TU WIEN Informatics

CM2ML: A generic, portable, and extensible framework for encoding the structure of conceptual models

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jan Patrick Müller
Matrikelnummer 12102339

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dr. Dominik Bork
Mitwirkung: Univ.Ass. Syed Juned Ali, BSc MSc

Wien, 18. November 2024

Jan Patrick Müller

Dominik Bork

TU WIEN Informatics

CM2ML: A generic, portable, and extensible framework for encoding the structure of conceptual models

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

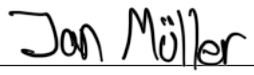
Jan Patrick Müller
Registration Number 12102339

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dr. Dominik Bork
Assistance: Univ.Ass. Syed Juned Ali, BSc MSc

Vienna, 18th November, 2024



Jan Patrick Müller

Dominik Bork

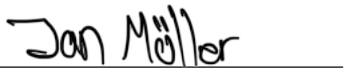
Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Jan Patrick Müller

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. November 2024



Jan Patrick Müller

Acknowledgements

With deepest gratitude I would like to thank my family for their unwavering moral and financial support throughout my studies. Their belief in my abilities encouraged me to surpass any challenges I encountered throughout the years, with their wisdom being an unquantifiable benefit.

Further, I also want to thank my supervisors Associate Prof. Dr. Dominik Bork and Univ.Ass. Syed Juned Ali BSc MSc for their guidance and knowledge throughout the creation of this thesis. Their insight and expertise proved invaluable and greatly aided the thesis' progress.

Kurzfassung

Das konzeptuelle Modellieren erstreckt sich über ein breites Spektrum an Disziplinen in der Welt der Informatik. Eine große Zahl an Anwendungen macht konzeptuelles Modellieren zu einem allgegenwärtigen Werkzeug für Designer, Entwickler, Projektmanager und andere Gruppen. Zuletzt wurde Machine Learning in mehreren Anwendungsfällen eingeführt, wofür jedoch Modell-Kodierungen benötigt werden. Während ein beträchtlicher Aufwand in die Entwicklung und Verbesserung dieser Kodierungen fließt, fehlt es an Effizienz-Vergleichen für konkrete Anwendungen. Zudem werden Implementierung von Kodierern häufig an eine spezifische Modellierungssprache gekoppelt. Das Auswählen der besten Kodierung für eine Aufgabe ist für Laien daher schwer.

Wir implementieren Conceptual Models to Machine Learning (CM2ML), ein Framework zum Kodieren der Struktur von konzeptuellen Modellen, auf Basis der Design Science Research Methodik. Eine Entkopplung von Modellierungssprachen und Kodierungen durch Einführen einer Zwischenrepräsentation macht es generisch. CM2ML ist modular und erweiterbar, womit ein flexibler Einsatz sowie eine Erweiterung durch Parser für neue Modellierungssprachen und Kodierern möglich ist. Experimentieren wird durch seine komponierbaren Plugins und deren deklarative Parameter, welche eine strukturierte Schnittstelle für die Konfiguration von Parser- und Kodiererverhalten bieten, erlaubt. CM2ML inkludiert Adapter für Kommandozeilen und REST Server, während seine Software Bibliothek Scripting und eine Implementierung neuer Adapter erlaubt. CM2ML inkludiert einen Parser für UML sowie drei konfigurierbare Kodierer. Neben graphen- und baumbasierten Kodierern verfügt es über ein Bag-of-Paths (BoP) Framework zum Kodieren von Modellelementen und deren Kontext in einer anpassbaren Textform. Eine Webanwendung stellt zudem Visualisierungen für die Zwischenrepräsentation und Kodierer bereit, womit Nachvollziehbarkeit zwischen Modellen und Kodierungen gegeben ist.

In einer experimentellen Evaluation wird die Effizienz von CM2MLs graphen- und baumbasierten Kodierern bei der Klassifizierung von Elementen verglichen. Dafür werden verschiedene Konfiguration für den UML Parser und beide Kodierer evaluiert. Die Ergebnisse zeigen einen deutlichen Vorteil für die graphenbasierte Kodierung sowie Verbesserungen der Effizienz mit bestimmten Konfigurationen. In einem zweiten Experiment wird der BoP Kodierer so konfiguriert, dass er ein modernes Framework für die Graph-Sprachmodellierung erfolgreich emuliert. CM2ML kann also zum Vergleichen von Kodierungen und zum Experimentieren mit Parsern und Kodierern verwendet werden.

Abstract

Conceptual modeling spans a broad range of disciplines in the areas of software engineering and computer science. A large number of use cases for conceptual modeling render it an omnipresent tool for designers, engineers, project managers, and other groups of users. Recently, researchers started introducing machine learning to certain use cases, which requires some form of model encoding. While considerable effort is put into developing and improving those encodings, a lack of comparisons regarding their efficiency for a given task makes it difficult for non-experts to choose the one best suited. This is reinforced by implementations of encoders often being tied to specific modeling languages.

We implement Conceptual Models to Machine Learning (CM2ML), a framework for encoding the structure of conceptual models, following the Design Science Research methodology. It's generic in the sense of decoupling modeling languages from encoder implementations through an intermediate representation (IR). CM2ML is modular and extensible, allowing it to be used flexibly and extended with support for new languages and encodings. Its composable plugins and their declarative parameters provide a structured interface for configuring the behavior of parsers and encoders, enabling experimentation. CM2ML includes adapters for a command-line interface and a REST server, while its software library may be used to implement custom adapters or scripts. CM2ML has a built-in parser for UML and three configurable encoders. Besides a raw graph and a tree-based encoder, it features a Bag-of-Paths (BoP) framework for encoding model elements and their context in a customizable textual representation. A web application that provides visualizations for the IR and each encoder provides traceability between a model and a resulting model encoding.

In an experimental evaluation, the efficiency of CM2ML's raw graph and tree-based encoders for the task of node classification is compared, including different configurations for the UML parser and both encoders. The results show a clear advantage for the raw graph encoding and efficiency improvements for certain configurations. In a second experiment, CM2ML's BoP encoder is configured to successfully emulate the output of a state-of-the-art graph language modeling framework. As such, CM2ML may be used to compare encodings effectively and experiment with parameters for parsers and encoders.

Contents

CHAPTER 1

Introduction

Conceptual modeling is an ever-present aspect of several disciplines in the world of computer science and software engineering. For example, the latter leverages conceptual modeling for tasks such as architectural design, business process modeling, documentation, and code generation [BAD23; SR17]. At the center of conceptual modeling, modeling languages such as the Unified Modeling Language (UML), ArchiMate, and Ecore enable the definition of models that represent abstract concepts.

Recently, the conceptual modeling community has started introducing machine learning (ML) to different stages of modeling workflows [BAR23]. For the definition stage, this includes model creation itself, as well as the completion of partial models [Di +21; WSS22]. In the later stages, ML may be used to transform models, e.g., by generating model transformations or analyzing models to gain insights into concepts such as model domain or model similarity [Bur+22; Lóp+22; Ngu+19]. These tasks, or at the very least their respective ML training stages, use conceptual models as their input.

As an abstract representation, conceptual models may be encoded in various formats. Each encoding can capture structural and semantic information about a model to varying degrees and levels of detail. Currently, there are many different approaches for encoding the structure of conceptual models, including raw graphs, graph kernels, tree-based encodings, and Bag-of-Paths (BoP) frameworks [Bur+22; Kha+22; Lóp+22; WSS22]. Research on these encodings often does not compare their efficiency with other approaches but focuses on a single encoding for a given use case. Few comparisons of encodings are conducted, resulting in a lack of understanding which encoding may be best suited for a given task [Lóp+22]. This is reinforced by implementations of encoders being most commonly tied to a specific modeling language [AB24; BAD23; WSS22]. Further, research into encodings for conceptual models often proposes configurable parameters [AB24; BC17; FSG22]. However, manually adapting existing implementations of encoders to reflect these parameters is a time consuming task, as no common interface for parameter configurations exists. Encoders are often implemented with different means for configuring

said parameters, e.g., code or implementation changes, interactive prompts, and command-line interface (CLI) arguments with different formats are commonly used [AB24; Bur+22; FSG22]. Given these circumstances, testing different encodings and comparing their efficiency for a specific ML task can be challenging.

1.1 Methodology

This thesis follows the Design Science Research (DSR) methodology [Hev+04; Off+09], because its focus on iterative development and creating artifacts for evaluation enables a continuous evaluation and validation of an implementation. In the context of this thesis, the artifacts to evaluate are a modular software framework as a whole but also its individual modules. We follow the phases of the DSR methodology as described below.

1.1.1 Problem Identification and Motivation

Through a literature review and an expert interview with the thesis' supervisors, two senior researchers with backgrounds in the area of conceptual modeling, a gap in the research around encoding conceptual models was identified. In particular, research is often focused on implementing a certain encoding for a specific modeling language. These encodings are often not configurable and not compared with different encodings for a specific task [Bur+22; WSS22; Kha+22]. Further, evaluative comparisons are focused on specific ML tasks and different ML models and do not include variations of a specific encoding [Lóp+22]. As a result, we identify the need for a framework for encoding conceptual models. The framework should be generic to decouple encodings from concrete modeling languages. It should also be modular to support an extension with additional encodings and support for new modeling languages. Finally, it should be configurable in the way conceptual models are represented and encodings created to facilitate experimentation.

1.1.2 Objective Definition

Based on the results of the problem identification, we define a list of measurable requirements for the framework. Further, the thesis' supervisors experience in university-level teaching revealed additional requirements beyond technical aspects. In particular, visualizing the inner workings of encodings in order to provide traceability for relating model to model encoding is an important criteria in teaching. Thus, the goal of the thesis is to implement a framework, called Conceptual Models to Machine Learning (CM2ML), for encoding the structure of conceptual models. This framework should meet the requirements defined in section 1.2.

1.1.3 Design and Development

Based on the defined objectives, the CM2ML framework is implemented iteratively. In addition, modular design is used to guide the development of the framework's architecture.

In each iteration, a functional framework is created as an artifact. As some implementation requirements or limitations only materialize during development, continuous design readjustments of the frameworks and its modules are made as required.

1.1.4 Demonstration

The framework's functionality is validated by conducting a comparison of two encodings for the ML task of node classification. Further, its third built-in encoding is validated by configuring it to resemble a state-of-the-art encoding. Finally, the runtime performance of the framework is demonstrated in an experiment. All three demonstrations simulate real-world applications to highlight how the CM2ML framework is usable for such scenarios.

1.1.5 Evaluation

The CM2ML framework is evaluated using test-driven development (TDD) [Ast03] to validate its artifacts during and after each iteration. This approach enables an early detection of issues and limitations, supporting iterative development. Further, the three demonstrations described above form an assessment of the requirements defined as part of the objective definitions.

1.2 Goal of the Thesis

Based on the problem identification and objective definition phases of the DSR methodology, we define the following requirements for the CM2ML framework.

[**REQ1**] **Genericity:** The framework should be generic. An intermediate representation (IR) should decouple encoders from concrete modeling languages. A parser for UML models using the Eclipse Papyrus serialization format should be implemented.

[**REQ2**] **Modularity:** The framework should be modular. Its individual modules, i.e., parsers and encoders, should be available as standalone components for developers.

[**REQ3**] **Portability:** To enable various use cases, the framework should be portable across different environments. This includes a software library, a CLI, a representational state transfer (REST) server, and a web application.

[**REQ4**] **Extensibility:** The framework should be extensible. The software library should support an extension of the CLI and REST environments with custom parsers and encoders.

[**REQ5**] **Configurability:** The framework should be configurable. Parameters of parsers and encoders should be configurable without modifying the framework's implementation.

[**REQ6**] **Traceability:** A dedicated web application should provide visualizations for the IR and outputs of all built-in encodings. These visualization should show how different parts of an encoding output correspond to specific elements of the IR.

[**REQ7**] **Performance:** The framework and encoders should be performant enough to encode batches of at least 1,000 conceptual models in reasonable time, i.e., one minute, on consumer hardware.

1.3 Research Questions

The thesis aims to answer the following research questions:

[**RQ1**] Can a framework fulfilling the requirements of genericity, modularity, portability, extensibility, configurability, traceability, and performance be realized?

[**RQ2**] Can such a framework be used to effectively compare the efficiency of different encodings?

1.4 Structure of the Thesis

Chapter 2 describes the thesis' background and related work. Afterward, chapter 3 details the implementation of the CM2ML framework, followed by a presentation of its visualizer in chapter 4. Next, chapter 5 presents the built-in encoders and configurable parameters of CM2ML, which are then evaluated in chapter 6. Finally, chapter 7 summarizes the thesis' results and presents opportunities for future work.

CHAPTER 2

Background and Related Work

This chapter first describes the background of conceptual modeling with UML. Afterward, a dataset of UML models used throughout the thesis is presented. Finally, work related to the thesis is detailed.

2.1 Conceptual Modeling with UML

This section provides an overview of conceptual modeling, using UML as an example. Given the size of the conceptual modeling field, it is not exhaustive and instead presents topics relevant for the scope of the thesis. Since UML is complex by itself, we only present key concepts of the UML modeling language. For further reading, the UML specification describes all inherent elements of the modeling language [Obj17].

2.1.1 Classes

UML has hierarchically organized classes, also referred to as types in this thesis. Each class may have zero, one, or multiple generalizations. Generalizations are a form of inheritance, i.e., a class inherits associations, attributes, and transitive generalizations from each of its generalized classes. A number of categories of UML classes exist. For example, structural classes like `Interface` or `Property` describe the structure of their instances. Behavioral classes such as `Event` and `Activity` can be used to define behavior, while others like `ActivityGroup` or `Package` may be used to group their contained elements. Finally, the `Comment` class is only concerned with adding textual annotations to UML models.

2.1.2 Elements

Elements are the foundational building blocks of UML models. Each element has an assigned type, i.e., a UML class. In essence, elements are instances of classes with

attributes and associations. In UML modeling software, elements are usually depicted as nodes of different shapes and sizes.

2.1.3 Associations

Associations are connections between two or more elements, with many associations having defined types for their connected elements. They can represent relationships, references, or containment and are commonly depicted as lines connecting the nodes of elements. Classes may redefine associations from their generalizations and, e.g., add semantic information.

2.1.4 Attributes

Both associations and elements may have attributes in UML. Attributes have a name, a type, and a value. Classes of the UML metamodel have attributes with primitive types and optional default values. In particular boolean, numeric, string, and enumeration literals are used as values in the UML metamodel.

2.2 Dataset

The development of the UML parser and the evaluation of CM2ML's encoders is guided by a dataset containing 46,731 UML models. This dataset is a subset of the MAR dataset [LC20], a collection of UML models crawled by the MAR model search platform from sources such as public GitHub repositories. Similar to the ModelSet dataset [LCC22], this dataset is curated by removing any identical duplicates. The curation and management of the dataset was conducted by the thesis' assistant advisor. For reproducibility, the dataset is published online[1]. This dataset is chosen over the popular ModelSet for its wider coverage of UML classes, an important factor for implementing a UML parser with TDD.

2.3 Related Work

As the usage of ML applications for conceptual models increases, researchers have continued developing and improving encodings of conceptual models. This section will first present seven state-of-the-art structural encodings, followed by existing research that deals with comparing different encodings. Further, it will analyze existing evaluation approaches for structural encodings.

2.3.1 State-of-the-Art Structural Encodings

Based on a systematic review of encodings for conceptual models by Ali et al. [Ali+23], a number of structural encodings relevant to the thesis are identified and presented below.

[1] `https://owncloud.tuwien.ac.at/index.php/s/QiHnZEOaWPyxzrW`, password "cm2ml".

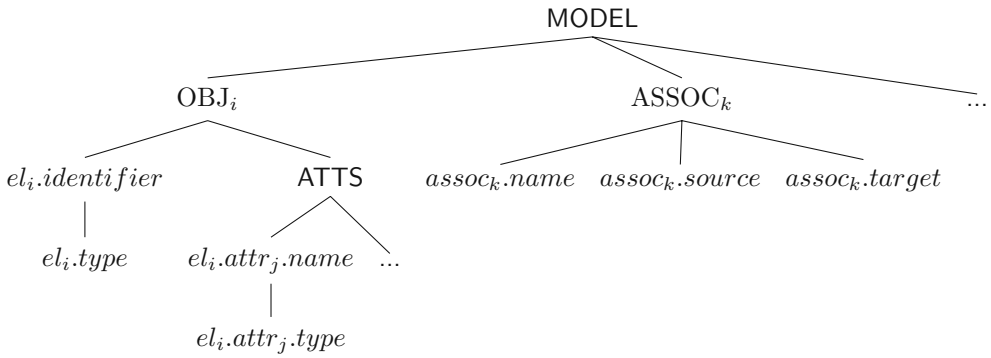


Figure 2.1: Tree-based encoding for models using globally enumerated associations.

Recently, Smajevic [Sma22] implemented a generic transformation process for creating graphs based on conceptual models. The proposed process is generic and able to transform different modeling languages into a common graph structure. In particular, the implementation is able to transform the three modeling languages Ecore, ArchiMate, and UML[2]. This work is of relevance to the thesis, because it shows that graphs are capable of representing models from different modeling languages. Further, it demonstrates that a graph structure that extends simple labeled nodes and edges with attributes is able to capture both structural and semantic information of conceptual models.

While not targeting a generic end-to-end process, some research into encodings of conceptual models considers support for multiple modeling languages. For example, Burgueño et al. [Bur+22] detail a generic transformation process for a tree-based encoding. Their process includes several steps that are modeling language-independent. The proposed encoding is a tree, with a structure as shown in Figure 2.1, that captures the structural information of conceptual models. While not explicitly mentioned, their transformation process can be adapted to support a generic graph-based IR, as long as identifiers and types of elements are accessible. To evaluate their encoding, a tree long short-term memory network (Tree-LSTM) is trained to transform class models to relational models. In a second experimental evaluation, Burgueño et al. [Bur+22] train a Tree-LSTM to generate Java code from UML models.

Weyssow et al. [WSS22] also propose a tree-based encoding. While their use case is different from that of Burgueño et al., their tree structure is similar and exhibits only few differences. Namely, association nodes are children of the node corresponding to the association's source element, as depicted in Figure 2.2. In addition, only the identifier of an element, and not also its type, has a distinct node separate from other attributes. Finally, the attributes are encoded differently. Instead of attribute name and value, types and names of attributes are included in their tree structure.

Fumagalli et al. [FSG22] present a pattern-discovery workflow for conceptual models.

[2]The serialization format used by the Eclipse Papyrus UML tool is supported.

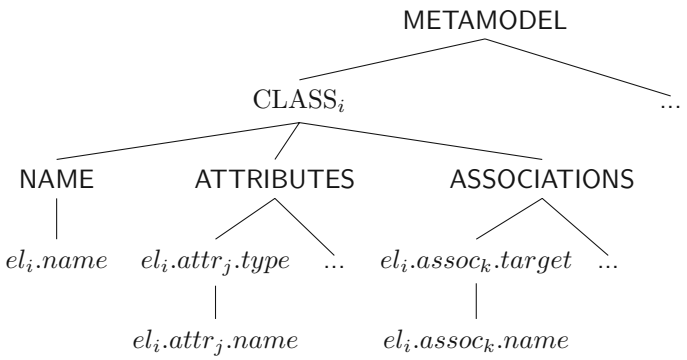


Figure 2.2: Tree-based encoding with locally enumerated associations.

Their proposed workflow consists of several tasks, the first being a language-dependent import of conceptual models. This first task creates a graph for each imported conceptual model. The next tasks of the workflow, i.e., discretization, normalization, embedding, and mining, are all language-independent. Their goal is to support different modeling languages. They realize this goal by making all tasks, except the model import, language-independent.

López et al. [Lóp+22] implement a range of different semantic and structural encodings. Among the latter, they include a raw graph encoding for graph neural network (GNN) models. They transform conceptual models to graphs and create adjacency lists from the resulting edges. To encode elements as feature vectors, they enumerate attribute values and replace each value with its corresponding index.

Khalilipour et al. [Kha+22] also propose a graph-based IR. Their encoding leverages graph kernels to compare the structure of graphs and extract features. While only implemented for Ecore, their method also considers compatibility with other modeling languages such as UML, hence the usage of a language-independent IR.

Ali and Bork [AB24] present a BoP approach with their Graph Language Modeling framework for Conceptual Models (GLaM4CM) that operates on conceptual models from a graph-based perspective. It enriches data from individual nodes with contextual information of neighboring nodes by connecting them through paths. Afterward, these paths are encoded in a textual representation.

2.3.2 Encoding Comparisons

López et al. [Lóp+22] conduct a comparative study evaluating the efficiency of different encodings for the task of model classification. They implement a framework that enables a comparison of various structural and semantic encodings and their performance for this tasks. As required by the structure of the encodings at hand, they use different ML models for their evaluation. Their framework differs from the goal of the thesis, because it is focused on a singular task, i.e., model classification. It is not concerned

with encoding configurability and experimentation. Further, it does not meet some of CM2ML's requirements, e.g., traceability or portability. For structural encodings, their framework uses graphs as an IR. Their mapping approach from conceptual models to graphs transforms each model element to a graph node, each association to an edge, and assigns each node a name and class, i.e., type.

CHAPTER 3

Framework

This chapter describes the CM2ML framework and its modularity concept. CM2ML is published as a monorepo on GitHub[1]. The framework is split into a number of separate modules, also referred to as packages or components. This architectural choice enables a usage of individual components in environments that may not support other parts of the framework. In particular, the CLI and REST packages are not supported by, and thus redundant for, browser environments, in which the software library may be used. One core concept of the CM2ML framework is the separation of parsers and encoders. Parsers turn a serialized conceptual model into an instance of the framework's IR, which is then consumable by any encoder, as they transform IR instances into encoder-specific data structures.

First, the core concepts of the framework, plugins and adapters, and their underlying technology are detailed. Next, the IR enabling arbitrary combinations of parsers with encoders is presented. Afterward, the implemented UML parser and built-in deduplication approach are outlined. Finally, an overview of the framework's architecture is given.

3.1 Technology

The technology behind CM2ML is determined by its requirements presented in section 1.2. In the following subsections, the individual technologies are presented.

3.1.1 TypeScript

To meet the portability requirement, we implement the framework using the TypeScript programming language [Micb]. As Typescript transpiles to JavaScript, the CM2ML framework may be used in browser applications. In addition, JavaScript runtimes such as

[1] https://github.com/borkdominik/CM2ML

Node.js [Dah] and *Bun* [Sum] enable a usage of the framework in a number of scenarios, including CLIs, desktop apps, or servers. Another strong point of TypeScript is the ease-of-distribution for the CM2ML framework, as users may simply install the framework using a package manager of their choice, like *npm*, *pnpm*, or Bun.

We choose TypeScript over JavaScript for three reasons. First, we leverage its type checking for static code analysis [BAT14]. Each component and plugin of CM2ML has typed inputs, parameters, and outputs. With type checking, we are able to, e.g., ensure that composing two components is only possible if their outputs and inputs match. Next, TypeScript code is usually of higher quality, indicated by a lower number of code smells per line of code [BM22]. Finally, the inclusion of type declarations in code artifacts may provide auto completion to users of CM2ML's software library, even if they are not using TypeScript themselves.

3.1.2 Turborepo

We use the monorepo development strategy, i.e., the usage of a single, shared git repository for all individual components, to organize the development of the CM2ML framework. *Turborepo* [Ver], a build system for TypeScript and JavaScript projects, allows the definition of *tasks* with dependencies, inputs, and outputs. Using Turborepo allows us to orchestrate code analysis, testing, and building of all components that constitute the entirety of the CM2ML framework. Beyond that, the task orchestration also enables a semi-automated execution of the evaluations performed in chapter 6. Turborepo ensures that all components are built and up-to-date, then executes the framework for a selected encoder and finally starts the evaluating ML Python application by executing a respective script.

3.1.3 Libraries

A number of software libraries, i.e., *packages* in JavaScript terminology, are used across every component of the CM2ML framework. These are presented below, while packages only used in specific components of CM2ML are detailed in the respective sections.

Vite

While *Vite* [Voi] is foremost a development server for web applications, it may also be used to bundle, i.e., build TypeScript packages from source code. We also implement *vite-plugin-lib* [Mül], a Vite plugin that automatically configures a range of settings for bundling TypeScript packages without requiring additional configuration. vite-plugin-lib also creates type declarations for bundled packages. These are required to maintain type information, as the bundled code is transpiled JavaScript. vite-plugin-lib is published on npm and used to build every component of the framework. Vite also enables a usage of Vitest [Vit], a modern unit testing library used extensively across CM2ML to validate implementations.

Zod

Zod is a library for declaring data schemas and validating input [McD]. As CM2ML deals with user provided input for parameter configuration, we employ Zod across the framework to define schemas for parameters. User-provided input is then automatically validated by the framework using Zod. Should input not adhere to a parameter's schema, a human-readable error message is generated by Zod and presented to users.

3.1.4 Continuous Integration

A continuous integration pipeline based on GitHub Actions ensures that each Turborepo task can be successfully executed in a clean environment for every commit. These tasks include building packages, running static analysis with *ESLint* [Ope], executing unit and End-to-End (E2E) tests, and performing type checking. It is also configured to publish packages with npm's *provenance* statements. These statements allow users to validate how, when, and where a published package was built. As a result, supply-chain security is increased [npm].

3.2 Plugins

Plugins are the building blocks of any workflow within the framework and implemented in the package `@cm2ml/plugin`. In simple terms, a plugin is a named execution step of a workflow with a list of input parameters. Each plugin has typed input and output with optional parameters, similar to a regular TypeScript function. The main goal of plugins is not to prevent code duplication and enable reuse, but to establish a well defined interface for plugin adapters and encapsulate parameter definitions.

3.2.1 Parameters

A parameter is defined by a name, datatype, description, default value, and an optional group. Depending on the datatype, it may also have a list of allowed values. Plugins have a validation schema for input parameters that is automatically derived from their parameter declaration. These validation schemas may be used by adapters that deal with user-provided input to ensure the validity of parameter configurations. The following parameter types are available and used across the framework.

`boolean`

A simple boolean value, that can either be `false` or `true`.

`number`

Any integer or floating point number. The allowed values may be restricted and validated by plugins at runtime.

`string`

Any string, or an enumeration value if a list of allowed values is defined. If such a list is given, the parameter type is treated as an enumeration and user-provided values are validated by the parameter's schema.

`list<string>`

A list of strings, or a list of enumeration values if a list of allowed values is defined. Validation behavior for enumeration values is analogous to that of `string` parameters. Beyond that, list parameters can be flagged as *unique* and *ordered*. Uniqueness treats the list as a set without duplicates, while an ordered list is never sorted by the framework.

3.2.2 Composition

Plugins can be combined and modified in a number of ways using helper functions. Most importantly, two plugins can be chained. This is simply referred to as *composition* and results in a new plugin that invokes the second plugin with the output of the first plugin. To propagate the type-safe nature of plugins, the input type of a plugin created through composition matches that of its first plugin, while the output type matches that of its second. The parameters of such a plugin equal the merged parameters of its composed inputs. The composition mechanism ensures that there are no parameter naming conflicts at runtime.

Next, plugins can be modified to accept batch input by wrapping them using the `batch` function. These batch plugins invoke their underlying plugin for every item of an input batch and return the corresponding list of outputs. Of course, a plugin implementation may also work on batches directly, but this utility enables a more streamlined definition of plugins that do not have to process batch metadata.

In order to support per-item error handling for batch plugins, the `trying` and `catching` primitives are implemented. The former is a modifier that wraps a plugin with try-catch logic. When invoked, it either returns the underlying plugins output or, should an error be thrown, an `ExecutionError`. The latter is a standalone plugin that accepts input of a given type as well as instances of `ExecutionError`. When invoked, it acts as a passthrough for non-`ExecutionError` inputs. Every caught `ExecutionError` is thrown again, unless its `continueOnError` parameter is enabled. This enables batch workflow executions that can ignore invalid input and continue with valid items.

3.2.3 Structured Output

To provide a common output structure for the framework, we define the `Structured-Output` interface. It is a generic record with two fields, `data` and `metadata`. For each field the interface has a generic type, so that it may be used for different data structures. By convention, the `data` field is used to store output for a single batch entry, or in the case of batch plugins, store an output array matching the size of an input batch. The

`metadata` field on the other hand is only used for data that is shared for every entry of a batch. This metadata should be identical by reference for every item, to reduce memory usage for large batches by preventing duplication. If this is not possible, the `metadata` field must be equal by value for all items of a batch.

3.3 Adapters

The `@cm2ml/plugin-adapter` package exports the `PluginAdapter` class, which includes utility methods for applying, i.e., registering, plugins in an execution environment. Plugin adapters have a typed input and output, which all of their applied plugins must adhere to. The following two plugin adapters are implemented as part of the framework. Both receive a list of strings, i.e., serialized conceptual models, as input and emit `StructuredOutput`. The `data` of this structured output is typed as an array with elements of the type `unknown`, while `metadata` is also typed as `unknown`. Since plugin adapters are only responsible for serializing output as JSON, they do not need to know the actual output datastructures of applied plugins.

3.3.1 CLI adapter

The CLI adapter, found in the `@cm2ml/cli-adapter` package, can be used to define and instantiate a CLI that exposes each applied plugin as a separate subcommand. An executable CLI configured to include all of the built-in parsers and encoders is provided in the `@cm2ml/cli` package. It is implemented with *CAC* [EGO], a lightweight framework for creating CLIs. For each plugin, the adapter creates two subcommands, one for processing a single conceptual model and the other for processing an entire batch at once. Subcommand names are equal to a plugin's name, with the batch version being prefixed with "`batch-`".

Conceptual models are provided via a path for the single-model subcommand. For batch subcommands, the path of the directory containing the input models must be provided instead. The CLI adapter then resolves the files and reads their content, which is, in turn, provided to plugins as input. Output is either written to an output file, also referenced by its target path, or to the console. For batch subcommands, the output will be written as a JSON dictionary with the names of input files as keys and their corresponding encodings as values.

Plugin parameters are exposed as options, i.e., arguments, for each respective subcommand. The adapter automatically configures the subcommand options for each loaded plugin based on its respective parameter schema. Should a plugin have the `continueOnError` parameter, it is ignored for the single-input variant. Batch subcommands have two additional options on the other hand. `start` allows users to define an offset from the beginning of the list of files in a directory. Further, `limit` enables the configuration of an upper limit for the number of files to process. Both have no default values, thus all files in a directory are processed by default for batch subcommands.

3.3.2 REST adapter

Located in the `@cm2ml/rest-adapter` package, the REST adapter configures an HTTP server that exposes a route for every applied plugin. Powered by the low-overhead *Fastify* [The] web framework, it enables the remote encoding of models. Analogous to the CLI, a pre-configured executable of a REST server with all built-in parsers and encoders is available through the `@cm2m/rest` package.

Only two endpoints are static. The first, `/health`, is a GET endpoint and returns the number of applied plugins. This minimal endpoint may be used to validate uptime and check server status. Next, `/plugins` is a GET endpoint returning the metadata of all applied plugins. This metadata included each plugin's name and parameter data, i.e, parameter name, type, default value, and description.

Each applied plugin has a dynamically generated POST endpoint `/plugins/{name}`, where `{name}` is the plugin's name. When invoked, these dynamic endpoints first validate the request body, which must be valid JSON. The body must also contain the key `input`, with its value being a non-empty array containing the serialized input models. Further, values for each parameter of a plugin may be configured by providing them under the respective parameter's name in the body. Unlike the CLI, the REST adapter has no special endpoint for processing a single input model. The response of a plugin-specific endpoint is JSON-serialized `StructuredOutput`. Encodings are contained in the `data` array and may be positionally matched to each provided entry in the `input` array of the request body.

3.4 Software Library

Each module of the framework is installable as a standalone package. For convenience, we provide two additional packages, `@cm2ml/builtin` and `@cm2ml/cm2ml`. The former exports all individual parser and encoder plugins of the framework, as well as utility functions for composing a list of parsers and encoders to create ready-to-use plugins for adapters. In addition, it exports a list of pre-composed plugins that combine every built-in parser and with every built-in encoder. We refer to these pre-composed plugins as *prepared plugins*. The second package, `@cm2ml/cm2ml`, includes all individual modules of the framework as well as the two pre-configured CLI and REST adapters and their executable binaries.

3.5 Intermediate Representation

An IR for conceptual models is the key enabler for being able to use one encoder implementation for different modeling languages. It is particularly important that an IR is expressive enough to capture the structural and semantic information from models of different modeling languages, while also providing the data required for various encodings. Of course, it is not possible to foresee every possible modeling language and encoding,

thus potentially limiting future usage of the IR. As such, CM2ML's IR implementation is based on the related work presented in section 2.3. The implemented IR is a composite data structures that captures two distinct perspectives on conceptual models, which are presented below. Each instance of an IR model can also be marked as *strict* by its instantiating parser. Strict models will have additional validations performed across the framework to ensure that input models are valid.

3.5.1 Metamodel Information

It is necessary for some encoders to have access to certain metamodel information. Thus, for each model of the IR, we include a reference to metamodel information based on a concrete modeling language. Available information includes a list of available attribute names, a list of possible types for elements, as well as the names of the attributes used for storing identifiers, types, and element names. With this static configuration data, accessor functions for these special attributes are automatically generated. As a result, it is possible to use those accessor functions for, e.g., accessing a model element's type or identifier in an encoder, without knowing which modeling language is used or having to differentiate between languages.

3.5.2 Models as Graphs

The IR's first view on models treats them as graphs to capture connections, e.g., UML associations, between model elements. Elements are treated as nodes, while associations, i.e., references between elements, are treated as edges. Both nodes and edges can have attributes, with certain attributes, i.e., identifiers, types, and names, receiving special treatment. Attributes can be typed to support type-sensitive use cases like, e.g., encoding attribute values. The available types are `unknown`, `string`, `category`, `integer`, `float`, and `boolean`. It should be highlighted that all values are stored as strings to prevent the need for type casting at runtime. With this approach, the types of attributes only have to be used when necessary, e.g., during the above mentioned encoding of attribute values. Nodes and edges can also have tags, which may be initialized by a parser as desired but are designed to match a keyword used for a node or edge in a modeling language's serialization.

3.5.3 Models as Trees

Second, any hierarchical relations between model elements are captured by also presenting the nodes of IR models as a tree structure. For example, UML models have the concept of *containment*. The tree structure captures that concept by exposing *contained* elements as children of their *container*. As a limitation, all models must have exactly one root element. For modeling languages with no concept of a single root element, a possible workaround might be the inclusion of a *virtual* root element, that has the actual root elements as children. Since the creation of the tree structure is parser-specific, such a workaround would require no changes to the framework.

3.5.4 IR Post-Processor

The framework includes a plugin designed for modeling language-independent processing of IR models. This plugin may be used by parser authors as desired. It validates a number of postconditions for an IR instance, like ensuring each node has a unique identifier. For non-strict IR models, the post-processor plugin will remove any invalid nodes. These include non-root nodes without any edges or identifiers.

In addition it implements a number of optional transformations that are configurable through parameters. In particular, the plugin offers the two `boolean` parameters `nodeTagAsAttribute` and `edgeTagAsAttribute`. If enabled, the post-processor will add an attribute to nodes or edges that contains the respective tag. Finally, the `unifyTypes` `boolean` parameter may be enabled to, if applicable, merge type attributes for all nodes and edges. Since some modeling languages can use different attributes for type information, e.g., `xmi:type` and `xsi:type` in UML, this flag may be required to ensure a predictable categorical encoding of their values.

3.6 Parsers

In the CM2ML framework, parsers are plugins that transform serialized conceptual models, i.e., strings, into instances of the IR. Through composition with encoder plugins, which are detailed in chapter 5, parsers enable a usage of their modeling language with every implemented encoding. This thesis implements a parser for UML models using the serialization format of the Eclipse Papyrus modeling software. As a reusable foundation, an Extensible Markup Language (XML) parser is implemented. Both are described below.

3.6.1 XML Parser

The framework's reusable XML parser is implemented using the *htmlparser2* [Win] library. It should be highlighted that the XML parser is not usable standalone and requires configuration from parser authors. In particular, a metamodel configuration as described in subsection 3.5.1 and a handler for text XML nodes are required.

The parser itself is a plugin that consumes an XML string and emits an instance of the IR. However, this IR instance does not yet match a modeling language's metamodel. The XML parser only performs a number of modeling language-independent tasks. First, it creates an IR node for every XML element in its input. Next, it registers the node of each element as a child of the element's parent's node. In addition, it parses the XML attributes of each element and creates a corresponding attribute for the element's IR node. By default, the parser ignores text XML nodes and delegates the handling of such nodes to concrete parsers through a configurable callback. The output of the XML parser are instances of IR models without any edges. The XML parser has two `boolean` parameters, `debug` and `strict`, that default to `false`. The values of both parameters are stored in IR models to be checked by other plugins that are invoked after

the XML parser. If `debug` is enabled, plugins may print debugging information useful for plugin development. `strict` is more important, as it signals plugins to perform input validation, as described for the IR post-processor in subsection 3.5.4.

3.6.2 UML Parser

The framework includes a parser for UML models in the serialization format used by the Eclipse Papyrus modeling software. It is a composition of the XML parser, a plugin referred to as a *metamodel refiner* for UML models, and the IR post-processor described in subsection 3.5.4. The concepts of the UML parser are presented below.

Pre-processing

In a first step, the parser removes a number of explicitly unsupported elements. These include certain XML extensions and legacy UML types that are not present in version 2.5.1 of the UML specification. Further, it generates an identifier for each IR node that does not yet have one. The reason for this is that some encodings require identifiers for all IR nodes to be present.

Refinement

Since the XML parser already creates IR instances with nodes, the task of the metamodel refiner is to create edges for UML associations. In addition, the UML parser is responsible for inferring the UML types of elements. To complete both tasks, it leverages a hierarchical execution of *handlers* for UML types. To refine an IR node, the refiner first determines which handler is applicable by searching a handler registry for a node's type or tag. If no handler is found, the element is removed in non-strict mode and an error is thrown in strict mode. Next, the handler for the node's UML type is invoked. Handler's are implemented on a type-basis and thus know which attributes and associations an element of their type should and can have. Handlers set default values for attributes that are not yet present on an IR node.

Further, handlers resolve all corresponding IR nodes for associations their respective type might have according to the UML metamodel. Associations may be resolved through XML attributes or tags of child elements. For example, the `importedElement` association of an `ElementImport` instance may be referenced through an XML attribute of the same name or by one of its child elements having a tag of the same name. Refiners resolve elements using both approaches at the same time. In the case of references via XML attributes, the refiner removes the XML attributes as they are not valid UML attributes according to the specification. While resolving a given association, a handler has knowledge about the expected type of resolved elements according to the metamodel. At this point it attempts to infer or narrow types for resolved elements. If a resolved element has no type information attached, the expected type of the resolved association is assigned. On the other hand, if a resolved element has a type but it is a generalization

of the resolved association's type, it will be overwritten with the inferred type because the inferred type is more specific.

Once a handler has resolved all associations and inserted default values for missing UML attributes, it invokes the handlers of UML generalizations of its own type. E.g., the handler for the `Namespace` type invokes the `NamedElement` handler, which invokes the root handler for the *Element* type.

A number of UML associations can only be resolved after all IR nodes have been refined. These include `inheritedMember` and `importedMember` due to their transitive nature. These are resolved iteratively until no further UML associations are created. Since UML package merges are mostly used by metamodel builders, not recommended for regular modeling, and the matching, i.e., merging, rules for elements are often ambiguous, package merges are not resolved by the parser [DDZ08].

Filters

The UML parser offers customizable whitelists and blacklists for allowed UML classes, attributes, and associations. By default, all six of these lists are empty and, thus, ignored. A node, an attribute, or an edge is allowed by a blacklist if its type, name, or tag is not present in the list. Similarly, a node, an attribute, or an edge is allowed by a whitelist if the whitelist is empty or contains the type, name, or tag respectively. A node, attribute, or edge will only be included in the parser's output if it passes both the applicable blacklist and whitelist.

External References

UML models may contain external references to, e.g., other UML files. We ignore and remove such references, e.g., by deleting `href` attributes from elements. This is most relevant for UML profiles, a mechanism used to customize the UML modeling language. Profiles and the diagram-visualization UML classes presented in Annex B of the UML specification [Obj17] are not supported by the framework. However, the parser is able to handle all other UML classes listed in the core specification, totaling 193 classes and 49 abstract classes.

Validations

In its strict mode, the parser does not accept invalid or unknown input data. In addition, some validation checks are executed to ensure that input models are well defined. While not guaranteed, in non-strict mode the parser can also handle invalid models or ones using a different serialization format that is similar to that of Eclipse Papyrus. In such cases, the invalid data is ignored and removed from the IR.

Parameters

A summary of the UML parser's configurable parameters is given in Table 3.1. Through composition, it also inherits the parameters of the IR post-processor described in subsection 3.5.4.

Parameter	Type	Default Value	Description
`debug`	`boolean`	`false`	If enabled, log debugging information.
`strict`	`boolean`	`false`	If enabled, perform validations and do not accept invalid models.
`onlyContainment-Associations`	`boolean`	`false`	If enabled, do not create IR edges for non-containment associations.
`relationships-AsEdges`	`boolean`	`false`	If enabled, transform IR nodes of UML relationships to edges.
`nodeWhitelist`	`list<string>`	`[]`	Whitelist of UML element types to include. Root nodes will never be removed. Ignored if empty.
`nodeBlacklist`	`list<string>`	`[]`	Blacklist of UML element types to exclude.
`edgeWhitelist`	`list<string>`	`[]`	Whitelist of association tags to include. Ignored if empty.
`edgeBlacklist`	`list<string>`	`[]`	Blacklist of association tags to exclude.
`attributeWhitelist`	`list<string>`	`[]`	Whitelist of attribute names to include. Ignored if empty.
`attributeBlacklist`	`list<string>`	`[]`	Blacklist of attribute names to exclude.
`randomizedIdPrefix`	`boolean`	`false`	Use a randomized prefix for generated ids, instead of "`eu.yeger`".

Table 3.1: Configurable parameters of the UML parser.

Test-Driven Development

In an automated unit test suite, the parser is invoked for every model of the dataset described in section 2.2, facilitating TDD. First limited to a batch of 100 models, increasing the number of included models step-by-step enables an iterative implementation of the UML parser. By failing tests for models who have elements with no assigned type, the TDD approach highlights handlers with missing type inference. With the additional validations performed by the UML parser, this approach also identifies unresolved associations. If an association remains unresolved, either an unknown attribute used for this association or child XML elements with a respective tag are present. This knowledge may then be used to identify and extend the handler responsible for such an unresolved association.

In this test suite, all 46,731 UML models are parsed four times with different parser configurations. We test all combinations of values for the two `boolean` parameters `onlyContainmentAssociations` and `relationshipsAsEdges` and save all four

encodings for each model as a snapshot. Through these snapshot, the unit tests ensure no unexpected changes to the parser are introduced by accident.

During the TDD, we also identified 78 invalid models in the dataset. The two most common reasons for a model being invalid are an identifier being used twice or an element having two generalization associations to the same target element. We are able to use these invalid models to also test the strict mode's validations.

3.7 Deduplication

López et al. [Lóp+22] describe the occurrence of duplicate conceptual models in datasets as a threat to the accuracy of results of ML evaluations. In particular, the presence of duplicates may result in an increased accuracy for ML architectures that resemble memorization approaches. This may skew results in favor of such architectures. As such, the thesis' literature review identified deduplication as an important feature for the CM2ML framework. We thus provide a composable plugin `deduplicate` that is automatically applied to all built-in encoders. While disabled by default, the `boolean` configuration parameter `deduplicate` may be activated to filter all duplicate output entries. Only the first occurrence for each group of duplicated entries remains in the output. For the built-in encoders, the deduplication is the last step of the encoding process.

Again, taking separate executions of encoding processes for different datasets into account, the `deduplicate` plugin can also be configured to deduplicate batch data based on one or multiple previously processed batches. For this, the `deduplicationData` parameter accepts a list of strings, which must be serializations of framework output in batch mode. For the current batch, the plugin will then also remove duplicates of any those previously generated output values.

3.8 Architecture

The CM2ML framework's architecture is shown in Figure 3.1. In this context, a workflow is a prepared plugin, i.e., a composition of a parser plugin, an encoder plugin, and the deduplication plugin.

In a typical usage flow, a user first submits input, i.e., a batch of conceptual models and parameters for configuration to an adapter. After reading user input, an adapter will start the workflow execution by invoking a prepared plugin associated with the user's input. For example, the CLI adapter invokes the plugin whose name is associated with a command entered by a user and the REST adapter selects the correct plugin according to which endpoint which is called. The invocation of a prepared plugin automatically validates parameters provided by users. Regardless of which parser is contained within the invoked plugin, it emits instances of the IR. These IR models are then passed on to the encoder of the prepared plugin, which consumes them to create an encoding. As the

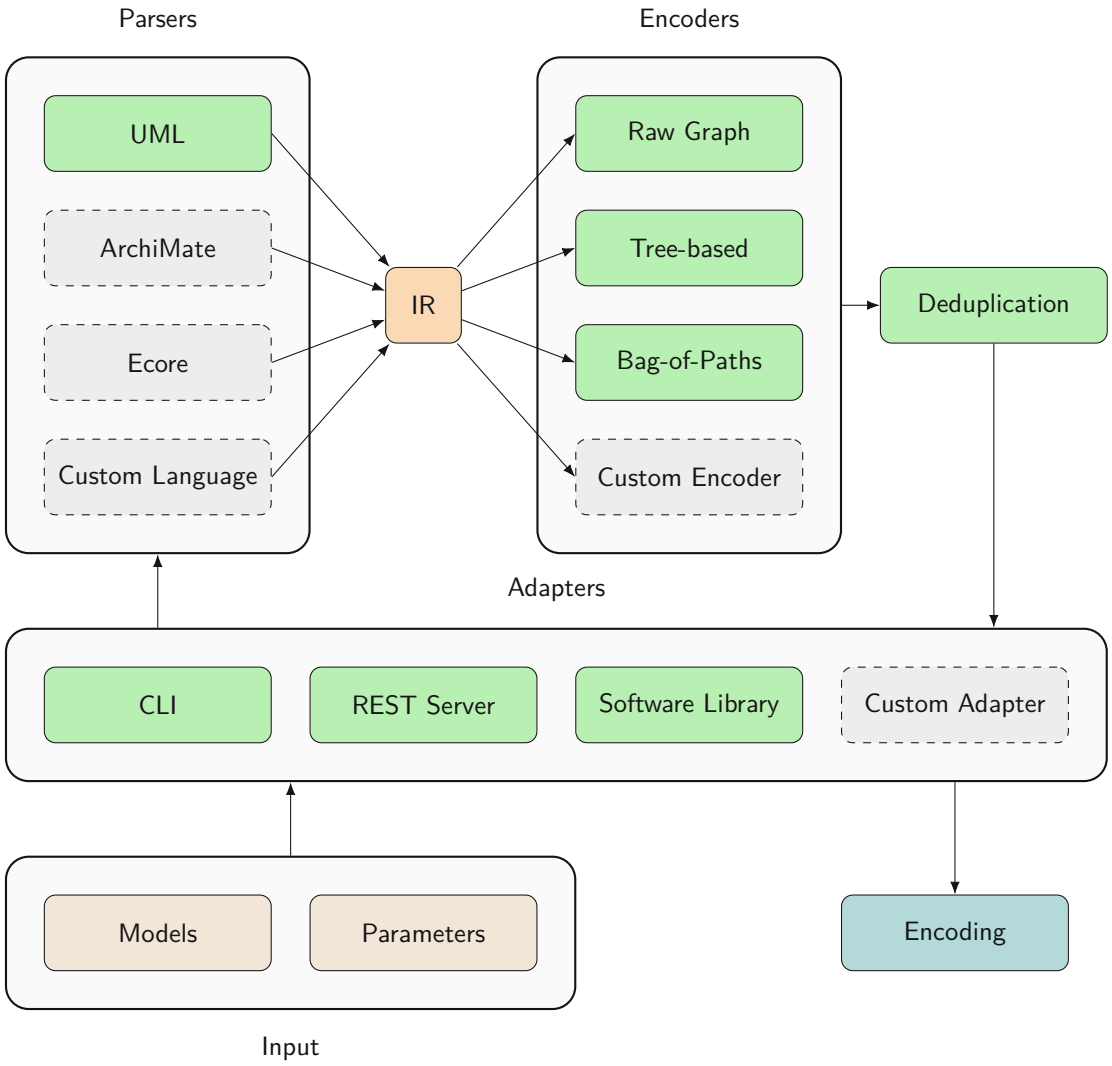


Figure 3.1: Architecture of the CM2ML framework. Components with dashed borders (gray) are not implemented and represent possible extensions.

second to last step, the deduplication plugin will, if enabled, remove any duplicates in the encoding batch. Finally, an adapter will handle plugin output, e.g., perform serialization or report errors.

CHAPTER 4

Visualizer

In order to aid the understandability of the implemented encoders and to fulfill the traceability requirement [**REQ6**], we implement a visualization frontend in form of a local-first web application[1]. After detailing the technology and state management used for implementing the visualizer, this chapter presents its user interface. Finally, the testing approach of the visualizer is briefly described.

4.1 Technology

To enable a fast development, we use React [Met] as our frontend framework and *shadcn/ui* [sha] as a library for customizable and accessible user interface components. The visualizer is built as a progressive web application (PWA) with client-side rendering and no backend connectivity. Because the visualizer uses the CM2ML framework as a software library, both the parsers and encoders can be executed client-side in the browser. This allows the visualizer to be installed as a desktop or mobile app using browsers such as Google Chrome or Safari. Once installed, the visualizer works without an internet connection.

4.2 State Management

The visualizer's entire state is stored on-device using the `localStorage` application programming interface (API). It uses *zustand* [Hen] for state management, as this complex part of the application is shared in a number of places. The application has four different categories of state. First, it stores user settings such as the theme, i.e., light or dark mode, the selected view for the IR and the active layout. Next, the model state includes, if available, a selected parser, the serialized input model, and the parameter configuration

[1] `https://jan-mueller.at/external/cm2ml`

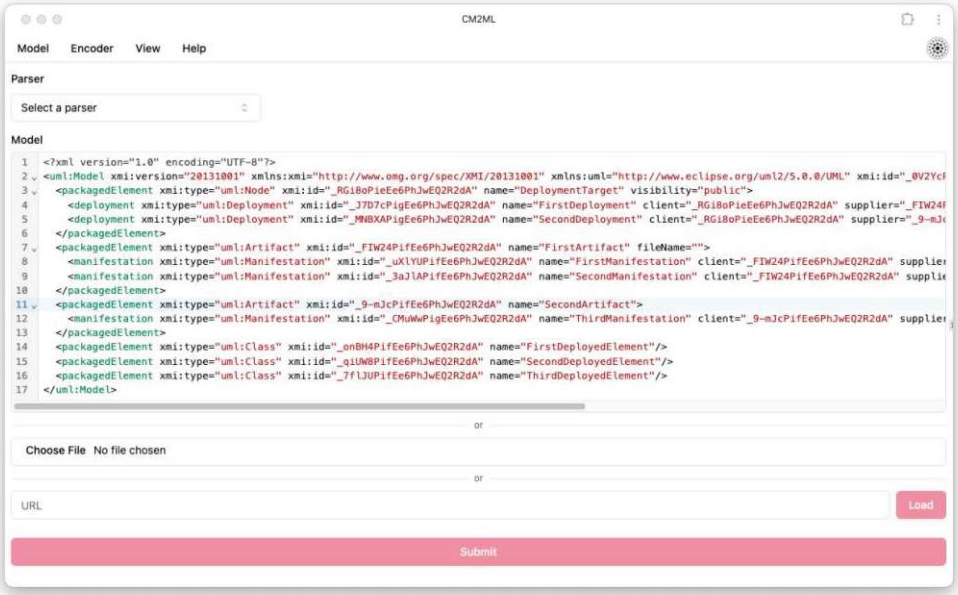


Figure 4.1: Input methods for conceptual models in the visualizer.

provided by users. Analogously, the encoder state contains a selected encoder and its parameter configuration. Finally, the selection state, which is not persisted across sessions, contains information about any active selection of model elements, i.e., nodes or edges. This information is used to synchronize selections between IR and encoder views.

4.3 User Input

The visualizer enables users to configure both input models as well as parameters of parsers and encoders interactively. In the following subsections, both areas are detailed.

4.3.1 Model Inputs

The visualizer offers three choices for configuring models. First, a syntax-aware editor, depicted in Figure 4.1, may be used to either enter or paste serialized models. While the syntax highlighting may depend on the selected modeling language, the visualizer currently supports XML as it's used by Eclipse Papyrus' UML format. XML is also used by default if no language parser is selected.

Below the textual input, users may alternatively choose a model file using their browser's built-in file selection or specify a remote URL from which a model should be retrieved. For both of these options, the serialized models will be shown in the text editor after reading an input file or retrieving it from the given URL.

4.3.2 Parameter Inputs

An input form for parameters of both parsers and encoders is derived automatically from their parameter declarations. Within these forms, parameters are grouped and ordered alphabetically. The user interface for boolean, string, and numeric parameters are simple checkboxes and HTML inputs. For string parameters with a list of allowed values, a dropdown with the available choices is shown instead.

The user interface for list parameters is more complex, depending on whether such a parameter has a list of allowed values or is configured as ordered. For the former, users are shown a dropdown for adding new values from the available choices to their selected list, as indicated in Figure 4.2a. This dropdown also supports filtering the set of allowed values. Further, values of ordered list parameters may be reordered with drag-and-drop gestures, as shown in Figure 4.2b. If a list is not restricted to a list of allowed values, users may also edit list items inline. Each value may be removed individually or the entire list can be cleared at once. The buttons for both removal options as well as the highlighted inline editing are shown in Figure 4.2c.

4.4 IR Visualizations

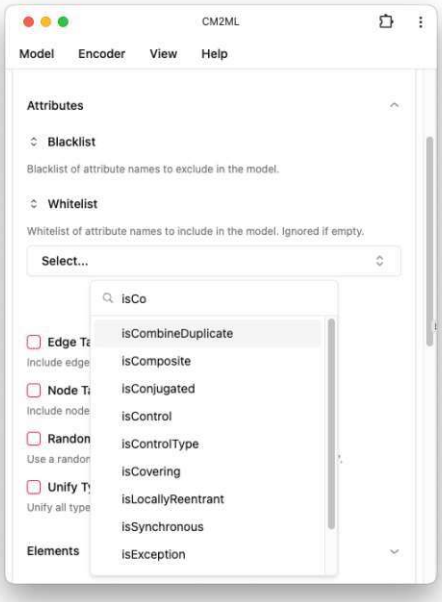
Because the IR supports two different perspectives on conceptual models, i.e., graph and tree views, the visualizer also offers two visualizations for IR instances. Beyond that, it shows metadata about conceptual models and detailed information for any selected nodes or edges in its details panel.

4.4.1 Graphs

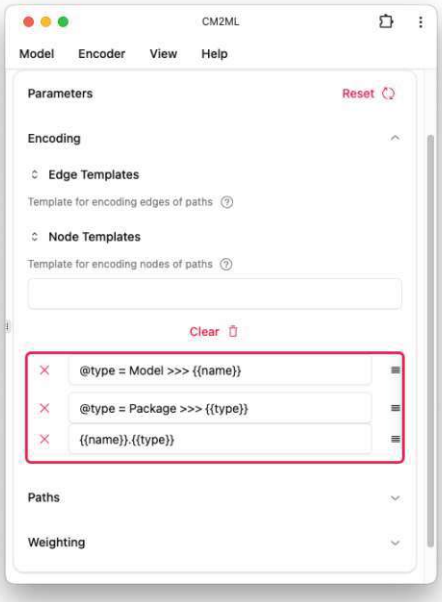
The graph visualization of IR models is shown in Figure 4.3. It is built using the *vis-network* [Alm] library and, thus, fully interactive with support for panning, zooming, and moving nodes. Nodes and edges of the IR correspond to nodes and edges in the visualization respectively. In this view, all edges between two IR nodes are represented as a single, undirected connection. This decision is based on readability issues caused by a large numbers of edges between two nodes. The labels of nodes consist of, if available, a node's name followed by its tag, i.e., its type in the case of UML models. The shape of nodes is determined by their number of connections. If a node has four or more neighboring nodes, it is displayed with a circular shape. Otherwise, a more compact rectangular shape is used. Clicking on nodes or edges selects them, while dragging a node moves it around the view.

4.4.2 Trees

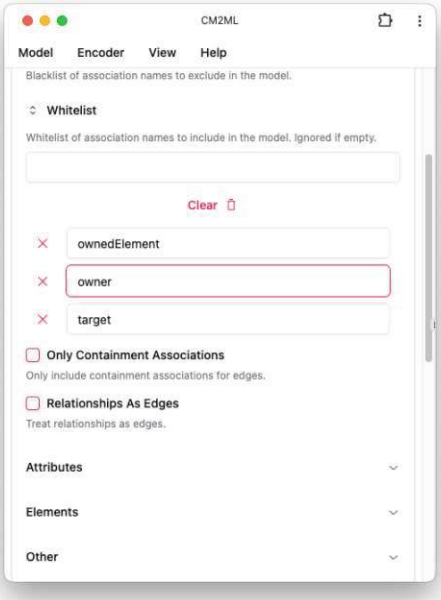
The tree visualization of IR models, shown in Figure 4.4, does not include IR edges. Instead, it depicts IR nodes in a hierarchy that is based on the tree structure of an IR instance. Implemented with *React Flow* [web] its does not allow for nodes to be rearranged as an optimized hierarchical layout is calculated by the visualizer instead.



(a) Dropdown for selecting list parameter values from a list of allowed values.



(b) Reordering values of ordered list parameters with drag-and-drop.



(c) Inline editing of list parameter values without a restriction to allowed values.

Figure 4.2: User interface for list parameters.

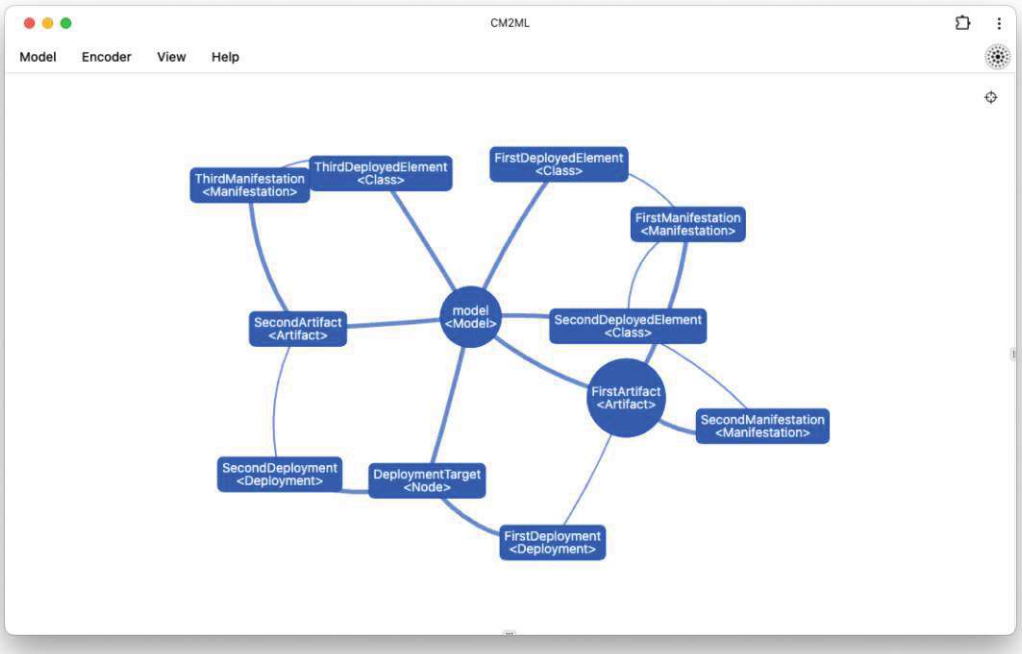


Figure 4.3: Graph visualization for IR models.

The labels of nodes in the tree view are identical to those of the graph view, and clicking on a node selects its corresponding IR node analogously.

4.4.3 Details Panel

The details panel can either show selected nodes, selected edges, or metadata about the current IR model if no selection is made. The last case is depicted in Figure 4.5a. For all modeling languages, the metadata includes the number of nodes, edges, and attributes across the entire IR instance. The right-hand-side column of metadata is parser-dependent and customizable by parser authors. For the UML parser, a model's name as well as the versions of the UML and XMI specification are displayed.

An example for detailed information about a selected node is displayed in Figure 4.5b. This view lists all attributes of a node alongside their values. Below that, it lists a node's parent and children, followed by its two groups of outgoing and incoming edges. Both edges and all references to nodes, i.e., parents, children, and nodes connected through edges, are highlighted in blue. Clicking on such a reference discards the current selection and selects the node or edge corresponding to the reference instead. This enables a fast exploration of model elements and their detailed information.

An example of the edge details view is shown in Figure 4.5c. This view groups edges

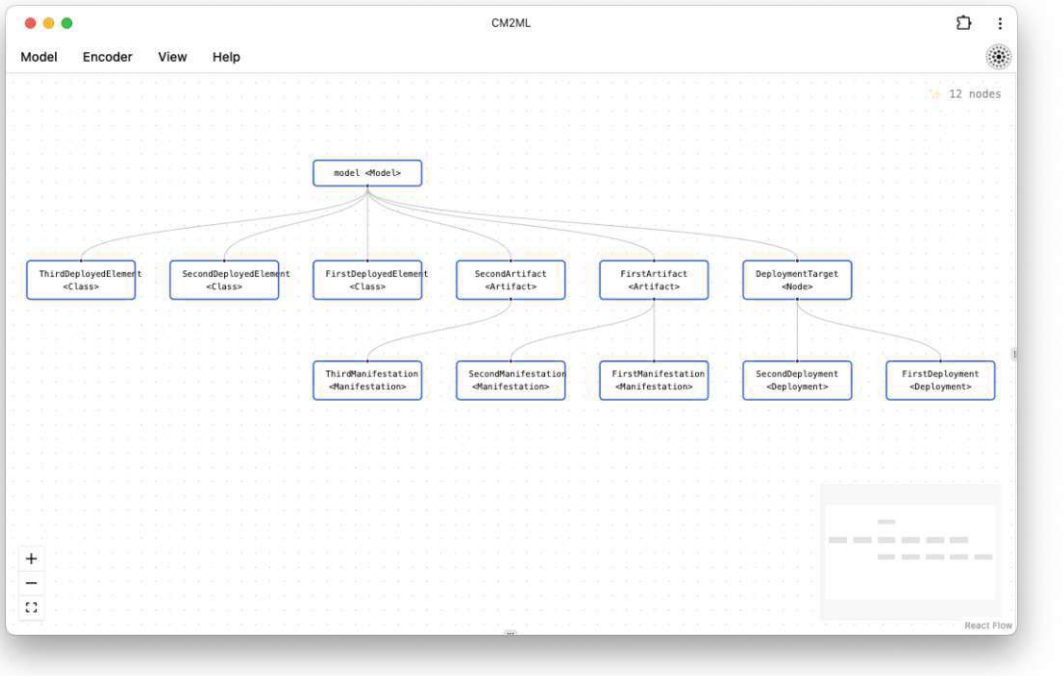


Figure 4.4: Tree visualization for IR models.

by their source and target, hence both edges between the two nodes of the example are included there. Below the clickable references to source and target nodes with the same functionality as the references of the node details view, attributes and their values are listed.

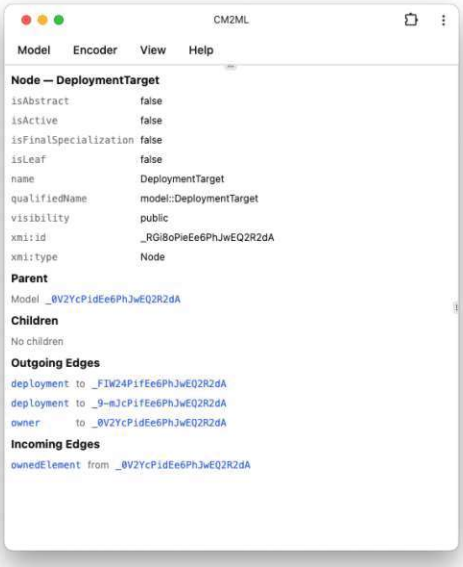
4.5 Layout

To accommodate as many client devices as possible, the entire visualizer is implemented using a responsive and resizable layout. Users may select between *compact* and *extended* layouts. Both layouts consist of panels that are split horizontally and vertically. Each panel is independently resizable to allow for customization of the user interface.

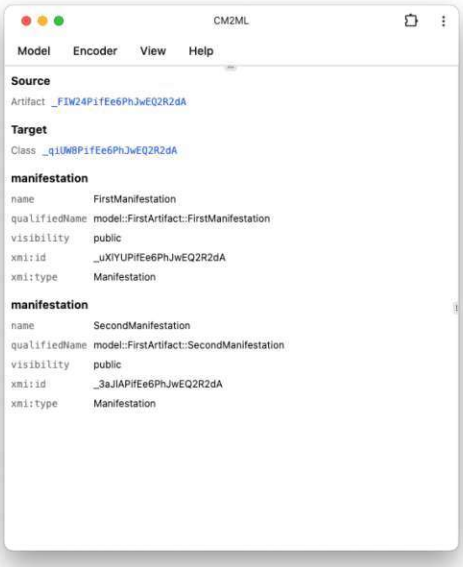
The extended layout has three columns, as shown in Figure 4.6a. The left column contains the model input and parser configuration, followed by two vertical panels for the IR visualization and details panel in the middle column. The right-most column contains the encoder configuration and visualization, again distributed across two panels. This layout enables experimentation with parser and encoder parameters with immediate feedback through updated visualizations.

On the other hand, the compact layout, visible in Figure 4.6b, is optimized for smaller screens. It only uses two columns and requires users to switch between input forms and

(a) Metadata about the current IR model.



(b) Detailed information about a selected node.



(c) Detailed information about selected edges.

Figure 4.5: Details panel for IR metadata, selected nodes, and selected edges.

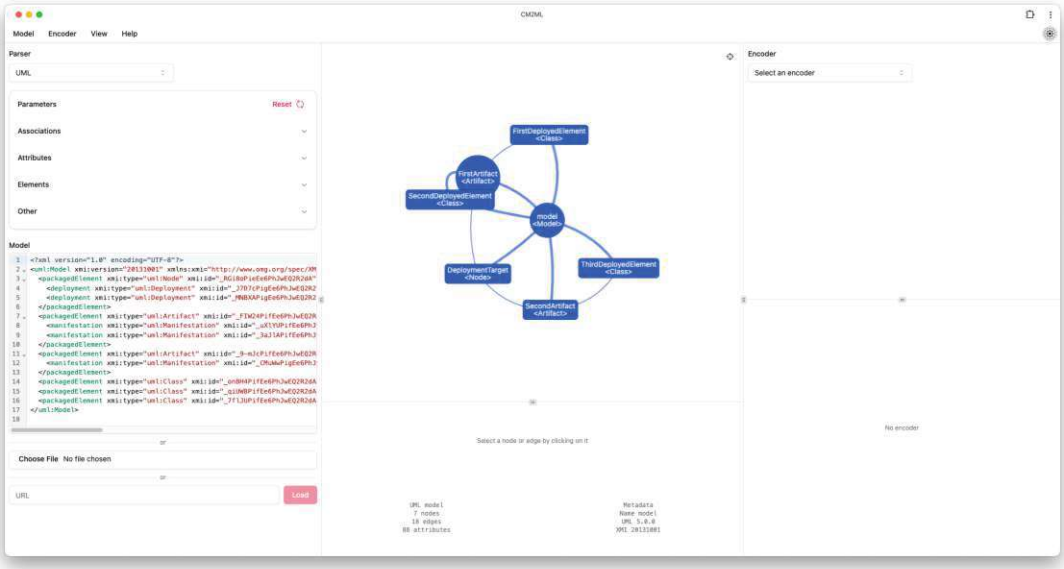
visualizations. The left column either shows the model input and parser configuration form or the IR visualization and details view in two vertical panels. Analogously, the right column shows the encoder configuration or the visualization for the encoding. As a downside, users are not able to see changes to parameter configurations immediately and must submit their input first.

4.6 Encoding Visualizations

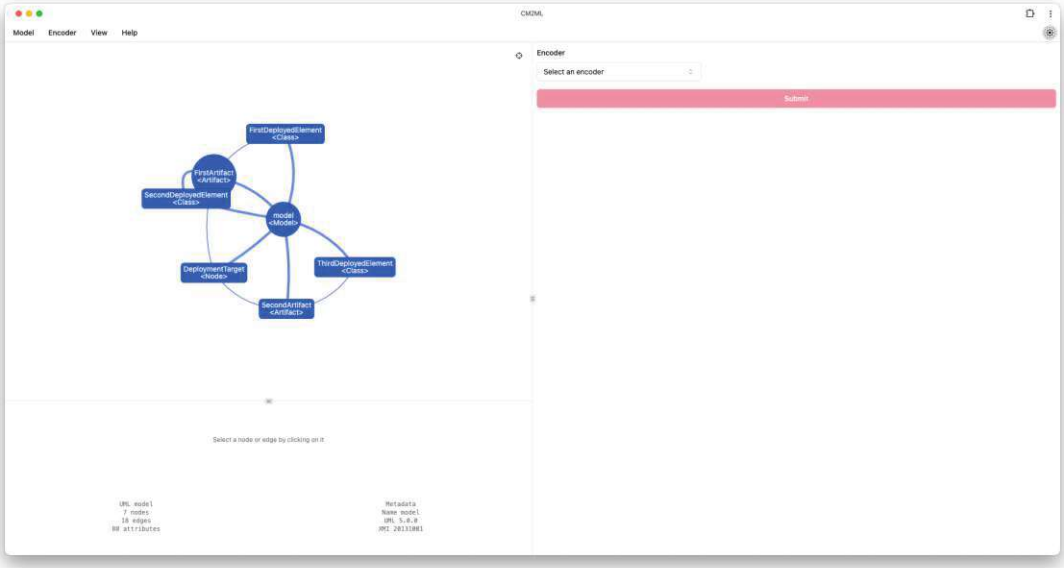
Each built-in encoding of the framework has its own visualization. Inspired by the usage of interactive visualization by Sanchez-Lengeling et al. [San+21], the encoding visualizations each show to which element of an IR instance a part of an encoding corresponds to, i.e., they provide traceability between a model and a model encoding. This is achieved by encoders emitting additional mapping data where required. This mapping data in turn is used to bidirectionally translate selections of IR elements to selections in an encoding visualization and vice versa. E.g., users are able to select a node or edge in the IR graph view and, if it exists, a corresponding part of an encoding will be selected and highlighted as well. The same holds for the other direction. If an encoding has multiple representations for an element of an IR model, this mechanism will select all matching elements. The concrete visualizations for each encoder are detailed in the encoder's respective section. Visualizations for adjacency matrices and lists of the raw graph encoding are presented in subsection 5.2.1 and subsection 5.2.2 respectively. The tree's of the tree-based encoder are detailed in subsection 5.3.1. Finally, the BoP encoder's paths are shown in subsection 5.4.3. The output of encoders without a dedicated visualization is presented in a readonly JSON text view that offers syntax highlighting analogous to the model input form's editor.

4.7 End-to-End Testing

We implement E2E tests for the visualizer using *Playwright* [Mica], a framework for automated user interface testing. Playwright interacts with the user interface like a human user would. Because the visualizer's core functionality is providing traceability **[REQ6]**, the E2E tests ensure that the synchronization between selections in the two IR views and all three encoder visualizations works bidirectionally. Since each E2E test includes a configuration of both parser and encoder parameters, the dynamically generated parameter forms and their input components are covered by the tests. Twenty-two tests are executed with Chromium, Firefox, and Webkit rendering engines, totaling 66 test executions. As a result, the visualizer's cross-browser compatibility is validated.



(a) Extended layout with three columns.



(b) Compact layout with two columns.

Figure 4.6: Details panel for IR metadata, selected nodes, and selected edges.

CHAPTER 5

Encoders

This chapter details the implementation of the raw graph, tree-based, and BoP encoders. They are implemented as plugins that operate on batches of the framework's IR format. Further, this chapter presents a dedicated plugin for encoding features, i.e., attributes.

5.1 Feature Encoder

To facilitate the framework's modularity, a dedicated plugin for feature encoding is implemented. It operates in a two-stage process. The first stage collects all values of each attribute in all models of a batch. Based on this information, the feature encoder determines how categorical attributes should be encoded. The second stage happens in subsequent plugins that use functions provided in the feature encoder's output. These functions support the creation of feature vectors for IR nodes and edges, as well as encoding the values of individual attributes.

5.1.1 Encoding Types

Each of the IR's six types for attribute values can be encoded as described below.

integer and float

Integer and float attribute values are not encoded, as they are immediately usable as input for ML models.

boolean

Boolean values use the common encoding of mapping `true` to 1 and `false` to 0. This approach is aligned with programming languages such as C, JavaScript, and Python.

category

Categorical attributes have textual values with a finite domain, e.g., behave like an enumeration. The encoding process for categorical values leverages the two-stage encoding process. During the collection stage, all values for each categorical attribute in a batch of IR models are enumerated and an index is assigned. Then, the encoding functions output by the feature encoding plugin can encode categorical values by replacing them with their respective index. This method of encoding categorical values is also known as label encoding. Since the domain of categorical attributes is finite, the maximum index for any categorical value does not increase with batch size.

string

While string values are similar to categorical values, their infinite domain poses a problem for label encoding. Namely, the unlimited vocabulary problem may occur with increasing batch sizes [Bur+22]. As a workaround, semantic embeddings of text may be used. However, these reach into the realm of semantic encodings for conceptual models, which are not within the scope of the thesis. To keep the structural encodings simple, the framework offers the option to treat values of string attributes as categorical values and use a label encoding, ignoring the unlimited vocabulary problem. However, configuration parameters of the feature encoder plugin also allow users to either output string values in their raw, un-encoded form, or omit them entirely.

unknown

Some attributes, and thus also their types, may not be known to a modeling language's parser implementation. Since values of such attributes cannot be sensibly encoded, the feature encoding plugin throws an error if any are encountered. This approach also has the benefit of guiding the development of parsers for additional modeling languages by highlighting missing metamodel information.

5.1.2 Encoding Alignment

One issue of this approach for feature encoding is a possible misalignment of feature vector size, layout, and attribute value encoding. For example, consider two batches, with the first batch containing model elements with attributes as well as categorical attributes with values that both do not occur in the second batch. Then, feature vectors for elements of the second batch will not match those of the first batch, due to an attribute not being present at all. If the second batch also has attributes not occurring in the first batch, it may match the feature vector shape but have entirely different attributes for some given indices. Further, the presence of different categorical values in the second batch will result in a different value enumeration and thus an encodig mismatch for those values.

To circumvent this issue, the feature encoding plugin offers two parameters for injecting feature encoding metadata generated by previous executions of the plugin. One parameter

is for attributes of IR nodes, the second for attributes of edges. This metadata is a serialized JSON object that contains a list of attributes for feature vectors, as well as known values for categorical features. If one of those parameters is configured, features will be encoded with two restrictions. First, only attributes also present in the metadata will be included. Additional attributes will be ignored to maintain feature vector shape. Attributes that are part of the metadata, but not present in a model element will be encoded as 0. Next, the known values for categorical attributes are included during the collection stage and prepended to the enumeration. This ensures that no mismatch can occur for categorical values.

5.1.3 Standalone Usage

The feature encoder may also be used as a standalone encoder. Its output is functionally identical to that of the raw graph encoder, without any adjacency information related to edges being present. No dedicated visualization is implemented for this standalone encoder.

5.1.4 Parameters

The configuration parameters of the feature encoder are listed in Table 5.1. Beyond the two parameters for alignment, the encoder's parameters are designed to enable filtering attributes by their type. A possible strategy is enabling `onlyEncodedFeatures` and any of the `raw` parameters to exclude an entire group of parameters.

5.2 Raw Graph Encoder

The implementation of the raw graph encoder is based on the concept of the raw graph encoding presented by López et al. [Lóp+22]. Its output consists of two parts. First, feature vectors for nodes and attributes of IR instances are created using the feature encoder detailed in section 5.1. To match feature vectors to their corresponding IR nodes, the latter are first enumerated and sorted by their identifiers. While this does not fix the problem of permutation invariance [San+21], it ensures a stable order and thus deterministic output. Feature vectors may then be matched to their nodes by their indices. In addition, adjacency information is captured in a user-selectable format. The encoder offers two formats, which are based on the work of Sanchez-Lengeling et al. [San+21]. The raw graph encoder supports optional weighting for both formats. An edge's weight is in the range $(0, 1]$ and anti-proportional to the number of incoming edges of its target node. In other terms, the higher the number of edges sharing a single target, the lower each edge's weight is.

5.2.1 Adjacency Matrix

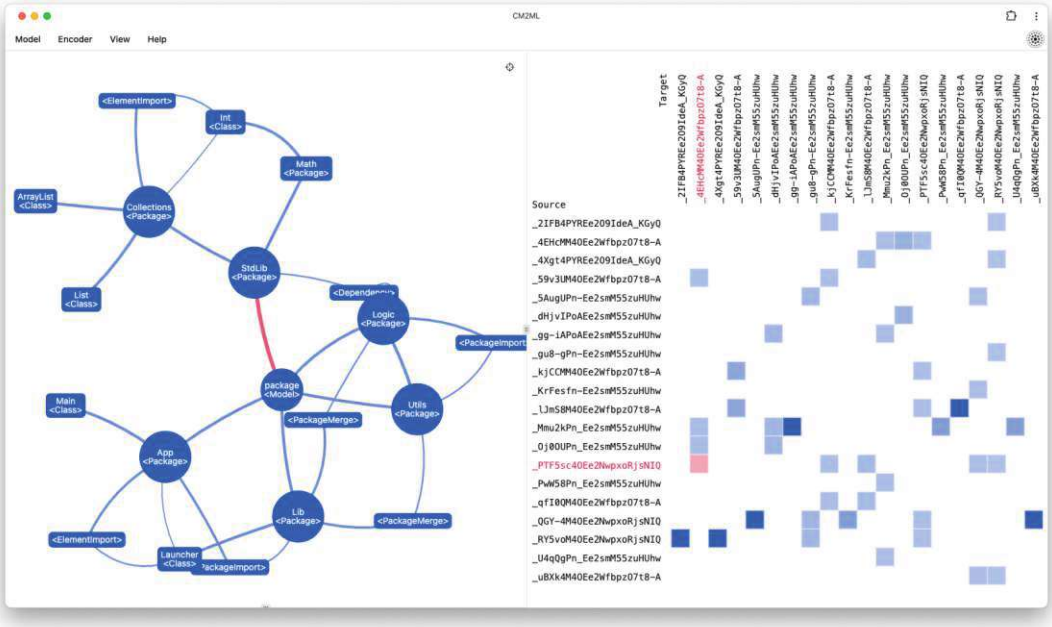
An adjacency matrix of an IR model with n nodes has $n \times n$ cells. The nodes of an IR model are enumerated and their indices correspond to rows and columns of an adjacency

Parameter	Type	Default Value	Description
`rawFeatures`	`boolean`	`false`	If enabled, no attribute will be encoded.
`rawBooleans`	`boolean`	`false`	If enabled, `boolean` attributes will not be encoded.
`rawCategories`	`boolean`	`false`	If enabled, `category` attributes will not be encoded.
`rawNumerics`	`boolean`	`false`	If enabled, `integer` and `float` attributes will not be encoded.
`rawStrings`	`boolean`	`false`	If enabled, `string` attributes will not be encoded.
`onlyEncoded-Features`	`boolean`	`false`	If enabled, un-encoded attributes will be ignored.
`nodeFeatures`	`string`	—	Serialized feature metadata as described in subsection 5.1.2.
`edgeFeatures`	`string`	—	Serialized feature metadata as described in subsection 5.1.2.

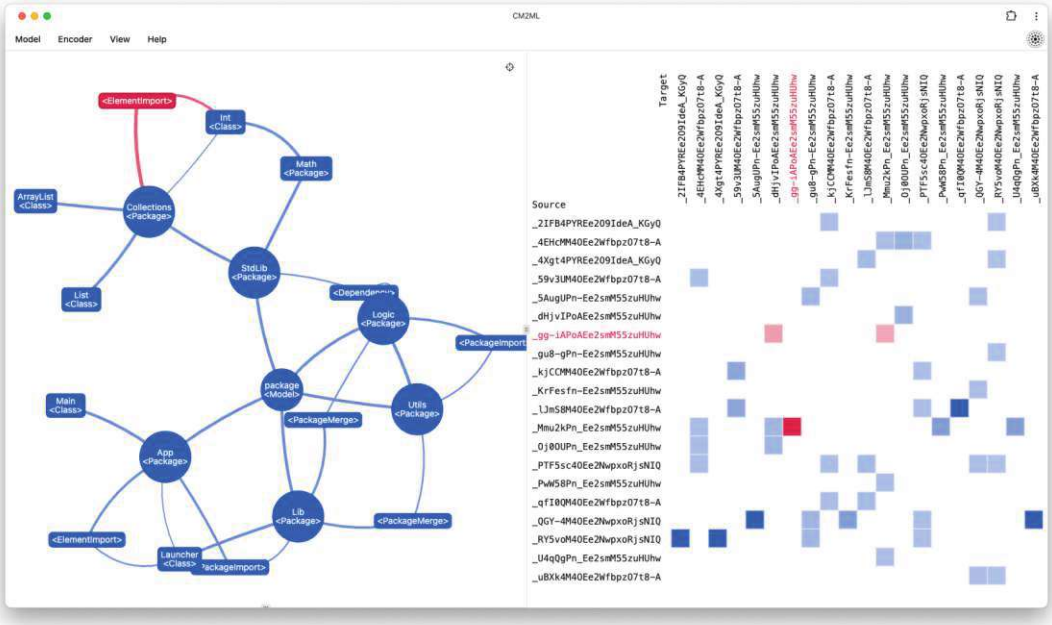
Table 5.1: Configurable parameters of the feature encoder.

matrix. Each cell is a numeric value that indicates if the node associated with its row has an outgoing edge to the node associated with the cell's column. A cell value of 0 represents no connection between two nodes, while a value of 1 indicates a connection. If weighting is enabled, connection weights will be used instead of the fixed value of 1. The adjacency matrix format has a number of drawbacks. First, the matrices are not memory efficient for graphs with few connections [San+21]. E.g., a graph with n nodes and just $n - 1$ edges still has an $n \times n$ adjacency matrix. In addition, each cell of a matrix may only represent one edge between two nodes. Since the IR supports multiple edges between nodes, adjacency matrices suffer from a loss of information. Because of this loss of edge information and no inherent order of edges, the raw graph encoder does not emit edge feature vectors if this format is selected.

The visualization for weighted adjacency matrices is depicted in Figure 5.1. The saturation of a cell's color indicates the weight of its corresponding edge, with a less saturated cell having a lower weight. Figure 5.1a shows how an edge selection in the matrix is reflected in the IR view by highlighting the matrix cell associated with an edge. Analogously, Figure 5.1b presents how a selected node results in all matrix cells corresponding to its outgoing and incoming edges being highlighted.



(a) Edge selection in adjacency matrix (right) and IR view (left).



(b) Node selection in adjacency matrix (right) and IR view (left).

Figure 5.1: Adjacency matrix visualization with selections.

Parameter	Type	Default Value	Description
`format`	`string`	`list`	The format of adjacency data. Either `list` or `matrix`.
`weighted`	`boolean`	`false`	If enabled, the adjacency data will include connection weights.

Table 5.2: Configurable parameters of the raw graph encoder.

5.2.2 Adjacency Lists

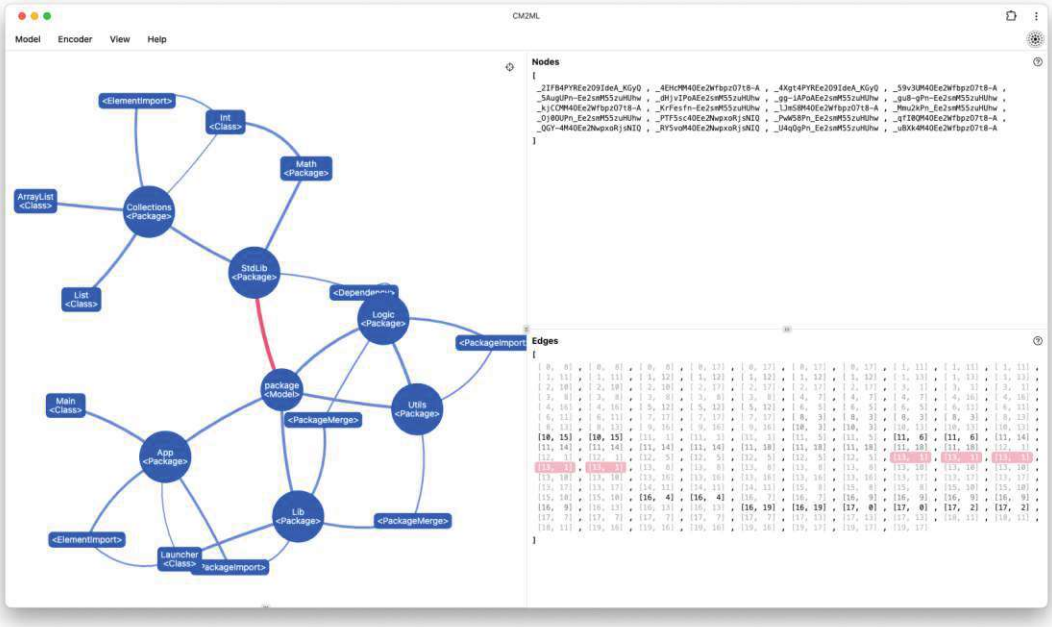
To circumvent the disadvantages of adjacency matrices, adjacency lists are used as the encoder's default format. These lists contain an entry for every edge of an IR model. Each entry is a tuple with two integer values that correspond to the indices of an edge's source and target nodes in an enumeration of a model's nodes. If weighting is enabled, an edge's weight is added as an entry's third value. One immediate advantage of adjacency lists is space efficiency [San+21]. A graph with n nodes and e edges has e list entries. Further, the concept of having one entry for each edge allows adjacency lists to contain separate entries for multiple edges between two nodes. This also allows edge feature vectors to be emitted. These can be matched to their corresponding edge through their indices within their respective lists, analogous to feature vectors of nodes.

Figure 5.2 shows an example for the visualization of adjacency lists. The top panel contains an enumeration of an IR instance's nodes, while the bottom panel shows an adjacency list. The list's entries contain the above described indices of its source and target nodes within the enumeration. If enabled, weighting is represented through opacity of list entries, with a higher opacity indicating a higher weight. Clicking on a list entry selects its corresponding edge in the IR. The reverse direction, i.e., clicking on a connection in the IR view, highlights all edges represented by the bidirectional connection, as shown in Figure 5.2a. Node selections are shown both in the node enumeration and by highlighting all entries for a selected node's outgoing and incoming edges, as visible in Figure 5.2b.

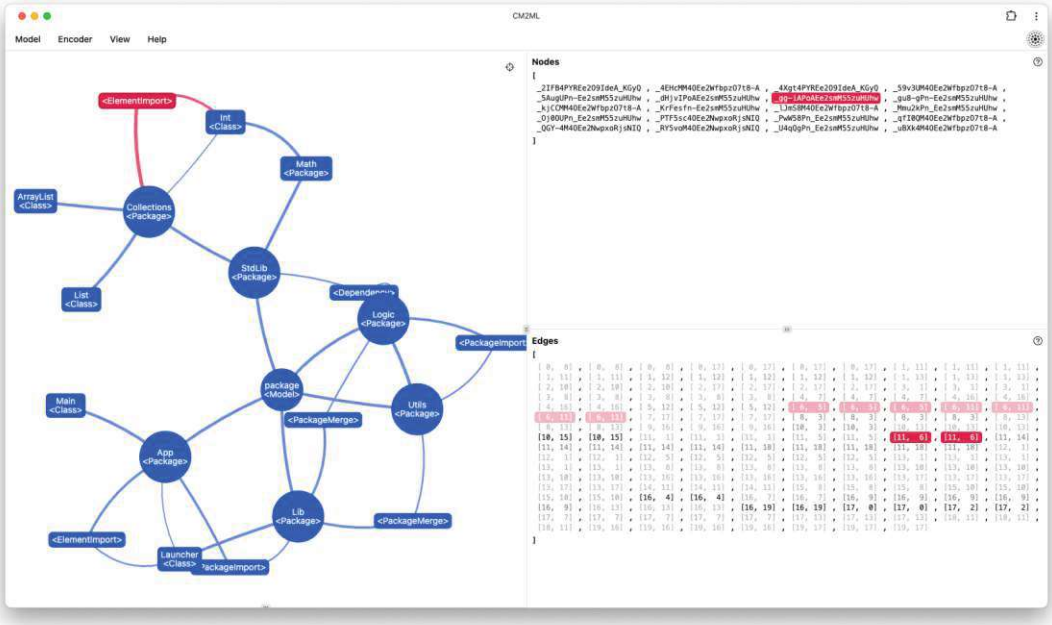
Further, the visualization includes feature metadata like the attribute names and types of features as well as concrete feature vectors as shown in Figure 5.3a. Finally, exact edge weights are also provided through tooltips as visible in Figure 5.3b.

5.2.3 Parameters

The raw graph encoder offers the two parameters detailed in Table 5.2 as well as all of the feature encoder's parameters described in subsection 5.1.4.



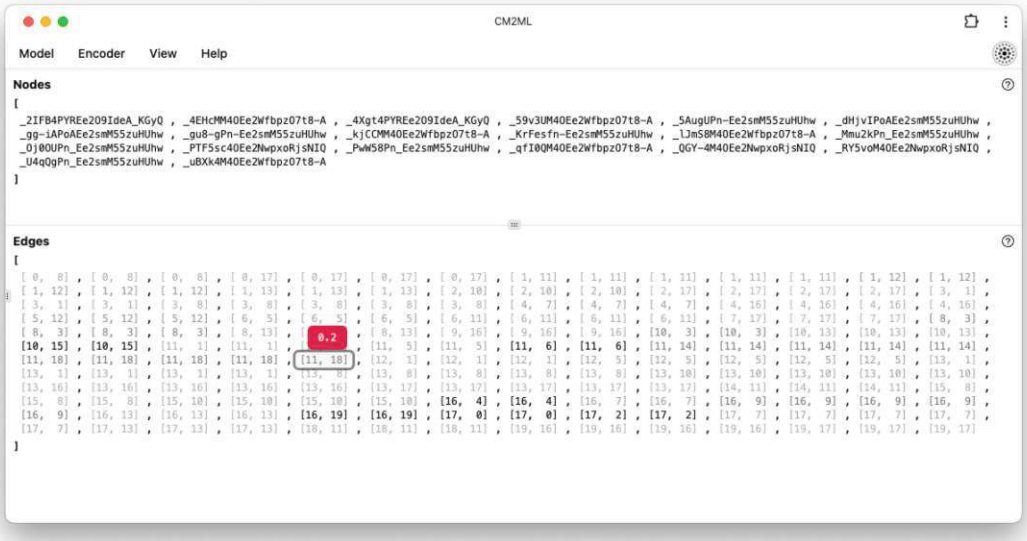
(a) Edge selection in adjacency list (right) and IR view (left).



(b) Node selection in adjacency list (right) and IR view (left).

Figure 5.2: Adjacency list visualization with selections.

(a) Tooltips showing node features (left) and feature metadata (right).



(b) Tooltip showing edge weight information.

Figure 5.3: Feature and weight tooltips.

5.3 Tree-based Encoder

The tree-based encoder, as the name implies, encodes models as trees. Each tree node has a string or numeric value and a variable number of children. The set of all values found in a tree's nodes is referred to as a vocabulary. Depending on the user-provided configuration, attribute values may be encoded by the feature encoder detailed in section 5.1. Encoded trees have a well-defined structure dictated by different formats, which are described below.

5.3.1 Formats

The tree-based encoder offers two encoding formats, `global` and `local`. The name of a format refers to the location of association nodes in trees. The visualization of the tree-based encoding is format-agnostic and implemented with reactflow, similar to the IR's tree view.

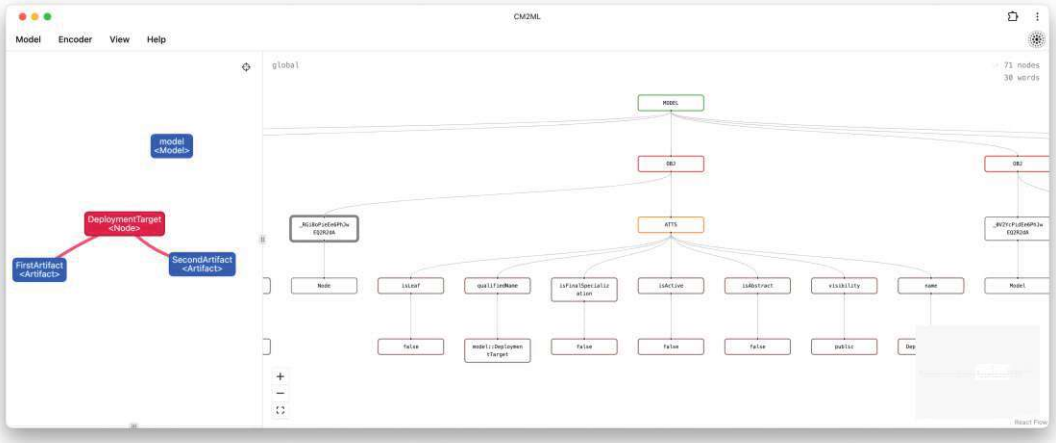
`global` Format

The `global` format creates trees in accordance with the work of Burgueño et al. [Bur+22]. Also described by the pseudocode in algorithm 5.1, the `global` tree format is generated as follows. First, a single root node with value `MODEL` is created. For each IR node of a model, a tree node with the value `OBJ` is added to the root. Next, a tree node containing the IR node's identifier is added to the `OBJ` node. This identifier node in turn receives a child node containing the IR node's type. Then, an `ATTS` node is added to the `OBJ` node. For each attribute of an IR node, excluding its type attribute, two tree nodes with the attribute's name and encoded value are created. The node containing the value is added to the tree node containing the attribute's name, which is added to the `OBJ` node of the IR node. Now, for each edge of an IR model, a node with value `ASSOC` is added to the root node. These `ASSOC` nodes have three child nodes, containing the identifiers of the edge's source IR node, target IR node, and tag respectively.

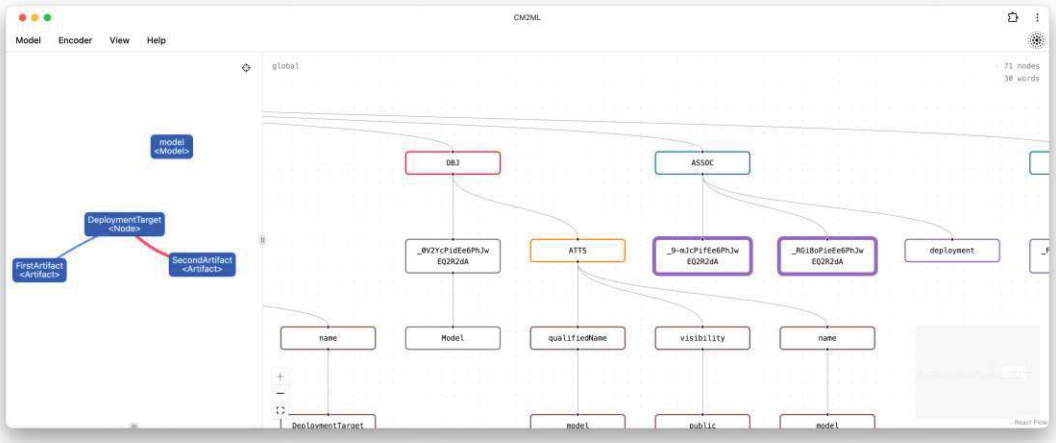
An example of the `global` tree format's visualization is given in Figure 5.4. First, Figure 5.4a shows how a selected node is highlighted by marking the identifier node in its corresponding subtree. Next, Figure 5.4b depicts a selected edge, which is highlighted by marking source and target nodes of its corresponding subtree. To aid navigation in large models, selecting a node or edge in an IR view automatically centers the marked elements in the tree-based encoding view.

`local` Format

The tree structure suggested by Weyssow et al. [WSS22] is made available, with slight modifications, as the `local` format. Instead of the combination of attribute type and name they use for their their tree format, CM2ML's `local` format uses attribute name and value analogous to the `global` format. This change is necessary because the work of Weyssow et al. is only concerned with trees representing metamodels.



(a) Node selection in `global` tree (right) and IR view (left).



(b) Edge selection in `global` tree (right) and IR view (left).

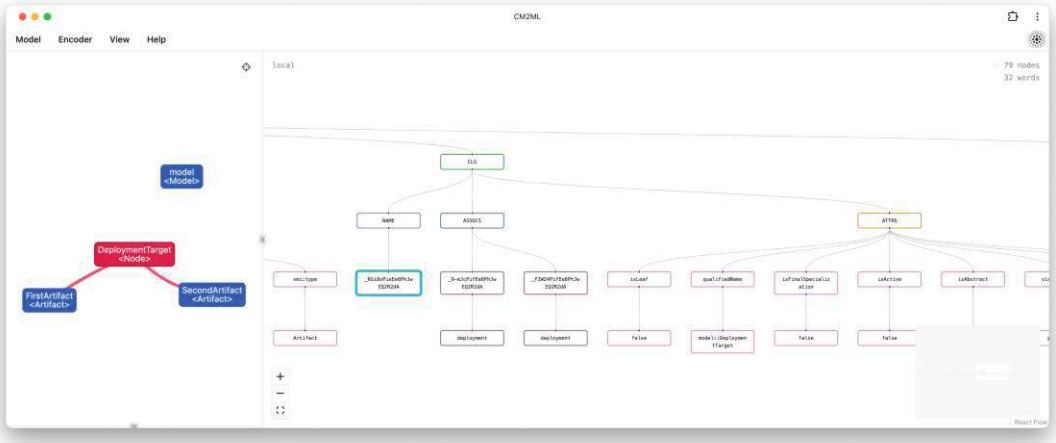
Figure 5.4: Visualization of a `global` tree with selections.

Algorithm 5.1: Pseudocode of the `global` tree format encoding.

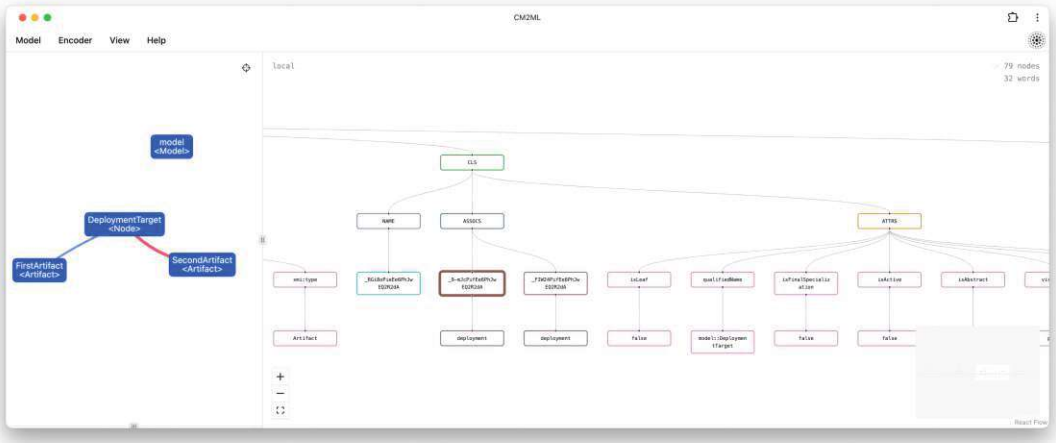
```
   Input: IR instance model
   Output: Root node of encoded tree
 1 define function createNode(value, parent)
 2
 3 rootNode ← createNode("MODEL", null)
 4
 5 foreach node ∈ model.nodes do
 6     objNode ← createNode("OBJ", rootNode)
 7
 8     idNode ← createNode(node.id, objNode)
 9     createNode(node.type, idNode)
10
11     attributesNode ← createNode("ATTS", objNode)
12     foreach attribute ∈ node.attributes do
13         if isIdentifier(attribute) or isType(attribute) then
14             continue
15         end
16         attributeNode ← createNode(attribute.name, attributesNode)
17         encodedValue ← encodeAttribute(attribute)
18         createNode(encodedValue, attributeNode)
19     end
20 end
21
22 foreach edge ∈ model.edges do
23     associationNode ← createNode("ASSOC", rootNode)
24     createNode(edge.source.id, associationNode)
25     createNode(edge.target.id, associationNode)
26     createNode(edge.tag, associationNode)
27 end
28
29 return rootNode
```

Trees of the `local` format are created according to the following steps, which match the pseudocode of algorithm 5.2. First, a root node with the value `MODEL` is created. For each IR node of a model, a tree node with value `CLS` is added to the root node. Three nodes, `NAME`, `ATTRS`, and `ASSOCS` are added to each `CLS` node. A tree node containing an IR node's identifier is added to the corresponding `NAME` node. For each attribute of an IR node, a tree node with the attribute's name is added to the IR node's `ATTRS` node. Further, a tree node containing the attribute's encoded value is added as the child to the above described node. For each outgoing edge of an IR node, a tree node with the edge's target identifier is added to the IR node's `ASSOCS` node. In addition, a node containing the edge's tag is added to the above detailed node.

The visualization of the `local` tree format is analogous to that of the `global` format. Both node and edge selections are shown in Figure 5.5a and Figure 5.5b respectively.



(a) Node selection in `local` tree (right) and IR view (left).



(b) Edge selection in `local` tree (right) and IR view (left).

Figure 5.5: Visualization of a `local` tree with selections.

Algorithm 5.2: Pseudocode of the `local` tree format encoding.

```
Input: IR instance model
Output: Root node of encoded tree
define function createNode(value, parent)

rootNode ← createNode("MODEL", null)

foreach node ∈ model.nodes do
    clsNode ← createNode("CLS", rootNode)

    nameNode ← createNode("NAME", clsNode)
    createNode(node.id, nameNode)

    attributesNode ← createNode("ATTRS", clsNode)
    foreach attribute ∈ node.attributes do
        if isIdentifier(attribute) then
            continue
        end
        attributeNode ← createNode(attribute.name, attributesNode)
        encodedValue ← encodeAttribute(attribute)
        createNode(encodedValue, attributeNode)
    end

    asssociationsNode ← createNode("ASSOCS", clsNode)
    foreach edge ∈ node.outgoingEdges do
        targetNode ← createNode(edge.target.id, asssociationsNode)
        createNode(edge.tag, targetNode)
    end
end

return rootNode
```

5.3.2 Attribute Encoding

Analogous to the raw graph encoder, the tree-based encoder leverages the framework's feature encoder to encode attribute values. Unlike the feature vectors of the raw graph encoding however, encoded trees have variable numbers of attribute nodes for each IR node. As a result, values can no longer be matched to attributes by their index but only by traversing a tree toward its root node. To improve the verbosity of encoded attribute values, the encoder offers the `verboseFeatureValues` parameter. If enabled, all attribute values are prefixed with their attribute's name and type, e.g., `isAggregation_boolean_true` or `isComposite_boolean_true` instead of a plain `true` for both attributes.

Because the identifiers of IR nodes are always included in either tree format, regardless of feature encoder configuration, the parameter `replaceNodeIds` is made available.

Parameter	Type	Default Value	Description
`format`	`string`	`local`	The tree format to use. Either `local` or `global`.
`replaceNodeIds`	`boolean`	`false`	If enabled, replace identifiers of nodes with enumerated identifiers.
`verboseFeature-Values`	`boolean`	`false`	If enabled, prefix attribute values with attribute name and type.
`wordsToIds`	`boolean`	`false`	If enabled, convert values of tree nodes to vocabulary tokens.
`idStartIndex`	`number`	0	Starting value for tokens if `wordsToIds` is enabled.

Table 5.3: Configurable parameters of the tree-based encoder.

While the number of unique identifiers in one model is limited by its number of elements, increasing the number of models in batches could result in vocabulary size increasing as well. With the `replaceNodeIds` parameter enabled, the nodes of each IR model are enumerated and assigned a unique index within the model, that is however shared across models. In particular, the nth node of each IR instance is assigned the identifier id_n.

As a last step, the tree-based encoder is able to replace each tree node's string value with a numeric token. These tokens start at zero by default and are continuously incremented, although the start value is configurable by users. Enabled via the parameter `wordsToIds`, this enables an immediate usage of the encoded trees for ML models that require numeric input.

5.3.3 Parameters

The tree-based encoder offers a parameter for format selection, two parameters for attribute encoding, and two parameters for tokenization, as listed in Table 5.3. Through composition, it also inherits all parameters of the feature encoder detailed in subsection 5.1.4.

5.4 Bag-of-Paths Encoder

The implemented BoP encoding is based on the graph language modeling framework GLaM4CM described by Ali and Bork [AB24]. Its concept is encoding a conceptual model as a collection of paths, which are derived from traversing model elements via their connections. The notion of a step refers to an edge and its target node in a path, while a segment is either a node or an edge, which occur alternating in paths. The only node of a path not also part of a step is the starting node. Each segment of a path, i.e., both its visited nodes and their connecting edges, are encoded with a textual

representation. While this representation is fixed in the work of Ali and Bork, CM2ML's BoP encoder uses a custom expression and templating language for highly customizable textual representations. In the following, we first present this custom language, followed by the BoP encoder's encoding process and configurable parameters.

5.4.1 Expression and Templating Language

We implement a parser for the encoder's custom expression and templating language using the JavaScript library *Ohm* [War]. The language's grammar is made available to users of the framework through help text attached to parameters that use the language. Ohm is able to generate parsers from a parsing expression grammar, which can then be enriched with semantic operations in JavaScript. The BoP encoder uses Ohm to implement its custom expression and templating language with three entry points. These are templates for assigning a numeric weight to a step, as well as encoding nodes and edges of steps to a textual representation. Thus, we implement the following concepts and validate each through an extensive suite of unit tests.

Selectors

The language offers selectors for both nodes and edges. Shared selectors for nodes and edges are attribute, keyword, and path selectors. The former is able to select the value of a node's or edge's attribute by its name, e.g., `attr.isAggregation` selects the value of the `isAggregation` attribute. Keyword selectors can target the identifier, type, name, and tag of both nodes and edges through the keywords `id`, `type`, `name`, and `tag` respectively. These are language-agnostic access methods for attributes corresponding to each respective keyword. Path selectors can select metadata about a current path, e.g., its length and current step index. The two available path selectors are `path.length` and `path.step`.

A source or target selector can be applied as an edge selector. These select an edge's source or target node and must be followed by a node selector. An example is `source.type`, which selects the type of an edge's source node.

For nodes, the language allows a selection of both incoming and outgoing edges that match a user-defined condition. On these edges, an edge selector must then be applied. For example, the node selector `edges.out[tag = owner].target.name` selects the name of the target node of a node's first outgoing edge with the tag `owner`. In other words, this example selector yields the name of a node's owner in UML models.

Conditions

Conditions can be used to select a specific edge of a node as described above, or to apply expressions conditionally. A node or edge condition is a node or edge selector followed by a comparison operator and a literal value. Alternatively, a selector may be combined with an existence or non-existence check. For example, `attr.isAggregation.exists`

is satisfied for nodes and edges that have an attribute named *isAggregation*. Available comparison operators are $=$, $\neq$, $\leq$, $\geq$, $<$, and $>$. Literal values may include digits, letters, as well as the special characters `:`, `.`, `,`, `=`, `-`, `(`, `)`, and `$`. An example for a node condition with a comparison operator is `name = MyModel`, which is only satisfied by nodes named *MyModel*.

Replacements

Replacements are segments of a template expression that are replaced with information about a node or edge. Alternatively, a replacement can also be a simple literal value that remains in template output as-is. For both nodes and edges, the language offers regular replacements and conditional replacements. Regular replacements are wrapped in double curly brackets and contain a node or edge selector respectively. When a template is applied to a node or edge, each replacement will be substituted with its selector's result. E.g., the replacement `{{node.id}}` will be replaced by a node's identifier.

Limited by double square brackets, conditional replacements contain a condition followed by `>>` as a separator and one or more regular replacements. When a template containing a conditional replacement is applied to a node or edge, one of two cases holds. First, a node or edge does not satisfy the conditional replacement's condition and the replacement is removed from the template's output. Second, its condition is satisfied and the conditional replacement is substituted with the result of its contained regular replacements. For example, the conditional replacement `[[type = Package >> Package: {{name}}]]` is replaced with *Package:* followed by the name of `Package` elements, while it is removed entirely for elements of a different type.

Templates

A node or edge template is a sequence of replacements, optionally preceded by a node or edge condition. Templates without conditions are considered universally applicable, while templates with conditions can only be applied to nodes or edges that satisfy their conditions. When a condition-less template or a template with a satisfied condition is applied to a node or edge, a textual representation is generated by executing each replacement as detailed above. E.g., a template with a condition such as `@type = Package >>> {{name}}` is only applicable to elements of type `Package`.

Step Weighting

A step weighting expression is either a number, or an edge-condition followed by a number. Similar to templates, step weighting expressions without conditions are considered universally applicable and a step weighting expression with a condition is only applicable to steps with edges that fulfill this condition. For example, the step weighting expression `@path.step >= 3 >>> 42` assigns a weight of 42 only to the third step and onward of a path.

5.4.2 Encoding Process

The BoP encoding is created in a four-step process. These steps are detailed below.

Paths

In the first step, all paths in an IR model that match user-provided criteria are enumerated and sorted. Configurable parameters for paths include minimum and maximum path length, and the option to allow paths containing cycles.

Weighting

Each step in a path is assigned a numeric weight. This weighting of steps uses the encoder's custom expression language. Users may configure a list of expressions, which are checked for applicability in their provided order. The first weighting expression that is universally applicable or has its condition satisfied by a step is used to calculate this step's weight. To optimize performance, step weights are cached and only computed as required. After the weights of all steps have been evaluated, the weights of paths are calculated. Depending on the encoder's configuration, a path's weight is either equivalent to its number of steps, the sum of individual step weights, or the product of individual step weights. In combination with configurable minimum and maximum weight limits as well as a sort order and an optional upper limit for the number of paths, this approach enables a fine-grained prioritization and filtering mechanism for paths.

Encoding

The third step is the crucial step of deriving textual representations for each path. For this, the implemented custom template grammar is leveraged to provide a highly customizable encoding of path nodes and edges. Analogous to step weighting expressions, templates are checked for applicability in their provided order. The first applicable template is used to derive a textual representation for a node or edge. Again, the results of template applications are cached to avoid redundant evaluations.

Pruning

Depending on configured templates, distinct paths with different nodes may be transformed to identical encoded textual representations during the encoding step. In addition, the list of paths may contain sub-paths of other paths starting at the same node, i.e., prefixes when viewing the paths as concatenated textual representations. If the customized encoding is identical for nodes and edges of prefixes and their subsuming path, the prefixes contain no additional information. Through the customizable `pruneMethod` parameter, users may choose to either keep or remove such paths. Removal can also be limited to paths that do not only have an identical encoding, but also contain identical nodes.

5.4.3 Visualization

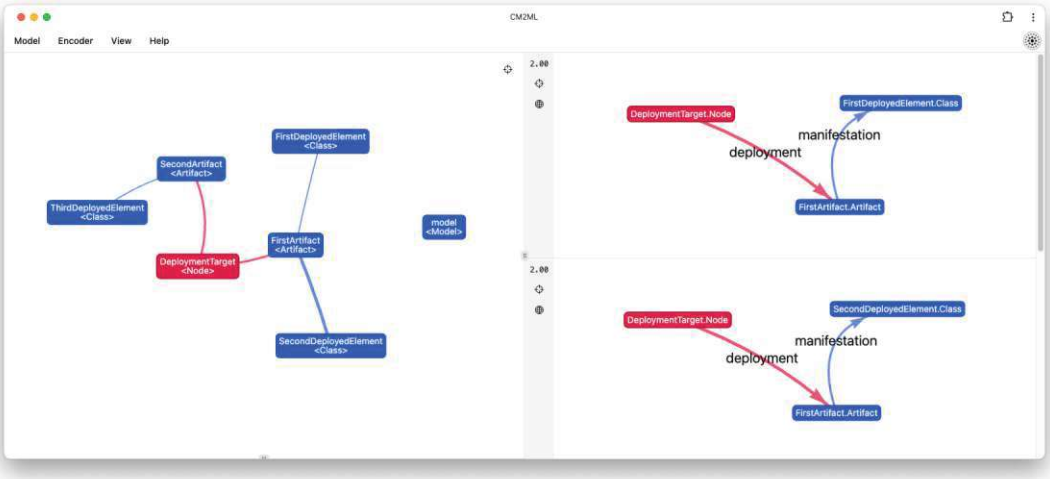
An example of the BoP encoding's visualization is shown in Figure 5.6. It includes a vis-network graph view for each path of an encoding, sorted by their weight according to the configured sort order. Nodes of these path views correspond to encoded IR nodes and edges to encoded IR edges. Nodes and edges in path views use their encoded textual representations as labels. Figure 5.6a shows how every occurrence of a selected node is highlighted in the path views. Similarly, Figure 5.6b depicts a selected edge having each occurrence in path views highlighted as well.

In the example of Figure 5.7a, the encoder has been configured to allow cycles and use the node template `{{name}}`. As a result, the output path also contains a cycle and exactly one node for each IR node of its path. However, configuring the encoder to encode each occurrence of a node in a path differently results in a special situation. As visible in Figure 5.7b, changing the node template to `{{name}}.{{path.step}}` can result in a path containing multiple encoded nodes corresponding to a single node of the IR. In such cases, a selection highlights every node in a path view corresponding to the selected node.

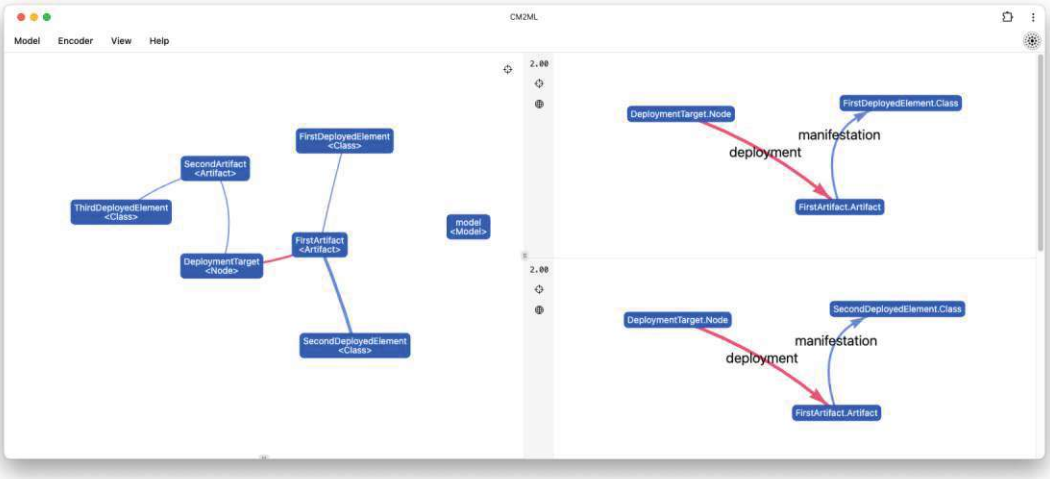
It is also possible to select all edges of a path at once by pressing the globe icon on the left side of a path view. Figure 5.8 shows how this is reflected both in the views of other paths and the IR graph view.

5.4.4 Parameters

All parameters of the BoP encoder are listed in Table 5.4. Every `list<string>` parameter is declared as unique and ordered. The order in particular is of importance since it determines which step weighting expressions and templates are used. It should also be highlighted that removing the limit for the maximum number of paths by setting `maxPaths` to a valuer smaller than 1 may result in very large outputs depending on the degree of connectivity a model exhibits.

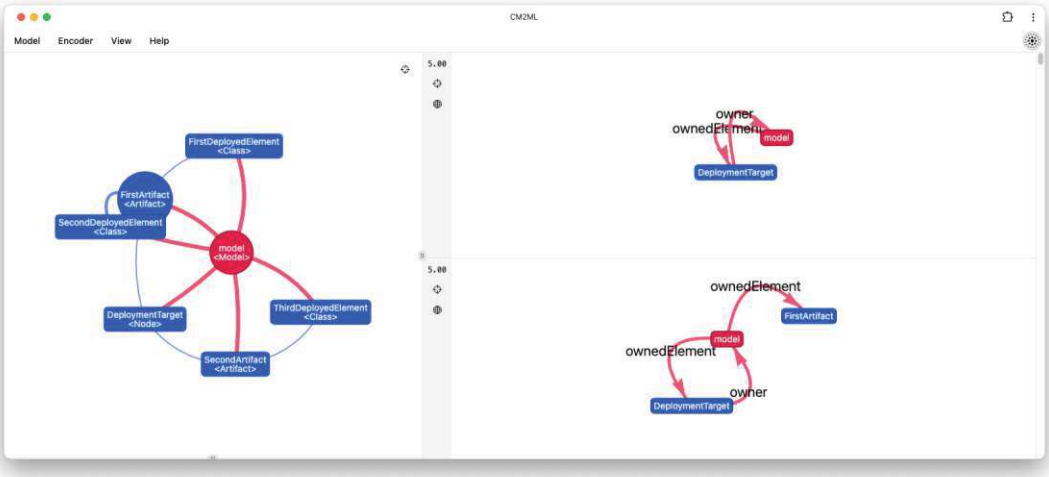


(a) Node selection in path (right) and IR view (left).

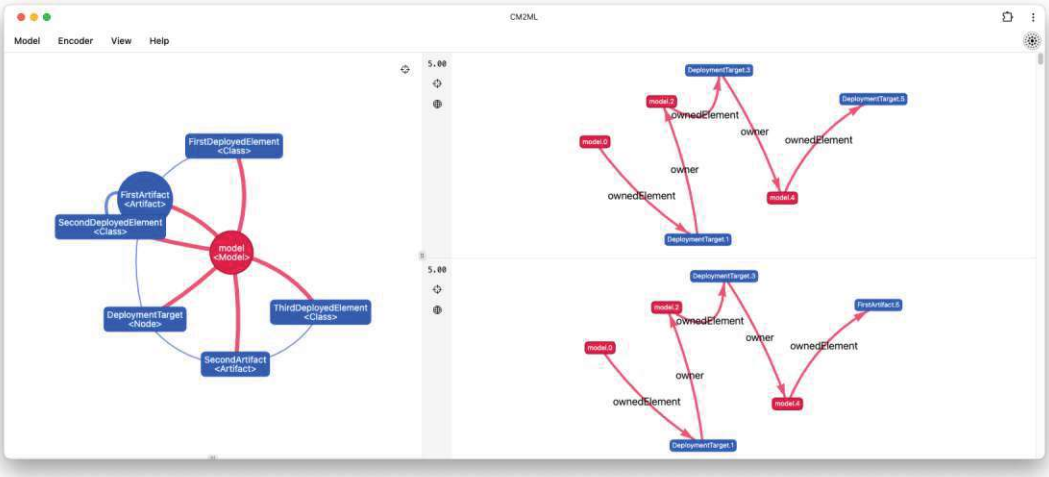


(b) Edge selection in path (right) and IR view (left).

Figure 5.6: BoP visualization with selections.



(a) Path with cycles (right) and IR view (left).



(b) Path with cycles and unique representations (right) and IR view (left).

Figure 5.7: BoP visualization with cycles.

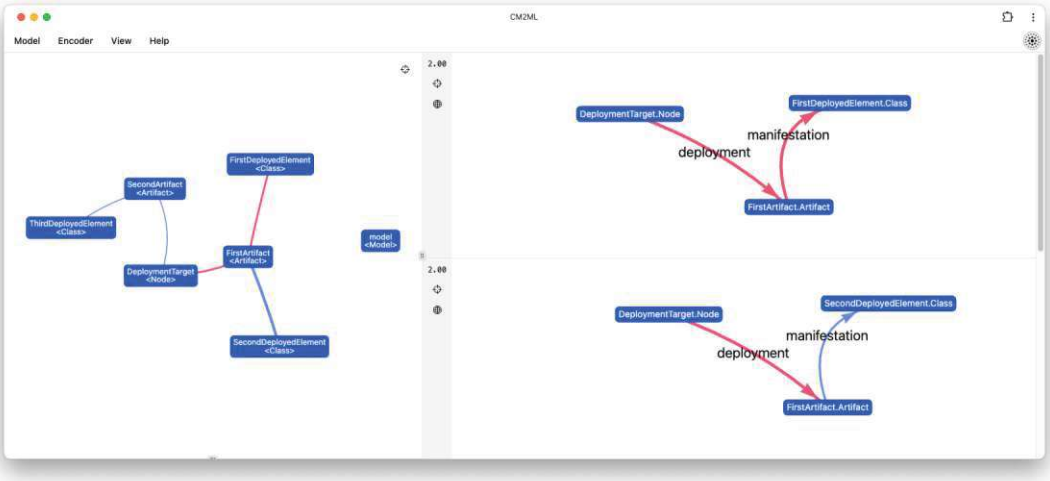


Figure 5.8: BoP visualization with entire path selected (top right) and IR view (left).

Parameter	Type	Default Value	Description
`allowCycles`	`boolean`	`false`	If enabled, paths may include cycles.
`minPathLength`	`number`	2	Minimum path length.
`maxPathLength`	`number`	3	Maximum path length.
`stepWeighting`	`list<string>`	`['1']`	Ordered list of step weighting expressions.
`pathWeight`	`string`	`step-sum`	Path weighting strategy. Either `step-sum`, `step-product`, or `length`.
`minPathWeight`	`number`	0	Minimum weight of paths.
`maxPathWeight`	`number`	`MAX_SAFE_INTEGER`	Maximum weight of paths.
`order`	`string`	`desc`	Path sort order based on path weight. Either `asc` or `desc`.
`maxPaths`	`number`	10	Maximum number of paths. Values smaller than 1 disable the limit.
`nodeTemplates`	`list<string>`	`['{{name}}.{{type}}']`	Ordered list of node template expressions.
`edgeTemplates`	`list<string>`	`['{{tag}}']`	Ordered list of edge template expressions.
`pruneMethod`	`string`	`none`	Prune method for encoded paths. Either `none`, `node`, or `encoding`.

Table 5.4: Configurable parameters of the BoP encoder.

CHAPTER 6

Experimental Evaluation

In this chapter, we present experimental evaluations of the three implemented encodings. For the raw graph and tree-based encodings, the evaluation trains and tests machine learning models to perform a node classification task using the dataset described in subsection 6.2.1. The goal of this first evaluation is to validate that the framework may be used to compare encodings effectively. The BoP encoding is evaluated using a unit-testing approach validating that its configurability enables the generation of output that is similar to that of the GLaM4CM framework of Ali and Bork [AB24]. This second evaluation also aims to validate that the framework facilitates experimentation through configurability. Afterward, an evaluation of CM2ML's encoding time performance is conducted. Finally, we describe threats to validity that have been identified during the evaluation.

6.1 Reproduction Package

The evaluations are not part of the CM2ML framework and instead implemented using industry-standard Python libraries for ML. However, they are included in the framework's monorepo and may be viewed as an additional testing layer of the framework's components in addition to their individual unit tests. The main Python libraries used for this evaluation are *NumPy* [Har+20], *scikit-learn* [Ped+11], *PyTorch* [Pas+19], and *PyTorch Geometric* (PYG) [FL19]. We use *Anaconda* [Ana] to create isolated environments with versioned libraries. The dataset of the experimental evaluations is described in subsection 6.2.1.

As reproducibility is an important factor, all evaluations are implemented with executable scripts and Turborepo tasks. Users may easily replicate an evaluation by first downloading the dataset and copying its files to the dataset folder, followed by installing the framework's dependencies and starting the Turborepo task corresponding to the desired encoding. In

addition, all evaluations use fixed seeds for random number sources to ensure repeated runs have identical results.

6.2 Methodology

The focus of the evaluation is to answer the research questions and validate that the framework's configurability enables an experimentation with encodings and their parameters. As such, our aim is not to train optimized models and emphasize the tuning of ML-related hyperparameters. Instead, we aim to produce observable classification performance differences with various encodings and their customizable configurations. Thus, certain ML techniques such as optimizing dataset splitting are not performed, as their effects are not of interest. All evaluations are performed using a 2020 MacBook Pro with an Apple M1 processor and 16GB of unified memory.

6.2.1 Dataset

The same datasets of UML models are used for each of the evaluated encodings. We employ a three-way split and create a train, validation, and test dataset for each encoding. The train datasets are used during model training to optimize weights. Validation datasets are also used during training, but only to optimize hyperparameters and perform early-stopping. Finally, test datasets are used to evaluate the models after training. They are also referred to as hold-out datasets, as models do not get to see their data until their training is finished.

Due to extensive training time, we use a subset of 1,000 UML models from the dataset described in section 2.2. Because of this smaller number of models, we employ a three-way split with 600 models for the training dataset. Our validation and test datasets contain 200 UML models each. For this evaluation, the framework's encoders are configured to adhere to these dataset sizes by settings the `start` and `limit` parameters respectively. The former is also adapted for the validation and test datasets to prevent any overlapping between them or with the training dataset.

Ad-hoc experiments with larger datasets for the models with shorter training time show either none or only slightly improved metrics. A possible reason for this may be that additional UML models mostly contain the same, more common elements as the UML models we do include. The less common elements still occur rarely enough to result in comparable metrics.

Deduplication

We enable the framework's built-in deduplication for all encoders used during the evaluation. Since the evaluations of the raw graph and tree-based encodings consist of three separately created datasets, we also use the `deduplicationData` parameter to ensure no duplicates exist between those datasets.

Creation Method

All datasets are created through separate invocations of the framework's encoders using the built-in CLI. The parameters for each invocation are identical, with the exception being `start`, `limit`, `deduplicationData`, `nodeFeatures`, and `edgeFeatures`. The reasons behind the usage of the latter two parameters are described in detail for each encoding below.

6.2.2 Evaluated Metrics

As described by Géron [Gér19], using accuracy as a metric for classification problems may result in misleading values. In particular, an uneven distribution of classes in the datasets at hand may result in a large number of false positive predictions. An analysis of the 1,000 UML models from our dataset reveals that it is indeed skewed. Each label, i.e., class, has an average of 557.8, but just a median of 32 occurrences. Further, the maximum number of occurrences is 12,260 for the class `PrimitiveType`. The reason for this is the structure of UML models, where some classes such as `PrimitiveType`, or `Property` are usually more common in a model than, e.g., `Class`. Elements of type `Class` may each contain multiple elements of type `Property`, which in turn often contain a `PrimitiveType`. This natural hierarchy skews the distribution of classes in UML models.

To circumvent the issue of a skewed dataset, we use a confusion matrix approach as suggested by Géron [Gér19]. The scikit-learn library provides a utility function for generating classification reports, including precision, recall, and F_1 score. We use these metrics instead for the evaluation, while the accuracy is used for short-circuiting during training. They are defined as follows, with TP, FP, TN, and FN representing the *true positives*, *false positives*, *true negatives*, and *false negatives* respectively:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{6.1}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{6.2}$$

$$\text{F}_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6.3}$$

It should be highlighted that the F_1 score in the evaluation's results will not match this formula, as it is averaged across all classes, while the formula applies to a single class. In particular, we measure both the macro average and weighted average. Macro-averaging uses the arithmetic mean of the individual metrics for each class. In contrast, weighted-averaging considers the number of occurrences in the datasets for each class to calculate a weighted arithmetic mean.

According to Géron, which of these three metrics is most relevant for the evaluation of an ML model depends on its application and requirements. For example, a high precision is preferable for applications where a false positive can result in negative consequences. As an example for this category, Géron uses a classifier for child-safe videos that whitelists, i.e., detects, safe videos. On the other hand, a higher recall may be preferred for applications that benefit from most if not all positives being classified as such and are not strongly negatively impacted by a number of false positives. An example for this second category are ML applications that classify medical scans, where false positives are not immediately harmful but false negatives lead to untreated conditions. Lastly, the F_1 score prefers balanced precision and recall values. Since our evaluative application does not fall in either of the first two categories mentioned above, we use the accuracy and F_1 score to compare the encodings and models.

It should be highlighted that under certain circumstances the classification report may encounter a division by zero. In particular, if a class is never predicted by the ML model, both the true and false positives will be zero. Since precision is defined as in Equation 6.1, a division by zero may occur. To prevent this issue, we exclude all never-predicted classes from the precision calculation.

6.2.3 Evaluation Process

The evaluation process, depicted in Figure 6.1, is similar for all encodings and models. The goal of the models is to classify model elements, with their UML types being used as the classes. We use this approach because the CM2ML framework infers those types from embedded metamodel information, thus generating valid data without the need for manual labeling. From the emitted encodings, we are thus able to derive two things. First, the input for the ML models, which has been stripped of the relevant type information, is extracted. This step is encoding-specific. Second, we use the removed type information to create labels for each entry in the dataset. Again, this process is encoding-specific and described in detail in the respective sections for each encoding. After each model is trained, the test dataset is then used to evaluate it.

For each raw graph encoder configuration, we evaluate two ML models ten times each with different seeds for random number initialization. After all evaluations are completed, we calculate the average of the metrics for each ML model and configuration respectively. The same process applies to the tree-based encoding, but with three different seeds for each encoder configuration. The reason behind this reduced number of runs is the longer training time for each run of the Tree-LSTM model.

6.3 Raw Graph Encoding

In related work, various GNN implementations are often used in combination with raw graph encodings for node classification [Lóp+22; Vel+18]. In this section, we evaluate two different GNNs in combination with CM2ML's raw graph encoder. After presenting

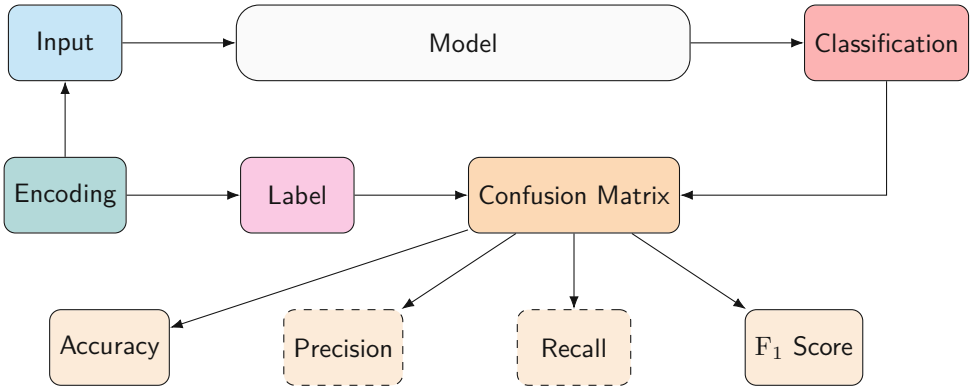


Figure 6.1: Process of generating evaluation metrics.

the raw graph encoder configurations, we describe the process of input masking and label creation. Afterward, the architecture of both ML models and their respective hyperparameters are detailed. Finally, we present and analyze the results.

6.3.1 Encoder Configuration

To treat both `xmi:type` and `xsi:type` attributes identically, as a difference exists only in the serialization, we enable the `unifyTypes` parameter. The encodings created by the framework now only contain the former attribute. This also has the benefit of preventing mismatches in the encoding of the type attributes. Without this parameter enabled, the UML type of an element could have two different numeric encodings depending on which attribute is used.

We leverage the implemented feature encoder to emit node encodings, i.e., feature vectors, that are usable by GNNs without further processing, as all values are numeric. However, we omit all string attributes by enabling both configurable parameters `rawStrings` and `onlyEncodedFeatures`. The reason for this is the unlimited vocabulary problem that arises if no text embedding is used. For all other attribute types, the numeric encodings are enabled. To prevent mismatches during feature encoding between the datasets, we use the `nodeFeatures` and `edgeFeatures` parameters of the encoder to match the vector shape and feature indices.

An advantage of GNNs is the ability to include neighboring nodes through edge connections in calculations and operate on irregular data [FL19; Luo+17; Vel+18]. While the implemented encoder is able to output both adjacency matrices and lists, we choose to use the latter for the evaluation due to their size efficiency.

Finally, by enabling both `strict` and `continueOnError`, we ensure that no invalid model is used for the evaluation.

Table 6.1 shows the different configurations we use for obtaining the datasets, excluding the parameters that are identical for each configuration. With each configuration we

Configuration	relationshipsAsEdges	onlyContainment-Associations	edgeTagAsAttribute
A	true	true	true
B	true	true	false
C	true	false	true
D	true	false	false
E	false	true	true
F	false	true	false
G	false	false	true
H	false	false	false

Table 6.1: Configurations of raw graph encoder parameters.

Metric	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
Node features	57	57	57	57	60	60	60	60
Edge features	12	11	12	11	1	0	1	0
Classes	123	123	123	123	145	145	145	145

Table 6.2: Analysis of raw graph encodings by configuration.

remove 48, 18, and 30 duplicate encodings from the train, validation, and test datasets respectively using the built-in deduplication. Further, the models of the test dataset contain one invalid model, which is removed due to the enabled `strict` mode. As a result, each configuration yields 552, 182, and 169 encoded models for the train, validation, and test datasets respectively.

Further, we analyze the datasets created with each configuration and calculate the following metrics shown in Table 6.2. The size of node and edge feature vectors determines the correct sizes for the embedding layer, while the number of classes directs the size of the classification layer. This is presented in more detail in subsection 6.3.4.

6.3.2 Input Masking and Label Creation

Figure 6.2 visualizes the input masking and label creation for the graph-based encoding. The feature vectors of nodes contain their classification, i.e., their UML type, as highlighted in Figure 6.2a. We prepare the feature vectors for usage in the ML model evaluation by setting all type values to zero, as shown in Figure 6.2b. In parallel, we generate matching labels from the framework's output as shown in Figure 6.2c. A node's type value is used as the classification label.

Besides the masking of classification data in the node feature vectors, no preprocessing is required. The adjacency list generated by the CM2ML framework is immediately

Attribute	$Node_1$	$Node_2$	$Node_3$
`isAbstract`	0	1	0
`visibility`	2	0	1
`xmi:type`	**1**	**3**	**2**

(a) Raw graph encoding feature vectors.

Attribute	$Node_1$	$Node_2$	$Node_3$
`isAbstract`	0	1	0
`visibility`	2	0	1
`xmi:type`	**0**	**0**	**0**

(b) GNN input feature vectors.

$Node_1$	$Node_2$	$Node_3$
1	**3**	**2**

(c) Label for expected output.

Figure 6.2: Example of input masking and label creation for raw graph encoding.

usable as input for both graph convolutional network (GCN) and graph attention network (GAT) models. Further, feature vectors of edges do not require any masking, as the evaluation only performs node classification.

6.3.3 Model Architecture

The implemented raw graph encoding is evaluated using two GNNs, namely a GCN and a GAT. Both are designed to operate on graph structures, but only the latter can optionally use edge feature vectors as well. Either GNN emits a vector of probabilities for each corresponding input node. These probabilities indicate how likely it is that an input node belongs to any given class, according to an ML model's prediction. To get the final classification, we apply the `argmax` operator to both ML models' outputs. This operator picks the class with the highest predicted probability for each node. While GNNs may contain multiple convolutional layers, our models use a fixed number of two layers, matching the model of López et al. [Lóp+22]. The two architectures and their differences are detailed below.

GCN

For the GCN model, an implementation of Kipf and Welling's [KW17] graph convolutional layer from the PYG library is used. PYG contains an extensive range of different models and individual layers for graph-based learning that are based on their respective papers. The GCN model consists of a graph convolutional embedding layer, followed by a rectified

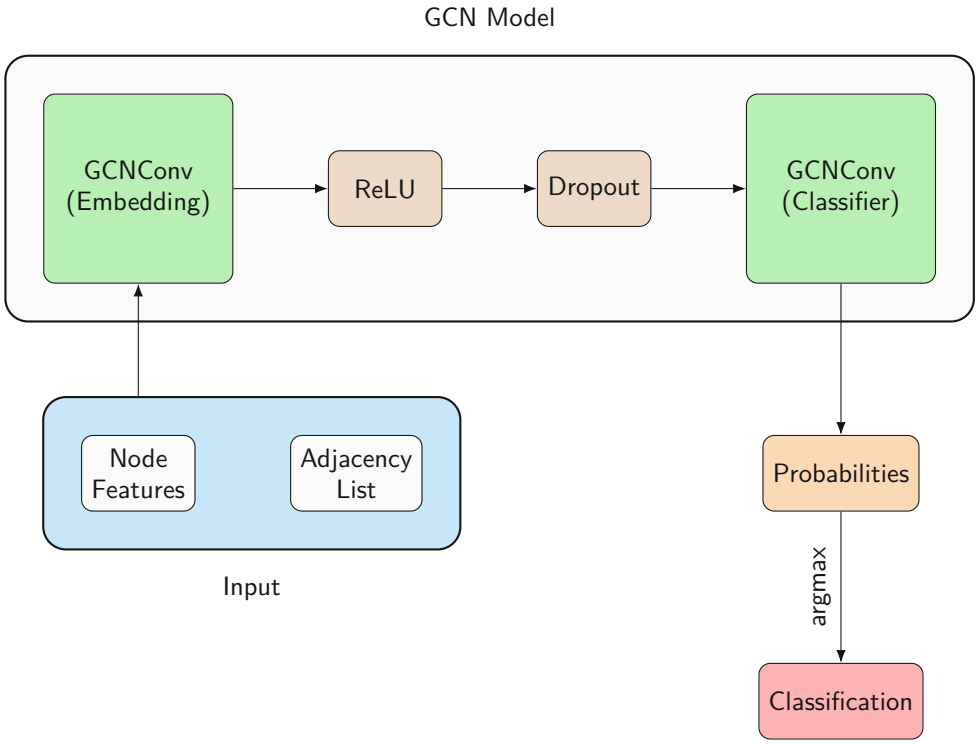


Figure 6.3: Architecture of the GCN model.

linear unit (ReLU) [Aga19] for activation, and a dropout layer. The dropout layer lessens the effect of overfitting [Luo+17]. Finally, a second convolutional layer is used as a classifier for the output layer. The GCN's architecture is shown in Figure 6.3.

GAT

The architecture of the GAT model is similar to the GCN model's and depicted in Figure 6.4. The major difference is the replacement of graph convolutional layers with graph attention convolutional layers as introduced by Veličković et al. [Vel+18]. Again, these are implemented by PYG. Analogous to the GCN model, a ReLU for activation and a dropout layer form a hidden layer between the two graph attention convolutional layers. A GAT may use multiple attention heads in each convolutional layer. The number of heads is a hyperparameter and independent for each layer. It should also be highlighted that the graph attention convolutions can optionally use the feature vectors of edges, which are also computed by the framework.

6.3.4 Hyperparameters

As ML model architecture and hyperparameter optimization is not the goal of this evaluation, we perform ad-hoc experiments using encoder configuration C to determine

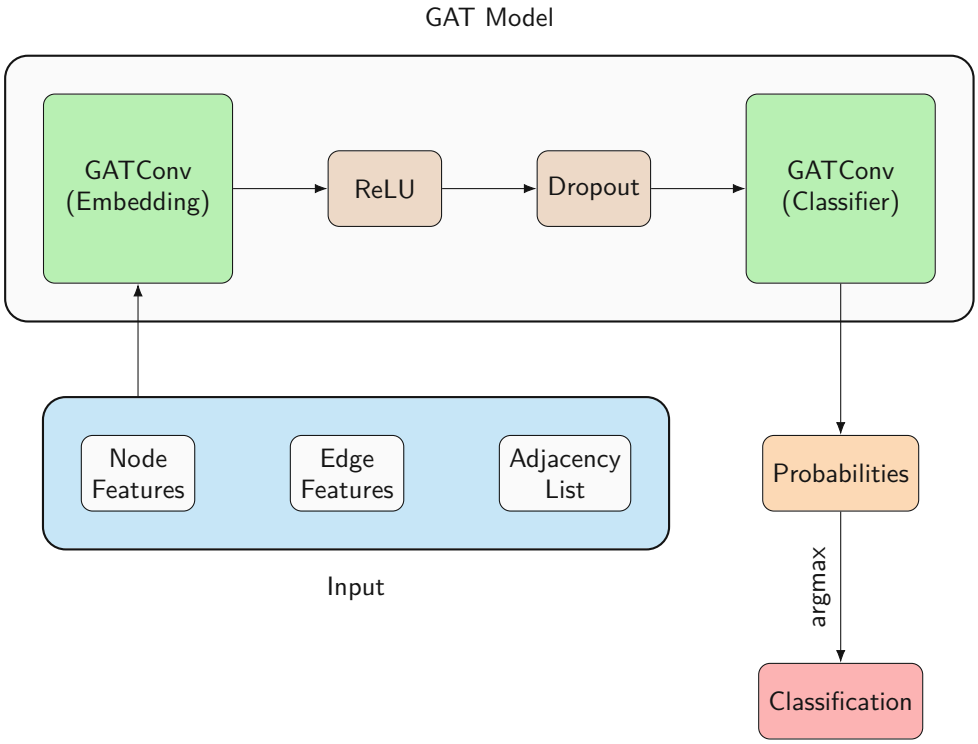


Figure 6.4: Architecture of the GAT model.

hyperparameters. For both models, we limit training to a maximum of 100 **epochs**. Additionally, **early stopping** with a **patience** of 10 is enabled for the training stage of the GAT and GCN models. This means that training will be terminated after epoch n, if all epochs $n-i$ with $i \in [0,9]$ have a lower accuracy than epoch $n-10$. Model-specific hyperparameters are discussed below. All hyperparameters are either selected based on related work or empiric experiments conducted with encoder configuration C.

GCN

Table 6.3 provides a summary of the GCN model's hyperparameters. The **embedding layer's** size is determined by the number of features each node possesses and is thus dynamically set, depending on the encoder configuration. Analogously, the size of the **classifier layer** is determined by the number of classes and also derived from the dataset of a concrete configuration. The size of the **hidden layer** may be selected independently of those variables. Based on empiric experiments, we use twice the number of classes as the hidden layer's size. A **dropout** of 0.2 and a **learning rate** of 0.01 are chosen analogous to Kipf and Welling [KW17].

Hyperparameter	Value
Maximum epochs	100
Patience	10
Layers	2
Embedding size	57 (A, B, C, D), 60 (E, F, G, H)
Hidden size	246 (A, B, C, D), 190 (E, F, G, H)
Classifier size	123 (A, B, C, D), 145 (E, F, G, H)
Dropout	0.2
Learning rate	0.01

Table 6.3: Hyperparameters for the GCN model.

GAT

An overview over all hyperparameter used for the GAT model is given in Table 6.4. We evaluated the usage of either eight **attention heads** or a single head for both the embedding and classification layers, based on Veličković et al. [Vel+18]. Results of ad-hoc experiments show that the configuration of eight heads for both layers performs best in our case. Also based on their findings, we choose a **dropout** of 0.6. Regarding layer sizes, the GAT model follows the GCN configuration and uses the number of node features and number of classes for the **embedding** and **classification layers** respectively. Since the usage of attention heads increases memory consumption, we only use the number of classes as the **hidden layer's** size, not its doubled value. Additionally, we use the number of edge features as the optional **edge embedding** size.

6.3.5 Results

In the following, the results of each ML model are first presented independently, followed by a comparison of the two models' best results. For each ML model, we view the two groups of configurations A, B, C, and D as well as E, F, G, and H separately. The reason for this is that the datasets of configurations belonging to each group share the same number of classes. Finally, we provide an explanation for the results based on their respective encoding configuration.

GCN

The GCN's results are shown in Table 6.5. We omit the results of the configurations B, D, F, and H. The reason is that those configurations only differ from their respective

Hyperparameter	Value
Maximum epochs	100
Patience	10
Layers	2
Embedding size (nodes)	57 (A, B, C, D), 60 (E, F, G, H)
Embedding size (edges)	12 (A, C), 11 (B, C), 1 (E, G), 0 (F, G)
Hidden size	123 (A, B, C, D), 145 (E, F, G, H)
Classifier size	123 (A, B, C, D), 145 (E, F, G, H)
Dropout	0.6
Learning rate	0.01
Attention heads	8

Table 6.4: Hyperparameters for the GAT model.

Configuration	Accuracy	Weighted-averaged F_1	Macro-averaged F_1
A	81.583 ± 0.623	79.206 ± 0.532	36.274 ± 1.580
C	$\mathbf{82.316 \pm 2.045}$	$\mathbf{80.578 \pm 2.294}$	$\mathbf{40.324 \pm 2.595}$
E	$\mathbf{85.206 \pm 0.497}$	$\mathbf{83.222 \pm 0.837}$	40.569 ± 1.628
G	84.262 ± 1.491	82.449 ± 1.822	$\mathbf{44.633 \pm 1.878}$

Table 6.5: Results of the GCN model by configuration. All values are percentages with standard deviation.

immediate predecessor configuration in the `edgeTagAsAttribute` parameter. Since this parameter only affects edge attributes and those are not used by the GCN model, these four configurations provide no additional insight.

Configuration C is the best performing configuration with `relationshipsAsEdges` enabled. While accuracy and weighted-average F_1 score of configuration A only exhibit a small decrease, the macro-avergage F_1 score is over 4% worse. As we can see an almost identical difference in the F_1 scores of the configurations E and G, it stands to reason that enabling `onlyContainmentAssociations` and thus omitting all non-containment associations from the graphs is detrimental to the prediction performance concerning

less often occurring classes. On the other hand, the configurations C and G, i.e., the configurations with all associations of the UML models being present in the output, exhibit a larger standard deviation.

Finally, disabling `onlyContainmentAssociations` in configuration G decreases the accuracy and weighted F_1 score compared to configuration E. This result may imply that when relationship UML elements are not treated as edges in the IR and thus the output graphs, a better prediction performance for commonly occurring classes may be achieved by omitting all non-containment UML associations from the encoding.

GAT

The evaluated metrics of the GAT model are displayed in Table 6.6. First, we can see that the configurations B, D, F, and G perform worse than their immediate predecessor configuration. The reason behind this is that those configurations disable the `edgeTagAsAttributes` parameter. As a result, the implemented UML parser creates edges from associations that do not have any attributes. Even though the GAT model uses edge feature vectors as input, they are void of any information unless the feature vector is created for an edge that represents an UML relationship. Hence, we see an even larger drop in performance for the configurations F and H, compared to B and D, as the latter two still contain some edge feature information through their relationship edges.

The two configurations D and H show the worst performance. Both can be characterized by including a large number of edges that do not have a single attribute associated to them.

In contrast to the GCN's results, we see the configurations A and E performing best in their respective groups, with the latter outperforming the former in all metrics. A conclusion may be that excluding all non-containment UML associations is beneficial to the GAT in particular, whether UML relationships are represented as nodes or edges.

Similar to the GCN, we see an increase in standard deviation when including non-containment UML associations.

Conclusion

The GCN model shows only small differences when including or excluding non-containment UML associations, except in the macro-average F_1 score. Treating UML relationships as nodes in the IR has a larger positive impact on the evaluated metrics, although the scenarios are not immediately comparable because of the different number of classes.

The GAT model performs better than its GCN counterpart across every configuration and metric, including each respective standard deviation. While the GAT performs best when omitting non-containment UML associations, it is mostly indifferent to the representation of UML relationships. However, it is sensitive to edges without attributes. Configurations that include such edges exhibit the worst performance across both models.

Configuration	Accuracy	Weighted-averaged F_1	Macro-averaged F_1
A	**85.030 ± 0.517**	**82.444 ± 0.419**	**41.835 ± 1.511**
B	76.833 ± 0.732	75.183 ± 0.739	38.802 ± 1.891
C	83.919 ± 0.795	81.640 ± 0.955	41.717 ± 2.425
D	59.131 ± 4.030	54.603 ± 5.131	33.585 ± 3.043
E	**86.222 ± 0.363**	**83.968 ± 0.240**	**46.501 ± 1.571**
F	77.481 ± 0.545	75.625 ± 0.468	36.253 ± 1.539
G	84.370 ± 0.941	82.515 ± 1.198	45.932 ± 3.152
H	60.086 ± 4.771	53.895 ± 6.068	29.296 ± 1.634

Table 6.6: Results of the GAT model by configuration. All values are percentages with standard deviation.

Using the framework's configurable whitelist for UML associations may enable fine-tuning for the configurations *C* and *G*, where including certain additional associations beyond containment may improve performance, but allowing all associations results in a negative change.

In addition to the differences in metrics, both ML models show a large gap in training time. Training the GCN model requires 33.5 seconds on average. In contrast, to complete training the GAT model requires 296.9 seconds on average, depending on which configuration is used. For both models, training time is increased in general by including the additional non-containment UML associations. However, due to the usage of early-stopping, training time may also differ based on the number of epochs and the randomized weight initialization.

6.4 Tree-based Encoding

While the goal of Burgueño et al. [Bur+22] is automating model transformations, it is essentially a matter of generating tree structures from input trees with different data and structures. As such, the problem of node classification can be reduced to generating a tree that contains a classification for each node of the input, as visualized in Figure 6.5. We build upon their Tree-LSTM implementation and adapt it to perform node classification. The following subsections describe the encoder configurations used for the evaluation, followed by the ML model's architecture and hyperparameters. Finally, the results are presented.

6.4.1 Encoder Configuration

Because the implemented Tree-LSTM requires a non-trivial amount of memory depending on the size of input trees, we enable the `onlyContainmentAssociations` parameter. The removal of all non-containment UML associations reduces the size of

Configuration	format	relationshipsAsEdges	verboseFeatureValues
A	global	true	true
B	global	true	false
C	global	false	true
D	global	false	false
E	local	true	true
F	local	true	false
G	local	false	true
H	local	false	false

Table 6.7: Configurations of tree-based encoder parameters.

the encoded trees by around 49.16% for our dataset. For example, with configuration *A* and `onlyContainmentAssociations` disabled, the dataset's encoded trees have 3,437.89 nodes on average. In contrast, enabling `onlyContainmentAssociations` reduces this number to 1,747.69. Analogous to the evaluation of the raw-graph encoding, the framework is configured to omit string-based attributes by enabling both `rawStrings` and `onlyEncodedFeatures`. By configuring the `nodeFeatures` and `edgeFeatures` parameters for each dataset we ensure that the indices of encoded feature values are consistent. In contrast to the raw graph encoding, the identifiers of nodes are not treated as attributes by the encoder and, thus, not covered by the removal of string-based attributes. Thus, the parameter `replaceNodeIds` is enabled as well. In combination with the omitted attributes, this keeps the vocabulary sizes in check and prevents the unlimited vocabulary problem. Next, the `unifyTypes` parameter is enabled once again for the same reasons described in subsection 6.3.1. Since the tree-based encoder supports two different formats for trees, we perform the evaluation for both. Table 6.7 shows the different encoder parameter configurations we consider, excluding parameters that are identical for each configuration.

Because of restrictions regarding memory consumption and training time, we also omit and ignore all tree encodings with more than 1,000 nodes. This includes a large share of the models in our datasets. As shown in Table 6.8, the number of nodes in encoded trees and, thus also, the amount of omitted tree encodings depends on the chosen tree format. The listed vocabulary sizes are relevant to determine layer sizes for the Tree-LSTM model and only represent the vocabulary of the filtered and masked trees. The table also shows that the datasets of configurations using the more compact `local` format include more UML models than those created with the `global` format. A similar approach is also used by Weyssow et al. [WSS22], who encode metamodels and omit any metamodel with more than fifteen elements.

Metric	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
Train size	553	553	553	553	553	553	553	553
Train size (filtered)	187	187	179	179	201	201	191	191
Validation size	182	182	182	182	182	182	182	182
Validation size (filtered)	54	54	49	49	56	56	52	52
Test size	170	170	170	170	170	170	170	170
Test size (filtered)	41	41	38	38	45	45	44	44
Input vocabulary size	240	158	221	138	251	169	237	153
Classes	86	86	105	105	86	86	105	105

Table 6.8: Analysis of tree-based encodings by configuration.

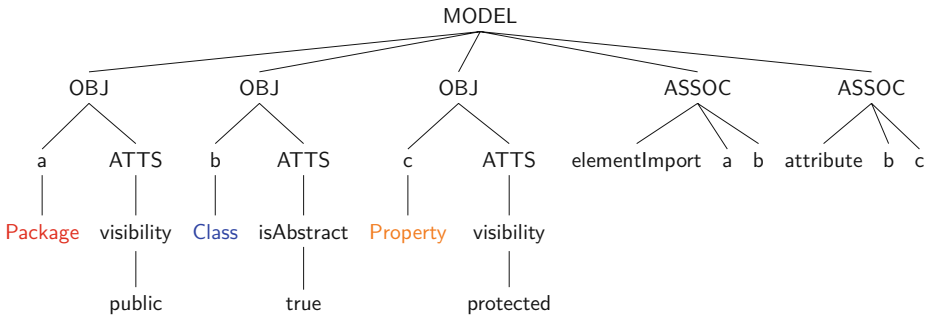
6.4.2 Input Masking and Label Creation

Figure 6.5a shows an example output of the framework for the `global` tree format and Figure 6.5c the corresponding label for the Tree-LSTM model. Figure 6.5b shows the input of the Tree-LSTM, where all classification data has been removed to facilitate learning based on model structure. The input masking and label extraction for the `local` format is analogous, but since it treats the type as a regular attribute, we remove the corresponding attribute node instead. It should be highlighted that no identifiers are present in labels. Instead, a model element and its classification are matched by their positional index within their respective trees.

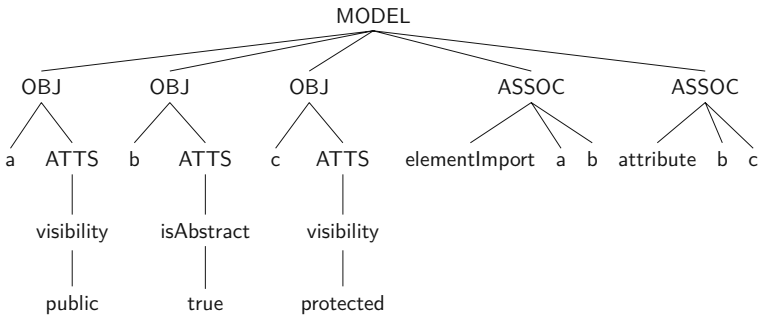
Not represented in the figure is the process of replacing each tree node's textual value with a numeric token. While the framework's encoder is able to handle this step internally, we instead perform it in the evaluation's code. This allows us to create separate sets of tokens, i.e., vocabularies in the context of the Tree-LSTM, for the input and output trees. A separation like this ensures that output trees may only contain tokens that represent the class of an element, in addition to a special token for the required root node. Unrelated tokens, e.g., tokens representing names or values of attributes, will thus never be included in the output. In addition, the Tree-LSTM implementation reserves a range of tokens for internal usage. These special tokens allow the model to represent a tree's structure in a serialized form, i.e., a one dimensional array.

6.4.3 Model Architecture

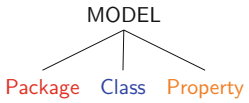
For the evaluation of the tree-based encoding, the neural network architecture of Burgueño et al. [Bur+22] is used. The implemented evaluation reuses their custom Tree-LSTM model, but recent changes to PyTorch require modifications to tensor instantiation and usage. In addition, we move from a pure accuracy-based evaluation of the test dataset to the classification report described in subsection 6.2.2. The Tree-LSTM's architecture is visualized in Figure 6.6. As a recurrent neural network, the model iteratively emits a



(a) Tree-based encoding.



(b) Tree-LSTM input.



(c) Label for expected output.

Figure 6.5: Example of input masking and label creation for tree-based encoding.

complete tree one node at a time. The completion of a full tree is then marked with a special token. In the following, we talk about entire trees and not individual nodes when referring to the Tree-LSTM's output.

The output of the Tree-LSTM is a tree and not immediately usable for classification without knowledge of its structure. Thus, we derive a classification vector, matching the shape to the GCN and GAT models' output. For any given output and label pair, one of three cases holds. First, the number of nodes in the output matches that of the label. This case is shown in Figure 6.7a, where the left side represents both an output and a label. We derive the classification for the model by removing the root node and enumerating its children. However, the Tree-LSTM output may also have more or fewer nodes than its assigned label. If an output tree contains additional tree nodes, they do not correspond to the classification for any element of its input model. We ignore any such superfluous tree nodes from output trees by limiting their serialized length to that

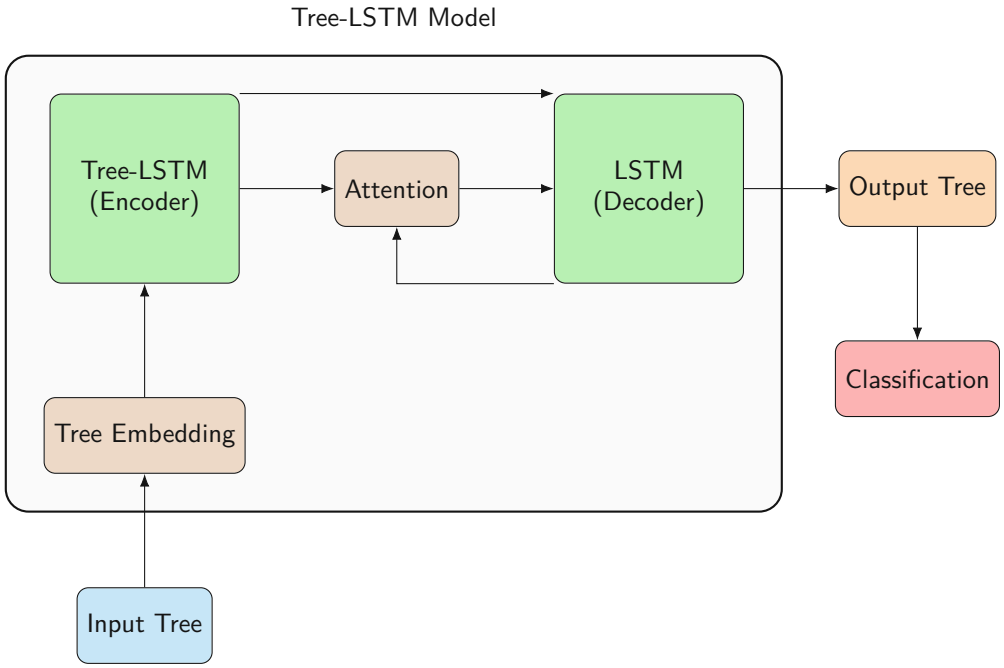
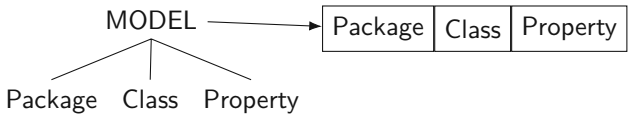


Figure 6.6: Architecture of the Tree-LSTM model.

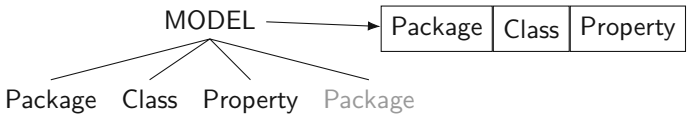
of the corresponding label, as shown in Figure 6.7b. On the other hand, trees generated by the Tree-LSTM may also contain fewer nodes than the corresponding label. For such cases, we pad the output with a reserved marker token to match its corresponding label's length, as displayed in Figure 6.7c. Since this reserved token is not part of the output vocabulary, this cannot result in random correct guesses.

6.4.4 Hyperparameters

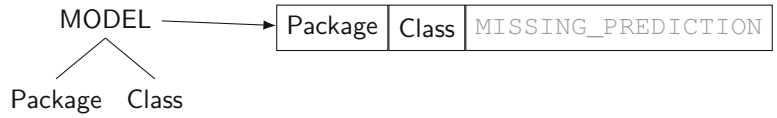
A summary of the Tree-LSTM's hyperparameters is given in Table 6.9. Analogous to the evaluation of the raw graph encodings, we determine hyperparameters based on ad-hoc experiments with encoder configuration *E*. As the foundation and starting point, our initial hyperparameters are based on the empirical findings of Burgueño et al. [Bur+22], with slight modifications to fit the different dataset. Again, early stopping is employed to determine the number of training **epoch**s, with an upper limit of 30 epochs. For the Tree-LSTM model, we use a **patience** of 20, but with the validation dataset's loss instead of accuracy as the fundamental metric. Because **embedding** and **hidden size** strongly drive memory consumption, we set both dynamically to each configuration's respective input vocabulary size. According to Burgueño et al., this is the required minimum value. Next, a **learning rate** of 0.005 ensures that overshooting is reduced and small weight adjustments are possible. Burgueño et al. recommend using a **single layer** with a **dropout** of 0.75 to reduce overfitting. Ad-hoc experiments show a performance overhead



(a) Deriving classification target (right) from label or Tree-LSTM output (left).



(b) Deriving classification (right) from Tree-LSTM output with superfluous nodes (left).



(c) Deriving classification (right) from Tree-LSTM output with too few nodes (left).

Figure 6.7: Deriving classification from Tree-LSTM output and labels, using the label from Figure 6.5c as an example.

Hyperparameter	Value
Maximum epochs	30
Patience	20
Layers	1
Batch size	32
Dropout	0.75
Embedding & hidden size	240 (A), 158 (B), 221 (C), 138 (D), 251 (E), 169 (F), 237 (G), 153 (H)
Learning rate	0.005
Learning rate decay	10% every third epoch

Table 6.9: Hyperparameters for the Tree-LSTM model.

of up to 20% when using two layers. Since training time is already much higher than that of the GNN models, we keep the single layer approach. As our dataset is smaller than that of Burgueño et al., we also adjust the decay of the **learning rate** to 10% every third epoch. Lastly, we use a reduced **batch size** of 32 instead of 64 to alleviate memory consumption.

Configuration	Accuracy	Weighted-averaged F_1	Macro-averaged F_1
A	**37.961 ± 2.064**	**28.956 ± 2.016**	**13.191 ± 0.592**
B	32.208 ± 1.618	23.739 ± 0.957	9.469 ± 0.622
C	34.995 ± 0.504	24.547 ± 0.372	9.681 ± 0.140
D	32.926 ± 2.113	23.253 ± 2.006	8.344 ± 1.361
E	**38.080 ± 1.987**	**29.482 ± 1.138**	**13.201 ± 0.409**
F	35.895 ± 0.713	26.016 ± 0.987	11.477 ± 0.686
G	33.508 ± 0.397	23.794 ± 1.261	8.875 ± 1.184
H	31.315 ± 2.643	23.010 ± 2.125	8.720 ± 0.794

Table 6.10: Results of the Tree-LSTM model by configuration. All values are percentages with standard deviation.

6.4.5 Results

The results of the Tree-LSTM model are shown in Table 6.10. We identify the best performing configuration for each format. For both formats, treating relationships as edges and using verbose feature values achieves the highest accuracies and F_1 scores. The `local` format has a small advantage, though the difference between both formats is smaller than the standard deviations. Also similar for both formats, we see that using non-verbose feature values strongly reduces all observed metrics.

Compared to the results of both GNN models, we observe considerably lower scores for all three metrics across all eight configurations. In addition, on average only 284.25 of the 905 encoded trees per dataset could be included in the evaluation due to memory consumption limits. Finally, an average training time of 37.2 minutes for each configuration significantly limits the usability of our Tree-LSTM for the task of node classification, especially with regard to the reduced dataset sizes.

6.5 Bag-of-Paths Encoding

We evaluate the BoP encoder's ability to be fine-tuned by configuring it to emulate the first two steps of the graph language modeling framework GLaM4CM proposed by Ali and Bork [AB24]. The first step, a transformation of conceptual models to conceptual knowledge graphs, is analogous to CM2ML's UML parser and IR. Then, their context generation is handled by the BoP encoder through a particular configuration. Since stereotypes serve different use cases in UML and the modeling language used by Ali and Bork, OntoUML, we adapt the target slightly. Instead of including element name, type, and stereotype, we only include the former two and mask types instead of stereotypes. In theory, this encoding would then also be usable for the node classification task of predicting UML types we presented above.

6.5.1 Encoder Configuration

The configuration of the encoder is more complex than the previous ones. First, we once again enable `strict` mode and deduplication with `deduplicate`. With `continueOnError`, and a `limit` of 1,000 we include all valid models in the batch of the first 1,000 models.

Since we are only interested in UML relationships being considered as edges, we first enable `relationshipsAsEdges` and `onlyContainmentAssociations`. Then, we add `owner`, `ownedElement`, `association`, and `associationClass` to the `edgeBlacklist` parameter. The first two cover the containment associations, which we do not want to include. The latter two are two forms of relationships that we also exclude.

Regarding the BoP encoder, we first configure the included paths. Using a `minPathLength` of 1 and a `maxPathLength` of 3 ensures that we are able to include all paths also covered by the context length analyzed by Ali and Bork. The defaults are used for all parameters related to step and path weighting, i.e., each step has the same weight and each path is thus weighted by its length. No minimum or maximum weights are defined and the sort order is descending. With a `maxPaths` value of 0, we do not impose a limit on the number of paths, thus including all paths matching the path criteria from above.

Using the `nodeTemplates` parameter, we encode each UML element with its name and type. Elements without names are only encoded with their type and prefixed with "unnamed". In addition, we surround types in node encodings with the string `$eu.yeger$`, which we previously identified as not being present anywhere in the dataset. We will use this special string later to apply masking in our stubbed ML application.

Edges are encoded with a fixed name for each type, creating a format more readable to humans. Finally, we prune the encoding for each UML model by setting the `pruneMethod` parameter to `encoding`. This prevents any paths with identical encodings from being present in the output. A complete overview of all parameters is given in Table 6.11.

For our dataset, we see one invalid and 359 duplicate models with this specific configuration. Across 10 CLI invocations of the encoder, we achieve an average runtime of 2.94 seconds.

6.5.2 Input Masking and Label Creation

Since input masking and the creation of labels is part of a specific ML application, neither are implemented within the CM2ML framework's BoP encoder. Instead, we use a Python script to handle this final step of the evaluation. Analogous to Ali and Bork, we create two groups of nodes across all BoP encodings from the dataset. The first group contains 80% of all nodes and does not mask node types. For this group, we remove the `$eu.yeger$` marker from all encoded node segments. In contrast, the second group has both the markers and their framed types replaced with `<MASKED>`. The removed type is

Parameter	Value
strict	true
deduplicate	true
continueOnError	true
relationshipsAsEdges	true
onlyContainment-Associations	true
edgeBlacklist	owner, ownedElement, association, associationClass
minPathLength	1
maxPathLength	3
maxPaths	0
nodeTemplates	@name.exists >>> {{name}} $eu.yeger${{type}}$eu.yeger$, unnamed $eu.yeger${{type}}$eu.yeger$
edgeTemplates	@tag = abstraction >>> abstracts, @tag = communicationPath >>> communicates with, @tag = componentRealization >>> realizes, @tag = dependency >>> depends on, @tag = deployment >>> deploys, @tag = elementImport >>> imports, @tag = extend >>> extends, @tag = extension >>> extends, @tag = generalization >>> generalizes, @tag = include >>> includes, @tag = informationFlow >>> informs, @tag = interfaceRealization >>> realizes, @tag = manifestation >>> manifests, @tag = packageImport >>> imports, @tag = packageMerge >>> merges, @tag = profileApplication >>> applies, @tag = protocolConformance >>> conforms to, @tag = protocolTransition >>> transitions to, @tag = realization >>> realizes, @tag = substitution >>> substitutes, @tag = templateBinding >>> binds, @tag = transition >>> transitions to, @tag = usage >>> uses
pruneMethod	encoding

Table 6.11: Configurations of BoP encoder parameters.

Encoded Path	Labels
`Package_16 Package depends on unnamed Operation`	–
`Package_16 Package depends on unnamed <MASKED>`	`PrimitiveType`
`FinalState <MASKED> transitions to State_3 State transitions to InitialState <MASKED>`	`FinalState, Pseudostate`
`SizeVisitor <MASKED> realizes Visitor <MASKED>`	`Class, Interface`

Table 6.12: Masking and label creation for the BoP encoding.

then considered the label for this node. Table 6.12 shows the final result for a number of examples from our dataset.

6.5.3 Comparison and Conclusion

The masked paths and labels of our BoP evaluation closely match those of Ali and Bork. Due to the difference in modeling languages, we omit the stereotype and instead focus on UML types, but match format of their context strings. This result validates the configurability of the encoder.

6.6 Encoder Performance

We measure the execution times of the framework's encoders with varying batch sizes using the Unix `time` command. The runtimes are measured for the configurations A for the raw graph encoder and tree-based encoder, and the configuration of the BoP encoder described in subsection 6.5.1. With the framework's CLI and the Bun runtime, each encoder is executed five times for 100, 1,000, and 10,000 models of the dataset described in section 2.2 respectively. For the raw graph and tree-based encoders, the dataset split ratios are maintained for each of the different total sizes. To achieve comparable results, we normalize execution times to a per-model average. It should be highlighted that the execution times of encoders can vary strongly depending on the configuration of parser and encoder parameters. The resulting execution wall clock times are averaged and displayed in Figure 6.8. To fulfill [**REQ7**], per-model execution times may not exceed 60 milliseconds. The results show that none of the three encoders exceeds 30 milliseconds on average, regardless of the batch size. Further, the raw graph and BoP encoders' averages remain below 20 milliseconds. The reduction of per-model execution times for larger batch sizes may be explained by the overhead of reading input files and validating configuration parameters being distributed across a larger number of models. For the tree-based encoding, we hypothesize that its relatively large serialized output results in its unique increase in per-model execution time for a batch size of 10,000 models.

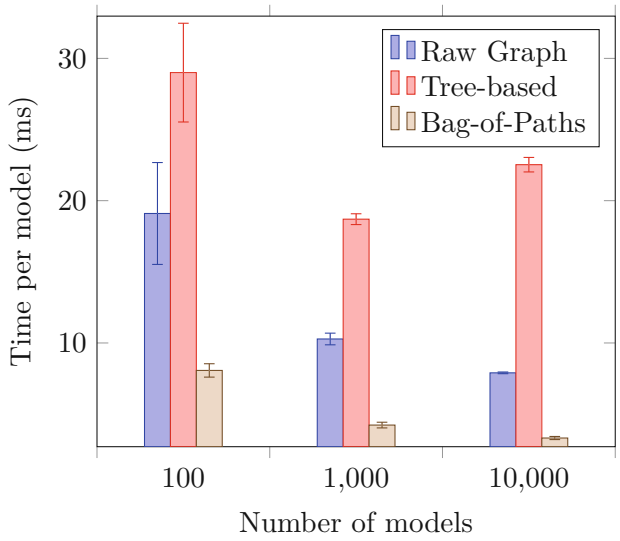


Figure 6.8: Per-model encoder execution times for 100, 1,000, and 10,000 models.

6.7 Threats to Validity

We identify a number of threats to this evaluation's validity. First, the size of the dataset used for the ML evaluations in section 6.3 and section 6.4 only consists of 1,000 elements. For the latter evaluation, additional filtering reduces the number of models in the dataset further. This may skew results for the tree-based encoding in particular, as the dataset may not be large enough to train the ML model properly. Further, CM2ML's parsers and encoders offer a wide range of configurable parameters, while the parameter configurations used for the evaluations only have three varying parameters each. This limited number of experiments, restricted by time, may not suffice to evaluate each parser and encoder and their high configurability. Further, the results of the two ML evaluations may also be affected by hyperparameters more than encoder configurations. However, an optimization of hyperparameter for each encoding configuration would result in a very large search space.

CHAPTER 7

Conclusion

In this chapter, we first summarize the implemented framework, its capabilities, and the resulting implications for the requirements and research questions. Afterward, possible extensions of the framework and opportunities for future work are presented.

7.1 Summary

This section addresses the requirements of section 1.2 and research questions presented in section 1.3 by summarizing the CM2ML framework's capabilities.

7.1.1 Requirements

The evaluation process detailed above allows us to validate that all requirements for the CM2ML framework are indeed met. In the following, we address each requirement individually.

Genericity

While we only evaluate our framework with UML models, the implemented ML node classifiers are entirely independent of any concrete modeling language. Neither the GNNs nor the Tree-LSTM model contain any references to UML in particular. Instead, any information about the concrete modeling language at hand is emitted by the framework as part of the encoding output. Further, the IR used by CM2ML is able to hander common concepts of modeling languages and is also used successfully in related work [FSG22; Kha+22; Lóp+22; SB21]. Associations and references between elements, attributes with types for both nodes and edges, and containment are all representable in the IR. Further, we implement a parser for UML models in the Eclipse Papyrus serialization format and validate that it can successfully parse the 46,731 models from our dataset. We conclude that [**REQ1**] is fulfilled.

Modularity

The UML parser and all encoders are implemented as separate packages, which are usable standalone through our software library. Further, the reusability of CM2ML's composable modules like the feature encoder is displayed within the framework itself. Beyond that, the visualizer uses the framework's modules through their software libraries. Thus, the implemented CM2ML framework is modular and [**REQ2**] is met.

Portability

We ensure the usability of CM2ML's software library and REST server through unit tests and its CLI is enabling the evaluations of chapter 6. Since CM2ML is implemented with TypeScript, it may be used on any platform supporting a JavaScript runtime, including web browsers. Hence, [**REQ3**] is fulfilled.

Extensibility

Both built-in adapters of the framework support a user-configurable extension with custom parsers and encoders in a declarative manner. The provided software library allows users to specify a list of parsers and encoders to be applied to any given adapter. Both the preconfigured CLI and REST use the same software library API to load the built-in parsers and encoders. As a result, the CM2ML framework meets [**REQ4**].

Configurability

With the above evaluations we show that the UML parser and both encoders allow configuration through parameter. The different configurations exhibit both small and large differences in performance for our evaluated task of node classification. This enables experimentation with and fine-tuning of encodings, and guarantees that [**REQ5**] is fulfilled.

Traceability

The visualizer, presented in detail in chapter 4, supports all built-in encoders. Its synchronization between IR and encoder views show how elements within an input model correspond to an output encoding. An extensive suite of E2E tests validates that this synchronization is working as required. Thus, the CM2ML framework's visualizer fulfills [**REQ6**].

Performance

The framework shows a good performance for large batches. Execution times depend on batch size, model complexity, as well as each concrete encoder and its configuration. We consider [**REQ7**] as successfully met based on the execution times measured in section 6.6.

7.1.2 Research Questions

The information presented above answers [**RQ1**]. It is indeed possible to realize a framework fulfilling all requirements of section 1.2.

Further, our evaluation of the raw graph and tree-based encodings for the task of node classification answer [**RQ2**]. The CM2ML framework enables the evaluation of different encodings for this concrete task. CM2ML does not include the final ML applications and their models, unlike the work of López et al. [Lóp+22]. Instead, the framework is focused on rapid iterations during experimentation with encodings and achieves this goal. Through the three supported environments, i.e., CM2ML's two adapters and software library, users are able to customize any built-in encoding to meet their needs in a declarative manner.

7.2 Future Work

During CM2ML's development, a number of possible extensions for future work have been identified. These are presented below in no particular order.

First, an extension of the framework with parsers for additional modeling languages, such as Ecore or OntoUML, would immediately grant access to CM2ML's three built-in encodings to more members of the conceptual modeling community. Beyond that, implementing further structural encodings, e.g., graph kernels or n-grams, may facilitate more extensive comparisons of different encodings.

Next, the visualizer is limited to parsing and encoding a single model at a time. A possible extension could be a batch mode, where users may encode a number of conceptual models at once. Because batch executions are the expected modus operandi for the CLI and REST adapters, users would be able to visualize the encoding for those batch executions as well.

While usually treated as nodes in an IR model, UML relationships are often presented as edges in UML modeling software. The UML parser already offers a sort of *interpretation* of UML elements through the `relationshipsAsEdges` parameter. A future extension of the UML parser could, e.g., introduce an option to interpret UML properties in a similar manner by removing their IR nodes and adding them as IR attributes to their owner's IR node.

The expression and templating language of the BoP encoder offers highly customizable, modeling language-independent expressions. A possible extension of the CM2ML framework could leverage the conditional expression of this language in a number of places. For example, conditional expressions could be used in the IR post-processor for user-customizable filters that further refine an IR instance by removing nodes, edges, or attributes not satisfying any provided condition. Of course, an extension of the language is also possible. In particular, introducing logical operators to conditions and allowing selectors on the right-hand-side of comparison operators would allow for a number of

interesting expression, e.g., a condition `edge.source.id = edge.target.id` that is only satisfied by reflective edges.

List of Figures

List of Tables

List of Algorithms

Acronyms

API application programming interface. 25, 82

BoP Bag-of-Paths. ix, xi, 1, 8, 32, 35, 48, 49, 51–55, 57, 75–78, 83, 85, 87

CLI command-line interface. 2, 3, 11, 12, 15, 16, 22, 59, 76, 78, 82, 83

CM2ML Conceptual Models to Machine Learning. ix, xi, 2–4, 6, 9, 11–13, 17, 18, 22, 23, 25, 43, 49, 57, 60, 62, 75, 76, 79, 81–83, 85

DSR Design Science Research. 2, 3

E2E End-to-End. 13, 32, 82

GAT graph attention network. 63–69, 72, 85, 87

GCN graph convolutional network. 63–69, 72, 85, 87

GLaM4CM Graph Language Modeling framework for Conceptual Models. 8, 48, 57, 75

GNN graph neural network. 8, 60, 61, 63, 74, 75, 81

IR intermediate representation. xi, 3, 4, 7–9, 11, 16–23, 25–27, 29–33, 35–41, 43–48, 51–55, 68, 75, 81–83, 85

ML machine learning. 1–3, 6, 8, 12, 22, 35, 48, 57, 58, 60–64, 66, 69, 76, 79, 81, 83

PWA progressive web application. 25

PYG *PyTorch Geometric*. 57, 63, 64

ReLU rectified linear unit. 63–65

REST representational state transfer. 3, 11, 16, 22, 82, 83

TDD test-driven development. 3, 6, 21, 22

Tree-LSTM tree long short-term memory network. 7, 60, 69–75, 81, 85, 87

UML Unified Modeling Language. 1, 3, 5–8, 11, 17–21, 23, 26, 27, 29, 49, 58–62, 68–70, 75, 76, 78, 81–83, 87

XML Extensible Markup Language. 18, 19, 21, 26

Bibliography

[AB24] Syed Juned Ali and Dominik Bork. „A Graph Language Modeling Framework for the Ontological Enrichment of Conceptual Models“. In: *Advanced Information Systems Engineering*. Ed. by Giancarlo Guizzardi et al. Vol. 14663. Series Title: Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2024, pp. 107–123. ISBN: 978-3-031-61057-8. DOI: 10.1007/978-3-031-61057-8_7. URL: https://link.springer.com/10.1007/978-3-031-61057-8_7 (visited on 09/06/2024).

[Aga19] Abien Fred Agarap. *Deep Learning using Rectified Linear Units (ReLU)*. Feb. 7, 2019. arXiv: 1803.08375[cs,stat]. URL: http://arxiv.org/abs/1803.08375 (visited on 10/11/2024).

[Ali+23] Syed Juned Ali et al. „Encoding Conceptual Models for Machine Learning: A Systematic Review“. In: *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)* (2023). DOI: 10.1109/MODELS-C59198.2023.00094.

[Alm] Almende B.V. *vis-network*. Version 9.1.9. URL: https://github.com/visjs/vis-network (visited on 10/31/2024).

[Ana] Anaconda. *Anaconda Software Distribution*. Version 24.5.0. URL: https://www.anaconda.com (visited on 10/31/2024).

[Ast03] Dave Astels. *Test Driven development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003. ISBN: 0-13-101649-0.

[BAD23] Dominik Bork, Syed Juned Ali, and Georgi Milenov Dinev. „AI-Enhanced Hybrid Decision Management“. In: *Business & Information Systems Engineering* 65.2 (Apr. 2023), pp. 179–199. ISSN: 2363-7005, 1867-0202. DOI: 10.1007/s12599-023-00790-2. URL: https://link.springer.com/10.1007/s12599-023-00790-2 (visited on 10/25/2023).

[BAR23] Dominik Bork, Syed Juned Ali, and Ben Roelens. *Conceptual Modeling and Artificial Intelligence: A Systematic Mapping Study*. Mar. 12, 2023. arXiv: 2303.06758[cs]. URL: http://arxiv.org/abs/2303.06758 (visited on 11/08/2023).

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. „Understanding TypeScript“. In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Red. by David Hutchison et al. Vol. 8586. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281. ISBN: 978-3-662-44202-9. DOI: 10.1007/978-3-662-44202-9_11. URL: http://link.springer.com/10.1007/978-3-662-44202-9_11 (visited on 10/25/2024).

[BC17] Önder Babur and Loek Cleophas. „Using n-grams for the Automated Clustering of Structural Models“. In: *SOFSEM 2017: Theory and Practice of Computer Science*. Ed. by Bernhard Steffen et al. Vol. 10139. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 510–524. ISBN: 978-3-319-51963-0. DOI: 10.1007/978-3-319-51963-0_40. URL: http://link.springer.com/10.1007/978-3-319-51963-0_40 (visited on 05/21/2024).

[BM22] Justus Bogner and Manuel Merkel. „To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub“. In: *Proceedings of the 19th International Conference on Mining Software Repositories*. May 23, 2022, pp. 658–669. DOI: 10.1145/3524842.3528454. arXiv: 2203.11115[cs]. URL: http://arxiv.org/abs/2203.11115 (visited on 10/25/2024).

[Bur+22] Loli Burgueño et al. „A generic LSTM neural network architecture to infer heterogeneous model transformations“. In: *Software and Systems Modeling* 21.1 (Feb. 2022), pp. 139–156. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-021-00893-y. URL: https://link.springer.com/10.1007/s10270-021-00893-y (visited on 05/28/2024).

[Dah] Ryan Dahl. *Node.js*. Version 22.8.0. URL: https://github.com/nodejs/node (visited on 10/31/2024).

[DDZ08] J. Dingel, Z. Diskin, and A. Zito. „Understanding and improving UML package merge“. In: *Software & Systems Modeling* 7.4 (Oct. 2008), pp. 443–467. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-007-0073-9. URL: http://link.springer.com/10.1007/s10270-007-0073-9 (visited on 04/11/2024).

[Di +21] Juri Di Rocco et al. „A GNN-based Recommender System to Assist the Specification of Metamodels and Models“. In: *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS). Fukuoka, Japan: IEEE, Oct. 2021, pp. 70–81. ISBN: 978-1-66543-495-9. DOI: 10.1109/MODELS50736.2021.00016. URL: https://ieeexplore.ieee.org/document/9592428/ (visited on 05/07/2024).

[EGO] EGOIST. *CAC*. Version 6.7.14. URL: https://github.com/cacjs/cac (visited on 10/31/2024).

[FL19] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. Apr. 25, 2019. arXiv: 1903.02428[cs]. URL: http://arxiv.org/abs/1903.02428 (visited on 10/31/2024).

[FSG22] Mattia Fumagalli, Tiago Prince Sales, and Giancarlo Guizzardi. „Pattern Discovery in Conceptual Models Using Frequent Itemset Mining". In: *Conceptual Modeling*. Ed. by Jolita Ralyté et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 52–62. ISBN: 978-3-031-17995-2. DOI: 10.1007/978-3-031-17995-2_4.

[Gér19] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*. OCLC: 1153040518. Place of publication not identified: O'Reilly Media, Inc., 2019. ISBN: 978-1-4920-3263-2.

[Har+20] Charles R. Harris et al. „Array Programming with NumPy". In: *Nature* 585.7825 (Sept. 17, 2020), pp. 357–362. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-2649-2. arXiv: 2006.10256[cs]. URL: http://arxiv.org/abs/2006.10256 (visited on 10/31/2024).

[Hen] Paul Henschel. *zustand*. Version 4.5.5. URL: https://github.com/pmndrs/zustand (visited on 11/11/2024).

[Hev+04] Alan R. Hevner et al. „Design Science in Information Systems Research". In: *MIS Quarterly* 28.1 (2004). Publisher: Management Information Systems Research Center, University of Minnesota, pp. 75–105. ISSN: 0276-7783. DOI: 10.2307/25148625. URL: https://www.jstor.org/stable/25148625 (visited on 10/31/2023).

[Kha+22] Alireza Khalilipour et al. „Categorization of the Models Based on Structural Information Extraction and Machine Learning". In: *Intelligent and Fuzzy Systems*. Ed. by Cengiz Kahraman et al. Lecture Notes in Networks and Systems. Cham: Springer International Publishing, 2022, pp. 173–181. ISBN: 978-3-031-09176-6. DOI: 10.1007/978-3-031-09176-6_21.

[KW17] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. Feb. 22, 2017. arXiv: 1609.02907[cs,stat]. URL: http://arxiv.org/abs/1609.02907 (visited on 10/09/2024).

[LC20] José Antonio Hernández López and Jesús Sánchez Cuadrado. *MAR: A structure-based search engine for models*. Aug. 26, 2020. arXiv: 2008.11858[cs]. URL: http://arxiv.org/abs/2008.11858 (visited on 04/11/2024).

[LCC22] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. „ModelSet: a dataset for machine learning in model-driven engineering“. In: *Software and Systems Modeling* 21.3 (June 2022), pp. 967–986. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-021-00929-3. URL: https://link.springer.com/10.1007/s10270-021-00929-3 (visited on 04/11/2024).

[Lóp+22] José Antonio Hernández López et al. „Machine learning methods for model classification: a comparative study“. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. MODELS '22: ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems. Montreal Quebec Canada: ACM, Oct. 23, 2022, pp. 165–175. ISBN: 978-1-4503-9466-6. DOI: 10.1145/3550355.3552461. URL: https://dl.acm.org/doi/10.1145/3550355.3552461 (visited on 07/15/2024).

[Luo+17] Zhiling Luo et al. „Deep Learning of Graphs with Ngram Convolutional Neural Networks“. In: *IEEE Transactions on Knowledge and Data Engineering* 29.10 (Oct. 1, 2017), pp. 2125–2139. ISSN: 1041-4347. DOI: 10.1109/TKDE.2017.2720734. URL: http://ieeexplore.ieee.org/document/7961216/ (visited on 05/22/2024).

[McD] Colin McDonnell. *Zod*. Version 3.23.8. URL: https://github.com/colinhacks/zod (visited on 10/31/2024).

[Met] Meta Platforms, Inc. *React*. Version 18.3.1. URL: https://github.com/facebook/react (visited on 10/31/2024).

[Mica] Microsoft Corp. *Playwright*. Version 1.48.2. URL: https://github.com/microsoft/playwright (visited on 11/01/2024).

[Micb] Microsoft Corp. *TypeScript*. Version 5.6.3. URL: https://github.com/microsoft/TypeScript (visited on 10/31/2024).

[Mül] Jan Patrick Müller. *vite-plugin-lib*. Version 2.1.3. URL: https://github.com/DerYeger/yeger/tree/main/packages/vite-plugin-lib (visited on 10/31/2024).

[Ngu+19] Phuong T. Nguyen et al. „Automated Classification of Metamodel Repositories: A Machine Learning Approach“. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS). Sept. 2019, pp. 272–282. DOI: 10.1109/MODELS.2019.00011.

[npm] npmjs.com. *Generating provenance statements*. URL: https://docs.npmjs.com/generating-provenance-statements (visited on 11/01/2024).

[Obj17] Object Management Group®. *Unified Modeling Language, v2.5.1*. Dec. 2017. URL: https://www.omg.org/spec/UML/2.5.1/PDF.

[Off+09] Philipp Offermann et al. „Outline of a design science research process“. In: *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology - DESRIST '09*. the 4th International Conference. Philadelphia, Pennsylvania: ACM Press, 2009, p. 1. ISBN: 978-1-60558-408-9. DOI: 10.1145/1555619.1555629. URL: http://portal.acm.org/citation.cfm?doid=1555619.1555629 (visited on 11/02/2024).

[Ope] OpenJS Foundation. *ESLint*. Version 9.13.0. URL: https://github.com/eslint/eslint (visited on 11/01/2024).

[Pas+19] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Dec. 3, 2019. arXiv: 1912.01703[cs]. URL: http://arxiv.org/abs/1912.01703 (visited on 10/31/2024).

[Ped+11] Fabian Pedregosa et al. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[San+21] Benjamin Sanchez-Lengeling et al. „A Gentle Introduction to Graph Neural Networks“. In: *Distill* 6.8 (Aug. 17, 2021), 10.23915/distill.00033. ISSN: 2476-0757. DOI: 10.23915/distill.00033. URL: https://distill.pub/2021/gnn-intro (visited on 10/27/2024).

[SB21] Muhamed Smajevic and Dominik Bork. „From Conceptual Models to Knowledge Graphs: A Generic Model Transformation Platform“. In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). Fukuoka, Japan: IEEE, Oct. 2021, pp. 610–614. ISBN: 978-1-66542-484-4. DOI: 10.1109/MODELS-C53483.2021.00093. URL: https://ieeexplore.ieee.org/document/9643650/ (visited on 10/11/2023).

[sha] shadcn. *shadcn/ui*. Version 2.1.2. URL: https://github.com/shadcn-ui/ui (visited on 10/31/2024).

[Sma22] Muhamed Smajevic. „A Generic Conceptual Model to Graph Transformation Framework and its Application for Enterprise Architecture Analysis“. PhD thesis. TU Wien, 2022. 130 pp. URL: https://repositum.tuwien.at/handle/20.500.12708/135936 (visited on 08/29/2024).

[SR17] Mohammad Ali Jabbari Sabegh and Jan Recker. „Combined Use of Conceptual Models in Practice: An Exploratory Study“. In: *Journal of Database Management* 28.2 (Apr. 2017), pp. 56–88. ISSN: 1063-8016, 1533-8010. DOI: 10.4018/JDM.2017040103. URL: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/JDM.2017040103 (visited on 11/09/2024).

[Sum] Jarred Sumner. *Bun*. Version 1.0.24. URL: https://github.com/oven-sh/bun (visited on 10/31/2024).

[The] The Fastify Team. *Fastify*. Version 5.0.0. URL: https://github.com/fastify/fastify (visited on 10/31/2024).

[Vel+18] Petar Veličković et al. *Graph Attention Networks*. Feb. 4, 2018. arXiv: 1710.10903[cs, stat]. URL: http://arxiv.org/abs/1710.10903 (visited on 10/10/2024).

[Ver] Vercel Inc. *Turborepo*. Version 2.2.3. URL: https://github.com/vercel/turborepo (visited on 10/31/2024).

[Vit] Vitest Team. *Vitest*. Version 2.1.3. URL: https://github.com/vitest-dev/vitest (visited on 11/01/2024).

[Voi] VoidZero Inc. *Vite*. Version 5.4.10. URL: https://github.com/vitejs/vite (visited on 10/31/2024).

[War] Alessandro Warth. *Ohm*. Version 17.1.0. URL: https://github.com/ohmjs/ohm (visited on 10/31/2024).

[web] webkid GmbH. *React Flow*. Version 11.11.4. URL: https://github.com/xyflow/xyflow (visited on 10/31/2024).

[Win] Chris Winberry. *htmlparser2*. Version 9.1.0. URL: https://github.com/fb55/htmlparser2 (visited on 10/31/2024).

[WSS22] Martin Weyssow, Houari Sahraoui, and Eugene Syriani. „Recommending Metamodel Concepts during Modeling Activities with Pre-Trained Language Models“. In: *Software and Systems Modeling* 21.3 (June 2022), pp. 1071–1089. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-022-00975-5. arXiv: 2104.01642[cs]. URL: http://arxiv.org/abs/2104.01642 (visited on 05/28/2024).