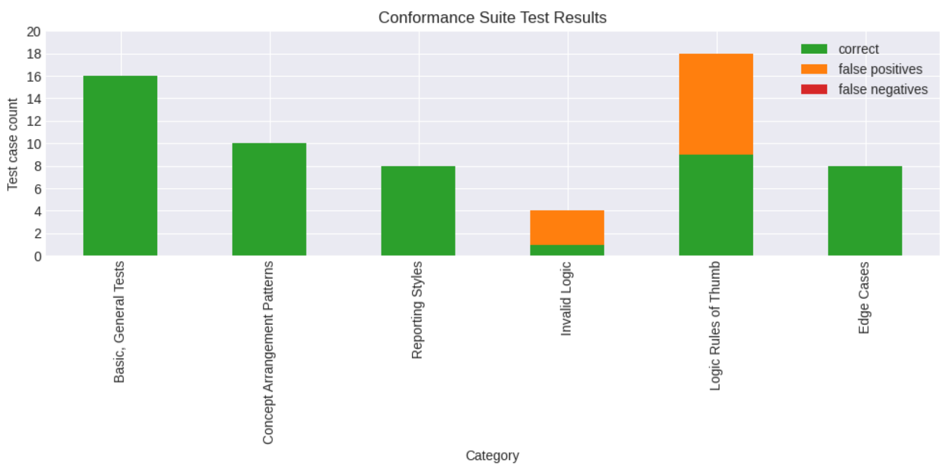



Figure 7.7: Conformance suite results

The conformance suite contains 77 test cases, of which 22 are considered logical checks. Of the 22 logical checks, Brel passes 10 and fails 12. Since Brel only deals with the syntactic part of the XBRL specification, the test cases it passed it passed by coincidence. We still list the results for completeness.

Of the 55 syntactic checks, Brel passes all of them. This is a good result, as it shows that Brel is able to parse XBRL reports correctly. However, it is important to note that the conformance suite is not exhaustive. Also, conformance suite test cases are not necessarily representative of real-world XBRL reports. They serve as a good starting point, but they are not enough to guarantee Brel's correctness.

### 7.2.2 Hand-picked XBRL reports

We will also look at a hand-picked XBRL report. We will compare the structures that Brel extracts from them against the structures that the XBRL specification prescribes. Since the source files of an XBRL report meant to be read by machines, we will use the XBRL platform Arelle to visualize the structures that Brel extracts from the reports. Arelle is a mature, robust and widely used XBRL platform and can therefore be considered a reliable source of truth. It closely follows the XBRL specification, including the OIM.

The hand picked report is from the SEC's EDGAR database [30]. In this case we will use Microsoft's 10K report for the fiscal year 2022. [3] This report will be loaded directly from the EDGAR database by providing the URL of the report to Brel.

We will look at the `ComprehenisveIncomeStatement` component and all facts associated with the concepts within its presentation network. Its presentation network acts as a good sanity check, since the information in it is understandable by non-accountants. Furthermore, Brel employs the same network parsing algorithm for all networks, so the results of this evaluation will be representative of Brel's correctness.

First, let us look at the comprehensive income statement as it is presented in Arelle.

| Concept | 2021-06-30 | 2022-06-30 | 2023-06-30 |
|---|---|---|---|
| 100020 - Statement - COMPREHENSIVE INCOME STATEMENTS | | | |
| Statement of Comprehensive Income [Abstract] | | | |
| Net income | 61,271,000,000 | 72,738,000,000 | 72,361,000,000 |
| Other comprehensive income (loss), net of tax: | | | |
| Net change related to derivatives | 19,000,000 | 6,000,000 | -14,000,000 |
| Net change related to investments | -2,266,000,000 | -5,360,000,000 | -1,444,000,000 |
| Translation adjustments and other | 873,000,000 | -1,146,000,000 | -207,000,000 |
| Other comprehensive loss | -1,374,000,000 | -6,500,000,000 | -1,665,000,000 |
| Comprehensive income | 59,897,000,000 | 66,238,000,000 | 70,696,000,000 |

Figure 7.8: Microsoft's 10K report cover page in Arelle

The figure 7.8 shows the comprehensive income statement of Microsoft's 10K report in Arelle. Brel works different from Arelle, as it does not visualize the network and its facts in a graphical manner. However, using the `viewer.py` script we have written in section 7.1, we can visualize the cover page in a textual manner. The figure 7.9 shows the hierarchy of cover page of Microsoft's 10K report in Brel.

[3] Available at `https://www.sec.gov/ixviewer/ix.html?doc=/Archives/edgar/data/789019/000095017023035122/msft-20230630.htm`

```
Component: .../Role_StatementCOMPREHENSIVEINCOMESTATEMENTS
Info: 100020 - Statement - COMPREHENSIVE INCOME STATEMENTS

link name: link:presentationLink
arc roles: ['.../parent-child'], arc name: link:presentationArc
└──[ABSTRACT] Statement of Comprehensive Income [Abstract]
    ├──[CONCEPT] Net Income (Loss)
    ├──[ABSTRACT] Other comprehensive income (loss), net of tax:
    │   ├──[CONCEPT] Net change related to derivatives
    │   ├──[CONCEPT] Net change related to investments
    │   ├──[CONCEPT] Translation adjustments and other
    │   └──[CONCEPT] Other comprehensive income (loss)
    └──[CONCEPT] Comprehensive Income (Loss), Net of Tax, Attributable to Parent

```

Figure 7.9: Microsoft's 10K presentation network in Brel

The figure 7.9 shows the hierarchy of the comprehensive income statement of Microsoft's 10K report in Brel. Note that the labels between the two figures are different, since Arelle might use different label roles from the ones used in Brel. The only difference between the content of the two figures is the fact that Arelle includes the facts associated with the concepts in the network, while Brel only includes the report elements and their relationships.
Luckily, we can use the `viewer.py` and filter all facts that have concepts present in figure 7.9 to visualize the facts associated with the concepts in the network. The output of the `viewer.py` script is shown in figure 7.10. Note that the output has been truncated. Also note that XBRL reports can contain duplicate facts, as they can be reported in different contexts. Arelle does not show duplicate facts, while Brel does.

```
+--------------------------------------------------------------------------------------------------------------------+
| Facts Table                                                                                                        |
+-----------------------------------------------------+------------------------------+------------+-------+-------------+
|                                             concept |                       period |     entity |  unit |       value |
+-----------------------------------------------------+------------------------------+------------+-------+-------------+
|                               us-gaap:NetIncomeLoss | from 2020-07-01 to 2021-06-30 | 0000789019 |  USD  | 61271000000 |
|                               us-gaap:NetIncomeLoss | from 2021-07-01 to 2022-06-30 | 0000789019 |  USD  | 72738000000 |
|                               us-gaap:NetIncomeLoss | from 2022-07-01 to 2023-06-30 | 0000789019 |  USD  | 72361000000 |
|          us-gaap:OtherComprehensiveIncomeLossCas... | from 2020-07-01 to 2021-06-30 | 0000789019 |  USD  |    19000000 |
|          us-gaap:OtherComprehensiveIncomeLossCas... | from 2021-07-01 to 2022-06-30 | 0000789019 |  USD  |     6000000 |
|          us-gaap:OtherComprehensiveIncomeLossCas... | from 2022-07-01 to 2023-06-30 | 0000789019 |  USD  |   -14000000 |
|             us-gaap:OtherComprehensiveIncomeLos... | from 2020-07-01 to 2021-06-30 | 0000789019 |  USD  | -2266000000 |
|             us-gaap:OtherComprehensiveIncomeLos... | from 2021-07-01 to 2022-06-30 | 0000789019 |  USD  | -5360000000 |
|             us-gaap:OtherComprehensiveIncomeLos... | from 2022-07-01 to 2023-06-30 | 0000789019 |  USD  | -1444000000 |
| us-gaap:OtherComprehensiveIncomeLossForeignCurren... | from 2020-07-01 to 2021-06-30 | 0000789019 |  USD  |   873000000 |
| us-gaap:OtherComprehensiveIncomeLossForeignCurren... | from 2021-07-01 to 2022-06-30 | 0000789019 |  USD  | -1146000000 |
| us-gaap:OtherComprehensiveIncomeLossForeignCurren... | from 2022-07-01 to 2023-06-30 | 0000789019 |  USD  |  -207000000 |
|                       us-gaap:OtherComprehensive... | from 2020-07-01 to 2021-06-30 | 0000789019 |  USD  | -1374000000 |
|                       us-gaap:OtherComprehensive... | from 2021-07-01 to 2022-06-30 | 0000789019 |  USD  | -6500000000 |
|                       us-gaap:OtherComprehensive... | from 2022-07-01 to 2023-06-30 | 0000789019 |  USD  | -1665000000 |
|               us-gaap:ComprehensiveIncomeNetOfTax | from 2020-07-01 to 2021-06-30 | 0000789019 |  USD  | 59897000000 |
|               us-gaap:ComprehensiveIncomeNetOfTax | from 2021-07-01 to 2022-06-30 | 0000789019 |  USD  | 66238000000 |
|               us-gaap:ComprehensiveIncomeNetOfTax | from 2022-07-01 to 2023-06-30 | 0000789019 |  USD  | 70696000000 |
+-----------------------------------------------------+------------------------------+------------+-------+-------------+

```

Figure 7.10: Microsoft's 10K facts in Brel

The figure 7.10 clearly shows that Brel is able to extract the facts from the comprehensive income statement of Microsoft's 10K report correctly. All facts present in figure 7.8 are also present in figure 7.10 and vice versa. Each individual fact is also correct, as it is associated with the correct concept, period, entity, unit, and value. Brel represents these characteristics differently from Arelle. For example, Arelle visualizes concepts using their labels, while Brel visualizes concepts using their QNames. However, the values themselves represent the same information.

The only difference is between the representation of the facts, as Arelle visualizes the facts as a table, where the rows are the concepts and the columns are the periods. Brel visualizes the facts as a table representing a hypercube.
We can conclude that Brel is able to extract the structures from the comprehensive income statement of Microsoft's 10K report correctly. This is a good result, as it shows that Brel is able to parse XBRL reports correctly. However, it is important to note that this evaluation only covers a single component within the report. It does not cover the full range of Brel's functionality or its correctness. This example serves as a vertical slice of Brel's correctness. Together with the conformance suite, it gives us a good idea of Brel's correctness.

## 7.3 Performance

TODO: Write this chapter

## 7.4 Robustness

The effectiveness of Brel in handling real-world XBRL reports is crucial to its utility. As such, we intend to assess Brel's robustness through the loading of 10K and 10Q reports [4] from the top 50 US companies by market capitalization as of the date of this document [5]. The companies selected for this evaluation are detailed in table 7.1. This selection was made by shortening the list of the top 100 US companies by market capitalization[5]. The focus on US companies is due to the SEC's requirement for all domestic companies to submit their financial statements in XBRL format[32].

| | | |
|---|---|---|
| Microsoft | Apple | Alphabet (Google) |
| Amazon | NVIDIA | Meta Platforms (Facebook) |
| Berkshire Hathaway | Eli Lilly | Tesla |
| Broadcom | Visa | JPMorgan Chase |
| UnitedHealth | Walmart | Exxon Mobil |
| Mastercard | Johnson & Johnson | Procter & Gamble |
| Home Depot | Oracle | Merck |
| Costco | AbbVie | AMD |
| Chevron | Adobe | Salesforce |
| Bank of America | Coca-Cola | Netflix |
| PepsiCo | Thermo Fisher Scientific | McDonald's |
| Cisco | Abbott Laboratories | T-Mobile US |
| Danaher | Intel | Intuit |
| Comcast | Disney | Wells Fargo |
| Amgen | IBM | Caterpillar |
| ServiceNow | Qualcomm | Nike |
| Union Pacific | | |

Table 7.1: The 50 largest US companies by market capitalization at the time of writing

Table 7.1 outlines the companies selected for assessing the robustness of Brel. The table arranges the largest three companies in the top row from left to right, with the subsequent three companies in the second row, continuing in this pattern. For

[4] 10K reports are annual filings, whereas 10Q reports are quarterly filings.
[5] Date of document: 29 January 2024

each company, the most recent ten 10K and 10Q reports accessible on the EDGAR database[30] will be utilized. Should a report be unavailable, it will be omitted, and the next available report will be considered. The reports will be processed through the Brel API to determine one of three possible outcomes:

1. Successful report loading without errors.
2. Successful report loading accompanied by a logged warning or error.
3. An exception is thrown by Brel during the report loading process.

Given Brel's eager loading approach, the focus will be solely on the loading of reports without conducting further analyses. This approach provides insight into Brel's practical robustness. Nevertheless, this evaluation does not encompass the entirety of Brel's capabilities or accuracy, aspects which have been previously addressed in section 7.2. The findings from the robustness evaluation are depicted in figure 7.11.

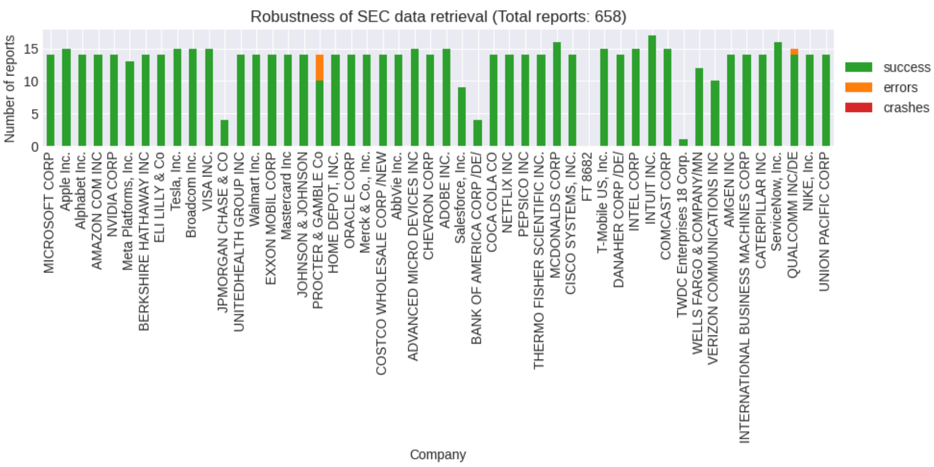



Figure 7.11: Robustness evaluation of Brel

Figure 7.11 presents the robustness assessment of Brel, indicating the number of reports loaded successfully, those loaded with warnings or errors, and instances where Brel encountered a failure. Remarkably, Brel managed to process all 10K and 10Q reports without any failures. Nonetheless, it encountered warnings or errors in 0.76% of these reports, specifically from companies like Procter & Gamble and Qualcomm. These issues arose from reports where dimensions referenced concepts rather than explicit members. While the XBRL specification technically permits dimensions to reference concepts, logically, a correct approach for modeling dimensions that reference concepts is to create a copy of the concept as a member. Hence, Brel issues warnings for such reports.

The variation in the number of reports evaluated per company is attributed to limitations within the SEC's report retrieval API, which lacks the functionality to filter reports by format. Consequently, the API returns reports in either XML or HTML format, but Brel is only equipped to process XML-formatted reports.

Despite logging warnings or errors for a small fraction of the reports, Brel's ability to load all reports demonstrates its robustness in processing real-world XBRL reports.

# Chapter 8

# Conclusion

This thesis introduces a Python API for processing XBRL reports, designed to abstract the complexities of the XBRL format from the user. We have successfully demonstrated that this API covers XBRL and the Open Information Model (OIM), and facilitates straightforward extraction and analysis of data from XBRL reports. Moreover, we have shown that this API effectively conceals the XML structure underlying XBRL reports from the end user.

This thesis aimed to answer three research questions, all of which have been answered in the previous chapters. The first question was whether it is possible to create a Python API that encompasses the OIM of the XBRL format. We have shown that it is indeed possible to create such an API in Section 5.4. The second question was whether the non-OIM sections of the XBRL format can be managed in a similar manner. We have shown that this is indeed possible in Section 5.7. The third question was how Brel can be implemented in a way that allows for XBRL reports in both CSV and JSON format to be used as input. We have shown that this is possible in Section 6.5.

Brel contributes the XBRL domain by offering an open-source Python API specifically for XBRL reports. While other similar APIs are available, none prominently feature the OIM. Arelle provides a Python API, but as a standalone application, it differs from Brel. Brel, being a Python package, integrates seamlessly into other Python applications, distinguishing its role in the field. The API has been developed into a Python package named `Brel`, which is accessible on the Python Package Index (PyPI) [1].

Reflecting on our results, we conclude that the API is robust across various XBRL reports as indicated in Section 7.4, and adheres to the XBRL standard, inclusive of the OIM, as demonstrated in chapters 4 and 5. A notable limitation of Brel is its inability to semantically interpret data in XBRL reports, meaning it cannot autonomously identify logical inconsistencies in the reports. While XBRL is primarily designed for machine-readability to facilitate automated processes, Brel does not account for the semantic layer of XBRL, limiting the potential of automated processes to rely on it. As a result, these processes must independently verify the logical consistency of the reports.

Currently, Brel operates predominantly as a syntactic interface for the XBRL format, without leveraging its semantic aspects. Despite this, it simplifies interaction with XBRL reports in Python by masking the technical intricacies of XBRL. Looking forward, Brel could integrate a semantic layer over its existing syntactic framework, enabling more sophisticated analyses of XBRL reports.

[1] Brel can be installed using the command `pip install brel-xbrl`. This command is compatible with all major operating systems, including Windows, macOS, and Linux.

## 8.1 Future Work

Future developments for Brel can be categorized into two areas: enhancing its current features and expanding its functionalities. The improvements and extensions discussed here stem from Brel's limitations, as addressed in section 2.3.

### 8.1.1 Support for Additional XBRL Formats

Presently, Brel is compatible only with the XML format outlined in the XBRL 2.1 specification [12]. With the introduction of the OIM, XBRL has expanded to include formats like CSV and JSON. Currently, these formats are not supported by Brel, and their specifications are still under development [2]. For instance, at this time, the CSV and JSON formats do not accommodate networks, an essential component of XBRL.
In addition to CSV and JSON, XBRL also endorses the Inline XBRL (iXBRL) format, merging XBRL with HTML. Given the SEC's use of iXBRL for financial reporting, its support is critical. Considering the structural similarities between HTML and XML, it is feasible to incorporate iXBRL support into Brel by enhancing its existing architecture.

### 8.1.2 Semantic Layer Integration

As discussed in chapter 2.3, Brel does not utilize XBRL's semantic layer. Currently, Brel processes XBRL reports without interpreting the data, such as not identifying logical inconsistencies like negative asset values. Future versions of Brel could detect and report these errors, potentially suggesting corrections.
As noted in section 4.8, Brel has the capability to support open hypercubes, which are defined by having dimensions with members that are yet to be identified. It is important to note that XBRL hypercubes encompass semantic information[3]. By leveraging this semantic data, Brel can significantly enhance the analysis process of XBRL reports.

### 8.1.3 Performance Enhancement

Although not a primary focus of this thesis, Brel's performance remains a vital metric. Future iterations can benchmark their performance against this initial version. Python, being a high-level language, may not offer optimal speed. Part of Brel's parser could be optimized within Python or rewritten in a more efficient language like C or Rust.

### 8.1.4 Enhancing Usability

To make Brel more user-friendly, developing a graphical user interface (GUI) is advisable. Currently functioning as an API, Brel requires users to write Python code. A GUI would eliminate the need for coding, thus broadening Brel's accessibility.

### 8.1.5 Conducting a Usability Study

As of now, Brel has been publicly available for a few weeks with limited user engagement. A comprehensive usability study would provide a scientific assessment of its user-friendliness. Such a study would also identify Brel's strengths and weaknesses, enabling more focused improvements.

[2] The OIM specifications contains a section for footnotes, which are a type of network.
[3] See `https://www.xbrl.org/WGN/dimensions-use/WGN-2015-03-25/dimensions-use-WGN-2015-03-25.html`

## 8.2 Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Ghislain Fourny, for his support and guidance throughout the development of Brel and the writing of this thesis. Without his expertise and encouragement, this project would not have been possible.

I am also grateful to Prof. Dr. Gustavo Alonso for supervising this thesis and letting me work on this project as part of the Systems Group at ETH Zurich.

Both the development of Brel and the writing of this thesis have been aided by a few indispensable tools. Handling XML and HTTP requests in Python would have been much more difficult without the help of the `lxml` and `requests` libraries. They have been invaluable in the development of Brel, and I am grateful to their developers for their hard work.

I would also like to thank OpenAI for developing ChatGPT[4] , and GitHub for publishing Copilot[5]. I am not a writer, I am a programmer. My writing skills do sometimes leave something to be desired, and ChatGPT has been a great help in improving the quality of my writing and expressing my thoughts in a more formal manner. ChatGPT does generate text with many logical errors, but when used correctly, it can provide an alternative perspective on a text that can be very helpful. Even though this thesis was written with the help of ChatGPT, none of the information in this thesis was generated by ChatGPT. ChatGPT was merely used for two purposes: to improve the quality of my writing, and to provide an alternative perspective on the text.

GitHub Copilot has also been a great help for implementing Brel, as it has helped me generate examples for libraries that I have never used before, and has helped me find the right functions to use in many cases. Similar to ChatGPT, GitHub Copilot does not necessarily generate correct code, but it can speed up the mundane parts of programming significantly. Copilot was used for two purposes: to generate examples for libraries that I have never used before, to speed up repetitive tasks in programming like creating getters and setters, and to give alternatives for how to name certain methods and classes.

Apart from these tools, I would like to thank my friends for their support during my studies at ETH Zurich, especially Pascal Strebel, who has been a great friend all throughout my ETH journey. I hope that we will continue to be friends for many years to come and that we will continue to support each other, no matter where life takes us.

I would like to thank my family for their unwavering support and encouragement. I know that I did not always have time for them during my studies, but I hope that they know that I love them and that I am grateful for everything that they have done for me.

And last but not least, I would like to thank you, the reader, for taking the time to read this thesis. I hope that you have found it interesting and that you have learned something new from it.

[4] `https://chat.openai.com/`

[5] https://github.com/features/copilot

# Bibliography


[1] European Banking Authority. Eba xbrl filing rules. https://extranet.eba.europa.eu/sites/default/documents/files/documents/10180/2185906/63580b57-b195-4187-b041-5d0f3af4e342/EBA%20Filing%20Rules%20v4.3.pdf?retry=1, 2018.

[2] European Banking Authority. Reporting frameworks. https://www.eba.europa.eu/risk-and-data-analysis/reporting-frameworks, unknown.

[3] Tim Berners-Lee. Using relative uri's. https://www.w3.org/DesignIssues/Relative.html, 2011.

[4] Richard Smith Bruno Tesnière and Mike Willis. the journal. https://www.pwc.com/gx/en/banking-capital-markets/pdf/120202thejournal.pdf, 2002.

[5] companiesmarketcap.com. Largest us companies by market cap as on january 29, 2024. https://companiesmarketcap.com/usa/largest-companies-in-the-usa-by-market-cap/, 2024.

[6] Microsoft Corporation. Annual report 2022. https://www.microsoft.com/investor/reports/ar22/index.html, 2022.

[7] Dr. Ghislain Fourny. *The XBRL Book: Simple, Precise, Technical*. Independently published, 2023.

[8] Charles Hoffman. The seattle method. https://xbrlsite.com/seattlemethod/.

[9] Charles Hoffman. Extensible business reporting language (xbrl). https://www.xbrl.org/, 2000.

[10] Charles Hoffman. Xbrl-based digital financial reporting conformance suite tests. http://xbrlsite.azurewebsites.net/2019/Prototype/conformance-suite/Production/index.xml, 2020.

[11] Apple Inc. 10-q (quarterly report) for quarter ending june 24, 2023. https://www.sec.gov/Archives/edgar/data/320193/000032019323000077/0000320193-23-000077-index.htm, 2023.

[12] XBRL International Inc. Extensible business reporting language (xbrl) 2.1. https://www.xbrl.org/Specification/XBRL-2.1/REC-2003-12-31/XBRL-2.1-REC-2003-12-31+corrected-errata-2013-02-20.html, 2003.

[13] XBRL International Inc. Extensible business reporting language (xbrl) dimensions 1.0. https://www.xbrl.org/specification/dimensions/rec-2012-01-25/dimensions-rec-2006-09-18+corrected-errata-2012-01-25-clean.html, 2006.

[14] XBRL International Inc. Extensible business reporting language (xbrl) generic links 1.0. https://www.xbrl.org/specification/gnl/rec-2009-06-22/gnl-rec-2009-06-22.html, 2009.

[15] XBRL International Inc. Extensible business reporting language (xbrl) 2.1 - 5.1.1 concept definitions. http://www.xbrl.org/specification/xbrl-2.1/rec-2003-12-31/xbrl-2.1-rec-2003-12-31+corrected-errata-2013-02-20.html#_5.1.1, 2013.

[16] XBRL International Inc. Extensible business reporting language (xbrl) 2.1 - 5.2.2 the labellink element. http://www.xbrl.org/specification/xbrl-2.1/rec-2003-12-31/xbrl-2.1-rec-2003-12-31+corrected-errata-2013-02-20.html#_5.2.2, 2013.

[17] XBRL International Inc. Extensible business reporting language (xbrl) 2.1 terminology - 1.4 terminology (non-normative except where otherwise noted). http://www.xbrl.org/specification/xbrl-2.1/rec-2003-12-31/xbrl-2.1-rec-2003-12-31+corrected-errata-2013-02-20.html#_1.4, 2013.

[18] XBRL International Inc. About xbrl us. https://xbrl.us/home/about/, 2021.

[19] XBRL International Inc. Open information model 1.0. https://www.xbrl.org/Specification/oim/REC-2021-10-13+errata-2023-04-19/oim-REC-2021-10-13+corrected-errata-2023-04-19.html#term-component, 2021.

[20] XBRL International Inc. xbrl-csv: Csv representation of xbrl data 1.0. https://www.xbrl.org/Specification/xbrl-csv/REC-2021-10-13+errata-2023-04-19/xbrl-csv-REC-2021-10-13+corrected-errata-2023-04-19.html, 2021.

[21] XBRL International Inc. xbrl-json: Json representation of xbrl data 1.0. https://www.xbrl.org/Specification/xbrl-json/REC-2021-10-13+errata-2023-04-19/xbrl-json-REC-2021-10-13+corrected-errata-2023-04-19.html, 2021.

[22] AICPA Karen Kernan. Xbrl - the story of our new language. https://us.aicpa.org/content/dam/aicpa/interestareas/frc/accountingfinancialreporting/xbrl/downloadabledocuments/xbrl-09-web-final.pdf, 2009.

[23] Mark V Systems Limited. Arelle - open source xbrl platform. https://arelle.org/arelle/,, 2010.

[24] Stack Overflow. Stack overflow developer survey 2020. https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents, 2020.

[25] Ghislain Fourny Robin Schmidiger. Brel api documentation. https://papedipoo.github.io/Brel/, 2024.

[26] Robin Schmidiger. Brel - business reporting extensible library. https://github.com/BrelLibrary/brel, 2024.

[27] European Securities and Markets Authority (ESMA). filings.xbrl.org. https://filings.xbrl.org/.

[28] European Securities and Markets Authority (ESMA). Esma publishes esef conformance suite. https://www.esma.europa.eu/press-news/esma-news/esma-publishes-esef-conformance-suite, 2020.

[29] U.S. Securities and Exchange Commission (SEC). Interactive data test suite. https://www.sec.gov/structureddata/osdinteractivedatatestsuite.

[30] U.S. Securities and Exchange Commission (SEC). Edgar. https://www.sec.gov/edgar/searchedgar/accessing-edgar-data.htm, 1996.

[31] U.S. Securities and Exchange Commission (SEC). 17 cfr § 229.402 - (item 402) executive compensation. https://www.govinfo.gov/app/details/CFR-2011-title17-vol2/CFR-2011-title17-vol2-sec229-402, 2011.

[32] U.S. Securities and Exchange Commission (SEC). Inline xbrl. https://www.sec.gov/structureddata/osd-inline-xbrl.html, 2018.

[33] U.S. Securities and Exchange Commission (SEC). 17 cfr § 229.402 - (item 402) executive compensation. https://www.ecfr.gov/current/title-17/part-229#p-229.402(v)(2)(iv), 2023.

[34] XBRL US. Xbrl us xule. https://xbrl.us/xule/, 2021.

[35] World Wide Web Consortium (W3C). Xml, xlink and xpoiner. https://www.w3schools.com/xml/xml_xlink.asp, 1999.

[36] World Wide Web Consortium (W3C). Qnames as identifiers. https://www.w3.org/2001/tag/doc/qnameids-2004-01-14.html, 2004.