Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 478

Systems Group, Department of Computer Science, ETH Zurich

Brel - A python library for XBRL

by

Robin Schmidiger

Supervised by

Prof. Gustavo Alonso, Dr. Ghislain Fourny

September 2023 – March 2024

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

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

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

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

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

Place, date	**Signature(s)**

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

Contents

Chapter 1

Preface

This thesis was written as part of Robin Schmidiger's master's degree in computer science at ETH Zurich. It was supervised by Prof. Gustavo Alonso and Dr. Ghislain Fourny.
Both the thesis and the source code of the library are available on GitHub[27]
The contents of this thesis are based heavily on both the XBRL standard[11] created by Charles Hoffman and "The XBRL Book"[10] 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[23]. 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.[19] 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)[34] and the European Banking Authority (EBA)[2] are progressively mandating the submission of reports in XBRL format. In the corporate domain, entities like JP Morgan Chase, Microsoft, and Hitachi are leveraging XBRL to streamline their financial reporting mechanisms.[4]

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

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

However, the current state of XBRL does not entirely meet this requirement. This situation has led to the emergence of a varied range of XBRL tools, most of which are proprietary and not freely available. Although there are some open-source alternatives, they are relatively few. One significant exception is Arelle, a well-known tool in the XBRL field[24]. Other open-source python libraries, like `python-xbrl`[5], `pysec`[9], and `py-xbrl`[28], are often limited in scope and do not support XBRL's latest key feature.

In 2021, XBRL International published a new specification termed Open Information Model (OIM)[20]. The OIM is a logical data model for XBRL reports that is independent from the XBRL syntax. One objective of the OIM is to make XBRL more approachable for both developers and non-technical users by iterating on XBRL's design. The OIM is not yet finalized and does not cover all aspects of XBRL.

2.2 Goals of this thesis

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

- 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, Brel will have to make some compromises.
The first limitation of this thesis is that the library will only support XBRL reports in XML format. Secondly, the library will only support reading XBRL reports, not creating or modifying them. Third, Brel will not semantically validate XBRL reports.
While the initial two limitations are straightforward, the third limitation necessitates further clarification. XBRL reports can be interpreted in terms of syntax and semantics, similar to the source code of a program. An XBRL report that is syntactically accurate adheres to the XBRL specifications, yet it may not always be logically coherent [1]. A semantically correct report is both syntactically correct and logically coherent. This thesis will not address the semantics of XBRL reports, as they are not explicitly detailed in the XBRL specification but are outlined in supplementary documents.
Nonetheless, semantic validation of XBRL reports can be achieved through the use of the Brel API.

2.4 Structure of this thesis

Grasping the underlying XBRL standard is essential to understand how Brel provides a pythonic API for XBRL reports. Hence, Chapter 4 will offer a concise introduction to XBRL. The chapter will present the fundamental concepts of XBRL within the OIM framework, followed by an exploration of the non-OIM aspects of XBRL. This chapter will concentrate on the concepts pertinent to this thesis, rather than delving into the technical specifics of the XBRL standard.
Subsequently, Chapter 5 will introduce the API of Brel. This chapter, which forms part of the thesis results, is positioned prior to the implementation of the API. The reason for this deviation from the conventional structure is that it is more logical to introduce the API before its implementation. The API chapter will answer research question 2.2 and 2.2.

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

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

Chapter 3

Related work

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

3.1 XBRL Specification

As indicated in Section 2, this thesis builds upon the XBRL standard originally authored by Charles Hoffman[11]. The XBRL standard comprises numerous components, and this thesis specifically focuses on the following components: the Open Information Model (OIM)[20], the XBRL 2.1 specification[13], the extension for dimensional reporting[14], and the specification for generic links[15]. Chapter 4 will delve into these components in detail.

3.2 The XBRL Book

Understanding the XBRL specification requires a good grasp of both XML and XBRL. To help newcomers, Dr. Ghislain Fourny has authored "The XBRL Book" [10], which serves as a comprehensive guide to XBRL. This book covers all important aspects of the XBRL standard, including the relatively recent OIM, making it an invaluable resource for those looking to learn about XBRL.

3.3 Arelle

Arelle[24] stands as an open-source XBRL platform. As of the current writing, Arelle holds the distinction of being the most comprehensive open-source platform in its category. It provides support for all features found in the XBRL 2.1 specification and the OIM. Similar to Brel, Arelle is implemented in Python and is available as open-source software. However, it's important to note that Arelle is a complete XBRL platform, in contrast to Brel, which is primarily a Python library. The reader can think of Arelle as the "Excel for XBRL."

3.4 Xule

Xule[36] serves as a rule language tailored for XBRL. This declarative language empowers users to craft rules for the validation of XBRL reports. The Arelle project employs Xule to validate reports. Conceptually, Xule can be likened to a domain-specific language designed specifically for XBRL validation rules. While Xule is not directly incorporated into this thesis, it is worth noting that Brel has the potential to offer support for Xule in the future.

3.5 EDGAR

The SEC, known as the U.S. Securities and Exchange Commission, operates a system referred to as EDGAR (Electronic Data Gathering, Analysis, and Retrieval) [32]. This EDGAR system serves as a public repository for XBRL reports submitted to the SEC[1].

3.6 ESEF filings

The European counterpart to the SEC is ESMA, the European Securities and Markets Authority, which operates a database containing ESEF filings [29]. ESEF stands for European Single Electronic Format, a standardized format for XBRL reports within the European Union.
Much like EDGAR, the ESEF database is accessible to the public[2]. An intriguing aspect of this database is its hosting by XBRL International, the organization responsible for maintaining the XBRL standard.

3.7 SEC - Interactive Data Public Test Suite

In order to verify the compliance of XBRL processors utilized by companies for generating XBRL reports, the SEC established the Interactive Data Public Test Suite [31]. This comprehensive test suite comprises a vast assortment of XBRL reports designed to assess the performance of XBRL processors. It is noteworthy that the SEC offers this test suite at no cost, and it can be accessed on their official website.
Brel employs the aforementioned test suite to evaluate its XBRL processor. Currently, Brel does not successfully pass all the tests contained within the suite, primarily due to its lack of support for certain features within the XBRL standard.

3.8 ESMA Conformance Suite

ESMA, the European Securities and Markets Authority, administers a test suite designed for XBRL processors, as well [30]. This test suite shares similarities with the SEC's Interactive Data Public Test Suite, albeit being under the administration of a different regulatory authority. One notable divergence is that the ESMA Conformance Suite is tailored to facilitate automated testing of xHTML reports.
As of the present, Brel does not possess the capability to support xHTML reports, rendering the ESMA Conformance Suite irrelevant to the scope of this thesis. Nonetheless, it is important to acknowledge that there are intentions for Brel to incorporate support for xHTML reports in the future.

[1]`https://www.sec.gov/edgar/search/`
[2]`https://filings.xbrl.org/`

Chapter 4

XBRL

4.1 Overview

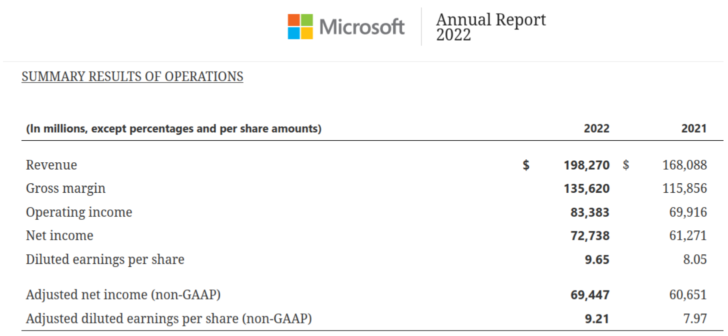
The content of this thesis is largely based on the XBRL standard[11] created by Charles Hoffman and Dr. Ghislain Fourny's interpretation of it in *the XBRL Book*[10]. Since the thesis builds on the foundation laid by the two, it is important to understand the ground work that they have done. This chapter will give a brief introduction to XBRL. The terms "report" and "filing" are used interchangeably in this thesis.
In essence, XBRL is a standardized format for representing reports. After all, XBRL stands for **eXtensible Business Reporting Language**.[11]
As Ghislain Fourny has put it in *the XBRL Book* [10]:

> 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
[8]

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. It is different from the XBRL instance document, which contains the actual facts of a report. 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).

[4]https://xbrl.us/us-gaap/

- **ifrs** [5] - Contains concepts for International Financial Reporting Standards (IFRS).
- **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.

[5] https://www.ifrs.org/
[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.

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.

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[18][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)`[35].

- `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`[18].
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:

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

```

Figure 4.5: Example of a presentationArc in XML syntax

The XML snippet in figure 4.5 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[20]. 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" [10] 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 [16]. 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`.[10]
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.

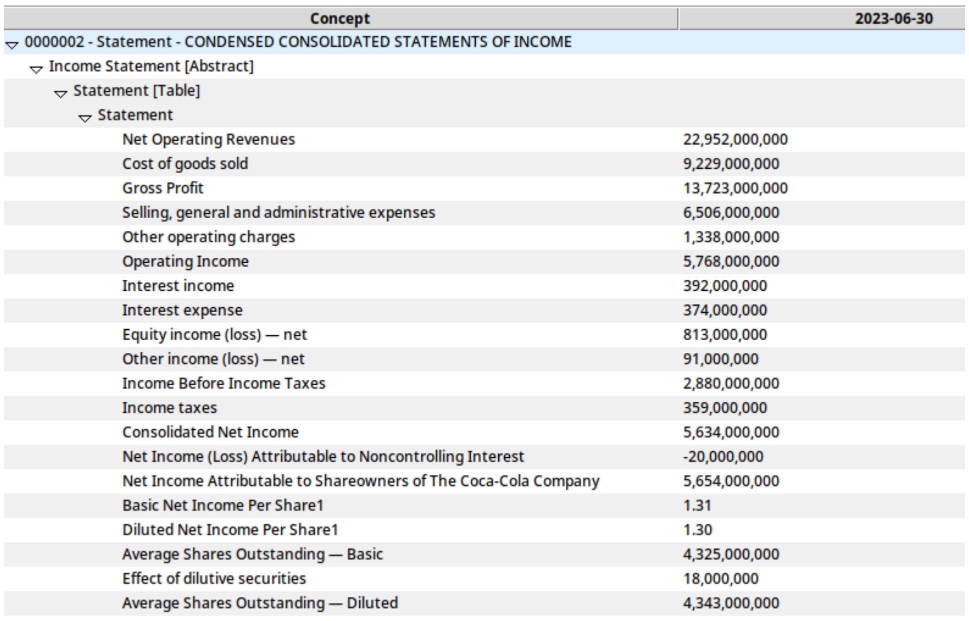
4.6.5 definitionLink

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[7] of Q2 2019 as displayed in Arelle[24].



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:

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


```

Figure 4.12: Example of a label in XML syntax

Note that we have omitted both the connecting arc and the locator in this example, as they 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[17].

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)` [33] [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.

[11] CFR stands for Code of Federal Regulations.

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.

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.[14].

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[11] created by Charles Hoffman and Dr. Ghislain Fourny’s interpretation of it in *the XBRL Book*[10].
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[26]. 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.

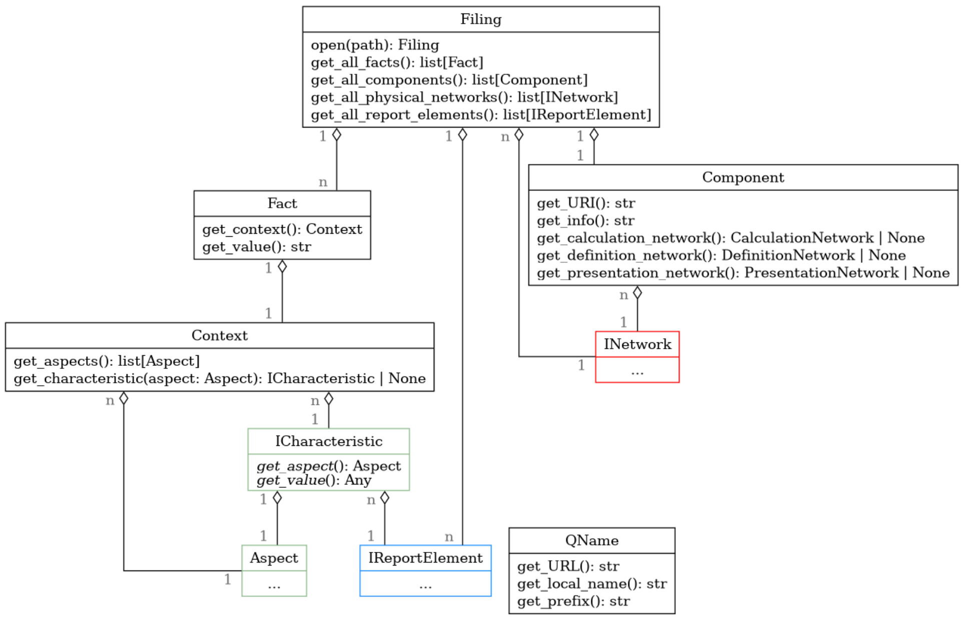

Filing
open(path): Filing
get_all_facts(): list[Fact]
get_all_components(): list[Component]
get_all_physical_networks(): list[INetwork]
get_all_report_elements(): list[IReportElement]
Fact
get_context(): Context
get_value(): str
Component
get_URI(): str
get_info(): str
get_calculation_network(): CalculationNetwork | 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)[20]. 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.

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`.

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

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.

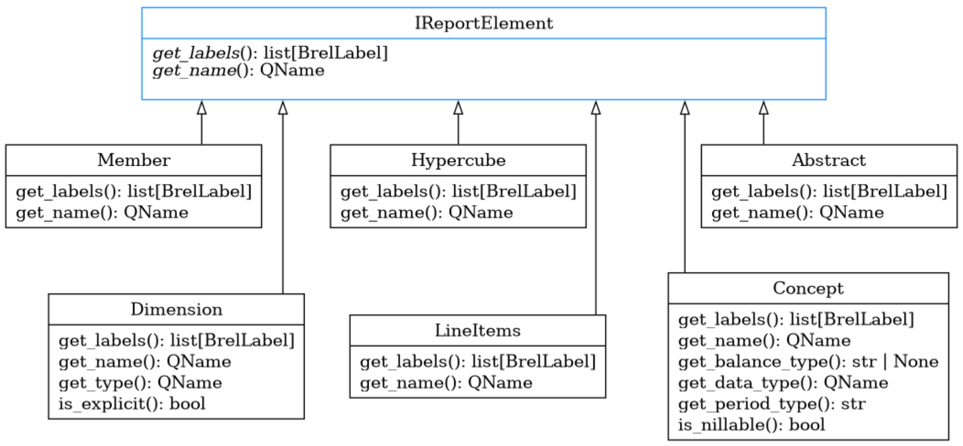


Figure 5.2: UML diagram of the report element classes in Brel

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 4.6.6. 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.

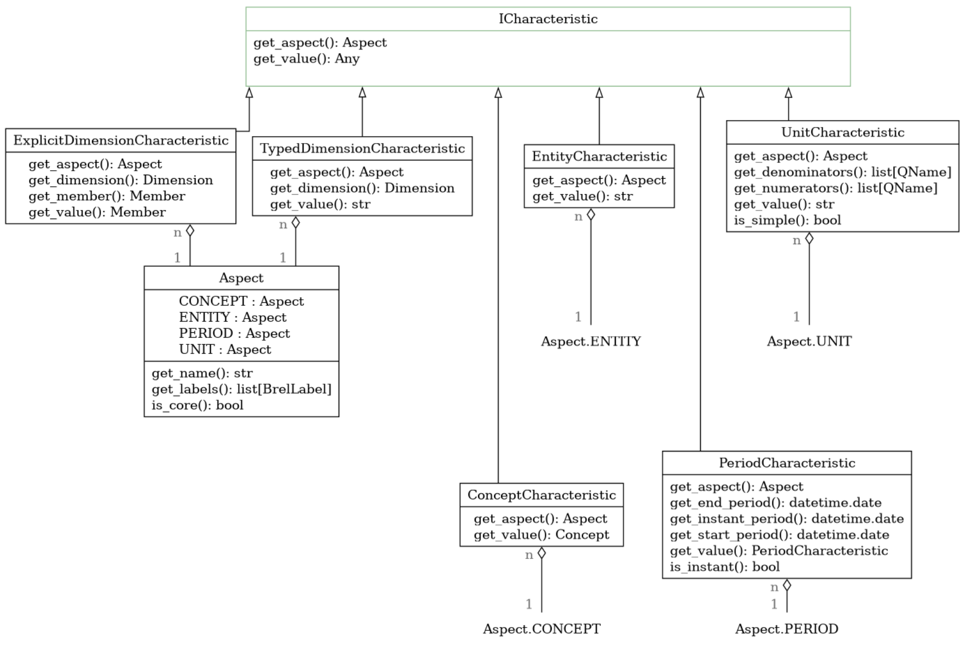


Figure 5.3: The interplay between aspects and characteristics

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] [38]
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,

[5]The expended name format of a QName is `namespace_prefix:local_name`.[39]

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.

5.3.6 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

[6] Schemes tend to be URLs.
[7] The notation is similar to the Clark notation for QNames.[38]

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 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.

[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`.

5.4 Answering Research Question 1

The Open Information Model (OIM) is a conceptual model for XBRL.[20] 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.

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

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.

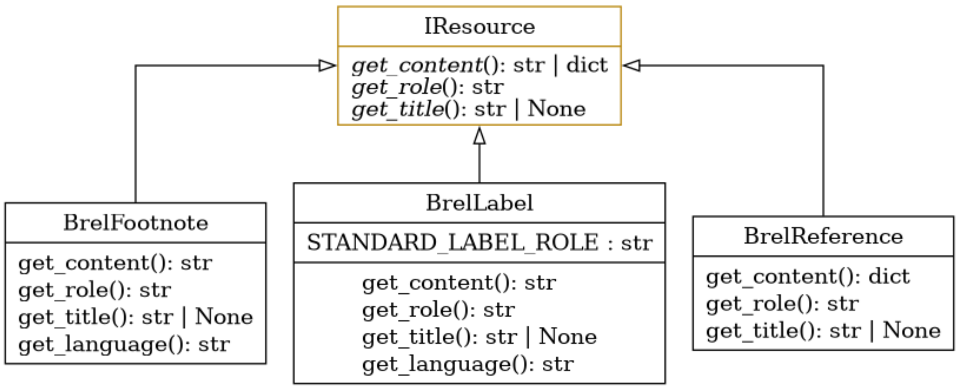


Figure 5.5: UML diagram of the resource classes in Brel

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.

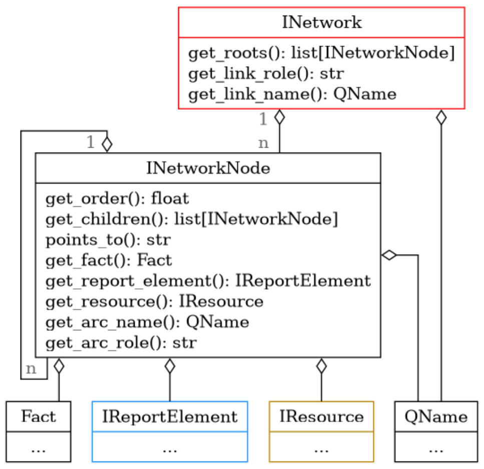
References

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 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

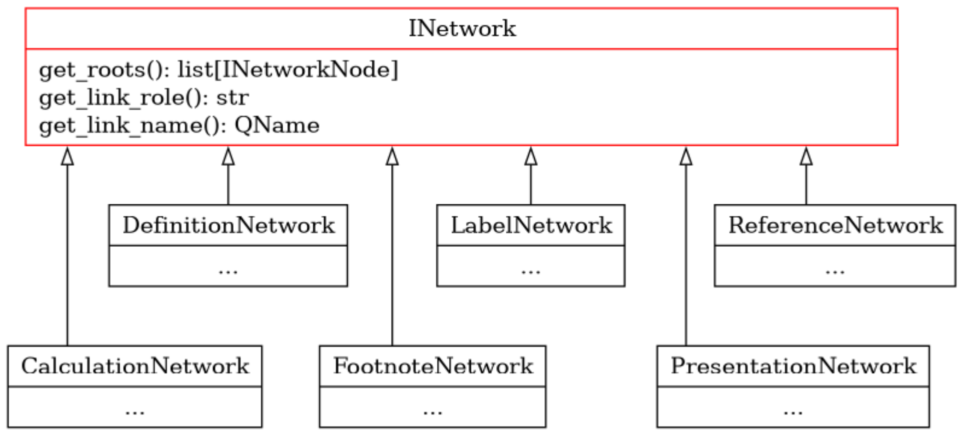
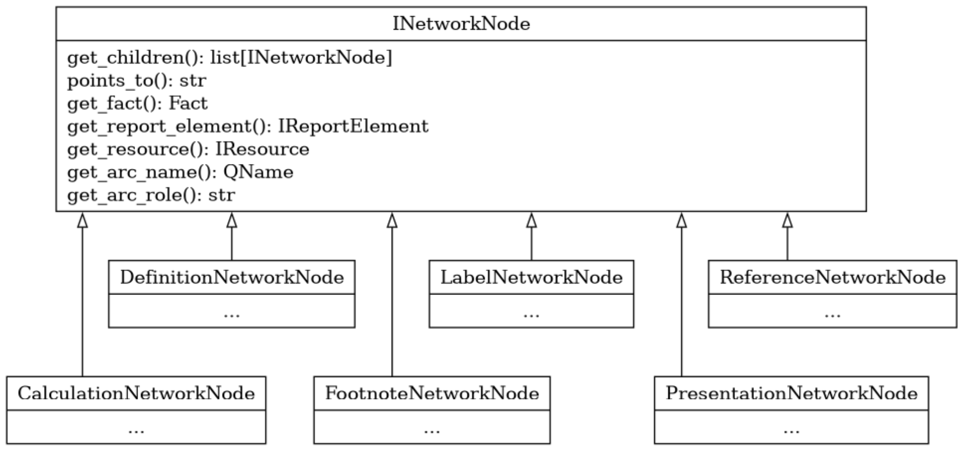


Figure 5.8: UML diagram of the node classes



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. This section aims to answer research question 2.2.

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

Research question 2.2 is twofold. First, the question asks how the non-OIM parts of XBRL can be converted into a python API. Secondly, it seeks to understand how this API can align with the OIM.

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

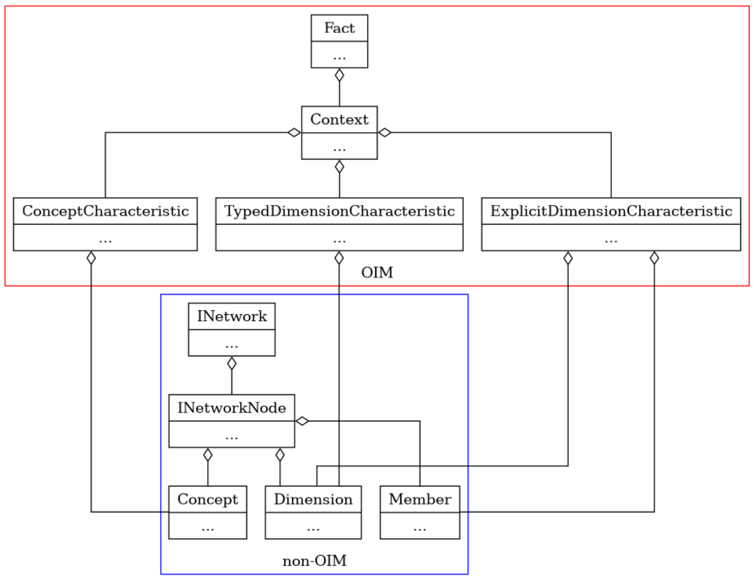


Figure 5.9: The Interaction Between Characteristics and Report Elements

In this setup, while the OIM utilizes characteristics to describe facts, networks employ the report elements within these characteristics. For instance, the `Concept` class could have implemented both the `IReportElement` and `ICharacteristic` interfaces. However, given that concepts in facts and networks serve different purposes, they should not be interchangeable. Consequently, Brel employs different classes for the two distinct use cases.
This marks the end of the chapter on the Brel API. It addressed both the OIM and non-OIM elements of XBRL. Additionally, it detailed how these components are integrated into a singular API, thereby addressing research questions 2.2 and 2.2. Following this comprehensive coverage of the API and the foundational XBRL standard, the subsequent chapter will focus on the implementation of the Brel API.

Chapter 6

Implementation

The development of the API aligns closely with the API's design presented in Chapter 5. Brel is created using the Python programming language. Python, a high-level language, is among the most favored programming languages globally. It also enjoys popularity beyond the realms of computer science and software engineering. The 2020 Stack Overflow Developer Survey ranks Python as the fourth most favored programming language among all participants, not limited to professional developers [25].
Readers can interpret Brel's implementation as converting XBRL reports into Python objects. The majority of these conversions from design to implementation are direct and will be briefly discussed in the initial section of this chapter.
Brel's implementation diverges from the standard XBRL in three main aspects: DTS caching, namespace normalization, and networks. Each aspect is detailed in its own section within this chapter.
This chapter is exclusively concerned with XBRL reports in the XML format.

6.1 General Implementation

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

6.1.1 Parsing Report Elements

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

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

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

6.1.2 Parsing Facts

Brel processes facts immediately following the parsing of report elements. Facts are analyzed prior to networks because footnote networks may reference facts.
Facts are solely defined in the instance document of the XBRL report. This document is an XML file containing a straightforward list of facts, syntactic contexts,

and units, represented as XML elements. It might also include a list of footnotes, which are detailed in section 6.2.
Fact XML elements hold the fact's value and references to both the syntactic context and unit. The XML element's tag represents the QName of the concept associated with the fact.
Syntactic context XML elements outline a part of a fact's characteristics. They differ from `Contexts` as defined in the Brel API. A `Context` in Brel encompasses all characteristics of a fact, while a syntactic context includes only the period, entity, and dimensions of a fact. During parsing, Brel initially uses syntactic contexts to create `Context` instances. Subsequently, it supplements the `Context` with the remaining characteristics - the concept and the unit.
Unit XML elements, as their name implies, define the unit of a fact.
The rationale for XBRL segregating facts, syntactic contexts, and units into distinct XML elements is to minimize redundancy. Multiple facts can share the same syntactic context and unit.
Brel parses all facts by identifying all fact XML elements and resolving their links to syntactic contexts and units. It reutilizes units, entities, and dimensions across various facts.

6.1.3 Parsing Components

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

6.2 Implementation of Networks

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

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

Technically, a network in Brel is a collection of root nodes. These root nodes are linked to their respective children, who then connect to their own children, and so forth. Thus, to navigate through a network, knowledge of its root nodes is sufficient. The `INetwork` class offers a method to access all the network's root nodes, and the `INetworkNode` class provides a method to retrieve the children of a node, enabling the traversal of the network.
An important aspect of Brel's network implementation is that networks cannot be devoid of nodes. They must contain at least one node.

6.2.1 Transforming Links into Networks

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

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

2. In the second step, with all nodes already established, Brel sifts through the edge list, appending the to-node as a child of the from-node.

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

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

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

6.2.2 Parsing Locators

As indicated in section 4.6, locators serve to reference report elements or facts. This section details the method Brel uses to interpret locators.
XBRL locators utilize XPointer[37] expressions for referencing other XML elements, potentially from different XML documents. These XPointers in XBRL take the form `filename#id`, where `filename` denotes the XML document's URI and `id` is the id of the XML element. To interpret a locator, Brel first identifies the targeted XML element. Subsequently, it translates this XML element into a report element or a fact.
Brel accomplishes this by tracking the id of each fact and report element it parses. It constructs a lookup table mapping ids to their corresponding facts and report elements. Whenever Brel encounters a locator during parsing, it interprets the locator by consulting the lookup table with the locator's id.

6.2.3 Parsing Resources

Resources, the alternate type of element referable by arcs, do not point to other elements within the report. Instead, they directly encapsulate the value of the element they signify.
The current XBRL 2.1 specification outlines three inherent resource types: label, reference, and footnote, though custom resources are feasible.
Resources comprise three components: a role, a label, and a value.
The role is a URI defining the resource's type. For instance, the role `http://www.xbrl.org/2003/role/terse` denotes a label resource offering a concise label for a concept, while `http://www.xbrl.org/2003/role/footn` indicates a footnote resource associated with a concept.
The label functions as an identifier for the resource, utilized by arcs to reference the resource. This label should not be mistaken for a concept's human-readable label.
The resource's value embodies the actual resource content. For labels, this means the label text itself. For references, it is typically a dictionary pointing to an external resource, like an article in the SEC's Code of Federal Regulations.
Given that resources contain all necessary information for parsing, Brel does not need to resolve external references to analyze them. Hence, their parsing is straightforward.

6.2.4 Consequences of Networks

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

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

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

Brel adopts the second strategy to differentiate line items from abstracts. Initially, it treats all report elements as abstracts. Then, during the parsing of a definition

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

6.3 Namespace Normalization

Both chapters 4 and 5 reveal that Brel utilizes QNames to identify various elements within the XBRL report. QNames are a fundamental concept in XML and XML-based languages, like XBRL. As such, for most QNames in Brel, the necessary information is directly retrievable from the corresponding XML elements in both the XBRL taxonomy and the XBRL filing. However, a key distinction exists between QNames in XML and those in Brel, particularly in terms of Namespace bindings.
Namespace bindings represent the associations between prefixes and namespace URIs. In XBRL, a URI typically links to a taxonomy file. The prefix is employed to succinctly reference the namespace URI within the XBRL filing. For instance, the prefix `us-gaap` might be bound to the namespace URI `http://fasb.org/us-gaap/2023`.
In XML documents, these namespace bindings can be specified for individual elements. Child elements inherit their parent elements' namespace bindings, except when they establish their own. This flexibility allows the creation of intricate namespace hierarchies, where each element can possess unique namespace bindings. Conversely, Brel's approach to namespace bindings is more simplified, maintaining a flat and globally defined structure.
This section is devoted to discussing the implementation of QNames in Brel, focusing particularly on namespace bindings. Given that XML documents include information irrelevant to namespace bindings, the figures in this section omit any extraneous information that is not relevant to namespace bindings and their hierarchical structure. An example figure is provided below.

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

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

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

6.3.1 Flattening Namespace Bindings

Flattening a tree structure into a flat one is a common challenge in computer science. A popular solution is the use of a depth-first search algorithm, which is the method

employed in Brel to flatten the XBRL taxonomy's namespace hierarchy.
It is important to remember that in XML, child elements inherit their parent elements' namespace bindings. Consequently, when flattening the namespace hierarchy, it is crucial to ensure that all parent namespace bindings are also present in the children, except where the children define their own namespace bindings.
To illustrate this process, the following figure depicts a flattening of the namespace hierarchy shown previously in figure 6.1:

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

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

In this representation, the namespace hierarchy is transformed into a flat structure [2]. All elements are positioned on the same level, and the sequence of elements is determined by the depth-first search algorithm.
Each child element inherits the namespace bindings from its parent. Therefore, `element2` and `element3` inherit the namespace bindings from `element1`.
To extract the namespace bindings from this flat structure, one can simply iterate over the elements and record the namespace bindings of each. For the example provided, the extracted list of namespace bindings would be as follows:

Figure 6.3: Extracted namespace bindings from the flattened hierarchy

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

```

6.3.2 Handling Namespace Binding Collisions

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

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

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

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

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

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

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

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

6.3.3 Resolving Version Collisions

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

Figure 6.4: Illustration of a Version Collision

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

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

6.3.4 Resolving Prefix Collisions

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

Figure 6.5: Illustration of a Prefix Collision

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

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

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

Figure 6.6: Representation of a Resolved Prefix Collision

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

6.3.5 Resolving Namespace URI Collisions

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

Figure 6.7: Illustration of a Namespace URI Collision

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

In Brel, namespace URI collisions are not permitted because they can cause errors where elements are not found. For instance, consider the scenario where a user searches for the QName `foo:baz` in the previously mentioned example. Brel would fail to locate it. However, since both `foo` and `bar` are linked to the identical namespace URI, Brel should be capable of finding the QName `bar:baz`. This rationale underpins the prohibition of namespace URI collisions in Brel.
To resolve such collisions, Brel selects one prefix as the preferred option and renames the other to eliminate the conflict Brel opts for the shorter prefix as the preferred one. If both prefixes are of equal length, the choice is based on alphabetical precedence.
In our example, `bar` is chosen as the preferred prefix. Consequently, Brel will rename the `foo` prefix, along with all its occurrences, to `bar`.
Figure 6.8 depicts the figure 6.7 after resolving the namespace URI collision.

Figure 6.8: Representation of a Resolved Namespace URI Collision

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

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

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

Figure 6.9: Table containing all special prefixes

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

6.4 Discoverable Taxonomy Set (DTS) Caching

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

6.4.1 Discovery Process in DTS Caching

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

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

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

6.4.2 Downloading Taxonomies

Retrieving the taxonomy file from a given URI is straightforward for most URIs. However, certain URIs are relative, meaning the URI specifies the location of another taxonomy file in relation to the current one[3]. Brel deduces the domain of relative URIs by recalling the domain from which the relative URI was referenced. It then merges the domain of the current taxonomy file with the relative URI to create a complete URI.
Relative URIs bring up an additional issue - the naming of taxonomy files. As Brel downloads these files from the internet, it must save them locally. Brel opts to store taxonomy files in a folder named `dts_cache`, without subfolders. Hence, Brel needs to ensure each taxonomy file has a distinct name. The initial option for naming is to use the local name from the taxonomy file's URI, but this may not be unique. Another possibility is to utilize the URI itself as the file name. However, since URIs can be relative and typically long, including characters unsuitable for file names, this is impractical. Brel's chosen solution is to create a unique file name derived from the complete URI of the taxonomy file, removing any characters that are invalid for file names.
By employing the discovery and downloading mechanisms, Brel successfully retrieves all taxonomy files referenced by the XBRL report. Brel then saves these files in the `dts_cache` directory, addressing the second crucial assumption outlined in section 6.1.

6.5 Addressing Research Question 3

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

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

To make Brel compatible with multiple formats, both the design and implementation of the Brel API need to be format-agnostic. The design of the Brel API is largely format-independent. Its first segment is grounded in the OIM, which is a logical data model and thus inherently format-agnostic. The latter half of the Brel API, while based on the XBRL XML syntax, largely abstracts away the specifics of this format.
The only exceptions are the `get_link_role` and `get_link_name` methods in the `INetwork` interface, as well as the `get_arc_role` and `get_arc_name` methods in the `INetworkNode` interface. These methods return attributes bearing the same names in XML, primarily serving debugging purposes. They are not essential to the API's functionality. Therefore, the second half of the Brel API is almost entirely format-agnostic, with the exception of debugging methods.
The primary aspect where Brel relies on the XBRL XML syntax is the `QName` class. This class mirrors the QName structure in XML, comprising a prefix, a namespace URI, and a local name. However, even though QNames originate from XML, they have been adopted in other XBRL specifications. Notably, both the JSON[22] and CSV[21] specifications of XBRL adopt QNames in a similar structure to the XML specification, rendering the `QName` class format-neutral.
Given the API's format-agnostic design, the aspect of Brel that relies on the XBRL XML syntax is exclusively the parser. Brel's parser is encapsulated in a distinct module, named `brel.parser.XML`. To support different formats, only this parser module needs modification, allowing the rest of Brel to remain as is.

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

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

- The facts in the report, which can be filtered by concept and a dimension. [1] For each fact, the viewer should display all characteristics of the fact in a easy-to-read manner.
- The components of a report together with their networks. The viewer should be able to display the relationships between report elements in the networks in a human-readable manner.

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

```
pip install brel-xbrl
```

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

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

```python
import argparse, brel

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

Listing 7.1: The implementation of the CLI XBRL report viewer

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

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

```python

# Load the report
report = brel.Filing.open(args.file)
```

Listing 7.2: The implementation of the CLI XBRL report viewer

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

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

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

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

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

Listing 7.3: The implementation of the CLI XBRL report viewer

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

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

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

Listing 7.4: The implementation of the CLI XBRL report viewer

The implementation of the viewer is now complete. The viewer can be used to display the facts and components of an XBRL report using only a few lines of code

and using python's built-in list comprehension.
The following two examples illustrate how the viewer can be used to display the facts and components of an XBRL report. Each example will show both the command that is used to display the information and the output of the command. For both examples, we will use the Q3 2023 10-Q report of Apple Inc. (AAPL) that is available on the SEC's website[12]. To keep the commands simple, we will use the `report.zip` file that contains the report. However, the viewer can also load reports from URLs.

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

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

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

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

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

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

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

As shown in figure 7.6, the command returns the component identified with URI `http://xbrl.sec.gov/ecd/role/InsiderTradingArrangements`. It clearly shows the URI, label, and networks for the component. The networks are pretty-printed in a human-readable manner and show the relationships between report elements in the networks. Additionally, the network shows the link/arc roles and names for each network, which can be useful for debugging.
This example shows that the Brel API is easy to use and can be used to implement a simple CLI XBRL report viewer. The viewer is able to display the facts and components using only a few lines of code and using python's built-in list comprehension. Of course, the viewer is not perfect and can be improved in many ways. The viewer also does not cover every feature of the Brel API, but it serves as a proof of concept for the Brel API. More examples are available both in the Brel

documentation[26] and in the Brel repository[27].

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 [3]. The list of companies that we will use is shown in table **??**. This list was generated by cropping the list of the 100 largest US companies by market capitalization[6]. We limited the list to US companies since the SEC mandates that all US companies must file their financial reports in XBRL[34].

Microsoft	Apple	Alphabet (Google)
Amazon	NVIDIA	Meta Platforms (Facebook)
Berkshire Hathaway	Eli Lilly	Tesla
Broadcom	Visa	JPMorgan Chase
UnitedHealth	Walmart	Exxon Mobil
Mastercard	Johnson & Johnson	Procter & Gamble
Home Depot	Oracle	Merck
Costco	AbbVie	AMD
Chevron	Adobe	Salesforce
Bank of America	Coca-Cola	Netflix

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

The table **??** shows the companies that we will use to evaluate Brel's robustness. The table shows the largest three companies in the first row from left to right, followed by the next three companies in the second row, and so on. For all companies, we will use the latest five 10K and 10Q reports that are available on the EDGAR database[32]. We will load the reports using the Brel API and check for one of three outcomes:

1. The report is loaded successfully without any errors.
2. The report is loaded successfully, but Brel logs a warning or an error.
3. Brel raises an exception while loading the report.

Since Brel loads reports eagerly, we will only load the reports and not perform any operations on them. This will give us a good idea of how robust Brel is in practice. However, it does not cover the full range of Brel's functionality or its correctness. This functionality was already covered in section 7.1. The results of the robustness evaluation are shown in table **??**.

[3]Time of writing: 29 January 2024

Chapter 8

Conclusion

This thesis introduces a Python API for processing XBRL reports, designed to abstract the complexities of the XBRL format from the user. We have successfully demonstrated that this API covers the Object Information Model (OIM) of XBRL, and facilitates straightforward and efficient extraction and analysis of data from XBRL reports. Moreover, we have shown that this API effectively conceals the XML structure underlying XBRL reports from the end user.
The API has been developed into a Python package named `Brel`, which is accessible on the Python Package Index (PyPI) [1]. We have established its robustness in interpreting XBRL reports from various companies. Additionally, we have discussed the capabilities and constraints of the API, which could serve as a foundation for the future development of Brel.
This thesis aimed to answer three research questions, all of which have been answered in the previous chapters. The first question was whether it is possible to create a Python API that encompasses the OIM of the XBRL format. We have shown that it is indeed possible to create such an API in Section 5.4. The second question was whether the non-OIM sections of the XBRL format can be managed in a similar manner. We have shown that this is indeed possible in Section 5.8. The third question was how Brel can be implemented in a way that allows for XBRL reports in both CSV and JSON format to be used as input. We have shown that this is possible in Section 6.5.
Brel contributes the XBRL domain by offering an open-source Python API specifically for XBRL reports. While other similar APIs are available, none prominently feature the Object Information Model (OIM) of XBRL. Arelle provides a Python API, but as a standalone application, it differs from Brel. Brel, being a Python package, integrates seamlessly into other Python applications, distinguishing its role in the field.
Reflecting on our results, we conclude that the API is robust across various XBRL reports as indicated in Section **??**, and adheres to the XBRL standard, inclusive of the OIM. A notable limitation of Brel is its inability to semantically interpret data in XBRL reports, meaning it cannot autonomously identify logical inconsistencies in the reports. While XBRL is primarily designed for machine-readability to facilitate automated processes, Brel does not account for the semantic layer of XBRL, limiting the potential of automated processes to rely on it. As a result, these processes must independently verify the logical consistency of the reports.
Currently, Brel operates predominantly as a syntactic interface for the XBRL format, without leveraging its semantic aspects. Despite this, it simplifies interaction with XBRL reports in Python by masking the technical intricacies of XBRL.

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

Looking forward, Brel could integrate a semantic layer over its existing syntactic framework, enabling more sophisticated analyses of XBRL reports.

8.1 Future Work

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

8.1.1 Support for Additional XBRL Formats

Presently, Brel is compatible only with the XML format outlined in the XBRL 2.1 specification [13]. With the introduction of the Object Information Model (OIM), XBRL has expanded to include formats like CSV and JSON. Currently, these formats are not supported by Brel, and their specifications are still under development [2]. For instance, at this time, the CSV and JSON formats do not accommodate networks, an essential component of XBRL.
In addition to CSV and JSON, XBRL also endorses the Inline XBRL (iXBRL) format, merging XBRL with HTML. Given the SEC's use of iXBRL for financial reporting, its support is critical. Considering the structural similarities between HTML and XML, it is feasible to incorporate iXBRL support into Brel by enhancing its existing architecture.

8.1.2 Semantic Layer Integration

As discussed in chapter 2.3, Brel does not utilize XBRL's semantic layer. Currently, Brel processes XBRL reports without interpreting the data, such as not identifying logical inconsistencies like negative asset values. Future versions of Brel could detect and report these errors, potentially suggesting corrections.

8.1.3 Performance Enhancement

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

8.1.4 Enhancing Usability

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

8.1.5 Conducting a Usability Study

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

[2] The OIM specifications contains a section for footnotes, which are a type of network.

8.2 Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Ghislain Fourny, for his support and guidance throughout the development of Brel and the writing of this thesis. Without his expertise and encouragement, this project would not have been possible.
I am also grateful to Prof. Dr. Gustavo Alonso for supervising this thesis and letting me work on this project as part of the Systems Group at ETH Zurich.
Both the development of Brel and the writing of this thesis have been aided by a few indispensable tools. Handling XML and HTTP requests in Python would have been much more difficult without the help of the `lxml` and `requests` libraries. They have been invaluable in the development of Brel, and I am grateful to their developers for their hard work.
I would also like to thank OpenAI for developing ChatGPT[3] , and GitHub for publishing Copilot[4]. I am not a writer, I am a programmer. My writing skills do sometimes leave something to be desired, and ChatGPT has been a great help in improving the quality of my writing and expressing my thoughts in a more formal manner. ChatGPT does generate text with many logical errors, but when used correctly, it can provide an alternative perspective on a text that can be very helpful. Even though this thesis was written with the help of ChatGPT, none of the information in this thesis was generated by ChatGPT. ChatGPT was merely used for two purposes: to improve the quality of my writing, and to provide an alternative perspective on the text.
GitHub Copilot has also been a great help for implementing Brel, as it has helped me generate examples for libraries that I have never used before, and has helped me find the right functions to use in many cases. Similar to ChatGPT, GitHub Copilot does not necessarily generate correct code, but it can speed up the mundane parts of programming significantly. Copilot was used for two purposes: to generate examples for libraries that I have never used before, to speed up repetitive tasks in programming like creating getters and setters, and to give alternatives for how to name certain methods and classes.
Apart from these tools, I would like to thank my friends for their support during my studies at ETH Zurich, especially Pascal Strebel, who has been a great friend all throughout my ETH journey. I hope that we will continue to be friends for many years to come and that we will continue to support each other, no matter where life takes us.
I would like to thank my family for their unwavering support and encouragement. I know that I did not always have time for them during my studies, but I hope that they know that I love them and that I am grateful for everything that they have done for me.
And last but not least, I would like to thank you, the reader, for taking the time to read this thesis. I hope that you have found it interesting and that you have learned something new from it.

[3]`https://chat.openai.com/`
[4]https://github.com/features/copilot

Bibliography

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

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

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

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

[5] Joe Cabrera. python-xbrl. https://github.com/greedo/python-xbrl, 2016.

[6] 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.

[7] 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.

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

[9] Jacob Fenton. pysec. https://github.com/lukerosiak/pysec, NaN.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

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

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

[28] Manuel Schmidt. py-xbrl. https://github.com/manusimidt/py-xbrl, 2021.

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

[30] 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.

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

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

[33] 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.

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

[35] 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.

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

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

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

[39] 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.