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 my 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[24] The contents of this thesis are based heavily on both the XBRL standard[9] created by Charles Hoffman and "The XBRL Book"[8] 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[20]. 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.[16] 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) and the European Banking Authority (EBA)[2] are progressively mandating the submission of reports in XBRL format.[30] 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. As a framework, XBRL embodies considerable intricacy, with many complexities stemming from historical design choices that contribute to its challenging usability. 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. Nevertheless, the present status of XBRL does not fully align with this requirement. The XBRL landscape is characterized by a diverse array of tools, with a majority being proprietary and not openly accessible. While there are open-source alternatives available, they are limited in number. Apart from Arelle, a notable platform in the XBRL domain[21], other open-source are often limited in scope.

In 2021, XBRL International published a new specification termed Open Information Model (OIM)[17]. 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, notably the OIM. The library is called `Brel` (short for **B**usiness **R**eporting **E**xtensible **L**ibrary) 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:

- How can the OIM be translated into an easy-to-use python API?
- How can the non-OIM sections of XBRL be converted into an easy-to-use python API that is consistent with the OIM?
- 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, the implementation of 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.
Whereas the first two limitations are self-explanatory, the third limitation requires some explanation. XBRL reports can be interpreted in terms of syntax and semantics, similar to the source code of a program. A syntactically correct XBRL report conforms to the XBRL specification, but it does not necessarily make sense [1]. A semantically correct report is both syntactically correct and is also logically consistent. This thesis will not cover the semantics of XBRL reports, which are not strictly defined by the XBRL specification, but rather through supporting documents.
However, by using the Brel API, one can semantically validate XBRL reports. The functionality for semantic validation is not part of the Brel library, but can be implemented on top of the Brel API. The reader can think of Brel as a syntactic wrapper around XBRL reports.

2.4 Structure of this thesis

To properly understand how Brel implements a pythonic API for XBRL reports, it is important to first understand the underlying XBRL standard. Therefore, chapter 4 will give a brief introduction to XBRL. It will introduce the core concepts of XBRL in the OIM, followed by the non-OIM sections of XBRL. The chapter will not delve into the technical details of the XBRL standard, but rather focus on the concepts relevant for this thesis.
Next, chapter 5 will introduce the API of Brel. This chapter, while constituting a portion of the thesis outcomes, is positioned prior to the implementation of the API. The reason for this deviation from the conventional structure is that it is

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

more logical to introduce the API before its implementation. The API chapter will answer research question 2.2 and 2.2.
Chapter 6 outlines the Brel library's implementation, connecting the API from Chapter 5 with the XBRL standards in Chapter 4. While aiming for a comprehensive overview, the chapter will particularly emphasize the more involved aspects of the implementation. It will also discuss the library's design and its potential to support various formats in future versions, addressing Research Question 2.2.
Chapter 7, the Results chapter, assesses the library's performance against the objectives established in the introduction. This chapter provides a detailed analysis of Brel's compatibility with various XBRL conformance suites. It also illustrates practical applications of the library, demonstrating how Brel can be utilized for reading and validating XBRL reports.
Chapter 8, the final chapter, will recapitulate the key findings of this thesis. It will also provide perspectives on potential directions for future research and development regarding Brel.

Chapter 3

Related work

The goal of this thesis is to create a new open source XBRL library that is largely based on the OIM. When it comes to work related to Brel, there are three main areas of interest. First, there is the XBRL specification itself and interpretations of it. Second, there are other XBRL libraries and platforms that are similar to Brel. This also includes public databases of XBRL reports. Third, there are the requirements set by authorities and other organizations that XBRL processors have to fulfill.

3.1 XBRL Specification

As mentioned in section 2, this thesis is based on the XBRL standard[9] originally created by Charles Hoffman. The XBRL standard is a complex standard consisting of many different parts. The parts of the XBRL standard that are relevant to this thesis are the Open Information Model (OIM)[17], the XBRL 2.1 specification[10], the extension for dimensional reporting[11] and the specification for generic links[12], all of which will be covered in chapter 4.
The XBRL 2.1 specification and the OIM overlap in most areas, but the OIM is a partial rewrite of the XBRL 2.1 specification. The OIM does not contain all the features of the XBRL 2.1 specification, but it is a lot easier to understand. The OIM also covers some aspects of XBRL dimensions, but does not contain any features from XBRL generic links.

3.2 The XBRL Book

To properly understand the XBRL specification, one already needs to have a good understanding of both XML and XBRL. This makes it difficult for beginners to get started with XBRL. To address this problem, Dr. Ghislain Fourny wrote "The XBRL Book"[8]. The book is a comprehensive guide for XBRL and covers all the important aspects of the XBRL standard, including the rather recent OIM.

3.3 Arelle

Arelle[21] is an open source XBRL platform. At the time of writing, Arelle is the most fully open source platform of its kind. It supports all the features of the XBRL 2.1 specification and the OIM. Like Brel, Arelle is also written in Python and is open source. Unlike Brel, Arelle is a complete XBRL platform and not just a python library. Think of Arelle as "Excel for XBRL".

3.4 Xule

Xule[32] is a rule language for XBRL. It is a declarative language that allows users to write rules for validating XBRL reports. It is part of the Arelle project and is used by Arelle to validate reports. Think of Xule as a domain-specific language for XBRL validation rules. Even though Xule is not part of this thesis, Brel should be able to support Xule in the future.

3.5 EDGAR

The SEC (US Securities and Exchange Commission) maintains a system called EDGAR [28]. [1] EDGAR is a database of financial reports submitted to the SEC. EDGAR stands for **E**lectronic **D**ata **G**athering, **A**nalysis, and **R**etrieval. Filings submitted to EDGAR have to be in XBRL format. The SEC provides public access to the filings submitted to EDGAR. [2]

3.6 ESEF filings

The european counterpart to the SEC is called ESMA (**E**uropean **S**ecurities and **M**arkets **A**uthority). It also maintains a database of ESEF filings[25]. ESEF stands for **E**uropean **S**ingle **E**lectronic **F**ormat, which is a standard for XBRL reports in the EU. Similar to EDGAR, the ESEF database is publicly accessible. [3] An interesting aspect of this database is that it is hosted by XBRL International, the organization that maintains the XBRL standard.

3.7 SEC - Interactive Data Public Test Suite

To ensure that the XBRL processors that the companies use to create their XBRL reports are compliant with the XBRL standard, the SEC created the Interactive Data Public Test Suite[27]. This test suite contains an extensive collection of XBRL reports that are used to test XBRL processors. The SEC provides this test suite free of charge and it is available on their website.
Brel uses this test suite to test its XBRL processor. At the time of writing, Brel does not pass all the tests in the test suite, since Brel does not support all the features of the XBRL standard. However, building a processor that passes all the tests is not a feat that can feasibly be accomplished in the scope of this thesis.

3.8 ESMA Conformance Suite

The European Securities and Markets Authority (ESMA) also maintains a test suite for XBRL processors[26]. This test suite is similar to the SEC's Interactive Data Public Test Suite, but it is maintained by a different authority. Another key difference is that the ESMA Conformance Suite facilitates automated testing of xHTML reports.
Brel does not support xHTML reports, so the ESMA Conformance Suite is not relevant for this thesis. However, Brel should be able to support 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

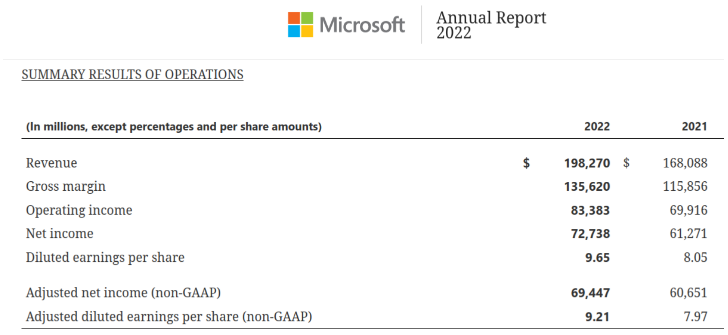
The content of this thesis is largely based on the XBRL standard[9] created by Charles Hoffman and Dr. Ghislain Fourny's interpretation of it in *the XBRL Book*[8]. Since the thesis builds on the foundation laid by the two, it is important to understand the ground work that they have done. This chapter will give a brief introduction to XBRL. Think of it as a crash course in XBRL.
In essence, XBRL is a standardized format for representing reports. After all, XBRL stands for **eXtensible Business Reporting Language**.[9]
As Ghislain Fourny has put it in *the XBRL Book* [8]:

> 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. Afterwards, I will introduce the more advanced concepts that are about putting facts into relation with each other, namely roles, networks, and report elements. The first half roughly corresponds to the OIM, while the second half covers the non-OIM parts of XBRL.
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 simple 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.
Furthermore, a lot of concepts in XBRL require knowledge of other XBRL concepts. There are also a few circular dependencies between the concepts, which makes it hard to explain them in a linear fashion. Therefore, there are a few sections in this chapter that will redefine concepts that have already been introduced. This is done to gradually introduce the reader to the more complex concepts of XBRL.

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.
Lets consider a simplified example involving a financial report of Microsoft Corporation for the fiscal year 2022. Microsoft's annual report can be found on the company's website[1]. It contains a lot of information about the company's financial situation, as well as information about the company's business activities. For this example, we will only consider the company's revenue for the fiscal year 2022. Microsoft chose to report this information as follows:



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

[7]

This table contains multiple facts about Microsoft for both fiscal years 2021 and 2022, as seen by the horizontal axis. The vertical axis describes what is being reported. The "what is being reported" part is called the **concept** of a fact. It reports the values for the concepts "Revenue", "Gross margin", "Operating income", . . . for both fiscal years. In summary, the table contains 14 facts across 7 concepts for 2 fiscal years.
For now, let us focus on the top left fact, which reports the company's revenue for the fiscal year 2022. In XBRL, a corresponding fact would be represented as follows:

- **Concept:** Revenue
- **Entity:** Microsoft Corporation
- **Period:** from 2022-04-01 to 2023-03-31 [2]
- **Unit:** USD
- **Value:** 198'270'000 [3]

In this example:

- The **concept** refers *what* is being reported. In this case, "Revenue" indicates that the fact is reporting information about the company's revenue.

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

- The **entity** refers to *who* is reporting. In the case of our example, the entity is "Microsoft Corporation". In our example this is implicit, since we are looking at Microsoft's annual report. However, the entity of a fact has to be explicitly stated in an XBRL report.
- The **period** refers to *when* the information is being reported. The period is defined as the fiscal year 2022. This is indicated by the column header "2022" in the table.
- The **unit** refers to *how* the information is being reported. In this example, the unit is "USD", which indicates that the information is being reported in US dollars. The unit is indicated by the dollar sign $ in the table.
- The **value** refers to *how much* is being reported. According to Microsoft's 2022 annual report, the company's revenue for the fiscal year 2022 was around 198 billion US dollars.

The concept, entity, period and unit of a fact are called its **aspects**. If necessary, additional aspects can be defined for a fact. These additional aspects are called **dimensions** and will be covered in section 4.8 The aspects that are not dimensions are called **core aspects**. Even though the name suggests otherwise, core aspects are not all mandatory for a fact. 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. One of the core aspects of a fact is its concept, which refers to what is being reported.
So for example, if a fact is reporting information about a company's revenue, then the concept of the fact is "Revenue". In this section, we will take a closer look at concepts and how they are defined in XBRL.
Concepts are the fundamental building blocks of XBRL and they are defined in what is called the **taxonomy**.

4.3.1 Taxonomy

Simply put, the XBRL taxonomy is a collection of concepts and their relationships. Each XBRL report defines its own taxonomy inside of a taxonomy schema file. The taxonomy defined by the report is called the **extension taxonomy**.
This extension taxonomy contains references to other taxonomies, which may contain references to even more taxonomies. So when a report and its extension taxonomy are loaded into memory, the entire taxonomy is loaded into memory. The transitive closure of all these references is called the **DTS** (short for **D**iscoverable **T**axonomy **S**et).
Note that most of the taxonomies in the DTS are not located on the same machine as the report. Instead, they are located on the internet and are downloaded on demand.
Some taxonomies commonly found in a report's DTS are:

- **us-gaap** [4] - Contains concepts for US Generally Accepted Accounting Principles (GAAP).
- **ifrs** [5] - Contains concepts for International Financial Reporting Standards (IFRS).

[4] https://xbrl.us/us-gaap/
[5] https://www.ifrs.org/

- **dei** [6] - Contains concepts for the SEC's Document and Entity Information (DEI) requirements.
- **country** [7] - Contains concepts for country codes.
- **iso4217** [8] - Contains concepts for currency codes.

Since a lot of DTSs from different reports share a lot of the same taxonomies, it makes sense to cache the taxonomies locally, instead of downloading them every time they are needed.

4.3.2 Concepts

Each concept in the DTS is identified by a **QName**. The technical intricacies of QNames will be covered in section 4.4, but for now think of them as a unique identifier for a concept. The QName of a concept tends to be human-readable and self-explanatory. However, accountants and business analysts tend to go overboard with with their naming conventions. Some examples of the QNames of concepts are:

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

But concepts do not only consist of a QName. They also constrain some of the aspects and values of the facts that reference them. Going back to our running example from section 4.2, the concept `us-gaap:Revenue` introduces some constraints. For example, it constrains the value to be a `monetaryItemType`. This means that the value of the fact must be a number, not just any arbitrary string.[9] It also constrains the unit to be a currency defined in the ISO 4217 standard.[1] Moreover, monetary facts have to be labeled as either "debit" or "credit" using the `balance` attribute. In this case, the concept `us-gaap:Revenue` constrains the fact to have a "debit" balance, since the company's revenue is an asset.
The concept `us-gaap:Revenue` also constrains the period of the fact to be of type "duration". This means that the period of the fact has to be a duration of time, such as a fiscal year or a quarter. Alternatively, the period could be of type "instant", which means that the period refers to a specific point in time.
Finally, the concept `us-gaap:Revenue` allows the fact to be null, which means that it is optional for a report to contain a fact for the concept `us-gaap:Revenue`. The vast majority of concepts allow facts to be null.
This wraps up our discussion about concepts in XBRL. In the next section, we will take a look at QNames and how they are used in XBRL. Together with the knowledge about concepts and facts, this will give us a solid foundation to understand the core parts of the XBRL standard.

[6] https://www.sec.gov/info/edgar/dei-2019xbrl-taxonomy
[7] https://xbrl.fasb.org/us-gaap/2021/elts/us-gaap-country-2021-01-31.xsd
[8] https://www.iso.org/iso-4217-currency-codes.html
[9] The `monetaryItemType` has additional constraints besides being an integer, but we will ignore them for now.

4.4 QNames

Although the motivation behind the XBRL processor Brel is to shield its user from the complexity of XML, we keep one key aspect of XML in our API: QNames.
QNames are a way to uniquely identify an XML element or attribute. They consist of three things: a namespace prefix, a namespace URI, and a local name. The prefix acts as a shorthand for the namespace URI.
For example the QName `us-gaap:Assets` identifies the element `Assets` in the namespace `us-gaap`.
In this example, the namespace prefix `us-gaap` is a shorthand for the namespace URI `https://xbrl.fasb.org/us-gaap/2022/elts/us-gaap-2022.xsd`, and together they form the namespace `us-gaap`.

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`

QNames are used in XBRL to identify concepts, facts and other elements. Since they provide a robust and easy way for identifying elements, we decided to use them in our API as well. However, there is one important difference between our QNames and the QNames used in the XBRL taxonomy: Currently, most XBRL filings are based on XML, where namespace prefixes are defined on a per-element basis. Therefore, the mapping from namespace prefixes to namespace URIs depends on where the QName is used.
In our API, there is a fixed, global mapping from namespace prefixes to namespace URIs. The motivation behind this decision is that it makes the API easier to use. More details about this mapping will be explained in section 6.3.

4.5 Segway to advanced XBRL

This concludes our overview of QNames as well as the overview of the core concepts of XBRL. Armed with this knowledge, we could already create functional XBRL reports, albeit with a lot of limitations.
The most glaring limitation is that the facts in the report are not structured in any way. Reports are just an unstructured set of facts. Since facts are not related to each other, it is impossible to verify if the values within the report are consistent.
Besides that, XBRL in its current form is not very user-friendly. The concepts for example are named using QNames, which tend to be human-readable but extremely verbose. Another limitation is that QNames are mostly in English, which makes it difficult for non-English speakers to understand the report.
All of these issues will be addressed in the next sections, which will introduce the advanced concepts of XBRL, the most important of which are networks.

4.6 Networks

This section marks the point where the OIM ends and the non-OIM parts of XBRL begin. Since XBRL is heavily based on XML, the remainder of this chapter will use XML syntax more heavily.
Up to now, all concepts in the XBRL taxonomy are treated as independent entities. The whole filing can be viewed as an unordered set of facts with no relation between them. However, in reality, concepts are not independent of each other. Instead, they are related to each other in some way.
For example, the concepts `Assets` and `Liabilities` are related to each other in the sense that they are both part of the concept `Balance Sheet`. Furthermore, the concept `Assets` can be further divided into the concepts `Current Assets` and `Non-Current Assets`. The aforementioned relations can be visualized using a directed graph, as shown 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 basic understanding of mathematics will recognize that the above example is a directed acyclic graph (DAG). More specifically, the above example is a tree. Graphs are commonly used to represent relations between entities. In the context of XBRL, graphs are used to represent all kinds of relations between concepts, facts, and other elements of an XBRL filing. From now on, I will refer to graphs that represent relations between XBRL elements as `networks`.
XBRL commonly uses the term `Extended Link` or `Link` to refer to networks or parts of networks. I will use the term `network` throughout this thesis, since it is more intuitive and less ambiguous. When I refer to a `link`, I am referring to something specific in the XBRL specification. When I refer to a `network`, I am referring to the general concept of a network.
XBRL groups multiple links together into so called `linkbases`. From a semantic perspective, linkbases do not have any meaning. From a technical perspective, linkbases are just XML elements that have children that are links.

4.6.1 Types of Networks

The XBRL 2.1 specification defines 6 built in types of networks[15][10]:

- `link:presentationLink`: A network that represents the hierarchy of concepts in a report. An example of this can be seen in figure 4.3.
- `link:calculationLink`: A network that represents how concepts are calculated from other concepts. For example, in figure 4.3, the concept `Assets` is calculated as the sum of the concepts "Current Assets" and "Non-Current Assets".
- `link:definitionLink`: A network that represents the typing relations between concepts. The meaning of this network will become clearer in section 4.6.5.

[10]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.

- `link:labelLink`: A network that associates report elements with human-readable labels.
- `link:referenceLink`: A network that links report elements to external resources. For example, 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)`[31].
- `link:footnoteLink`: A network that links report elements, facts and other elements to footnotes.

XBRL refers to these built in networks as `standard extended links`. If needed, XBRL allows users to define their own networks, which are referred to as `custom extended links`[15].
Technically speaking, XBRL does allow networks in XBRL to contain both directed and undirected cycles. However, in practice, networks in XBRL are almost always directed acyclic graphs (DAGs).
In the subsequent sections, I will describe how networks are implemented in XBRL on a conceptual level.
Even though labels will be covered in more detail in section 4.6.6, I will already use them throughout this chapter. Labels are used to associate report elements with human-readable labels. For example, the concept `us-gaap:CurrentAssets` might have the label "Current Assets" in the English language. The main motivation behind this editorial decision is that it makes the chapter easier to read.

4.6.2 presentationLink

The `link:presentationLink` network is used to represent the hierarchy of concepts in a report. I will describe presentationLinks in more detail compared to the other networks, since all other network types are implemented in a similar fashion as presentationLinks.
XBRL implements all its networks as a list of directed edges called `arcs`. Each arc has a source and a target. Duplicate arcs are not allowed.
Taking the example from figure 4.3, the presentationLink network would be represented as the following edge list:

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
```

Each arc in the example 4.4 is represented as a `link:presentationArc` element in XBRL. Besides presentationArcs, presentationLinks contain so called "locators" `link:loc` that represent the nodes in the network. In case of a presentation network, the locators are references to the concepts in the XBRL taxonomy. In other networks, locators can be references to other elements, such as facts.
Going back to the example in figure 4.4, the first arc `Balance Sheet -> Assets` would be represented as follows in XML syntax:

```
<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 contains two locators and one arc. The two locators represent the nodes of the arc, referencing the concepts `BalanceSheet` and `Assets` respectively. The arc represents the edge between the two nodes.
Let us look at the XML snippet 4.5 step by step.

- **Type**: The `xlink:type` attribute specifies the types for both the locators and the arc. For the former, the type is `locator`, whereas for the latter, the type is `arc`.
- **Connect nodes and edges**: Both locators contain an `xlink:label` attribute that uniquely identifies the locator. The arc links the two locators together using the `xlink:from` and `xlink:to` attributes.
- **Edge order**: The outgoing edges of a node are ordered using the `order` attribute of the arc.
- **Arcrole**: The `xlink:arcrole` attribute of the arc specifies the kind of relation between the source and the target of the arc. In case of a presentationLink, the `xlink:arcrole` attribute is always set to `parent-child`.

Locators and arcs form the basic building blocks of all networks in XBRL, notably presentationLinks. A presentationLink is just a container for locators and arcs. To build a fully featured presentation network such as in figure 4.3, we need to add more locators and arcs to the presentationLink. To avoid cluttering the chapter with XML snippets, I will only describe the XML syntax for the first arc in the network.

4.6.3 Motivation for Report Elements

Up to this point, I have only described how presentation networks are implemented in XBRL. I also mentioned that presentation networks indroduce a hierarchy of concepts, but this is not entirely true.
I have introduced concepts as the "what"-part of a fact. For example, if the company Foo reports a revenue of 1000 USD in 2019, the concept `Revenue` is the "what"-part of the fact.

However, if we look at the presentation network in figure 4.3, not all of the elements in the network can have a fact associated with them. An example of this is the concept `BalanceSheet`. In XBRL, concepts that can not have a fact associated with them are called `Abstract`.
XBRL combines abstracts and concepts under the umbrella term "report element". Report elements are, as the name suggests, elements that can appear in a report. Some of these report elements are used for facts, namely the concepts. Others are used to represent the structure of the report, namely the abstracts. There are six types of report elements in total[17]. I will introduce them as they come up in the subsequent sections.
With the introduction of report elements, our notion of a presentation network changes slightly. Instead of introducing a hierarchy of concepts, presentation networks introduce a hierarchy of report elements. However, our notion of a fact stays the same. A fact is still requires a concept, not a report element.

4.6.4 calculationLink

The `link:calculationLink` network is used to represent how concepts are calculated from other concepts. More specifically, it is used to represent how a concept is the sum of other concepts. Under the hood, calculationLinks are implemented in the same way as presentationLinks, but there are a few differences:

1. Arcs are now called `link:calculationArc`.
2. Links are `link:calculationLink`.
3. The `xlink:arcrole` attribute of the `link:calculationArc` element is set to `summation-item`.
4. The `link:calculationArc` element has an additional attribute called `weight`.
5. All locators in the link are references to concepts, not just report elements.

Most of these differences are self-explanatory and do not have any semantic implications. However, the last two differences are worth explaining in more detail.
The main motivation behind calculation networks is so that XBRL processors can either calculate the value of a concept or check if the value of a concept is computed correctly and consistently. In the case of our XBRL processor Brel, the main focus is on the latter. Chapter 6.4 of "The XBRL Book" [8] describes the different consistency checks in more detail.
Facts that have a concept within a calculation network are computed as a weighted sum of their children. The `weight` attribute of the `link:calculationArc` element specifies the weight of the child in the sum. Additionally, facts can only be associated with concepts, not just any report element. Therefore, all locators in a calculation network are references to concepts.
In the section on concepts 4.3, I have introduced the `balance` aspect of a concept. It specifies if the concept is a debit or a credit. The XBRL 2.1 specification enforces some constraints on the `balance` aspect of concepts in combination with the `weight` attribute of the `link:calculationArc` element [13]. If one concept has a `balance` of `debit` and another concept has a `balance` of `credit`, then their connecting arc must have a negative `weight`. If both concepts have the same `balance`, then their connecting arc must have a positive `weight`.

Concept 1	**Concept 2**	**Connecting edge weight**
Debit	Credit	≤ 0
Credit	Debit	≤ 0
Debit	Debit	≥ 0
Credit	Credit	≥ 0

Figure 4.6: Balance and weight constraints in calculation networks

A network that is consistent with the balance and weight constraints is called a `balance consistent network`.[8]

Balance consistency is not the only kind of consistency that calculation networks can be checked for. Another kind of consistency is `roll-up consistency` which comes in two flavors: `simple roll-up consistency` and `nested roll-up consistency`. Simple roll-up is roll-up consistency without any nested concepts. So the calculation network can only have a depth of 1.

Nested roll-up consistency is roll-up consistency with nested concepts. So the calculation network can have a depth of more than 1.

`Roll-up consistency` requires a calculation network as well as a presentation network and checks if the structure of the two networks is consistent. For example, if the calculation network contains the arc `Assets -> Savings Accounts`, then the presentation network must also contain the arc `Assets [Abstract] -> Savings Accounts`.

Remember that calculation networks can only contain concepts, not abstracts. So the report element `Assets` in the calculation network and the report element `Assets [Abstract]` in the presentation network are not the same. But how does XBRL know that they are related?

The answer is that the two report elements are related by the presentation network. It contains the arc `Assets [Abstract] -> Assets`.

Let us visualize this example in figure 4.7, which shows the calculation network and the presentation network side by side. The two networks are roll-up consistent with each other. The example is expanded a bit and also contains the concepts `UBS Savings Account`, `Raiffeisen Savings Account` and `Liabilities`. Note that the calculation network contains weights for the arcs, but the presentation network does not.

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

The presentation network in figure 4.7 is roll-up consistent with the calculation network. It contains the two abstracts `Assets [Abstract]` and `Savings Accounts [Abstract]`, which are not present in the calculation network. They take the role of the concepts `Assets` and `Savings Accounts` in the calculation network.
Besides roll-up and balance consistency, there is a third kind of consistency. XBRL does not have a name for it, but I will refer to it as `aggregation consistency`. It checks if in a calculation network, for each concept, the child concepts add up to their parent concept. Going back to our example in figure 4.7, the values of the UBS Savings Account and the Raiffeisen Savings Account should add up to the value of the Savings Accounts concept. Aggregation consistency uses weighted sums to add up the values of the child concepts, where the weight of a child concept is the weight of the connecting arc.
To check for aggregation consistency, we need to know the values of the facts that are associated with the concepts. So aggregation consistency checks a list of facts against a calculation network.
Aggregation consistency is best explained using an example. Consider the following list of facts 4.10 for 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 you can see, there are two facts reported against each concept, one for 2022 and one for 2023.
If there are multiple facts reported against a concept, then we iterate over them and "pin" all aspects except for the concept. Next, we go to the list of all facts and filter out all facts that have the same pinned aspects. Finally, we check for aggregation consistency using the filtered list of facts and the calculation network.
In our example, we would first check all facts for 2022, then all facts for 2023. The list of facts is aggregation consistent for the year 2022, since the values of the UBS Savings Account and the Raiffeisen Savings Account add up to the value of the Savings Accounts concept. Also, the value of the Savings Accounts minus the value of the Liabilities concept adds up to the value of the Assets concept.
However, the list of facts is not aggregation consistent for the year 2023. The reason being that the value of the Savings Accounts minus the value of the Liabilities concept does not result in the value of the Assets concept.

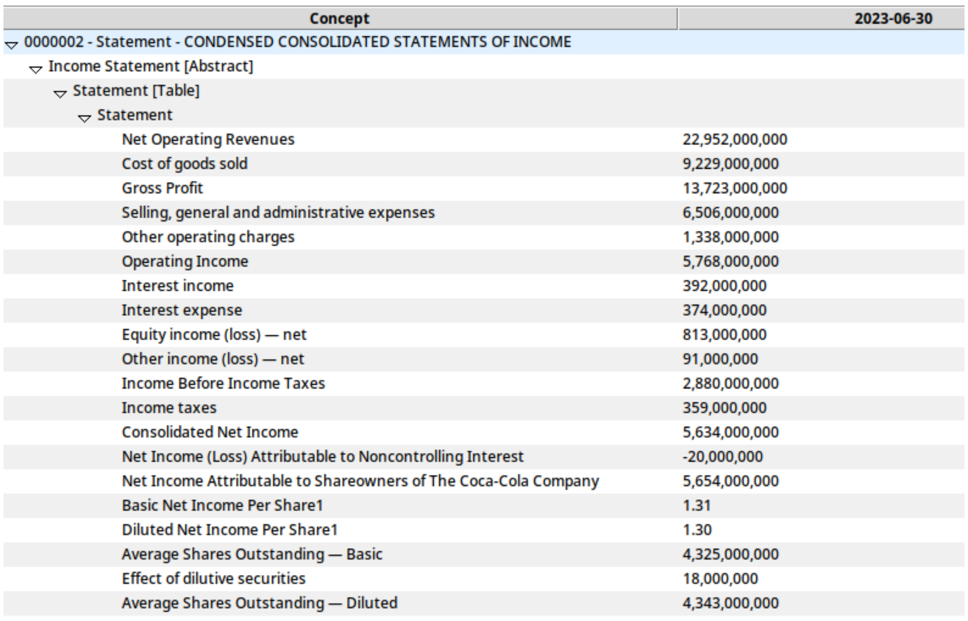
4.6.5 definitionLink

TODO

4.6.6 labelLink

The `link:labelLink` network is used to associate report elements with human-readable labels. Thus far, we referred to report elements using their QName. Obviously, if we were to open an XBRL report in a viewer such as Arelle, we would not be greeted with QNames for the concepts, abstracts, etc. Instead, we would see nicely formatted human-readable labels.

The following figure shows an example of the condensed consolidated statement of income of the Coca Cola Company[6] of Q2 2019 as displayed in Arelle[21].



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

Let us take a closer look at the first concept of the statement of income: Revenue. Even though the concept for the revenue is `us-gaap:Revenues`, Arelle displays it as "Net Operating Revenues".
Arelle achieves this by using the `link:labelLink` network that is part of the XBRL report. LabelLinks are another type of extended link that associates report elements with strings. They are implemented in the same way as presentationLinks and calculationLinks, but this time they not only contain arcs and locators, but also `link:label` elements.
From a semantic perspective, labels are different from report elements such as concepts and abstracts. Instead, they are a kind of `resource`. Resources are essentially metadata about report elements, facts, and other elements of an XBRL report.
The approach that XBRL takes to labels and other resources is quite interesting. Going back to our example in figure 4.11, the definition of the concept `us-gaap:Revenues` happens independently from any labels that are associated with it. The labels are later associated with the concept using the `link:labelLink` network. In fact, each report element can potentially have multiple labels in different languages or different degrees of verbosity. To categorize labels, XBRL uses the concept of `roles`, which are covered in more detail later in this chapter. We have already seen the `xlink:arcrole` attribute in the `link:presentationArc` 4.6.2. The role of a label works in a similar way.
In terms of the XML syntax, the `link:labelLink` network is implemented in the same way as the `link:presentationLink` network. The only addition is the `link:label` element, which is used to represent a label. It contains a few pieces of information:

1. The `xlink:label` attribute, which is used to reference the arc that the label is associated with. This must not be confused with the text of the label. Think of it as a unique identifier for the label.

2. The `xlink:role` attribute, which specifies the role of the label. More on this later.
3. The `xml:lang` attribute, which specifies the language of the label.
4. The `xlink:type` which is always set to `resource`.
5. The actual label text. This is the human-readable label that Arelle displayed in figure 4.11.

For example, for the label "Net Operating Revenues" 4.11, the following XML segment would be used:

```
<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 work in the same way as in all other networks.
The label in figure 4.12 has the role `http://www.xbrl.org/2003/role/terseLabel`. This role is used to indicate that the label text is short and concise. XBRL defines a few other roles of which the most important ones are:

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

Note that the 4.13 is not exhaustive. There are many more label roles that are used in practice. Users can even define their own label roles. For a complete list of standard label roles, refer to the XBRL 2.1 specification[14].

4.6.7 referenceLink

The `link:referenceLink` network is used to link report elements to external resources. For example, the concept `us-gaap:Revenues` might have an official definition in the SEC's Code of Federal Regulations (CFR). The reference establishes a link between the concept and external resource such as the CFR. Intuitively, think of the reference as a citation in a scientific paper.
Structurally, referenceLinks are implemented in the same way as LabelLinks - they contain arcs, locators, and resources. The only difference is that the resources are

references to external resources, not labels. Whereas labels are mostly just text, references are take the form of dictionaries.
The following figure shows an example of a reference in the XML syntax. Both the accompanying arc and locator are omitted for brevity. The only noteworthy changes to the arc are the tag `link:referenceArc` and the `xlink:arcrole` attribute, which is set to `concept-reference`.

```
<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 children of the `link:reference` element form a dictionary that describes the external resource that the reference points to.
In the example 4.14, the reference points `17 CFR 229.402(v)(2)(vi)` [29] [11]. References can point to any kind of external resource, not just the CFR. They can point to other XBRL reports, PDFs, websites, etc.
Usually, an XBRL report contains only one referenceLink, which is used to link the concepts in the report to the concepts to the underlying code of regulations.

4.6.8 footnoteLink

The `link:footnoteLink`, just like the `link:referenceLink` and the `link:labelLink`, is used to associate report elements with resources. One difference is that the resources are footnotes, not labels or references, which is a surface level difference. The other difference is that the locators in the footnoteLink can also reference facts, not just report elements. Other than that, footnoteLinks are implemented in the same way as the other 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 where `Roles` come into play.
Roles are a way to group networks together into a what is essentially a chapter of a report. Each set of networks is assigned a unique URI and potentially a label.

[11] CFR stands for Code of Federal Regulations.

For example, a report might have a role for the cover page, one for the balance sheet, one for the income statement, and so on. The balance sheet would only contain a presentation network, while the income statement would contain a presentation network, a calculation network, and potentially a definition network.

A role usually contains a presentation network, a calculation network, and a definition network. The other types of networks are not commonly used in roles. Rather, they belong to the report as a whole. An example of this would be a label network that contains all the labels for the entire report.

Roles in the XBRL XML syntax follow a simple structure, which I will explain using an example 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>
```

The role in figure 4.15 has the following properties:

- `roleURI` (required): The URI of the role. This URI is used to reference the role from other elements in the XBRL taxonomy. It is the primary identifier of the role.
- `definition` (optional): A human-readable description of the role.
- `usedOn`: A list of links that the role can be used in.

The networks that are associated with the role are not defined in the role itself. Rather, each link that uses the role has to declare the role in the `role` property, which is used to reference the role from the link.

Whenever a link references a role, the role must have a `usedOn` property that contains the type of the link. Going back to figure 4.15, if a definition network would reference the balance sheet role, a conformant XBRL processor would throw an error. This is because the balance sheet role does not declare the `definitionLink` type in its `usedOn` property.

4.8 Hypercubes

One key observation that can be made when looking at the facts of an XBRL report is that they are often structured like a hypercube. The aspects of a fact can be seen as the dimensions of a hypercube, whereas the value of the fact is the value of the hypercube at the given dimensions. Unlike networks, hypercubes are part of the OIM [12].

[12] Hypercubes are the reason why the OIM does not only cover the XBRL 2.1 core specification, but also the XBRL Dimensions 1.0 specification.

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

Hypercubes are a common way to structure data nowadays. Yet, back when XBRL was created, they were not as prevalent as they are today. In fact, the early versions of XBRL did not support hypercubes at all. They were retrofitted into the XBRL specification in 2006.[11].

4.8.1 Dimensions

When viewing facts as hypercubes, the cube ends up having four built in dimensions. These correspond to the four core aspects of a fact: `Period`, `Entity`, `Concept`, and `Unit`. XBRL allows for the creation of custom dimensions, which come in two flavors: explicit and typed.
Unfortunately, XBRL overloads the term "dimension". It refers to both the dimensions of a hypercube, as well as the two custom dimension types.

4.8.2 Explicit dimensions

Explicit dimensions are dimensions that have a predefined set of possible values. For example, let us assume that the Foo Company has two subsidiaries: Foo United States and Foo Europe. The Foo Company could then create a dimension called `Subsidiary` with the two possible values `Foo United States` and `Foo Europe`. The possible values of an explicit dimension are called `members`.
Both dimensions and members are defined using report elements, just like concepts and abstracts before them. To symbolize that a member belongs to a dimension, the member is defined as a child of the dimension in the definition network.
The members of a dimension can also have even more child members themselves. For example, the Subsidiary Foo Europe could have 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 are dimensions that do not have a predefined set of possible values. Instead, the values of a typed dimension are constrained by a data type.

For example, a dimension could be constrained to only allow values of the type `xs:integer`.
Similar to explicit dimensions, typed dimensions are defined using report elements. Unlike explicit dimensions, typed dimensions do not have members. They consist solely of the dimension report element, which defines the data type of the dimension.

4.8.4 Line items and hypercubes

With our current understanding of hypercubes, we can only view the whole report as a single, gigantic hypercube. Especially when considering the additional dimensions, most facts will use only a small subset of the possible dimensions. This makes the resulting hypercube high dimensional, with most of the dimensions being unused. Using the large hypercube as a basis for analysis would be very inefficient.
To solve this problem, XBRL introduces the `hypercube` report element. Conceptually, a hypercube is a sub-hypercube of the whole report hypercube. Hypercube report elements are usually defined an a per-role basis as part of a definition network. It picks a subset of the dimensions of the whole report hypercube. This subset is determined in the definition network, where "hypercube-dimension" arcs specify which dimensions are part of the hypercube.
Besides the hypercube report element, XBRL also introduces the `lineItems` report element. LineItems are used to specify which concepts are part of the hypercube. Reports can specify tens of thousands of concepts, but only a few of them are relevant for a particular role. The LineItems report element specifies the relevant concepts by listing them as children in the definition network.
If understanding lineItems proves to be difficult, consider the following: LineItems are to concepts what dimensions are to members.

4.9 XBRL Epilogue

As we have seen in this chapter, XBRL is a complex standard with many moving parts. It is a standard that has been in development for over 20 years, and it shows. In the first half of this chapter, we have seen how an XBRL report is, at its core, just a collection of facts. The second half of this chapter has shown us how these facts can be structured into networks, roles and hypercubes. This chapter was in no way exhaustive, and there are many more aspects of XBRL that we have not covered. However, the aspects that we have covered are the ones that are most relevant to this thesis. The chapter was also, as mentioned in the introduction, heavily based on both the XBRL standard[9] created by Charles Hoffman and Dr. Ghislain Fourny's interpretation of it in *the XBRL Book*[8].
The next chapter will introduce the API of Brel, and how it interprets the XBRL standard. The API chapter precedes the implementation chapter, which will explain how Brel implements the XBRL standard. Even though the API is part of the result of this thesis, it makes more sense to introduce the API before its implementation.

Chapter 5

API

This chapter describes the Brel API. It serves as a top-down overview of what Brel is capable of. The API differs from the underlying XBRL standard in key areas and aims to abstract and simplify XBRL, while still providing access to the full power of XBRL when needed.
The chapter is not intended to be a complete reference. Brel contains various helper functions and classes not described in this chapter. A complete reference of the Brel API can be found in the Brel API documentation[23]. The classes and methods described in this chapter define the minimal set of functionality needed to fully access all of Brel's features. They are designed to cover the underlying XBRL standard in a way that is easy to use and understand. Every additional feature of Brel could be implemented using the API described in this chapter. [1] The API of Brel can be divided into different parts, all of which are described in this chapter.

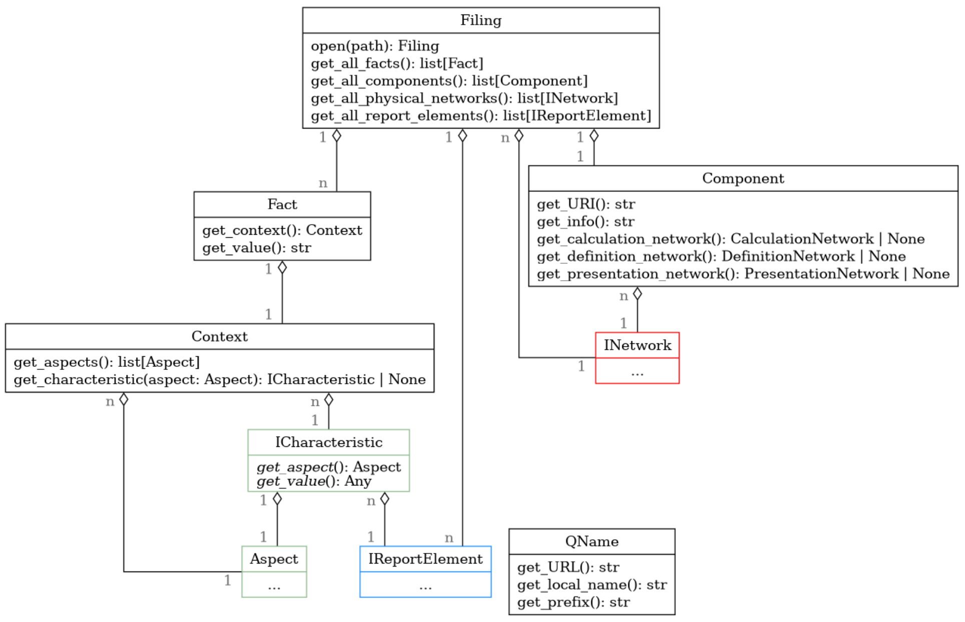
1. **Core** - The first part describes the `Core` of Brel, which consists of Filings, Facts, Components and QNames.
2. **Characteristics** - The second part describes the `Characteristics` of Brel, which covers Concepts, Entities, Periods, Units and Dimensions.
3. **Report Elements** - The third part covers `Report Elements`, specifically Concepts, Members, Dimensions, Line Items, Hypercubes and Abstracts.
4. **Resources** - The fourth part covers `Resources`, specifically Labels, References and Footnotes.
5. **Networks** - Finally, the `Networks` part describes how networks and their nodes are represented in Brel.

All of these parts should sound very familiar, since they were extensively discussed in chapter 4. Even though the previous chapter was already light on XML implementation details, the API described completely abstracts away the underlying XML structure. [2] This chapter will answer research questions 2.2 and 2.2.
Again, the current implementation of Brel is not complete. It does not allow for the creation of new filings, facts, components, etc. Its main purpose is to provide a way to access and analyze existing filings. Brel also does not analyze the semantics of the underlying reports.

[1]Obviously, some of the helpers directly access the underlying XBRL standard and are not implemented using the API to avoid unnecessary overhead.
[2]The only exception is the `QName` class, which is an almost direct representation of the XML QName type.

5.1 Core

The core of Brel consists of filings, facts, components and QNames, where each element is represented by one or more classes. In essence, each filing consists of a set of facts and components. QNames are used all across Brel, which is why they are considered part of the core. The following UML diagram shows the core of Brel.



Filing
open(path): Filing
get_all_facts(): list[Fact]
get_all_components(): list[Component]
get_all_physical_networks(): list[INetwork]
get_all_report_elements(): list[IReportElement]
Fact
get_context(): Context
get_value(): str
Component
get_URI(): str
get_info(): str
get_calculation_network(): CalculationNetwork | None
get_definition_network(): DefinitionNetwork | None
get_presentation_network(): PresentationNetwork | None
Context
get_aspects(): list[Aspect]
get_characteristic(aspect: Aspect): ICharacteristic | None
INetwork
...
ICharacteristic
get_aspect(): Aspect
get_value(): Any
Aspect
...
IReportElement
...
QName
get_URL(): str
get_local_name(): str
get_prefix(): str
1
n

Figure 5.1: UML diagram of the core of Brel

5.1.1 Filing

Starting with the `Filing` class, it represents a single XBRL filing. Obviously, it needs a method for loading a filing from a file, directory, or URL. The `Filing.open` covers all of these cases. It takes a single argument, which can be a file path, directory path or URL. The method will automatically detect the type of the argument and load the filing accordingly. It will also reject any invalid arguments.
The next two methods are `Filing.get_all_facts` and `Filing.get_all_components`, which return all facts and components of the filing, respectively. Both the facts and components are returned as a list of `Fact` and `Component` objects. The order of the facts and components is not guaranteed. Brel does provide helper methods for getting specific facts and components, but their functionality can be easily implemented using the `get_all_facts` and `get_all_components` methods.
There exist some networks that are not part of any Component, specifically `physical networks`. These networks cannot be accessed indirectly through Components, which is why the `Filing` class also provides a method for getting all physical networks. This method is aptly named `Filing.get_all_physical_networks`.
Similar to networks that are not part of any Component, there are also report elements that are not part of any network or fact. The method `Filing.get_all_report_elements` returns all report elements of the filing, including both report elements that are referenced by facts or networks and those that are not.
The two most important classes associated with a filing are `Fact` and `Component`. Both classes encompass two different core aspects of XBRL. All classes belonging to facts are covered by the Open Information Model (OIM)[17]. Their design will therefore answer research question 2.2. Classes belonging to components mostly

involve topics that are not covered by the OIM. Consequently, their design will answer research question 2.2.
The subsequent sections will first describe the `QName` class, since it is used by numerous other classes. The next few sections will cover the `Fact` class and all of its associated classes. Finally, the last subsections are dedicated to the `Component` class and its associated classes.

5.1.2 QName

The `QName` class represents a qualified name, which is a combination of a namespace and a local name. It is the only remnant of the underlying XML structure of XBRL in the Brel API. Since it already provides an elegant way of identifying elements across different namespaces, We chose to keep it in the API.
As described in chapter 4, a QName is a combination of a namespace URL, a prefix and a local name. Naturally, the `QName` class provides methods for accessing all three of these components. These methods are fittingly named `QName.get_URL`, `QName.get_prefix` and `QName.get_local_name`.

5.1.3 Fact

The `Fact` class represents, as the name suggests, a single XBRL fact. When boiled down to its core, a fact consists of a value and a set of characteristics that describe what the value represents. The value of a fact is represented by the `Fact.get_value` method, which returns the value as a string. The characteristics of a fact are represented by the `Fact.get_context` method. A context is a set of characteristics that describe the fact. In other terms, each fact occupies a position in a multi-dimensional space, where each characteristic represents a point along one dimension.
One design decision that seems questionable at first is the `Fact.get_value` method, which returns the value as a string. Not all values in XBRL are strings. Some are integers, decimals, dates, etc. The reason for this design decision is that Facts have a unit characteristic, which determines the type of the value. Therefore, units in XBRL are represented as a dimension of the fact's context. Since all values that XBRL facts can take are representable as strings, the `Fact.get_value` method returns the value as a string, since it is the most general representation of the value.
Brel does however provide helper methods for converting the value into the type that most appropriately represents the value. The helper method checks the unit of the fact and converts the value accordingly.

5.1.4 Context

Contexts, as described in section 5.1.3, are sets of characteristics what a fact represents. The `Context` class represents a single context. Each fact has its own context and each context belongs to exactly one fact. [3] Contexts provide two methods for accessing their characteristics. The `Context.get_aspects` method returns a list of all aspects for which the context has a characteristic. The `Context.get_characteristic` method returns the characteristic of the context for a given aspect. If the context does not have a characteristic for the given aspect, the method returns `None`.
To reiterate, a characteristic represents a point along a dimension single dimension. Multiple characteristics can be combined to form a point in a multi-dimensional space.

[3]There might be multiple contexts that have identical characteristics, but they are still represented as separate objects.

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". Both points point to values in a four-dimensional hypercube. [4]

5.1.5 Aspect and Characteristics

Aspects describe the dimensions along which characteristics are positioned. Their API is described in section **??**, which will be covered in a later portion of this chapter.

5.1.6 Report Elements

Report elements are the building blocks of XBRL filings. Probably the most important report element is the concept, initially explained in section 4.3. In figure 5.1, the interface `IReportElement` represents all report elements, not just concepts.
Obviously, some characteristics use report elements to describe the position of a fact along a dimension. Take the previous example 5.1.4 for instance. One of the characteristics uses the concept "net income" to describe the position of the fact along the concept dimension. In this case, the characteristic is uses the concept "net income".
Since there are multiple types of report elements, the `IReportElement` interface provides a method for getting the type of the report element. Report elements have their own dedicated section **??**, which will be covered in a later portion of this chapter.

5.1.7 Component

Moving on to the other side of figure 5.1, the `Component` class represents the chapters of a filing. Components are the first class not by the OIM.
Each component consists of a number of networks and an identifier. A component can have at most one network of each type. The available network types are calculation-, presentation- and definition networks. Additionally, a component can have an optional human-readable description of what the component represents.
The `Component.get_calculation_network`, `Component.get_presentation_network` and `Component.get_definition_network` methods return the calculation, presentation and definition network of the component, respectively. Each of these methods can return `None`, if the component does not have a network of the requested type.
The `Component.get_uri` method returns the identifier of the component, which is a URI that uniquely identifies the component within the filing.
The `Component.get_info` method returns the human-readable description of the component, if it exists. If the component does not have a description, the method returns an empty string.
The networks that are part of a component are represented by the `Network` class. Networks will be covered in the second half of this chapter. The first half focuses on OIM concepts, while the second half focuses on non-OIM concepts.

[4] In these examples, the four dimensions are entity, period, unit and concept.

5.2 Report Elements

Since characteristics use report elements, we introduce report elements first. Report elements were introduced in chapter 4. There are multiple types of report elements, which are all represented by different classes in Brel. All of these classes implement a common interface called `IReportElement`.
In total, we introduced six different types of report elements:

1. **Concept** - Concepts define what kind of information a fact represents.

2. **Abstract** - Abstracts are used for grouping other report elements.

3. **Dimension** - Dimensions are used to describe a custom axis, along which a fact is positioned.

4. **Member** - Members specify the point along a dimension that a fact is positioned at.

5. **LineItems** - Line items are used to group concepts into an axis, similar to how dimensions group members into an axis.

6. **Hypercube** - Hypercube elements describe a smaller sub-hypercube of the filing's global hypercube.

Technically, only concepts, members and dimensions are part of the OIM, whereas the remaining three are not. However, from an editorial point of view, it makes sense to describe all of them in one place. Brel chooses to implement report elements as seen in figure 5.2.

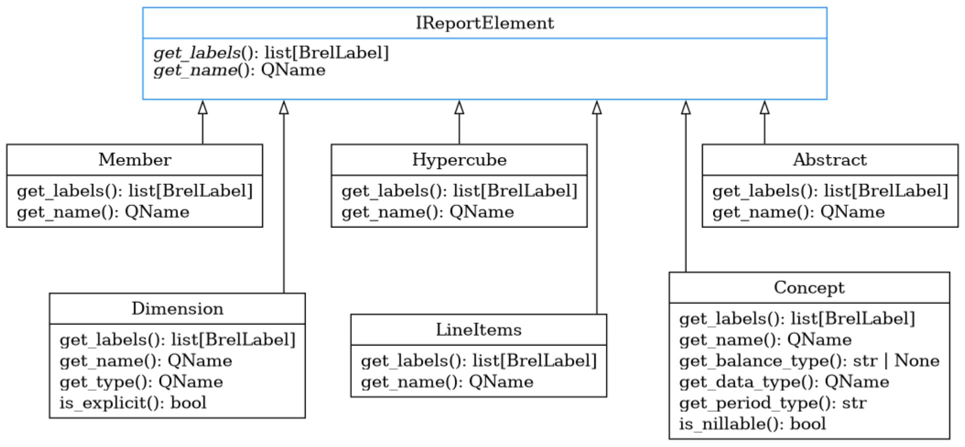


Figure 5.2: UML diagram of the report element classes in Brel

As a general rule of thumb, we designed Brel's inheritance hierarchy to be as flat as possible. Since all report elements share a common interface, the next section covers the interface first. The subsequent sections then describe the different types of report elements.

5.2.1 IReportElement

The `IReportElement` interface defines the common interface that all report elements share. In essence, a report element is nothing more than an name, in this case a QName. QNames were introduced in chapter 4 and their class was described in section 5.1.2. Since QNames are not completely human-readable and do not support multiple languages, Brel also provides a method for getting the label of a report element. This chapter has not yet covered labels, but they will be described in section **??**. However, they should be conceptually self-explanatory.
The `IReportElement` interface provides the methods `get_qname` and `get_labels`, which act as their names suggest. The `get_labels` method returns a list of labels, since a report element can have multiple labels.

5.2.2 Concept

The `Concept` is the most important report element in XBRL. Concepts are the type of report element that are required to be present in every fact. They define what kind of information a fact represents.
Besides the methods that are inherited from the `IReportElement` interface, the `Concept` also provides information about its associated facts.
TODO: finish this section

5.2.3 Dimension

Dimensions are used to describe a custom axis, along which a fact is positioned. Dimensions come in two different flavors, explicit and typed. Besides the methods that are inherited from the `IReportElement` interface, the `Dimension` has two additional methods. The first method `is_explicit` returns a boolean value, indicating whether the dimension is explicit or not. The second method `get_type` returns the type of the dimension, if it is typed. It raises an exception if the dimension is explicit.
There is no need for the method `get_members` in the `Dimension` class, since Brel models members differently. The dimension-member relationship is modeled as a parent-child relationship between the `Dimension` and `Member` objects in definition networks. So the members of a dimension change depending on the component that the dimension is part of.
We chose to combine both typed and explicit dimensions into a single class, since they are semantically very similar. They occupy the same position within networks, but have some slight differences in their behavior. From a user's perspective, these differences are negligible.

5.2.4 Abstract, Hypercube, LineItems, and Member

The `Abstract`, `Hypercube`, `LineItems` and `Member` classes are all very similar. Essentially, their only differentiating factor is their name. Besides that, they all provide the same methods and attributes. Namely, they implement the `IReportElement` interface. We decided to split them into four different classes, since they are semantically different.

5.3 Characteristics

Characteristics are used to describe the position of a fact along a dimension. Some of them rely on report elements for description, while others introduce new concepts. All characteristics share a common interface `ICharacteristic`. Each characteristic acts as a aspect-value pair, where the aspect characterizes the dimension's axis and the value details the position of the fact along the axis. The interplay between aspect and characteristics classes is illustrated in figure 5.3.

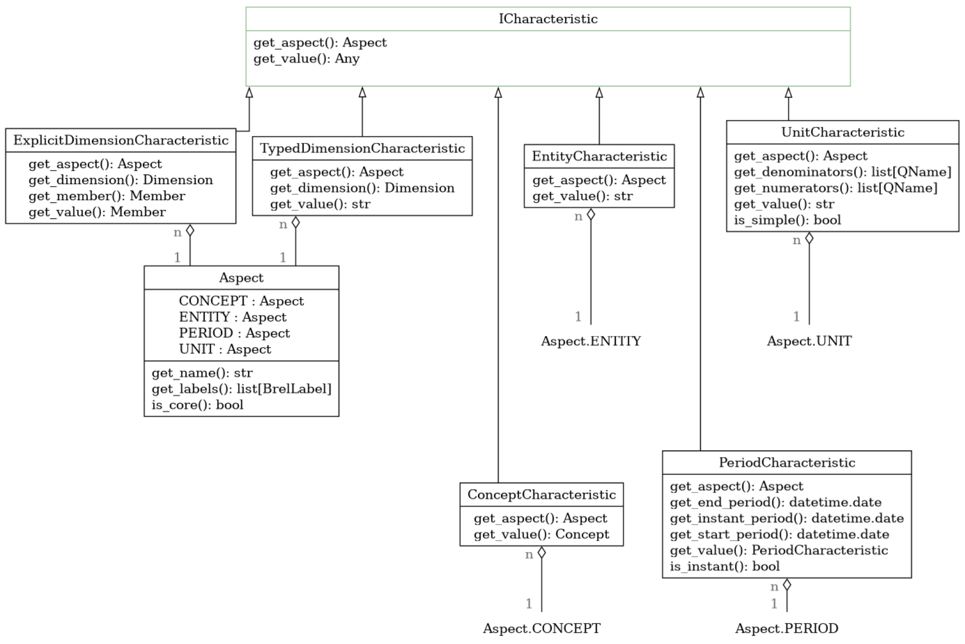


Figure 5.3: The interplay between aspects and characteristics

The `ICharacteristic` interface is integral to the Brel API, which is why it is described first. Next up is the Aspect class, followed by the different types of characteristics.

5.3.1 ICharacteristic

As I have pointed out in the previous section, characteristics are used to describe the position of a fact along a dimension using an aspect-value pair. An aspect is a description of the dimension's axis, while the value details the position of the fact along the axis.
The `ICharacteristic` interface follows this definition directly by providing the methods `get_aspect` and `get_value`.

5.3.2 Aspect

Aspects are used to describe the axis of a dimension. Each instance of the `Aspect` class represents a single aspect. The core aspects - Concept, Entity, Period and Unit - are all instances of the `Aspect` class. In addition, the core aspects are statically available as public constants of the `Aspect` class. The fields in question are `Aspect.CONCEPT`, `Aspect.ENTITY`, `Aspect.PERIOD` and `Aspect.UNIT`.
The `Aspect` class provides a method for getting the name of the aspect, called `get_name`. The name of an aspect is a string instead of the QName class. The

main reason for this is that the core aspects are available globally without the use of namespaces. Most facts exclusively use the core aspects. So the namespaces of QNames would only add unnecessary clutter. The only exception to this rule are dimensions, where the name of the dimension is a QName. Still, QNames can just be emulated by using strings using their expanded name format. [5] [34]
Similar to the `IReportElement` interface, the `Aspect` class also provides a method for getting the label of the aspect. The `get_labels` method returns a list of labels, since an aspect can have multiple labels.
Finally, the `is_core` method returns whether the aspect is a core aspect or not.

5.3.3 Concept Characteristic

The `ConceptCharacteristic` indicates which concept a fact uses. From the perspective of hypercubes, the `Aspect.CONCEPT` aspect is a dimension of concepts and the actual `Concept` report element is a point along that dimension. The dimension characteristic is the only characteristic that every context has to have.
The `ConceptCharacteristic` implements `ICharacteristic` as one would expect. `get_aspect` returns `Aspect.CONCEPT` and `get_value` returns the concept that the characteristic describes.

5.3.4 Entity Characteristic

The `EntityCharacteristic` dictates which entity a fact belongs to. From the perspective of hypercubes, the `Aspect.ENTITY` aspect is a dimension of entities An entity is a legal entity, such as a company and can be identified by a tag and a scheme[6] that acts as the namespace of the tag. Both of these values are combined into a single string using the notation {`scheme`}`tag`.[7]
The `EntityCharacteristic` implements the `ICharacteristic` interface, where `get_aspect` returns `Aspect.ENTITY` and `get_value` gives the string representation of the entity as described above.

5.3.5 Period Characteristic

The `PeriodCharacteristic` describes the period 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 the period respectively. If the period is instant, the methods raise an exception, since instant periods do not have a start or end date. 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 have an instant. All three methods return a date of type `datetime.date`, which is a standard Python class for representing dates.
Again, period characteristics implement the `ICharacteristic` interface. `get_aspect` returns `Aspect.PERIOD` and `get_value` returns the period characteristic itself. The reason why `get_value` returns itself is that there is no basic type for representing XBRL periods in python. The python `datetime` module, which is the de-facto standard for representing dates in python, does not provide a class for representing both instant and duration periods in a single class.

[5] The expended name format of a QName is `namespace_prefix:local_name`.[35]
[6] Schemes tend to be URLs.
[7] The notation is similar to the Clark notation for QNames.[34]

5.3.6 Unit Characteristic

The `UnitCharacteristic` describes the unit of a fact. Like all other characteristics before it, the `UnitCharacteristic` represents a point along the unit dimension and it implements the `ICharacteristic` interface. From a semantic point of view, the unit characteristic also defines the type of the fact's value. A fact with a unit of `USD` has a value of type `decimal` and a fact with a unit of `date` has a value representing a date.
Units come in one of two forms - simple and complex. Simple units are atomic units, such as `USD` or `shares`. Complex units are composed of multiple simple units, such as `USD per share`. All complex units are formed by dividing one or more simple units by zero or more 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 represents the complex unit in figure 5.4 using two lists of simple units. The method `get_numerators` returns the list of simple units in the numerator, `get_denominators` returns the denominators.[8]
Similar to the `PeriodCharacteristic`, the `UnitCharacteristic` does not have a basic type for representing XBRL units. Instead, it returns itself when `get_value` is called. The method `get_aspect` returns `Aspect.UNIT` as expected.

5.3.7 Dimension Characteristics

There are two categories of dimension characteristics in XBRL - typed and explicit. Typed dimension characteristics are used to describe a custom axis, along which a fact is positioned. The kind of values along this custom axis are of a specific type. Like every other characteristic, typed dimension characteristics implement the `ICharacteristic` interface.
As we have seen in section 5.2, custom dimensions are represented as a `Dimension` report element in Brel. So the aspect of a typed dimension characteristic should represent a `Dimension` report element. Luckily, `Dimension` objects are essentially just a name, which is represented by a QName. Therefore, the `get_aspect` method of the `TypedDimensionCharacteristic` class returns the QName of the dimension as a string.
The characteristic also provides direct access to the `Dimension` object itself via the `get_dimension` method. Think of `get_dimension` as a more complete version of `get_aspect`.
As the name suggests, the value of a typed dimension characteristic is of 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. [9] Naturally, Brel provides helper methods for converting the value into the type that most appropriately represents the value. These helper methods are

[8]The returned list of denominators does not contain the implicit denominator of 1.
[9]The `Dimension` object returned by `get_dimension` is guaranteed to be a typed dimension with `is_explicit` returning `False`.

not part of the minimal API described in this chapter, but they are part of the full 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.[17] 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.
Since the OIM is already quite tidy, the Brel API does not deviate much from it. Just like the OIM, the Brel API is not tied to any specific format for its underlying XBRL reports. It provides Reports, Facts, Concept-, Entity-, Period-, Unit and Dimension characteristics, which are all part of the OIM. [10]

- **Report** - The `Filing` class represents a single XBRL report. Just like the OIM, it acts as a wrapper around a list of facts. Aside from facts, a report also contains a taxonomy, which is a set of report elements, which are also accessible through the `Filing` class.
- **Fact** - The `Fact` and `Context` class represents a single XBRL fact. Just like the OIM, a fact consists of a value and a set of characteristics that describe what the value represents.
- **Characteristics** - Brel implements all of the characteristics described in the OIM as classes - concepts, entities, periods, units, explicit- and typed dimensions [11]

Therefore, the first half of this chapter offers a constructive answer to research question 2.2 by providing a python API that is based on the OIM.
Where the Brel API differs from the OIM so far is in its introduction of report elements. Yes, the OIM introduces concepts, dimensions and members, but it does not categorize them under a common umbrella term [12]. In the OIM, these three terms describe three completely unrelated things, and they are unrelated if one only considers the OIM. However, the Brel API also aims to cover the parts of XBRL that are not yet covered by the OIM. The non-OIM parts of XBRL are networks, components and resources. Networks require elements like concepts, dimensions and members to be treated in a homogeneous way. The exact reasoning behind this will be explained in the second half of this chapter.
The way Brel bridges the gap between the OIM and the non-OIM parts of XBRL is by introducing characteristics and report elements. Characteristics are used for facts, while report elements are used for networks. This is the reason why Brel uses `ConceptCharacteristic` instead of `Concept` when talking about the concept characteristic of a fact. Sure, a `ConceptCharacteristic` is in essence a wrapper around a `Concept`, but the two classes are used in different contexts.
The second half of this chapter will answer research question 2.2 by providing a python API that is based on the non-OIM parts of XBRL.

[10]The OIM does not use "characteristic" suffix. Brel uses it to avoid confusion with similarly named report elements.

[11]The OIM also introduces the language- and Note ID core dimensions. They are not yet implemented in Brel, but can be emulated using the typed dimension characteristic. They are rarely used in practice, which is why they are not yet implemented.

[12]The OIM would technically group concepts and members under the term "dimension", but it overloads the term "dimension" so many times that it is not clear what it refers to in any given context.

5.5 Resources

Before introducing networks, we first need to introduce Brel's approach to resources. Resources are XBRL's way of representing metadata, which links to other elements of the an XBRL report using networks.

Like report elements and characteristics before them, resources share a common interface. There are three types of resources in XBRL - label, reference and definition. Each of these types of resources is represented by a different class in Brel. The class diagram in figure 5.5 illustrates the relationship between the different resource classes.

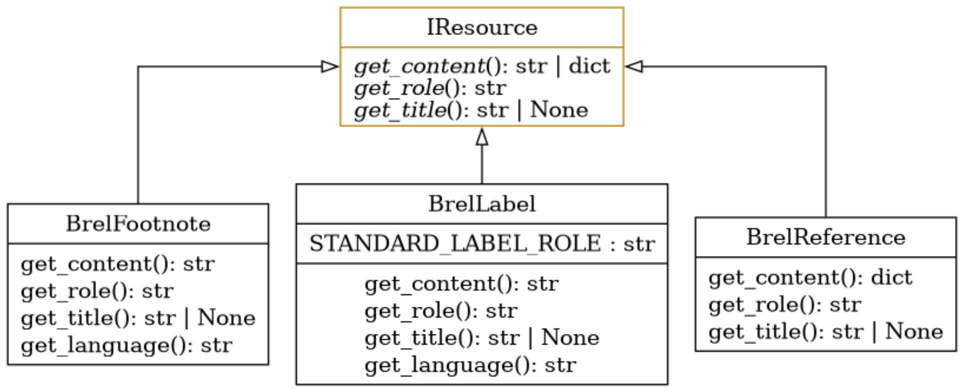


Figure 5.5: UML diagram of the resource classes in Brel

5.5.1 IResource

Resources consist of three parts - a role, a title and content.

The role acts as a type identifier for the resource. For example, even though each XBRL label is represented using a `BrelLabel` object, there are different types of labels. Different types of labels are distinguished using their role. As the name suggests, the `get_role` method returns the role of the resource.

Next up is the content of the resource, which is accessed using the `get_content` method. Usually, the content is a string. For references however, the content is embedded XML. Since Brel intends to remove any XML dependencies, it returns the embedded XML as a dictionary instead.

The title, accessed using the `get_title` method, is a human-readable description of the resource. For labels, the title is often omitted, since the label itself is already short and descriptive. However, the content of a resource can be arbitrarily long, which is why XBRL supports titles.

5.6 Labels and Footnote

Footnotes and labels are two types of resources that are used to link to other elements of an XBRL report. Even though they have their own classes, they are very similar from an API perspective.

Both of them implement the `IResource` interface. The methods `get_role`, `get_title` and `get_content` function virtually the same. Unlike their common interface, labels and footnotes have an additional method called `get_language`. The `get_language` method returns the language of the resource.

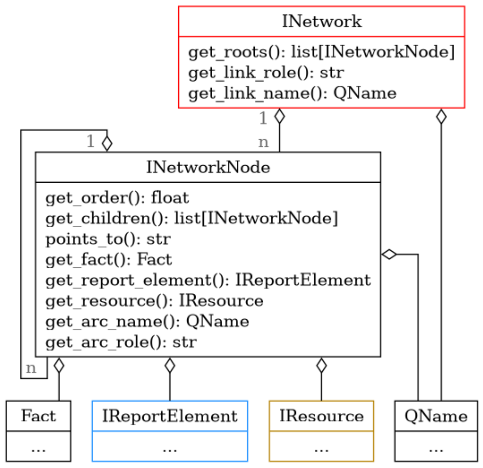
References

References are the third type of resource in XBRL. They are used to link XBRL reports to external resources. References are represented by the `Reference` class in Brel and implement the `IResource` interface. They do not have a `get_language` method like labels and footnotes do. Another difference is that the content of a reference is a dictionary instead of a string.
Now that we have covered resources, we can finally introduce networks.

5.7 Networks

Networks are the final piece of the puzzle that makes up the Brel API. Together with resources and components, they are the parts of XBRL that are not yet covered by the OIM. Brel aims to give them the same treatment as the OIM parts of XBRL. This section will describe how Brel represents networks and their nodes.
Essentially, 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.
Like other parts of Brel before them, networks implement a common interface. This time, there are two interfaces - `INetwork` and `INetworkNode`.

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



5.7.1 INetwork and INetworkNode

The `INetwork` interface acts as a wrapper around a list of root nodes. It provides a method for getting all root nodes, called `get_roots`.
Besides the list of roots, the `INetworkNode` also gives 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, so why are they part of the Brel API?
The reason is that they are useful for debugging. Networks are the part of XBRL where filers are most likely to make mistakes. Both the link role and the link name serve as sanity checks for filers and analysts alike. They might be removed from the API in the future, but for now they are useful for debugging.
The `INetworkNode` interface also provides a method for getting all child nodes of a node, called `get_children`. A node can not access its parent node directly, but since the graphs roots are accessible through the `INetwork` interface, the parent of a node can be found by traversing the graph from the roots.
These two methods cover all of the functionality that is needed to traverse a network. The next methods are about getting the elements that a node points to.
Since a node can point to different types of elements, the `INetworkNode` interface provides the method `points_to`. 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`.

The interface also defines the methods `get_fact`, `get_resource` and `get_report_element`, which do exactly what their names suggest. If the node does not point to the requested element, the methods raises an exception.

The final methods of the `INetworkNode` interface are `get_arc_role` and `get_arc_name`. Similar to the link role and link name, these methods expose the underlying XML structure of XBRL and are only used for debugging.

5.7.2 Different network types

As described in section 4.6, there are six different types of networks in XBRL. All of them have their own Network- and Node-classes in Brel. The classes are named after the network type they represent, with the suffix `Network` or `NetworkNode`. The following table shows 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 deviation. Since the interfaces are so simple, yet gives access to all the information in the network, there is no need to change them for each network type. The different semantic meanings of the networks can be expressed as helper function, 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

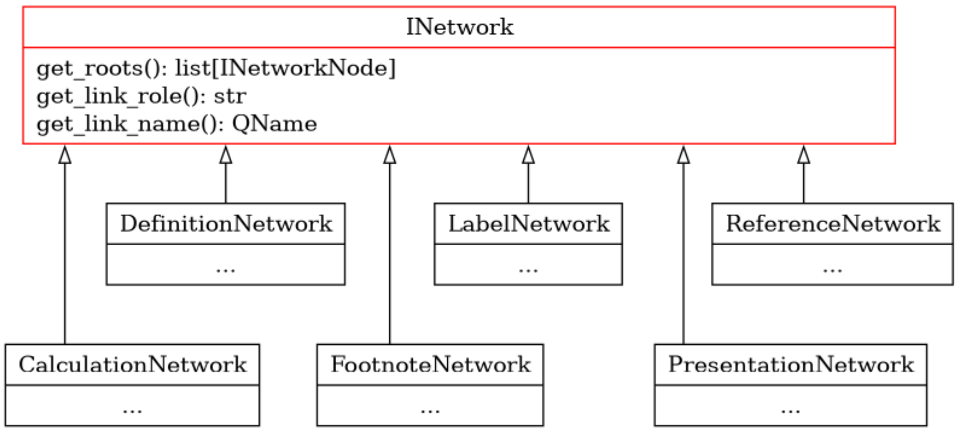
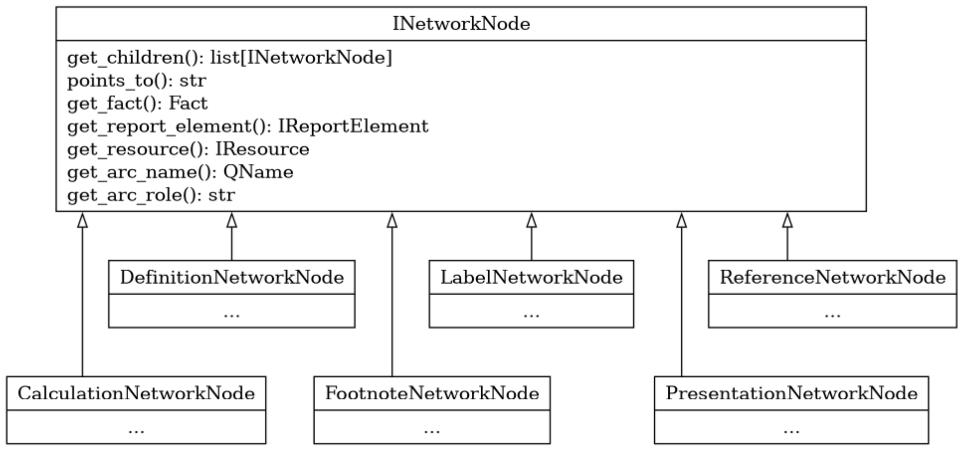


Figure 5.8: UML diagram of the node classes



With the network- and node-classes out of the way, Brel has covered all parts of XBRL it has set out to cover. The thing remaining with regards to the API is to see how the second half of this chapter answers research question 2.2.

5.8 Answering research question 2

The second half of this chapter introduced Brel's representation of networks and resources, both of which are not yet covered by the OIM. Therefore, a lot of the design decisions in this half of the chapter are not based on the OIM.
Research question 2.2 consists of two parts.

- First, the question asks how the non-OIM parts of XBRL can be converted into a python API.
- Second, it asks how this API can be made consistent with the OIM.

The first part of the question is answered by the second half of this chapter in a constructive fashion. This chapter an API for networks and resources, which break down the non-OIM parts of XBRL into their core components.
Answering the second part of the question requires a more abstract approach. In short terms, Brel marries the OIM and the non-OIM parts of XBRL by introducing report elements and characteristics.
The first half of this chapter already introduced a python API that is based on the OIM. The OIM's main goal was to report facts. Each fact has characteristics such as concepts, explicit dimensions, entities, etc.
The second half of this chapter was mostly concerned with networks and resources. Networks point to report elements among other things.
However, report elements were already introduced as part of the OIM, even though they are not strictly necessary for reporting facts. In fact, apart from concepts, dimensions and members, the OIM never even mentions any report elements.
The bridge between the two halves are characteristics and report elements, more specifically, there are three types of characteristics that are just wrappers around report elements. These characteristics are the concept characteristic, the explicit dimension characteristic and the typed dimension characteristic. The interplay between characteristics and report elements is illustrated in figure 5.9.

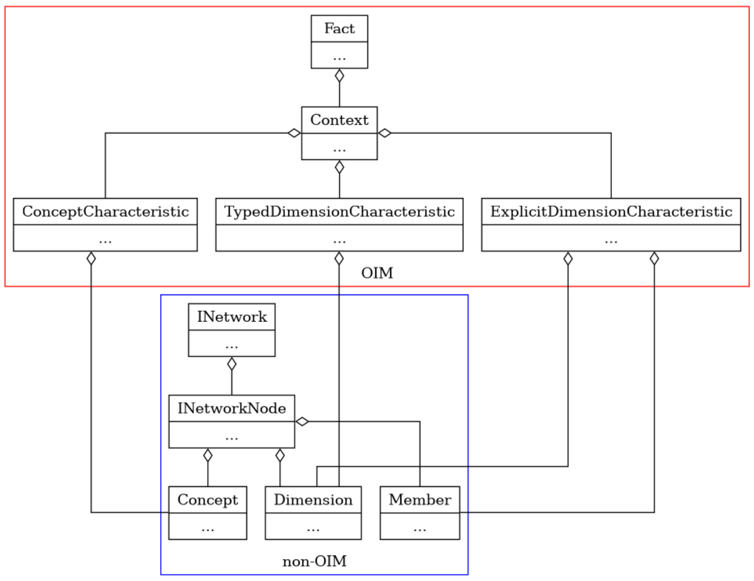


Figure 5.9: The interplay between characteristics and report elements

Where the OIM uses the characteristics to describe facts, networks use the report elements inside the characteristics. Of course, the class `Concept` for example could

have just implemented both the `IReportElement` and the `ICharacteristic` interface. However, concepts in facts and concepts in networks are used in different contexts, and they should not be used interchangeably. Therefore, Brel uses two different classes for the two different use cases.
This concludes the chapter on the Brel API. It covered both the OIM and the non-OIM parts of XBRL. It also explained how the two parts can be combined into a single API, in turn answering both research questions 2.2 and 2.2. With both the API and the underlying XBRL standard covered, the next chapter will introduce the implementation of the Brel API.

Chapter 6

Implementation

The implementation of the API closely follows the design of the API introduced in chapter 5. Brel is implemented in the programming language Python. Python is a high level language and one of the most popular programming languages in the world. Python is also popular outside of the computer science and software engineering communities as well. According to the 2020 Stack Overflow Developer Survey, Python is the fourth most popular programming language among all respondents, not just professional developers [22].
The reader may view the implementation of Brel as a translation from XBRL reports to python objects. Most of the translations from the design to the implementation are straightforward and will only be mentioned briefly, which the first section of this chapter will cover.
There are three key areas where the Brel's implementation differs from the underlying XBRL standard - DTS caching, namespace normalization and networks. Each area has its own dedicated section in this chapter.
This chapter exclusively focuses on XBRL reports in XML format.

6.1 General implementation

Brel parses XBRL reports using an eager bottom-up approach It starts with the smallest building blocks of XBRL reports - report elements. After all report elements have been parsed, Brel moves on to parsing facts and their associated characteristics. Next, Brel parses all networks and their associated resources. Finally, Brel parses the components of the report.
Brel chooses this bottom-up approach because both networks and facts depend on report elements. Networks depend on report elements since their nodes can point to report elements. Facts depend on report elements since their characteristics can refer to concepts, dimensions and members. Networks and facts often refer to the same report elements. Therefore, their python classes should share the same report element instances. The bottom-up approach ensures that all report elements are parsed before they are used by networks and facts. In the next four sections, this chapter will briefly cover the four stages of Brel's bottom-up approach.

6.1.1 Parsing report elements

Report elements are the smallest building blocks of XBRL reports. Therefore, they do not rely on any other XBRL elements and can be parsed first. Report elements are defined in the taxonomy set of the XBRL report, which is a collection of `.xsd` files in the XML format. These files are all stored locally on the user's computer.

Even though XBRL does not require the taxonomy set to be stored locally, Brel does. However, Brel automatically downloads the taxonomy set from the internet if it is not already stored locally. The mechanism for downloading the taxonomy set is discussed in section **??**.
Taxonomies contain three types of elements - linkbases, roles and report elements. Linkbases are discussed in section 6.2. Roles are discussed in section 6.1.3. Report elements are discussed in this section.
In XBRL, a taxonomy can refer to other taxonomies and associate them with a namespace prefix. For now, the reader can assume that different taxonomies agree on which namespace prefix and URI a given taxonomy should use. Brel ensures that this assumption holds true in a process called namespace normalization, which is discussed in section **??**. If all taxonomies agree on a prefix and URI for a given taxonomy, then all report elements defined in that taxonomy inherit the same prefix and URI as part of their QName.
Report elements within a taxonomy are organized in a flat list of XML elements Each XML element has a unique name attribute, which represents the local name of the report element's QName. Since the Brel API lists six different types of report elements, Brel needs to decide which type of report element each XML element represents. There is no single attribute in the XML element that indicates the type of report element. Instead, Brel uses a combination of different attributes to determine the type of report element. The process of determining the type of report element is outlined in the following table:

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

Table 6.1: Determining the type of report element

Brel uses the table 6.1 above to determine the type of report element. It traverses the table from top to bottom and selects the first row where all conditions are met. If a cell in the table is empty, then Brel ignores that condition.
The table above does not contain a row for the type "LineItems". The reason for this is that line items and abstracts can not be distinguished by their XML attributes. They can only be distinguished by their position within a definition network. Therefore, Brel parses line items and abstracts as abstracts. Brel later uses the definition network to determine which abstracts are line items, and fixes the type of these abstracts accordingly. This process is discussed in section 6.2.
Once all report elements have been parsed, Brel creates a lookup table for report elements. Given a QName, the lookup table returns the corresponding report element instance. This lookup table is used throughout the rest of the parsing process.

6.1.2 Parsing facts

Brel parses facts directly after parsing report elements. Facts are parsed before networks because footnote networks can point to facts.
Facts are exclusively defined in the instance document of the XBRL report. The instance document is an XML which contains a flat list of facts, syntactic contexts and units, which are represented by XML elements. It may also contain a list of footnotes, which are discussed in section 6.2.

Fact XML elements contain the value of the fact as well as references to the syntactic context and unit. The tag of the XML element is the QName of the concept of the fact.
Syntactic context XML elements describe a subset of the characteristics of a fact. They are different from `Contexts` as defined by the Brel API. A `Context` in Brel contains all the characteristics of a fact, whereas a syntactic context only contains the period, entity and dimensions of a fact. During parsing, Brel uses syntactic contexts as a starting point for creating `Context` instances. It then adds the remaining characteristics - the concept and the unit - to the `Context` instance.
Unit XML elements define, as the name suggests, the unit of a fact.
The reason why XBRL separates facts, syntactic contexts and units into three different XML elements is to reduce redundancy. Facts can share the same syntactic context and unit.
Brel parses all facts by finding all fact XML elements and resolving their references to syntactic contexts and units. It re-uses units, entities and dimensions across different facts.

6.1.3 Parsing Components

Components are the last section of the XBRL report that Brel parses. Before parsing components, Brel has already parsed all report elements, facts and networks. So far, this chapter has not discussed networks. Since parsing networks is complex, it is discussed in its own section - section 6.2. For now, the reader can assume that Brel has already parsed all networks, and that there is a lookup table for networks.
Components, like report elements, are defined in the taxonomy set of the XBRL report. XBRL chooses to call them "roleTypes" instead of "components". To parse all components, Brel scans all taxonomy files for roleType XML elements.
The roleType XML elements contain a three elements - A role URI, an optional description and a list of used-on elements. `Components` in Brel directly read both the role URI and the description from the roleType XML element. To get the networks that are associated with a component, Brel indexes the lookup table for networks by the role URI.
The used-on elements are a list of network types that are permitted to use the component. For example, if the network lookup returns a `PresentationNetwork` instance, then the roleType XML element must contain "presentationLink" in its list of used-on elements.
This concludes the section on the general implementation of Brel. Apart from networks, this section covered every part of XBRL and how Brel parses it. The next section will discuss network parsing in detail.

6.2 Implementation of Networks

6.2.1 Overview

In chapter 4 we introduced the concept of networks, followed by Brel's interpretation of networks in chapter 5. A network in Brel comprises of two different classes: `INetwork` and `INetworkNode`.
A network in Brel is a directed acyclic graph. [1] Each node in the graph has an ordered list of children and at most one parent. The network does not necessarily have to be connected, i.e. there can be multiple disconnected subgraphs in the network.

[1]The XBRL specification does allow cycles in networks, but Brel does not support them.

Technically speaking, a network in Brel is just a set of root nodes. These nodes point to their children, which in turn point to their children, and so on. So, to traverse a network, we only need to know the root nodes of the network. The `INetwork` class provides a method to get all root nodes of the network and the `INetworkNode` class provides a method to get all children of a node, which allows us to traverse the network.
One implementation detail of networks in Brel is that they cannot be empty. They have to contain at least one node. The functionality of an "empty" network can still be achieved by letting the component of the network point to `None`.

6.2.2 Parsing Links into Networks

Section 4.6 introduced the different types of networks, and how they are all just a collection of arcs, locators and resources. It also explained how locators and resources represent nodes and how arcs represent edges in a network. So parsing a link into a network is just a matter of converting a list of nodes and edges into a graph. Brel's algorithm for parsing links consists of four steps:

1. First, it iterates over all elements of the link. If the element is a locator or a resource, it creates an instance of the `INetworkNode` class. If the element is an arc, Brel stores the from and to attributes of the arc in an edge list.

2. For the second step, Brel can assume that all nodes have been created. So, it iterates over the edge list and adds adds the to-node as a child to the from-node.

3. Next, Brel iterates over all nodes and adds all nodes that do not have a parent to the root list of the network. The root list is then wrapped into a `INetwork` instance.

4. Finally, if the network has any implications on the report as a whole, Brel will apply these implications to the report. For example, if the label network adds a label to a concept, Brel will add the label to the concept in the report. The exact implications depend on the type of network.

Chapter 5 introduced the different types of networks. For each type of network, Brel provides both a node class and a network class, all of which inherit from the `INetwork` and `INetworkNode` classes respectively. Brel uses the factory pattern to create the correct network and node instances for a given link. Each type of network has its own factory. The factories are used by the algorithm 6.2.2 to create the correct network and node instances.

6.2.3 Resolving Locators

As mentioned in section 4.6, locators are used to point to report elements or facts. This section will explain how Brel resolves locators.
XBRL locators use XPointer[33] expressions to reference other XML elements, which are potentially defined in other XML documents. The XPointers XBRL uses are of the form `filename#id`, where `filename` is the name of the XML document and `id` is the id of the XML element. To resolve a locator, Brel first needs to find the XML element that the locator points to. Then, it needs to resolve the XML element into a report element or a fact.
Brel does this by remembering the id of each fact and report element that it has parsed. It builds a lookup table that maps ids to facts and report elements. Whenever Brel encounters a locator during the parsing process, it resolves the locator by looking up the id of the locator in the lookup table.

6.2.4 Parsing Resources

Resources are the other type of element that can be referenced by arcs. Unlike locators, resources are not used to point to other elements in the report. Instead, they directly contain the value of the element that they represent.
The current specification of XBRL 2.1 defines three built-in types of resources: label, reference and footnote. Custom resources are also possible.
Resources consist of three elements: a role, a label and a value.
The role is a URI that identifies the type of resource. For example, the role `http://www.xbrl.org/2003/role/terseLabel` identifies a label resource that contains a short label for a concept, whereas the role `http://www.xbrl.org/2003/role/footnote` identifies a footnote resource that contains a footnote for a concept.
The label acts as an identifier for the resource. The label is used by arcs to reference the resource. Do not confuse the label of a resource with a human readable label for a concept.
The value of a resource contains the actual resource. For labels, the value is the text of the label. For references, this is the dictionary that points to some exteral resource like an article in the SEC's Code of Federal Regulations.
Since resources have all the information that is needed to parse them, Brel does not need to resolve any references to parse them. Therefore, parsing them is very straightforward.

6.2.5 Implications of Networks

As mentioned in section 6.2.2, networks can have implications on the report as a whole. There are two implications that are common to all networks: Labels and LineItems promoting.

6.2.6 Add Labels to Report Elements

Labels are used to provide human-readable names for report elements. They are created using the label network `link:labelLink`. Labellinks are described in more detail in section 4.6.6. Brel parses report elements before it parses networks, since many networks contain locators that point to report elements. So, when Brel parses a label network, it already knows all report elements that are referenced by the locators in the network.
Once the label network has been parsed, Brel iterates over all labels in the network. It then adds the label to the report element that the locator of the label points to.

6.2.7 LineItems Promoting

Report elements are defined in the taxonomy and there six different types: concepts, abstracts, lineitems, members, dimensions and hypercubes. When looking at an XML element in the taxonomy, it is not always clear what type of report element it represents. Both abstracts and lineitems are represented by a structurally identical XML element. There are two approaches to tell them apart:

1. The first approach is to look at the QName of the element. Lineitems often have the substring "LineItems" in their QName. The problem with this approach is that this is more of a convention than a rule. There is no guarantee that the QName of a lineitem will contain the substring "LineItems".

2. The second approach is to look at how the element is used in networks. specifically, definition networks define the relationships between report elements. The arc role `hypercube-dimension` is used to define the relationship between

a hypercube report element and a dimension report element. Similarly, the arc role `all` is used to define the relationship between a hypercube report element and a lineitem report element.

Brel uses the second approach to tell lineitems and abstracts apart. Initially, Brel assumes that all report elements are abstracts. Then, when it parses a definition network, it looks at the arc roles of the arcs in the network. If the network contains an arc with the arc role `all` and the arc lies between a hypercube and an abstract, Brel assumes that the abstract is a lineitem.
This concludes the chapter on the implementation of networks. Together with previous sections of this chapter, it covered all parts of XBRL and how Brel parses them. However, section 6.1.1 made a key assumption about taxonomies that is not always true. Each taxonomy can import other taxonomies under a prefix-URI pair. The assumption was that all taxonomies agree on the prefix-URI pair of each taxonomy. This assumption is not always true. The next section will explain how Brel handles this situation.

6.3 QNames and namespace normalization

Both chapters 4 and 5 disclose that Brel leverages QNames to identify various elements across the XBRL report. QNames are a concept that is widely used in XML and XML-based languages, such as XBRL. Thus, for most QNames in Brel, the necessary information can be directly extracted from the corresponding XML elements in both the XBRL taxonomy and the XBRL filing. However, there is an important difference between QNames in XML and QNames in Brel - Namespace bindings.
Namespace bindings are the mappings between prefixes and namespace URIs. In XBRL, the URI often points to a taxonomy file. The prefix is used to refer to the namespace URI in the XBRL filing in a compact way. For example, the prefix `us-gaap` might be bound to the namespace URI `http://fasb.org/us-gaap/2023`.
In XML documents, namespace bindings can be defined on a per-element basis. Child elements inherit the namespace bindings of their parent elements, unless they define their own namespace bindings. This allows for the construction of complex namespace hierarchies, where each element can have its own namespace bindings. In Brel, however, namespace bindings are flat and defined on a global level.
The process of converting this hierarchical structure of namespace bindings into a flat structure is called **namespace normalization**. Namespace normalization not only flattens the namespace hierarchy, but also resolves collisions in namespace bindings.
The motivation for having a flat structure of namespace bindings is that it makes it easier for users to search concepts, types, etc. in an XBRL filing using QNames. Lets say a user wants to search for all facts that are associated with the concept `us-gaap:Assets`.
In XML, the prefix `us-gaap` might be bound to a different namespace URI in each element. So there are no guarantees that the prefix `us-gaap` is bound to just one namespace URI in the entire XBRL filing. A good XBRL parser will find all namespace bindings for the prefix `us-gaap` and search all of them for the concept `Assets`. However, this is not guaranteed and depends on the XBRL parser.
Furthermore, the filer of the XBRL report might choose to use the prefix `us-gaap1` instead of `us-gaap` in some sections to refer to the same namespace URI. So there might not be a concept called `us-gaap:Assets`, but a concept called `us-gaap1:Assets` instead.
The user could simply search both for `us-gaap` and `us-gaap1` to find the concept `Assets`. Obviously, this is extremely cumbersome and a suboptimal fix for the

problem. It also insufficiantly shields users from the complexity of XML and XML-based languages, such as XBRL, since they need to be aware of the structure of the underlying namespace hierarchy.
In Brel, the user can simply search for the QName `us-gaap:Assets` and will find all facts that are associated with this concept. Brel will automatically resolve the prefix `us-gaap` to the corresponding namespace URI. If the report uses the prefix `us-gaap1` instead of `us-gaap`, Brel will still be able to resolve the prefix to the correct namespace URI.

6.3.1 Namespace hierarchy notation

This section exclusively focuses on the implementation of QNames in Brel and namespace bindings in particular. Since XML documents tend to be verbose and contain a lot of information that is not relevant for namespace bindings, I will use a custom notation to represent namespace bindings in this section. I will refer to this notation as the *namespace hierarchy notation* and describe it using an example.
The namespace hierarchy notation works as follows:

- Each level of the namespace hierarchy is represented as a single line.
- Depending on the level of the namespace hierarchy, the element name has a different indentation.
- The name of the element is followed by a list of namespace bindings, separated by commas.
- Each namespace binding is represented as a key-value pair, where the key is the prefix of the namespace binding and the value is the namespace URI.
- If an element does not define any namespace bindings, the list of namespace bindings is omitted.
- XML attributes that are not namespace bindings are omitted in this notation.

Take the following XML snippet as an example:

Figure 6.1: Example of an XML snippet where namespace bindings are defined on a per-element basis

```
<element1
  xmlns:foo = "http://foo.com"
  color = "red"
>
    <element2 xmlns:bar = "http://bar.com"/>
        <element3 color = "blue">
    </element2>
</element1>
<element4
  xmlns:baz = "http://baz.com"
  xmlns:foo = "http://other-foo.com"
>
```

Using the namespace hierarchy notation, we can represent the same namespace hierarchy as follows:

Figure 6.2: Example of the same XML snippet in our custom notation

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

As we can see, the namespace hierarchy notation is much more compact than the XML snippet. The `color` attribute of `element1` and `element3` is omitted, since it is not a namespace binding.

6.3.2 Flattening namespace bindings

As mentioned in the previous section, namespace bindings in XML are defined on a per-element basis, which is arranged in a tree structure. In Brel, however, namespace bindings are defined on a global level, which is arranged in a flat structure. Thus, we need to flatten the namespace hierarchy of the XBRL taxonomy into a flat structure.

The process of flattening a tree structure into a flat structure is a common problem in computer science. One of the most common approaches to this problem is to use a depth-first search algorithm. This is also the approach that I use in Brel to flatten the namespace hierarchy of the XBRL taxonomy.

Remember that in XML, child elements inherit the namespace bindings of their parent elements. Thus, when flattening the namespace hierarchy, we need to make sure that all the namespace bindings of a parent are also present in its children, unless the children define their own namespace bindings.

To give an example of flattening, let us flatten the namespace hierarchy from the previous figure 6.2

Figure 6.3: Example of the same XML snippet in our custom notation, flattened

```
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"
```

As we can see, the namespace hierarchy has been flattened into a flat structure [2], meaning that all elements are on the same level. The order of the elements is determined by the depth-first search algorithm.

Each child element inherits the namespace bindings of its parent element. Thus, the `element2` and `element3` elements inherit the namespace bindings of the `element1` element.

To extract the namespace bindings of this flat structure, we can simply iterate over the elements and extract the namespace bindings of each element. For our example, this would result in the following list of namespace bindings:

[2]Technically, the namespace hierarchy is not completely flat, since the root element is still present. However, the root element does not contain any namespace bindings, so it does not affect the namespace hierarchy.

Figure 6.4: List of namespace bindings extracted from the flattened namespace hierarchy

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

6.3.3 Collisions in namespace bindings

An observant reader might have noticed that the list of namespace bindings from the previous section contains two bindings for the `foo` prefix. The first binding is defined as `foo = "http://foo.com"`, whereas the second binding is defined as `foo = "http://other-foo.com"`.
This is called a collision. Brel prohibits most collisions in namespace bindings, but allows some of them. The following section describes the different types of collisions and how they are handled in Brel.

6.3.4 Types of namespace collisions

There are three types of namespace collisions that can occur in Brel:

- **Version collision**: Two namespace bindings have the same prefix and the same namespace URI, but different versions of it. The version is defined as the numbers and dashes in the namespace URI that indicate if the URI is more recent than another URI.

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

- **Prefix collision**: Two namespace bindings have the same prefix, but different *unversioned* namespace URIs. A unversioned namespace URI is a namespace URI with all version information removed.

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

- **Namespace URI collision**: Two namespace bindings have the same *unversioned* namespace URI, but different prefixes.

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

Version collision

Version collisions occur when two namespace bindings have the same prefix and the same namespace URI, but different versions of it.
Version collisions are allowed in brel, but they do raise an interesting question: Lets say the creates a QName `foo:bar` to search for a concept in the taxonomy. Also assume that the XBRL filing contains the following namespace bindings:

Figure 6.5: Example of a version collision

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

Which namespace URI should be used for the QName `foo:bar`?
The mechanism that Brel implements is straightforward: Use the newest version.
First, Brel will remove all digits, dashes and dots from the URI versions. If the two URI versions are equal after this step, then they are considered the same URI with different versions. In our example, both URI versions transform into `http://foo.com/`.
Second, for each URI version, Brel will extract all numbers and compute their sum. The URI version with the higher sum is considered the newer version. For our example, the sum of the first URI version is 2024, whereas the sum of the second URI version is 2025. Thus, the second URI version is considered the newer version. So if the user searches for the QName `foo:bar`, the URI version `http://foo.com/01-01-2023` will be used.
Even though this mechanism is straightforward, it does have some drawbacks. Namely, theoretically it is easy to trick the mechanism into using an older version of a namespace URI. For example, the URI version `http://foo.com/31-12-2021` would be considered newer than `http://foo.com/01-01-2023`.
However, this is not a problem in practice since version collisions tend to be rare. On top of that, most taxonomies are released on a yearly basis and their URI just contains the year of the release. If the URI only contains the year, then the mechanism works as expected.
Furthermore, the mechanism used for searching and comparing QNames in Brel only uses the prefix and the local name of the QName. Therefore, even if the mechanism would be tricked into using an older version of a namespace URI, it would not affect the search results.

Prefix collision

A prefix collision occurs when two namespace bindings have the same prefix, but different *unversioned* namespace URIs. The following figure shows an example of a prefix collision:

Figure 6.6: Example of a prefix collision

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

Prefix collisions are not allowed in Brel. Brel will rename one of the prefixes to avoid the collision. In the case of our example above, Brel will `element2`'s binding to `foo1 = "http://other-foo.com"` and will replace all appropriate QNames with the new prefix.

Figure 6.7: Example of a resolved prefix collision

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

If the user searches for a QName `foo:bar`, Brel will both search for `foo:bar` and `foo1:bar`.

Namespace URI collision

A namespace URI collision occurs when two namespace bindings have the same *unversioned* namespace URI, but different prefixes. An example of a namespace URI collision is shown in the following figure:

Figure 6.8: Example of a namespace URI collision

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

Namespace URI collisions are not allowed in Brel. Brel will pick one of the two prefixes as the preferred prefix and will rename the other prefix to avoid the collision. In general, Brel will pick the shorter prefix as the preferred prefix. If both have the same length, Brel will pick the prefix that comes first alphabetically.

In the case of our example, `bar` will be picked as the preferred prefix. Brel will rename the prefix `foo` along with all occurences to `bar`.

Figure 6.9: Example of a resolved namespace URI collision

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

There are some prefixes that are considered special and will always be picked as the preferred prefix, regardless of their length or alphabetical order. These special prefixes do not even have to be defined in the XBRL taxonomy. If there is a namespace binding that points to the same namespace URI as one of the special prefixes, the special prefix will be picked as the preferred prefix.

The following prefixes are considered special:

- `xml = "http://www.w3.org/XML/<year>/namespace"`
- `xlink = "http://www.w3.org/<year>/xlink"`
- `xs = "http://www.w3.org/<year>/XMLSchema"`
- `xsi = "http://www.w3.org/<year>/XMLSchema-instance"`
- `xbrli = "http://www.xbrl.org/<year>/instance"`
- `xbrldt = "http://xbrl.org/<year>/xbrldt"`
- `link = "http://www.xbrl.org/<year>/linkbase"`
- `xl = "http://www.xbrl.org/<year>/XLink"`
- `iso4217 = "http://www.xbrl.org/<year>/iso4217"`
- `utr = "http://www.xbrl.org/<year>/utr"`
- `nonnum = "http://www.xbrl.org/dtr/type/non-numeric"`
- `num = "http://www.xbrl.org/dtr/type/numeric"`
- `enum = "http://www.xbrl.org/year/extensible-enumerations"`
- `enum2 = "http://www.xbrl.org/PR/<date>/extensible-enumerations-2.0"`
- `formula = "http://www.xbrl.org/<year>/formula"`
- `gen = "http://www.xbrl.org/<year>/generic"`
- `table = "http://www.xbrl.org/<year>/table"`
- `cf = "http://www.xbrl.org/<year>/filter/concept"`
- `df = "http://www.xbrl.org/<year>/filter/dimension"`
- `ef = "http://www.xbrl.org/<year>/filter/entity"`
- `pf = "http://www.xbrl.org/<year>/filter/period"`
- `uf = "http://www.xbrl.org/<year>/filter/unit"`
- `ix = "http://www.xbrl.org/<year>/inlineXBRL"`
- `ixt = "http://www.xbrl.org/inlineXBRL/transformation/<date>"`
- `entities = "http://xbrl.org/entities"`

If, for example, the XBRL filing contains a namespace binding `foo = "http://www.w3.org/2001/XMLSchema` then the prefix `xsi` will be picked as the preferred prefix and all occurences of `foo` will be renamed to `xsi`.

The special prefixes and the corresponding namespace URIs can be configured in in the `nsconfig.json` file.

With this knowledge of QNames and namespace normalization, we can now move on to the next section which explains how Facts are implemented in Brel. Facts use QNames for a few of their characteristics, most notably their concept characteristic.

6.4 DTS Cache

The second key assumption that section 6.1 makes is that every file of both the taxonomy set and the XBRL report is stored locally on the user's computer. In reality, this assumption does not hold true. In the most common case, only the XBRL report is stored locally on the user's computer. The taxonomy file within the XBRL report points to other taxonomy files, which are not stored locally. As mentioned in section 4.3, taxonomies may refer to other taxonomies. Brel needs to resolve and download the transitive closure of all taxonomy references in order to parse the XBRL report. Brel refers to this process as DTS caching.

6.4.1 Discovery

The letter D in DTS caching stands for "discoverable". It suggests that Brel first needs to discover all taxonomy files that the XBRL report refers to. Brel's approach is to parse the taxonomy file within the XBRL report and extract all taxonomy references from it. A taxonomy file might refer to other taxonomy files different ways.

- **schemaRef** - The most common way is to refer to other taxonomy files using the `schemaRef` element. The `schemaRef` XML element contains a `href` attribute, which contains a URL that points to another taxonomy file.
- **linkbaseRef** - A taxonomy file may also refer to other taxonomy files using the `linkbaseRef` element. Like the `schemaRef` element, the `linkbaseRef` element contains a `href` attribute.
- **import** - A taxonomy file may also refer to other taxonomy files using the `import` element. The `import` element contains a `schemaLocation` attribute containing an URI that points to another taxonomy file.
- **include** - A taxonomy file may also refer to other taxonomy files using the `include` element. Similar to the `import` element, the `include` element contains a `schemaLocation` attribute containing an URI that points to another taxonomy file.

Whenever Brel parses a taxonomy file, it extracts all taxonomy references from it and adds them to a working set of taxonomy references. Once Brel has parsed the current taxonomy file, it fetches the first taxonomy reference from the working set. Brel then repeats the process of parsing the taxonomy file and extracting all taxonomy references from it. The reader can think of this process as a breadth-first search of the taxonomy reference graph. If Brel has already parsed a taxonomy file, then it does not need to parse it again.

6.4.2 Downloading taxonomies

Given a URI, downloading the taxonomy file that the URI points to is trivial for most URIs. However, some URIs are relative URIs, which means that the URI indicates the location of the other taxonomy file relative to the current taxonomy file[3]. Brel infers the domain name of relative URIs by remembering the domain name of the taxonomy file that the relative URI was referenced from. It concatenates the domain name of the taxonomy file with the relative URI to form an absolute URI.
Relative URIs introduce another problem - File names for taxonomy files. Since Brel downloads taxonomy files from the internet, it needs to store them locally on the user's computer. Brel chooses to store taxonomy files in a directory called

`dts_cache` without using any subdirectories. This means that Brel needs to ensure that all taxonomy files have unique file names. The natural choice for file names is the local name of the taxonomy file's URI, which is not necessarily unique. The next option is to use the URI itself as the file name. Since URIs may be relative, there might be multiple URIs that point to the same taxonomy file. Additionally, tend to be very long and contain characters that are not allowed in file names. The approach that Brel chooses is to generate a unique file name based on the absolute URI of the taxonomy file. Brel strips the URI of all illegal characters to form a valid file name.
Using both the discovery and download mechanism, Brel is able to download all taxonomy files that the XBRL report refers to. Brel stores the taxonomy files in the `dts_cache` directory, which resolves the second key assumption that section 6.1 makes.

6.5 Answering research question 3

With the implementation of Brel complete for the XBRL XML syntax, we can now answer research question 2.2:

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

The answer to this question is that the Brel API is designed to be almost format agnostic. The first half of the Brel API is based on the OIM, which in itself is a logical data model. This makes it inherently format agnostic.
The second half of the Brel API is based on the XBRL XML syntax. However, the only methods that still expose the XBRL XML syntax are the methods `get_link_role` and `get_link_name` in the `INetwork` interface as well as the `get_arc_role` and `get_arc_name` methods in the `INetworkNode` interface. These methods return the XML attributes of the same name. However, the main purpose of these methods is debugging.
The only real remnant of the XBRL XML syntax in the Brel API is the `QName` class. However, Brel merely chooses to use the same data structure as QNames in XML, which is a combination of a prefix, a namespace URI and a local name. The Brel API does not require QNames to be in the XML format. In fact, both the JSON[19] and the CSV[18] specifications of XBRL use QNames in the same format as the XML specification. Therefore, the `QName` class is also format agnostic.
Since the Brel API is format agnostic, the only section of Brel that relies on the XBRL XML syntax is the parser. Brel implements the parser as a separate module, which is called `brel.parser.XML`. The parser module is the only module that needs to be changed in order to support other formats. The rest of Brel can remain unchanged.
This concludes the implementation chapter. This chapter covered the implementation of Brel. It used both chapter 4 and chapter 5 and explained how Brel converts XBRL reports in the XBRL XML syntax python objects that implement the Brel API. It also explained how Brel answers research question 2.2. The next chapter will evaluate Brel based both on correctness and performance.

Chapter 7

Results

After covering XBRL, the Brel API and its implementation, we can now evaluate Brel against the requirements that we defined in section 2.2. We will evaluate Brel based on correctness first and performance second. This chapter will also cover Brel's robustness and usability in a qualitative manner.
Even though Brel's performance is not part of the requirements set by this thesis, it still serves as an important metric. It will enable future versions of Brel to compare their performance against this initial version of Brel.
For testing Brels correctness, we will use XBRL conformance suites. Additionally, we will look at hand-picked XBRL reports and compare the structures that Brel extracts from them against the structures that the XBRL specification prescribes.
The usability of Brel will be evaluated by using it to implement a simple CLI XBRL report viewer. It will cover every feature of the Brel API and serve as a proof of concept for the Brel API.
Robustness is a qualitative metric that is hard to measure. We will evaluate Brel's robustness by loading the 10K and 10Q reports of the 30 largest US companies by market capitalization at the time of writing. This will give us a good idea of how robust Brel is in practice. The list of companies that we will use is

7.1 Correctness

TODO: Write this chapter

7.2 Performance

TODO: Write this chapter

7.3 Usability

TODO: Write this chapter

7.4 Robustness

Brel's robustness against real-world XBRL reports is paramount to its usefulness. Therefore, we will evaluate Brel's robustness by loading the 10K and 10Q reports of the 30 largest US companies by market capitalization at the time of writing [1].

[1] Time of writing: 29 January 2024

The list of companies that we will use is shown in table 7.1. This list was generated by using the list of the 100 largest US companies by market capitalization[5] and removing all companies that do not have any 10K or 10Q reports. We limited the list to US companies since the SEC mandates that all US companies must file their financial reports in XBRL[30].

Rank	**Name**	**Symbol**	**Market capitalization (USD)**
1	Microsoft	MSFT	3009100644352
2	Apple	AAPL	2975178948608
3	Alphabet (Google)	GOOG	1913839550464
4	Amazon	AMZN	1644346081280
5	NVIDIA	NVDA	1507465756672
6	Meta Platforms (Facebook)	META	1012884701184
7	Berkshire Hathaway	BRK-B	837177376768
8	Eli Lilly	LLY	606844485632
9	Tesla	TSLA	582537052160
10	Broadcom	AVGO	564053737472
11	Visa	V	542508843008
12	JPMorgan Chase	JPM	495597846528
13	UnitedHealth	UNH	465422254080
14	Walmart	WMT	442252623872
15	Exxon Mobil	XOM	411667300352
16	Mastercard	MA	411242921984
17	Johnson & Johnson	JNJ	383961169920
18	Procter & Gamble	PG	367400517632
19	Home Depot	HD	353616592896
20	Oracle	ORCL	316125544448
21	Merck	MRK	306160304128
22	Costco	COST	304787881984
23	AbbVie	ABBV	290254749696
24	AMD	AMD	286347395072
25	Chevron	CVX	281539051520
26	Adobe	ADBE	277496365056
27	Salesforce	CRM	270981922816
28	Bank of America	BAC	263945224192
29	Coca-Cola	KO	256680837120
30	Netflix	NFLX	249661423616

Table 7.1: The 30 largest US companies by market capitalization at the time of writing

Chapter 8

Conclusion

TODO: Write this chapter

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*. price waterhouse coopers, 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] Coca-Cola Company. 10-q (quarterly report) for quarter ending june 27, 2023. https://www.sec.gov/Archives/edgar/data/21344/000002134423000048/0000021344-23-000048-index.html, 2023.

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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

[24] Robin Schmidiger. Brel - business reporting extensible library. https://github.com/BrelLibrary/brel, 2021.

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

[26] 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.

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

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

[29] 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.

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

[31] 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.

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

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

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

[35] World Wide Web Consortium (W3C). Namespaces in xml 1.0 (third edition). https://www.w3.org/TR/2009/REC-xml-names-20091208/#dt-expname, 2009.