INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

WITNESSDB

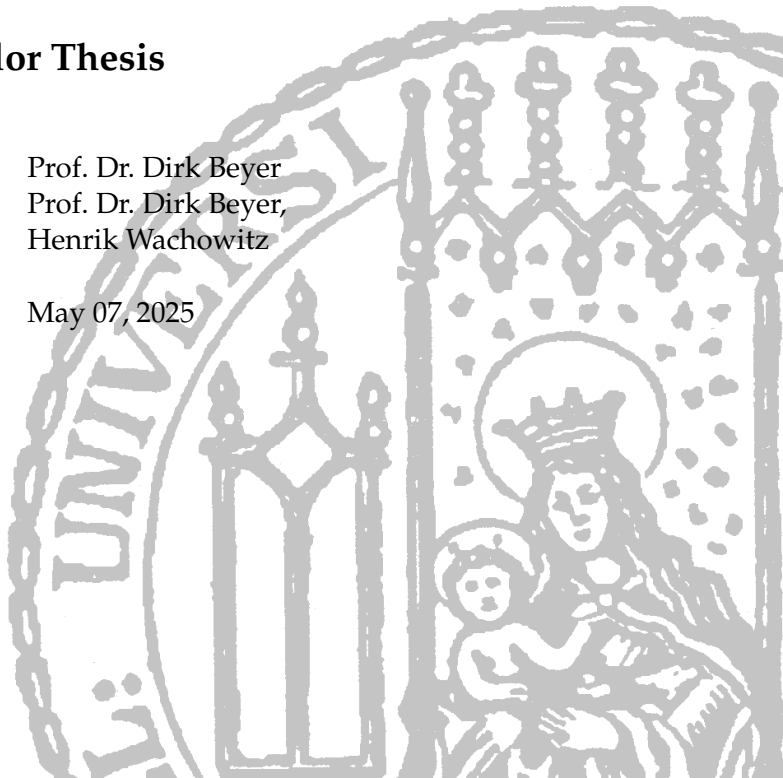
A Database for Verification Witnesses 2.0 (Neo4j)

Ivayla Ivanova

Bachelor Thesis

Supervisor Prof. Dr. Dirk Beyer
Advisor Prof. Dr. Dirk Beyer,
Henrik Wachowitz

Submission Date May 07, 2025



Statement of Originality

English:

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments.

Deutsch:

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, May 07, 2025 Ivayla Ivanova

Statement on Use of Artificial Intelligence

During the writing of this thesis, large language models were used to assist with wording, sentence structure, and overall readability. This assistance was strictly limited to language-related improvements. All ideas, research, analysis, and conclusions presented in this thesis are entirely my own original work.

München, May 07, 2025 Ivayla Ivanova

Abstract

Ensuring the correctness of software systems remains a critical goal in software engineering, with formal verification providing strong guarantees through mathematical proofs. Verification witnesses have been introduced as standardized artifacts that capture the verification process to enhance the transparency, reproducibility, and trustworthiness of verification results. Despite their value for validation, benchmarking, and research, these witnesses are currently stored only as individual files within large archives, which is inconvenient for large-scale querying, analysis, and visualization.

This thesis presents WitnessDB, a system for storing, validating, and exploring verification witnesses using the Neo4j graph database. WitnessDB transforms verification results encoded in Witness Format 2.0 (YAML) into graph structures and allows querying and interactive analysis through graphical and tabular interfaces. By leveraging the capabilities of a graph database, WitnessDB demonstrates how complex, interconnected verification data can be modeled in ways that potentially unlock new use cases and encourage new approaches to analysis.

Contents

1 Introduction

Ensuring the correctness of software systems has always been a fundamental concern in software engineering. Whether in low-level embedded systems or high-level application software, bugs and unintended behaviors can lead to critical failures, financial loss, or even endanger lives. Formal verification offers strong guarantees about system behavior by mathematically proving properties of programs [9]. Unlike traditional testing, which can only uncover the presence of bugs through a limited set of test cases, formal methods allow reasoning over all possible executions of a program.

In the ideal case, verifiers can determine whether a program satisfies a given specification by either proving its correctness or revealing a concrete violation [5]. However, verifiers are not immune to failure. As complex software systems themselves, they may contain implementation bugs [1, 3] and need to be further developed. Misleading results can also arise from imprecise specifications that fail to fully capture the intended behavior of the program. A common source of unreliability is the use of abstraction: to manage complexity, verifiers often simplify the program's behavior, which can result in missed bugs or incorrect confirmations if important details are lost [1, 5]. Verification tools also can often rely on assumptions about the environment or inputs that may not hold in real-world scenarios.

To improve the reliability of verification results, verification witnesses have been introduced as a solution [1, 5, 6]. Verification witnesses are structured artifacts that are additionally generated next to the results of a formal verification process. These artifacts serve as a way to make verification results more transparent, reproducible, and trustworthy by providing additional context on how the verification was carried out [5]. A key advantage of verification witnesses is that their format is standardized and human readable, which makes them exchangeable across different verification tools [1, 6]. This is particularly valuable in a competitive landscape, as it allows verification results from one tool to be validated by another, regardless of the underlying toolset. Competitions like SV-COMP allow different verifiers to be tested against a common set of benchmarks to highlight their strengths and areas for improvement [7].

There are plenty of verification witness datasets available, especially from competitions like SV-COMP, but they're not being used to their full potential because of how they're stored. These witnesses, which have potential applications ranging from research to tool development and benchmarking, are currently stored in files and bundled into large ZIP archives [2, 11]. This storage method makes it inconvenient to access, query, analyze, or visualize individual witnesses, especially at scale.

To address this challenge, this thesis presents *WitnessDB*, a system designed to store, manage, and explore verification witnesses using the *Neo4j* graph database. WitnessDB focuses on processing verification results represented in the standardized *Witness Format 2.0*, which encodes verification artifacts in YAML [1]. The system provides an integrated pipeline that transforms these structured YAML witnesses into a graph representation to enable storage, validation, querying, and interactive exploration through both graphical and tabular interfaces. The scope of this work includes the design of the graph schema, implementation of the import, export and validation processes and development of an interactive data explorer for analyzing the stored verification witnesses.

2 Background & Related Work

2.1 Witnesses

2.1.1 Overview

The introduction of *Witness Format 1.0* has been an important milestone in the standardization of verification results, establishing a common structure for exchanging correctness and violation information between verification tools [6]. To ensure a reliable verification process, specific data must be collected at different stages of program verification and persisted for later validation [5].

Figure 2.1 describes the process of verifying and validating a program.

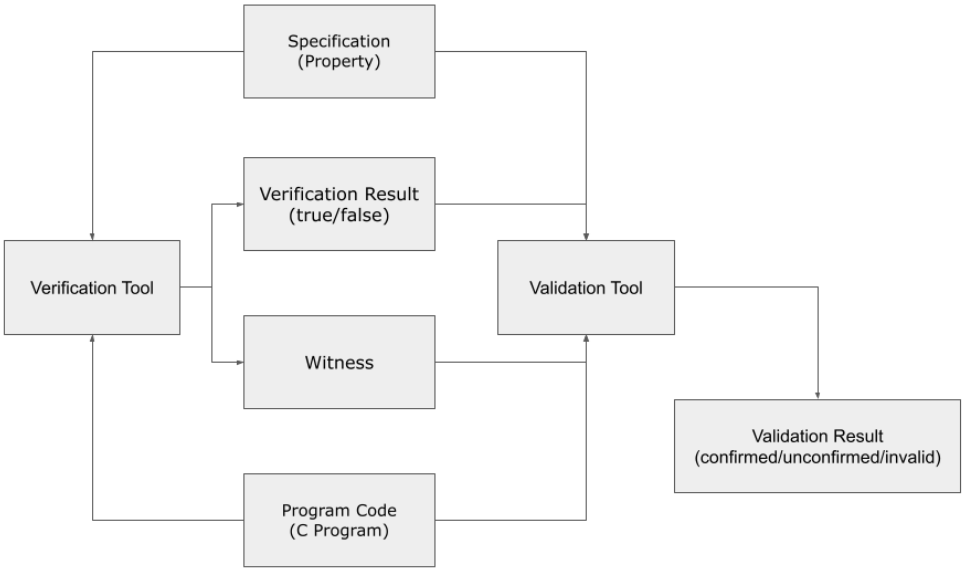


Figure 2.1: Program Verification and Validation Process

The following data is needed during this process [5]:

Property Specification. At the beginning of the process, the property to be verified must be clearly defined. This specification could be in the form of logical formulas, assertions, or reachability properties, often tied directly to the source code.

Program Code. The actual program being verified.

Verification Tool and Version. The tool used to verify the program, along with its version number to ensure reproducibility of results.

Verification Result (True or False). The outcome of a verification task is a binary decision indicating whether the program satisfies the given property. A result of true signifies that the property holds for all possible executions of the program, while false means that the property is violated in at least one execution path. Although some verification tools may also return unknown when the result cannot be determined [3], such cases are outside the scope of this work.

Witness. Alongside the result, a witness provides additional structured information to explain the outcome. A correctness witness is produced when the verification result is *true*. It helps confirm that the program is correct with respect to the specified property by providing evidence, such as invariants, that the verification process was performed correctly and that no counterexamples (violations) exist. A violation witness is produced when the result is *false*. It captures the specific state, path, or sequence of events leading to the property's failure. This makes it easier to identify exactly where the program fails and why, helping to identify bugs or unwanted behaviors in the code.

The initial Witness format was based on GraphML and modeled program behavior using control-flow automata, allowing tools to visualize and communicate program paths in a structured way.

While Format 1.0 enabled the first wave of interoperability in verification competitions, it soon showed limitations: the format was verbose, difficult to interpret manually, and lacked precise semantic definitions tied to source code. Validator support was limited, and rarely used features added unnecessary complexity.

To address these issues, Witness Format 2.0 was developed, replacing GraphML with a more concise and human-readable YAML format and defining semantics directly in terms of the programming language — typically C. It eliminates rarely used features, ensuring better validator compatibility and easier adoption.

2.1.2 Witness Format 2.0

Witness Format 2.0 provides a structured way to encode all relevant aspects of the verification process.

The `metadata` section of a witness captures the verification context: it includes the specification being checked, the path of the input files being verified along with SHA-256 hash of their content, information about the verification tool — its name and version, the creation time of the witness and additional configuration details. A snippet of a `metadata` section illustrating these fields is shown in Listing 1.

```
1 - entry_type: ...
2   metadata:
3     format_version: '2.0'
4     uuid: d127a331-bba7-45f1-b989-26fc7d0f064e
5     creation_time: '2024-10-31T21:57:28+01:00'
```

```
6       producer:
7         name: Automizer
8         version: 0.2.5-dev-4fc63b2
9       task:
10        input_files:
11        - /tmp/vcloud_worker_vcloud-master_on_vcloud-master/
              run_dir_f2b63a4e-8427-4ab0-9465-2c6a3eda9752/sv-
              benchmarks/c/nla-digbench-scaling/ps3-ll_unwindbound20
              .c
12        input_file_hashes:
13          ? /tmp/vcloud_worker_vcloud-master_on_vcloud-master/
                run_dir_f2b63a4e-8427-4ab0-9465-2c6a3eda9752/sv-
                benchmarks/c/nla-digbench-scaling/ps3-
                ll_unwindbound20.c
14          : 81
                afa9e511be2b9bae9d4ea2d6181d1112c109837a9a39236c346cbee76c712e

15        specification: |+
16          CHECK( init(main()), LTL(G ! call(reach_error())) )
17
18        data_model: ILP32
19        language: C
20      content:
21       ...
```

Listing 1: Snippet of the `metadata` section in a Witness Format 2.0 file

The outcome of the verification task is reflected in the `entry_type` field, which is set to either `invariant_set` (for a correctness witness) or `violation_sequence` (for a violation witness) to correspond to a verification result of `true` or `false`.

The `content` section of the witness holds the core data justifying the result, such as the invariants for correctness or the error path for violations. See Listings 2 and 3 for examples of a correctness and a violation witness.

```
1
2   - entry_type: invariant_set
3     metadata:
4       ...
5     content:
6     - invariant:
7         type: loop_invariant
8         location:
9           file_name: /tmp/vcloud_worker_vcloud-master_on_vcloud-
                master/run_dir_f2b63a4e-8427-4ab0-9465-2c6a3eda9752/
                sv-benchmarks/c/nla-digbench-scaling/ps3-
                ll_unwindbound20.c
10          file_hash: 81
                afa9e511be2b9bae9d4ea2d6181d1112c109837a9a39236c346cbee76c712e

11          line: 27
12          column: 5
```

```
13          function: main
14        value: (((__int128) 6 * x) == (((3 * ((__int128) y * y))
             + ((((__int128) y * y) * y) * 2)) + y))
15        format: c_expression
16    - invariant:
17        ...
```

Listing 2: Snippet of the `content` section in a Correctness Witness Format 2.0 file

```
1
2  - entry_type: "violation_sequence"
3    metadata:
4      ...
5    content:
6    - segment:
7      - waypoint:
8          type: "branching"
9          action: "follow"
10         constraint:
11           value: "false"
12         location:
13           file_name: "../../sv-benchmarks/c/ldv-regression/
                fo_test.i"
14           line: 1003
15           column: 2
16           function: "l_open"
17   - segment:
18     ...
```

Listing 3: Snippet of the `content` section in a Violation Witness Format 2.0 file

This overview does not cover every mandatory and optional field in the Witness Format 2.0 specification. For a more detailed understanding of all fields and their definitions see Ayaziová et al. [1].

2.1.3 Tools around Witnesses

2.1.3.1 Validation

As mentioned previously, a variety of tools exist that validate the content of both correctness and violation verification witnesses. These validators analyze the witness data in combination with the program code to confirm whether the witness truly justifies the verification result. An overview of the current state of witness validation tools, as used in SV-COMP, is provided by Beyer and Strejček [7].

WitnessLint is a tool that focuses on validating the syntax and structural correctness of Witness Format 2.0 YAML files. Unlike semantic validators, this tool does not interpret the witness in relation to the program behavior, but instead ensures compliance with the format specification. It verifying that all required fields are present and correctly typed. In this project, WitnessLint is used to ensure that only well-formed witnesses are accepted and stored in the database.

2.1.3.2 Visualization

Making verification processes more understandable through visualization has long been a focus in software verification [1]. Among other things, when introducing *Witness Format 1.0* in GraphML Beyer et al. [6] mention the potential for visualizations as one of its advantages, which offers developers the option to inspect error paths and makes verification results more accessible.

The *Verification-Aided Debugging* via a *Witness Store*, introduced by Beyer et al. [4], is one of the most significant projects for visualizing and exploring verification witnesses. It consists of three main subsystems: *Witness Store*, *Witness Validator* and *Interactive Bug Report*: The Witness Store acts as a persistent repository where violation witnesses in Witness Format 1.0 (GraphML) are stored, allowing users to revisit and validate previous verification results. The *Interactive Bug Report* provides visualizations upon successful validation that link error paths to system source code, control-flow graphs, and test values, offering deeper insights into the verification process. The Witness Validator is an online service with a web-based API what provides semantic validation of violation witnesses.

In addition to major projects like the Witness Store, several student works have explored visualization within the broader domain of formal verification. These include efforts such as developing a web frontend for visualizing CPAchecker's computation steps and results [16], introducing new approaches to visualize verification coverage and improving the user interface [14] and modernizing the architecture of BenchExec tables [8]. While these projects do not focus specifically on witness visualization, they highlight the growing interest in making verification results more accessible and interpretable through visual means.

2.1.3.3 Storage

During the most recent iteration of SV-COMP (2025) verification witnesses are archived and organized using a hash-based directory structure to ensure efficient access and integrity. Each witness is stored in the `fileByHash/` directory under a file name corresponding to its SHA2-256 hash, with extensions `.graphml` or `.yml`, depending on the format. Metadata for each witness is stored additionally in the `witnessInfoByHash/` directory as JSON files, also named using the hash of the corresponding witness. The `witnessListByProgramHashJSON/` directory maps program hashes to sets of associated witness metadata, enabling retrieval of all witnesses related to a specific program.

For further details on the structure and contents of the dataset see [11] and [2].

2.2 Graph Databases and Neo4j

Graph databases are designed to represent data as a network of interconnected entities, using a structure that emphasizes relationships. The most widely adopted graph model is the labeled property graph, which organizes data into two key components: nodes and relationships. Both nodes and relationships can store properties, which are key-value pairs that describe the characteristics of the element.

In this model:

- *Nodes* can have one or more labels that define their role or type within the graph.
- *Relationships* are directed and named, always connecting a start node to an end node.
- Both nodes and relationships can carry properties.
- Nodes can be connected by multiple relationships, each representing a different type of connection.
- There can only be one relationship of a given type in one direction between two nodes.

Neo4j is a labeled property graph database that leverages native graph storage and index-free adjacency, which distinguishes it from other graph and non-graph database systems [22].

Native graph storage refers to a storage architecture specifically designed to persist graph elements directly as graph structures, rather than mapping them onto relational tables or document formats. In such systems, nodes and relationships are stored as fixed-size records, and properties are kept in associated structures that can be quickly accessed [22, 24].

Index-free adjacency is a processing model where each node maintains direct references (often implemented as physical pointers or internal IDs) to its adjacent relationships and neighboring nodes. This means that traversing from one node to another does not require index lookups or joins. As a result, traversal performance becomes independent of the total size of the graph and instead depends solely on the local structure around the traversed nodes [22, 24].

The combination of the labeled property graph model, native graph storage, and index-free adjacency makes neo4j particularly well-suited for domains where relationships between data points are as important as the data itself.

Relational databases are optimized for tabular data and often struggle to efficiently represent or query complex relationships. Modeling highly interconneted data in a relational database typically requires multiple join tables, foreign keys,

and self-referencing tables, leading to expensive multi-way joins and degraded performance as the data becomes more connected [12].

Do et al. [12] provides a comparative query-based performance evaluation of Neo4j and MySQL across four common query types: selection/search, recursion, aggregation, and pattern matching. The authors modeled the same data both relationally and as a labeled property graph. The results demonstrate that Neo4j consistently outperforms MySQL, with particularly significant gains in queries involving complex joins.

Examples from the study in Figures 2.2, 2.3 and 2.4 illustrate the difference in query complexity between relational and graph databases. The dataset is taken from CareerVillage.org, a nonprofit platform that crowdsources career advice for young people.

Neo4j uses Cypher, a declarative graph query language specifically designed for working with property graphs [13, 18]. Instead of relying on joins, Cypher queries use a pattern-matching approach where nodes and relationships are represented using ASCII-art-like syntax. Cypher supports a wide range of operations, including filtering, aggregation, path querying, subqueries, and graph projections.

Question: Looking for students in a specific group and interested in a specific tag?

SQL Query

```
SELECT * FROM students s
JOIN group_memberships gm
ON students_id = gm.group_memberships_user_id
JOIN groups g ON
g.groups_id = gm.group_memberships_group_id
JOIN tag_users tu ON
tu.tag_users_user_id = s.students_id
JOIN tags t ON t.tags_tag_id = tu.tag_users_tag_id
WHERE t.tags_tag_name = 'colleg'
AND g.groups_group_type = 'youth program';
```

Cypher Query

```
MATCH (t:tags)<-[:HAS_TAG]-(s:students)-[:MEMBER_IN]->(b)
WHERE t.tags_tag_name='college'
AND b.groups_group_type='youth program'
RETURN s,t,b
```

Figure 2.2: Selection/Search Query Comparison

Question: Which tag has the most professionals?

SQL Query

```
SELECT tags.tags_tag_id, tags_tag_name,
COUNT(p.professionals_id)
AS number_of_professionals FROM tags
JOIN tag_users tu
ON tags.tags_tag_id = tu.tag_users_tag_id
JOIN professionals p ON p.professionals_id =
tu.tag_users_user_id
GROUP BY tags.tags_tag_id, tags_tag_name
ORDER BY COUNT(p.professionals_id)
DESC LIMIT 1;
```

Cypher Query

```
MATCH (p:professionals)-[:HAS_TAG]->(t:tags)
RETURN t.tags_tag_name AS TagName,
COUNT(p) ORDER BY COUNT(p)
DESC LIMIT 1
```

Figure 2.3: Aggregation Query Comparison

Question: Looking for students and professionals with the same group?

SQL Query

```
SELECT g.groups_id, professionals_id, students_id
FROM groups g
JOIN group_memberships gm ON
g.groups_id = gm.group_memberships_group_id
JOIN (SELECT group_memberships_group_id
AS group_id, professionals_id
FROM professionals p
JOIN group_memberships gm1 ON gm1.group_memberships_user_id =
p.professionals_id) pg ON pg.group_id =
gm.group_memberships_group_id
JOIN (SELECT group_memberships_group_id
AS group_id, students_id
FROM students s JOIN group_memberships gm2 ON
s.students_id = gm2.group_memberships_user_id) sg
ON sg.group_id= gm.group_memberships_group_id;
```

Cypher Query

```
MATCH (p:professionals)-[]->(g:groups)<-[]-(s:students)
RETURN p, g, s
```

Figure 2.4: Pattern matching Query Comparison

3 Implementation

3.1 System Architecture Overview

This thesis presents a modular architecture composed of four key components, each fulfilling a specific role within the data processing pipeline. Together, they enable users to upload verification witness files, validate them against the Witness Format 2.0 specification, transform and store their contents in a graph database, interactively explore the stored data, and finally reconstruct the original files when needed. To maintain consistency and streamline deployment, the entire architecture is built using Docker containers. The system follows a microservices pattern, with each component functioning as an independent service that communicates through well-defined APIs.

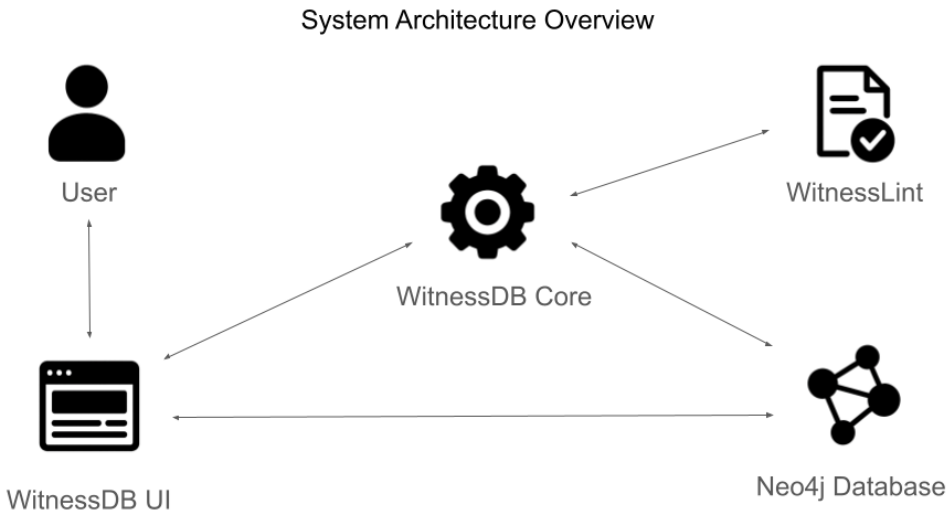


Figure 3.1: System Architecture Overview

For storing processed verification data, the system relies on a single Neo4j database instance. Depending on the selected schema, the data in Neo4j is structured either to replicate the original file layout or to reflect real-world domain relationships.

The next key component is WitnessLint, an external validation tool, as described in 2.1.3.1. It has been integrated into the project through a custom interface that provides an HTTP API for interaction with the core processing component. WitnessLint ensures that YAML-based witness files comply with the Witness Format 2.0 specification during both the initial upload and the reconstruction process,

before they are made available for download.

The *WitnessDB Core* component handles the core processing logic for witness files. It receives files from the user-facing component, validates them using WitnessLint, transforms them into an appropriate graph structure, and stores the resulting data in Neo4j. When requested, WitnessDB Core retrieves data from the database to reconstruct valid YAML files, which are then revalidated with WitnessLint before being provided to the user through the UI component.

The *WitnessDB UI* component serves as the user interface for the system, interacting with both the core processing component and the Neo4j database instance. As it is implemented as a single-page application (SPA), it makes it possible for users to perform all major operations without requiring full page reloads. Users can upload witness files for processing by the core component and download reconstructed files packaged as ZIP archives through the UI. The component also includes a dynamic explorer feature connected directly to the Neo4j database, which allows users to construct and execute custom Cypher queries. Query results are presented either visually as graphs or in tabular form. Export functionality is also available, enabling users to save graphs as PNG images or tables as CSV files.

3.2 Technology Stack

3.2.1 Neo4j Database Instance

The Neo4j database instance operates using two primary communication protocols: HTTP and the Bolt protocol. These protocols enable different ways to interact with the database, each with its own characteristics and use cases [17].

HTTP. The HTTP API allows web-based interaction with the database through interfaces like Neo4j Browser [19]. It also enables users to perform operations and execute Cypher queries via HTTP requests. While the HTTP API is very flexible, it typically has higher latency and lower throughput compared to the Bolt protocol.
Bolt protocol. It is the default method for communication between the Neo4j database and client applications. It is a binary protocol designed specifically for high-performance interactions with the database. The protocol is optimized for low-latency, high-throughput communication. Both the Neo4j Java Driver and the Neo4j JavaScript Driver, used in this project, connect to the database over the Bolt protocol.

3.2.1.1 Neo4j Community Edition Limitations

This project uses the *Neo4j Community Edition* due to its open-source nature , which comes with some limitations compared to the *Enterprise Edition*:

User Management. The Neo4j Community Edition supports only a single default user and does not offer advanced user or role management features available in the

Enterprise Edition. As a result, complex authentication and authorization are outside the scope of this project. Some permission control is handled programmatically within the application, which will be discussed in later sections.

Single Database Instance. The Neo4j Community Edition restricts deployment to a single database instance. This limitation influenced the application's architecture, which is designed around a single-node setup and will be discussed in later sections.

3.2.2 WitnessDB Core

The WitnessDB Core component is implemented in *Kotlin* using the *Spring* framework with its extensions.

Kotlin [15] was chosen primarily for its strong static typing and robust nullability support, which were critical given the challenges arising from the complexity of working with verification witness files. The language's type system provides compile-time guarantees, significantly reducing the risk of runtime errors caused by null references. Kotlin's data classes simplify the representation of data structures, which reduced boilerplate code in the graph transformations and file processing logic. Kotlin also emphasizes immutability by default, especially with collections like `List<T>`, ensuring data cannot be altered accidentally. Thanks to Kotlin's full compatibility with Java, it is possible to utilize a wide range of well-established Java libraries.

Spring [23] was selected due to its comprehensive support for microservice architectures. Key features include:

Embedded Web Server. By using an embedded Tomcat server, Spring Boot allows the application to run independently without requiring an external web server, which simplifies the deployment within Docker containers. Tomcat is a lightweight, open-source web server. It is designed to run Java-based web applications.

REST Controller Support. Spring Boot provides high-level support for building RESTful APIs, including easy request-to-method mapping, automatic handling of JSON serialization and deserialization and integrated exception management.

Dependency Injection. The framework automatically handles the instantiation and lifecycle of application components which reduces boilerplate code and simplifies the maintenance.

Integration with Neo4j. Spring Data Neo4j, part of the Spring ecosystem, simplifies the integration with the Neo4j graph database by managing the mapping of Kotlin or Java domain entities to graph data structures. It utilizes Spring Data's repository abstraction which provides a set of predefined methods for common database operations such as save, delete, and find. Entities are mapped to Neo4j nodes, relationships, and properties using domain annotations such as `@Node` and `@Relationship`. Queries can be executed with method name conventions or using custom Cypher queries defined via the `@Query` annotation.

For the serialization and deserialization of YAML witness files, the project uses the *Jackson* library. While Jackson was originally designed for JSON, it can handle other formats like XML and YAML through additional modules and it is widely recognized for its rich feature set.

3.2.3 WitnessDB UI

The WitnessDB UI is developed using *JavaScript* with the *Vue.js* framework [**?**]. Vue.js was selected primarily due to its component-based architecture and native support for building single-page applications with integrated routing capabilities.

Vue.js is built around several key principles:

Declarative Rendering. Vue allows developers to describe the desired UI structure directly in templates.
Reactivity System. Data-driven components automatically track changes in application state and update the DOM.
Component Composition. Complex UIs are built by composing small, reusable components, each managing its own data and behavior.
Routing Integration. Vue Router manages navigation without requiring full page reloads to ensure SPA behavior.

Graph rendering in the explorer view is handled using the *Neo4j Visualization Library (NVL)* [20]. NVL is a lightweight, framework-agnostic library designed for building interactive graph visualizations The library is also used in the enterprise visualization tool Neo4j Bloom and Explore. It renders graphs by receiving a container element, arrays of nodes and relationships, customizable options, and callback functions.

To communicate directly with the Neo4j database instance, the the UI component uses the official Neo4j JavaScript Driver, which offers a low-level API for executing Cypher queries over the Bolt protocol. The driver supports ResultTransformers, an interface that allows developers to customize how the results of Cypher queries are formatted. For the graph visualization the *Explore Data* feature relies on the `nvlResultTransformer` [21] that is specifically designed to make the query results compatible with the NVL.

To handle HTTP communication with the WitnessDB Core, the UI compoent uses *Axios*, a promise-based HTTP client that simplifies sending asynchronous requests.

3.2.4 WitnessLint

WitnessLint is written in Python, and to maintain a consistent stack for containerization, the custom API is also implemented in Python. The API server uses *Uvicorn*, a high-performance ASGI server designed for asynchronous web applications, making it well-suited for I/O-bound tasks. The API itself is built with FastAPI, which integrates well with Uvicorn.

3.3 File Upload Process and Storage Logic

This section describes the file upload process, from user interaction to data storage in Neo4j, including preprocessing, validation, and user feedback. Data modeling and transformation into the Neo4j schema as well as the other features such as data exploration and downloading are covered in later sections.

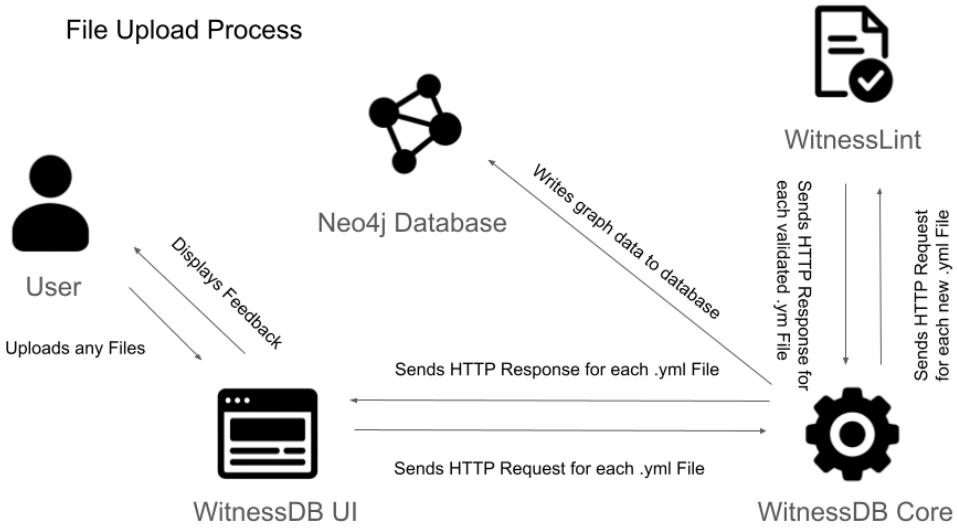


Figure 3.2: File Upload Process

3.3.1 User Interaction regarding File Upload

The file upload interface is available under the `/upload` route. Users begin by selecting the files they wish to upload, either via a traditional file dialog or through drag-and-drop functionality. Both `.yml` files and `.zip` archives containing YAML files are supported.

Before proceeding, the user must choose a data modeling strategy: either a *graph-oriented* schema, where each YAML file is represented as an isolated subgraph in the database, or an *interconnected* schema, which emphasizes structural overlap and shared entities across files. This choosen data schema applies to all uploaded files in the current session and influences how they are mapped into the database. The details around each model are discussed in later sections.

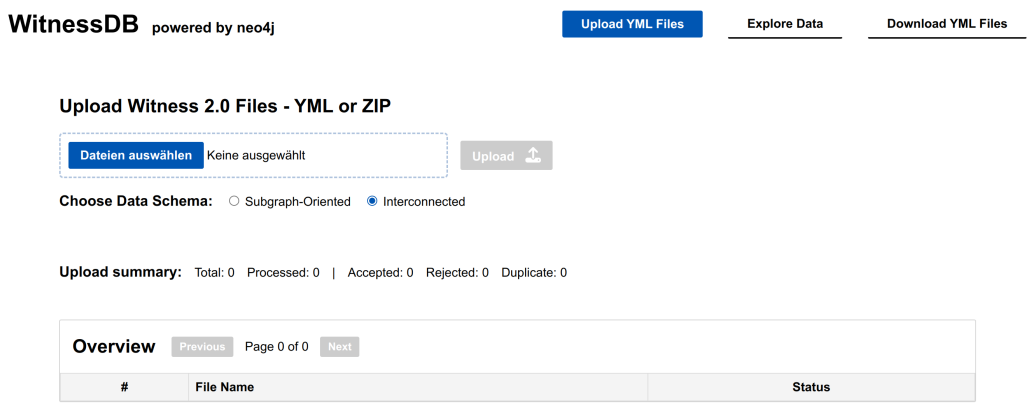


Figure 3.3: File Upload - Initial Setup

Once the files are selected and the data schema is chosen, the user initiates the upload by clicking a the *Upload* button. At this point, the WitnessDB UI component preprocesses the files on the client side, ensuring that only valid .yml files are extracted and sent to the WitnessDB Core component.

During the upload, a progress bar displays the overall completion percentage. A table is additionally rendered to show the upload status of individual files. Each entry initially includes a spinner that indicates ongoing processing. Once a response is received from the backend confirming that a file has been validated and stored, the corresponding spinner is replaced with a status indicator.

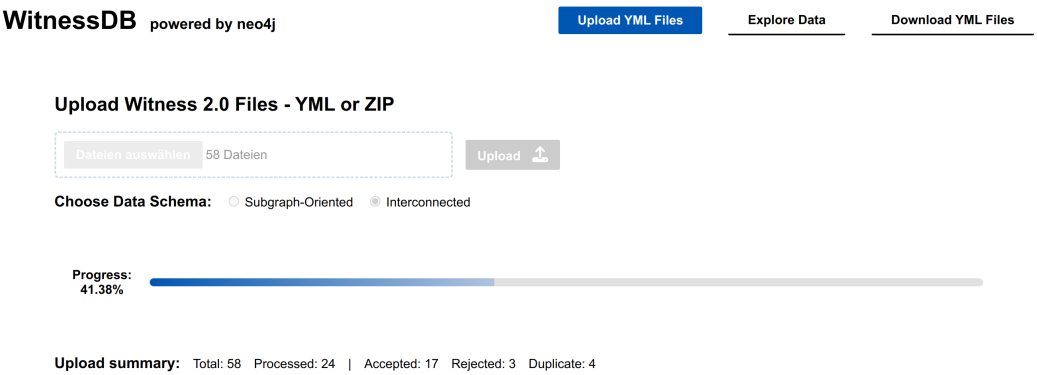
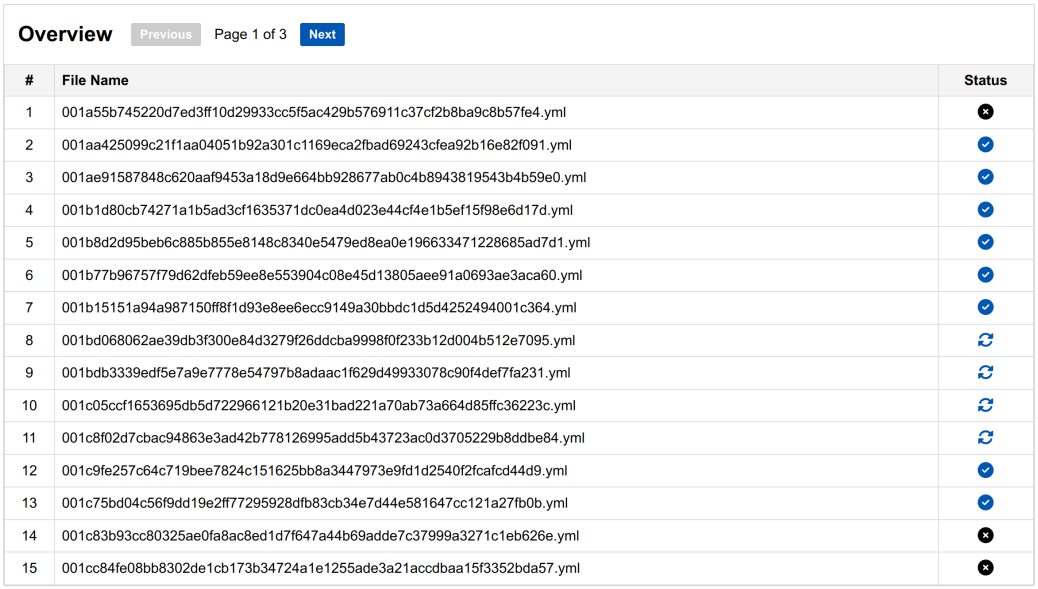


Figure 3.4: File Upload - Upload Area During Uploading



Overview Previous Page 1 of 3 Next

#	File Name	Status
1	001a55b745220d7ed3ff10d29933cc5f5ac429b576911c37cf2b8ba9c8b57fe4.yml	✗
2	001aa425099c21f1aa04051b92a301c1169eca2fbad69243cfea92b16e82f091.yml	✓
3	001ae91587848c620aaf9453a18d9e664bb928677ab0c4b8943819543b4b59e0.yml	✓
4	001b1d80cb74271a1b5ad3cf1635371dc0ea4d023e44cf4e1b5ef15f98e6d17d.yml	✓
5	001b8d2d95beb6c885b855e8148c8340e5479ed8ea0e1966334712286 85ad7d1.yml	✓
6	001b77b96757f79d62dfeb59ee8e553904c08e45d13805aee91a0693ae3aca60.yml	✓
7	001b15151a94a987150ff8f1d93e8ee6ecc9149a30bbdc1d5d4252494001c364.yml	✓
8	001bd068062ae39db3f300e84d3279f26ddcba9998f0f233b12d004b512e7095.yml	↻
9	001bdb3339edf5e7a9e7778e54797b8adaac1f629d49933078c90f4def7fa231.yml	↻
10	001c05ccf1653695db5d722966121b20e31bad221a70ab73a664d85ffc36223c.yml	↻
11	001c8f02d7cbac94863e3ad42b778126995add5b43723ac0d3705229b8ddbe84.yml	↻
12	001c9fe257c64c719bee7824c151625bb8a3447973e9fd1d2540f2fcafcd44d9.yml	✓
13	001c75bd04c56f9dd19e2ff77295928dfb83cb34e7d44e581647cc121a27fb0b.yml	✓
14	001c83b93cc80325ae0fa8ac8ed1d7f647a44b69adde7c37999a3271c1eb626e.yml	✗
15	001cc84fe08bb8302de1cb173b34724a1e1255ade3a21accdbaa15f3352bda57.yml	✗

Figure 3.5: File Upload - Status Table

The file status table grows progressively as files are processed. To maintain usability with large file sets, a basic pagination mechanism is included. The upload state is managed globally within the single-page application, which means that the upload process can only terminate earlier if the page is reloaded.

Details regarding file validation, error handling, and storage confirmation are covered in later sections.

3.3.2 File Preprocessing and Validation

Before a witness file is stored in the database, the WitnessDB Core component performs extensive validation to ensure the integrity and structure of the verification data. A series of technical and application-level checks are applied to each uploaded file, as summarized in Figure 3.6. This validation occurs before invoking the WitnessLint tool, which performs deep structural validation. Since linting involves invoking an subprocess and can be time-consuming, these early-stage validations help avoid redundant operations and improve overall system efficiency. Error handling during this stage is highly granular, so that the WitnessDB Core component can return informative HTTP responses for each type of failure to the WitnessDB UI component.

Validation Step	Purpose	Error Handling
File presence and non-empty check	Ensures the file has been properly received	Returns 400 Bad Request if file is missing/empty
File extension check (.yml)	Confirms correct file format	Returns 400 Bad Request if extension is invalid
MIME type check	Ensures the file has a recognized YAML content type	Returns 415 Unsupported Media Type
File Name sanitization	Prevents path traversal or injection through file name	Returns 400 Bad Request on invalid patterns
Duplicate file check (by hash)	Prevents re-upload of already stored witness files	Returns 409 Conflict if file already exists

Figure 3.6: File Preprocessing - Error Handling Overview

One key step is to check whether the witness file has already been stored in the database. As described in the background section, each witness file is uniquely identified by a filename that encodes a hash of its content. This approach allows the system to prevent redundant uploads by using a deterministic identifier. The WitnessDB Core component enforces this uniqueness constraint by querying the database for metadata nodes, which specifically contain information about the witness file (rather than the verification witnesses included within the file). If a file with the respective hash already exists in the database, the system avoids processing it again. This metadata, generated by the WitnessLint tool after validation, is stored in a dedicated node connected to the main entry node of each graph structure representing a verification witness. These nodes are referred to as *WitnessFile* or *WitnessFile_*. More information about the data model and storage will be discussed in later sections.

3.3.3 Data Transformation Pipeline

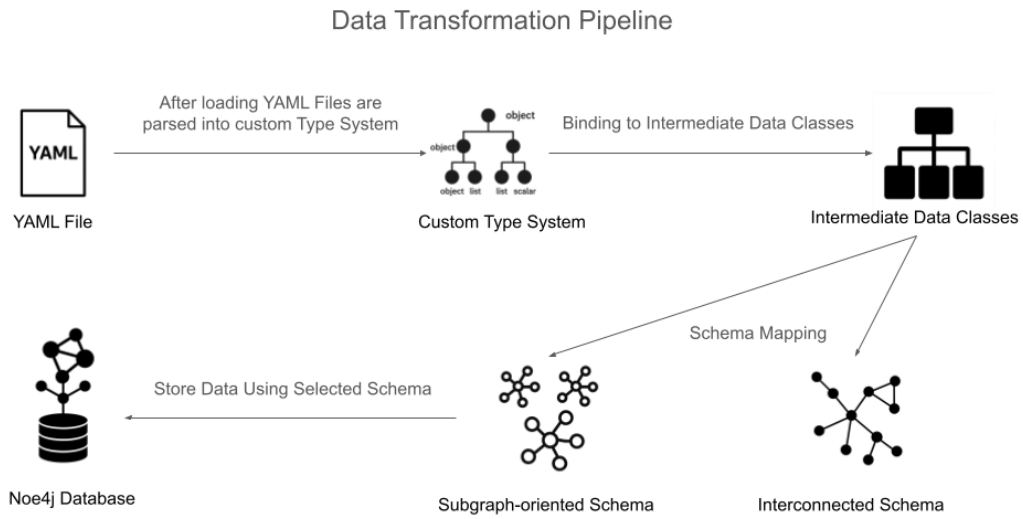


Figure 3.7: Data Transformation Pipeline Overview

3.3.3.1 Converting YAML to Kotlin Objects

To convert the uploaded YAML files into Kotlin objects, the program avoids relying directly on the general-purpose Jackson data structures. Instead, it employs a simplified, purpose-specific typing system and exceptions designed to improve readability and reduce deserialization complexity. This custom structure is designed specifically for Witness Format 2.0, which uses only a subset of YAML primitives and structures.

Rather than modeling the entire YAML structure with low-level tree representations, the deserialization process uses abstracted representations for values and containers. This makes it easier to convert YAML content into intermediate Kotlin data classes that precisely mirror the expected structure of a valid witness file. These intermediate data classes serve as a bridge and are later mapped to the actual domain entities used to direct the persisting process.

3.3.3.2 Mapping to Domain Entities

As previously mentioned, two distinct data schemas are supported in this project: the *subgraph-oriented* schema and the *interconnected* schema (for details, see Section **??**). These schemas dictate how the verification witnesses are logically structured in the database, and they influence the transformation process from intermediate parsed data into domain entities.
For the *subgraph-oriented* schema, the structure of the witness file is very close to the target data model. The intermediate data classes generated during the

deserialization step can be directly adapted into domain entities with only minor adjustments. This makes the mapping process relatively straightforward.

In contrast, the *interconnected* schema represents a more complex and normalized representation of the verification data. Mapping to this schema requires the explicit construction of domain objects that establish relationships between nodes. To manage the complexity, the core component includes dedicated services, inspired by the builder design pattern that encapsulate the logic for assembling these entities.

3.3.4 Data Storage in Neo4j

3.3.4.1 Integration with Neo4j

The Kotlin data classes representing the domain entities are annotated using `@Node`, `@Relationship`, and other annotations provided by Spring Data for Neo4j. These annotations define how the objects should be persisted in the Neo4j graph database. Spring Data automatically translates the annotated Kotlin classes into Cypher queries at runtime, abstracting away much of the boilerplate code that would otherwise be required for graph persistence. For example:

- `@Node` marks a class as a graph node.
- `@Id` and `@GeneratedValue` handle identity and automatic ID generation.
- `@Relationship` defines the type and direction of edges between nodes.

Spring Data for Neo4j handles the lifecycle of these objects, including creation, update, and deletion, by leveraging repositories defined through simple Kotlin interfaces.

3.3.4.2 Identifiers in Neo4j

In Neo4j, each node and relationship is automatically assigned an internal identifier by the database engine. While these IDs can be retrieved using the `Id()` function in queries, this approach is deprecated and not recommended for production use since these internal IDs are not stable across database imports, migrations, or backups. Newer versions of Neo4j introduce a 32-character UUID-like identifier accessible via the `elementId()` function. However, the Neo4j developers explicitly discourage relying on `elementId()` for application-level logic, as it is primarily intended for internal use and system-level tooling. Rather than relying on internal IDs, Neo4j recommends that developers define their own unique identifiers as node properties. These user-defined identifiers act as stable, meaningful equivalents for traditional primary keys. While Neo4j supports uniqueness constraints on single properties, it does not support composite keys consisting of multiple fields. For more information see the Cypher Manual [18]

In WitnessDB a domain-specific identifier is assigned to each node to ensure consistent referencing across the graph, regardless of the node's function within the data model. These identifiers are defined using the `@Id` annotation from Spring Data for Neo4j. Depending on the entity type, the identifier may be derived from meaningful domain attributes or generated programmatically when no natural key exists. Relationship entities, on the other hand, rely on Neo4j's internally generated IDs for identification. The Spring Data framework enforces the presence of identifiers through the `@Id` annotation and raises a compilation error if an entity lacks one. However, it does not automatically create database-level constraints such as uniqueness. To ensure data integrity, all properties for which uniqueness is essential to the correct functioning of the system have been manually annotated with uniqueness constraints using Cypher queries. Figure 3.8 shows some examples for applied uniqueness constraints.

```
SHOW CONSTRAINTS
```

Overview Previous Page 2 of 3 Next

#	name	type	entityType	labelsOrTypes	properties	ownedIndex
16	constraint_Metadata__identifier	UNIQUENESS	NODE	["Metadata_"]	["identifier"]	constraint_Metadata__identifier
17	constraint_Producer__identifier	UNIQUENESS	NODE	["Producer_"]	["identifier"]	constraint_Producer__identifier
18	constraint_Producer_identifier	UNIQUENESS	NODE	["Producer"]	["identifier"]	constraint_Producer_identifier
19	constraint_Segment__identifier	UNIQUENESS	NODE	["Segment_"]	["identifier"]	constraint_Segment__identifier
20	constraint_Segment_identifier	UNIQUENESS	NODE	["Segment"]	["identifier"]	constraint_Segment_identifier
21	constraint_Specification_identifier	UNIQUENESS	NODE	["Specification"]	["identifier"]	constraint_Specification_identifier
22	constraint_Task__identifier	UNIQUENESS	NODE	["Task_"]	["identifier"]	constraint_Task__identifier
23	constraint_Task_identifier	UNIQUENESS	NODE	["Task"]	["identifier"]	constraint_Task_identifier

Figure 3.8: Examples of Uniqueness Constraints

3.3.4.3 Indexing Data

Neo4j supports several types of indexes [18]:

- **Lookup Indexes.** Neo4j automatically creates lookup indexes to speed up label and relationship-type existence checks. These system indexes are used internally by the database to quickly find which labels exist or which nodes have a specific label or relationship type. They are not manually defined by the user.

- **Property Indexes.** These are the most commonly used indexes in Neo4j and allow for fast retrieval of nodes or relationships based on the value of a single property.

- **Composite Indexes.** These indexes cover multiple properties at once.

- **Full-Text Indexes.** These indexes support efficient text-based searches across one or more string properties.

- **Point Indexes.** These are specialized indexes that optimize queries involving spatial data, such as geographic coordinates.

In Neo4j, creating constraints automatically generates the corresponding indexes to support efficient lookups. Since Spring Data Neo4j does not generate these constraints or indexes by default, a custom initialization procedure is included that runs each time the WitnessDB Core component starts. This procedure checks for the existence of the required uniqueness constraints and creates them if they are missing. In addition to these automatically generated indexes, the initialization procedure also creates additional indexes on selected properties and composite property combinations to further optimize query performance. As a result, the system supports both constraint-driven indexes and manually defined property and composite indexes. Figure 3.9 shows some examples for composite indexes.

name	state	populationPercent	type	entityType	labelsOrTypes	properties	indexProvider
composite_index_location	ONLINE	100	RANGE	NODE	["Location"]	["file_name", "line"]	range-1.0
composite_index_producer	ONLINE	100	RANGE	NODE	["Producer"]	["name", "version"]	range-1.0

Figure 3.9: Examples of Composite Indexes

3.3.5 User Feedback

After a witness file is fully processed, the user receives feedback through a status icon in the UI. Currently, the system supports three feedback states: *success*, *failure*, and *duplicate* (if the file has already been uploaded). In addition to individual status indicators, the UI also provides an upload summary displaying the total number of uploaded files, the number of files processed, and a breakdown of processing outcomes by status. An example of this behavior can be found in Figure 3.4 and 3.5

If the file fails validation during pre-storage checks or if an error occurs during the write operation such as a violation of uniqueness constraints the WitnessDB Core component returns a detailed error message. While the WitnessDB UI component only displays a generic failure icon to the user, the full error response is available in the browser's developer console.

3.4 Data Modeling in Neo4j

3.4.1 File Structure-Based Model

This data model was developed to study how the contents of YAML files can be stored in a graph database with minimal transformation and maximum speed. As previously discussed in section 2.1.2 and shown in Listing 1, 2 and 3, Witness 2.0 Fromat supports several data structures, including sequences and mappings.

Sequences in YAML can be easily translated into lists or arrays. Mappings can typically be converted into objects when their structure is predictable. Neo4j, being a property graph database, maps such objects efficiently to nodes. However, nodes in Neo4j are limited in their ability to represent complex, deeply nested data structures.

In Neo4j, a node can only store primitive types, such as strings, numbers or even dates as properties. It can also store arrays of these primitive types. Nested or iterable maps are not supported directly as node properties. This limitation requires either flattening the data or representing complex mappings using additional nodes and relationships.

The transformation of Kotlin data classes into graph members is handled by the Spring Data Neo4j framework. It allows for automatic mapping between data classes and graph nodes or relationships, using only annotations and without the need for custom Cypher queries as previously explained in section 3.2.2. As a result, all elements of the YAML file are explicitly modeled in a way that is compatible with this framework.

3.4.1.1 Entry Point of the Witness 2.0 File Graph

This data model generates an independent subgraph for each YAML file, which is why it is referred to as a *subgraph-oriented* model. Each subgraph begins with an entry node, that stores all information produced by WitnessLint after validation during the upload process. This node is labeled `WitnessFile_`.

```
1  Witness Version: 2.0
2  Overview of checked witness:
3  Witness File: ./examples/safe-program-example.witness.yml
4  Witness Type: CORRECTNESS
5  Number of entries: 1
6  Witness Lines of Code: 33
7  Witness Size in Bytes: 1008
8  Amount of Invariants: 1
9  Amount of Segments: None
10 Amount of Waypoints: None
11 Specification: G ! call(reach_error())
12 Witness Hash:
       a9f98257a1daca6e78a59f68137ae51c639fc205341cb9f1d8c243ffe2d483f2
```

Listing 4: Result from WitnessLint validation of a witness file

A Witness 2.0 YAML file contains a sequence of entries, each representing an independent verification witness. Each of these entries is represented by a `Witness_` node, which stores only the type of the verification witness as a property. These `Witness_` nodes act as intermediate nodes that link to additional subgraphs representing the metadata and the content of the verification witness. The `WitnessFile_` node connects to one or more `Witness_` nodes, depending on how many verification witnesses are defined in the original YAML file.

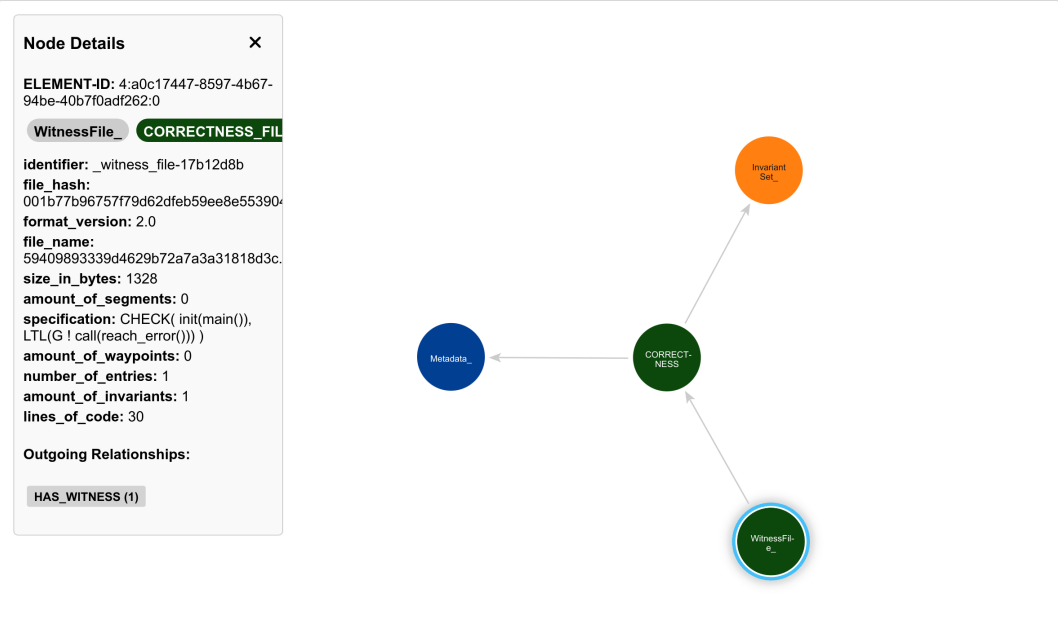


Figure 3.10: Example of `WitnessFile_` Node

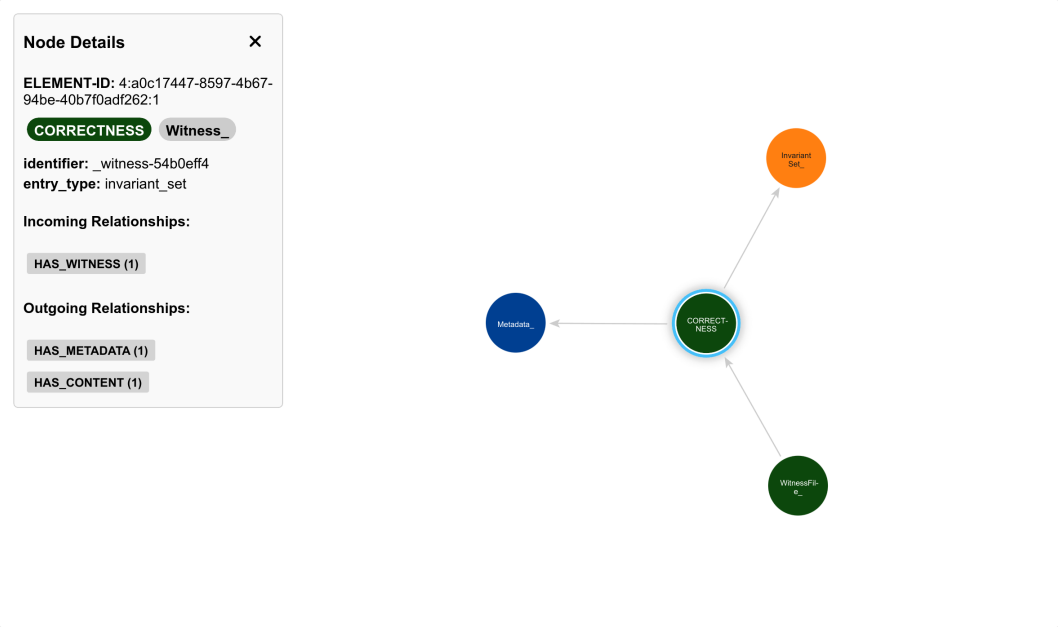


Figure 3.11: Example of `Witness_` Node

3.4.1.2 Metadata Subgraph

The metadata subgraph begins with the `Metadata_` that directly stores primitive properties extracted from the YAML file, such as `format_version`, `uuid`, `creation_time`.

More complex mappings within the metadata, such as producer and task, are represented using special nodes: `Producer_` and `Task_`. These nodes preserve the structure of their corresponding YAML mappings by storing each key-value pair as a property.

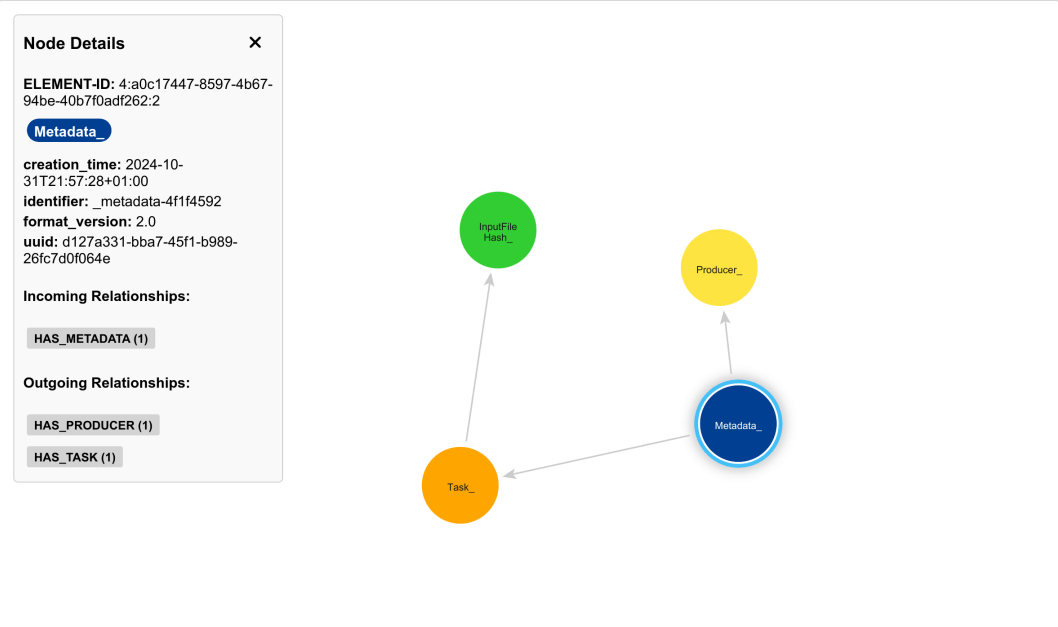


Figure 3.12: Example of `Metadata_` related Subgraph

The producer mapping contains several optional attributes. Neo4j natively handles nullable values, which means that optional attributes simply do not appear as properties in the `Producer_` node if they are absent from the YAML file. When queried, such non-existent properties return null, but Neo4j does not store any placeholder or explicit null value in the database.

The `Task_` node stores all primitive attributes of the task mapping directly as properties. The `input_files` is represented as an array property within the `Task_`-node, containing the stringified paths of the C program files being verified. Each input file path in the `input_files` sequence is associated with a hash value, as defined in a separate mapping in the YAML file. Because this mapping uses file paths as dynamic keys and is not easily transformed into a predictable object structure, each input file is represented by a separate node. These nodes store the file path and corresponding hash value as properties, and are connected to the `Task_` node. This structure allows the Spring Data Neo4j framework to automatically generate the full

metadata subgraph from Kotlin data classes. By using only annotations, without requiring any custom Cypher queries, the framework can serialize and persist the complete structure automatically.

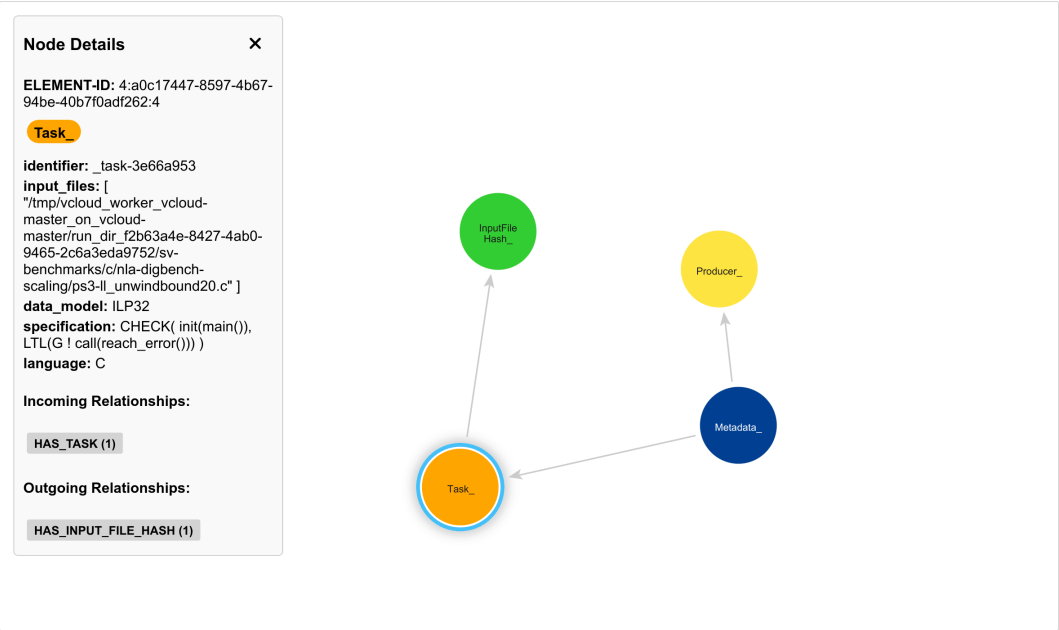


Figure 3.13: Example of `Task_` Node

3.4.1.3 Content Subgraph

This is the part of the graph where the actual output of verification tools, described in the background section, is stored. The goal was to create a unified representation for both correctness and violation witnesses, as both are represented in the YAML file as sequences: either of invariants or of segments containing error paths.

To make this possible, a type hierarchy with generic elements was designed. The entry point of the content subgraph is a node labeled `Content_`, which serves as a common base. Depending on the type of verification witness, this node is further specialized and labeled either `InvariantSet_` or `ViolationSequence_`. These nodes do not store any properties, only an identifier. Their primary role is to act as intermediate connectors between the actual content subgraph and the associated metadata nodes.

While Spring Data Neo4j supports class hierarchies, its support for generic types is limited. As a result, this type hierarchy is only partially compatible with the framework. This is not a problem during the persistence phase, when YAML files are uploaded and converted to graph structures. However, during the reconstruction process some manual intervention is required. The content subgraph can be automatically fetched and instantiated from the database, but it must be manually reconnected to the rest of the file's subgraph.

Both *Invariant Sets* and *Violation Sequences* contain information about important locations in the program's source code. This location data is captured using a unified mapping and represented in the graph by nodes labeled `Location_`. Each `Location_` node is connected either to an `Invariant_` node, in the case of a correctness witness, or to a `Waypoint_` node, which represents a point within an error-containing segment in the program. `Waypoint_` nodes are themselves linked to their corresponding `Segment_` node, forming part of a complete violation trace. The `ViolationSequence_` node serves as the entry point and is connected to one or more `Segment_` nodes, preserving the structure found in the original YAML file.

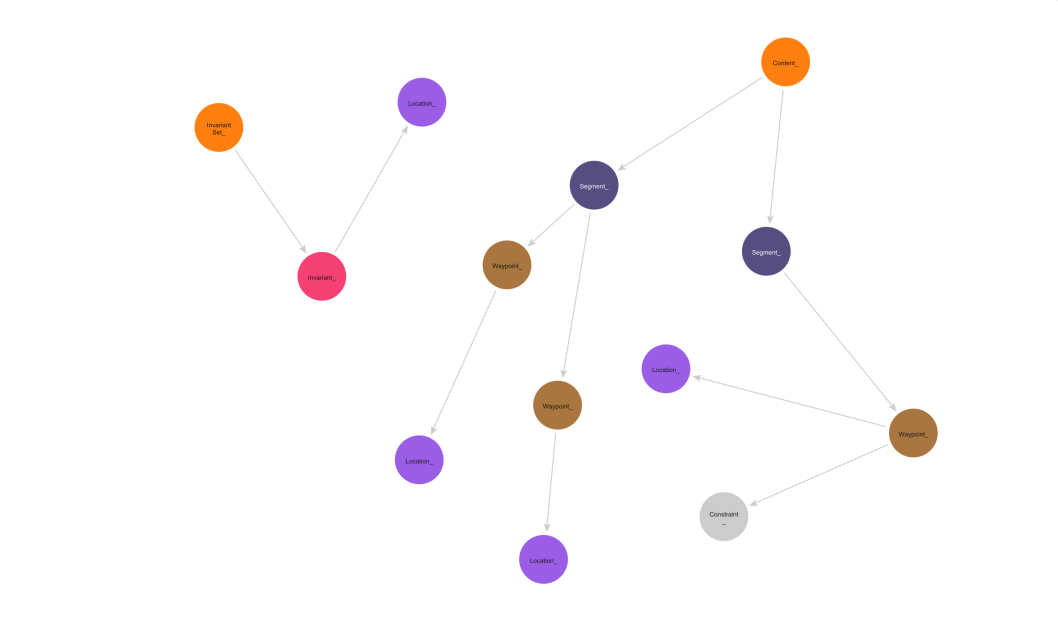


Figure 3.14: Example of Content Subgraphs for Correctness and Violation

As with the metadata subgraph, nodes in the content subgraph may include numerous optional properties. These are handled in the same way: if a property is absent from the YAML input, it is simply omitted from the node. Neo4j treats these absent fields as null when queried, without requiring any additional representation in the database.

3.4.2 Domain-Based Model

While the file structure-based model focuses on preserving the internal organization of individual YAML files, the domain-based model captures the broader semantic relationships that emerge across multiple files, tools, and verification tasks. This model reflects the interconnected nature of verification data, forming a global graph that links all verification artifacts by their roles, sources, and outcomes.

3.4.2.1 Metadata related Nodes

The interconnected data model has some structural similarities to the file-based model. In this model a metadata node called `WitnessFile` also is used to represent each stored YAML file. For every verification witness contained in such a file, there is a corresponding node of type `Witness`. However, unlike the isolated structure of the file-based model, these nodes are integrated into a larger graph of interconnected entities that collectively represent metadata related to the production of verification witnesses. The primary goal of this model is to capture the broader context in which a verification witness is generated. Most of the nodes that hold contextual metadata are globally defined. This allows each new `Witness` node to be mounted into an existing shared context. Since this model does not define a special `Metadata` node, the `Witness` node itself is directly connected to relevant metadata nodes such as Task or Producer. Attributes such as `creation_time`, `uuid` and `entry_type` are stored directly as properties of the `Witness` node. This design reduces the number of intermediate nodes.

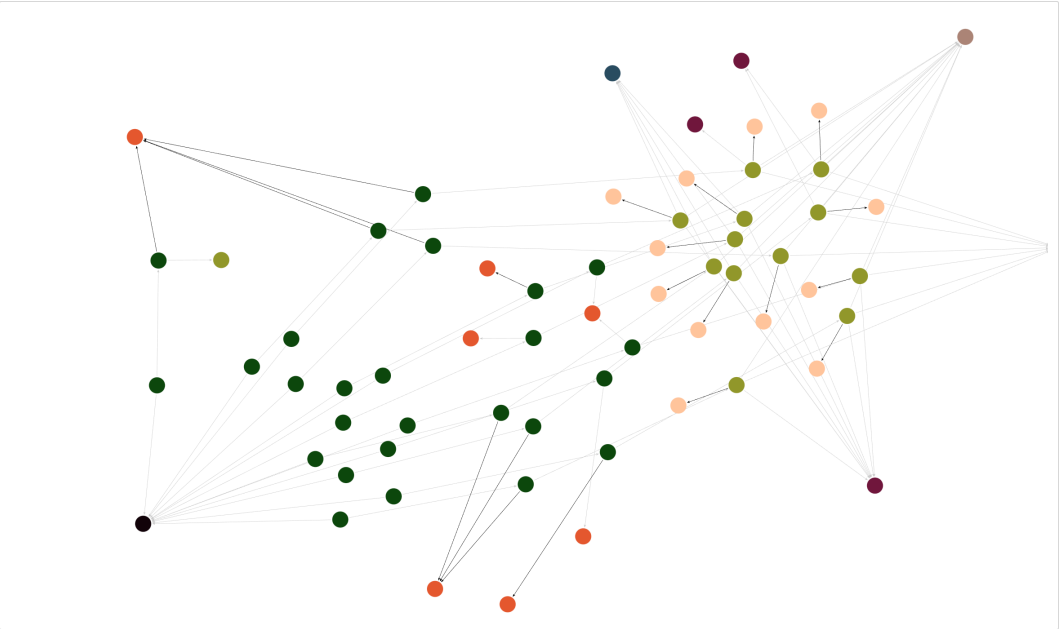
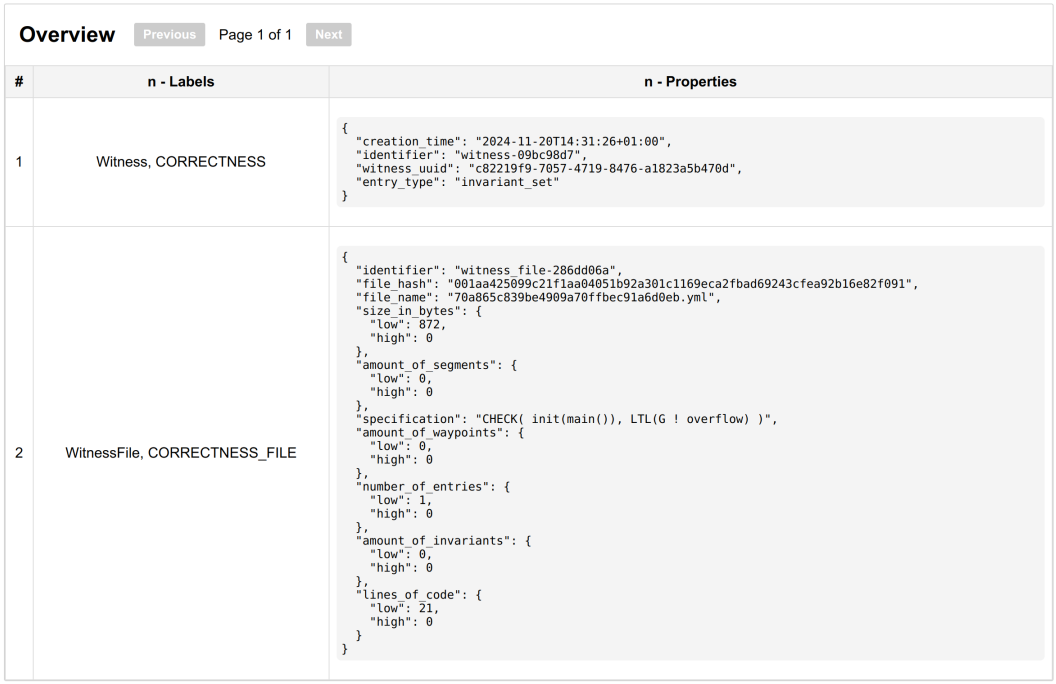


Figure 3.15: Example of Metadata related Subgraph



Overview Previous Page 1 of 1 Next

#	n - Labels	n - Properties
1	Witness, CORRECTNESS	`{ "creation_time": "2024-11-20T14:31:26+01:00", "identifier": "witness-09bc98d7", "witness_uuid": "c82219f9-7057-4719-8476-a1823a5b470d", "entry_type": "invariant_set" }`
2	WitnessFile, CORRECTNESS_FILE	`{ "identifier": "witness_file-286dd06a", "file_hash": "001aa425099c21f1aa04051b92a301c1169eca2fbad69243cfea92b16e82f091", "file_name": "70a865c839be4909a70ffbec91a6d0eb.yml", "size_in_bytes": { "low": 872, "high": 0 }, "amount_of_segments": { "low": 0, "high": 0 }, "specification": "CHECK(init(main()), LTL(G ! overflow))", "amount_of_waypoints": { "low": 0, "high": 0 }, "number_of_entries": { "low": 1, "high": 0 }, "amount_of_invariants": { "low": 0, "high": 0 }, "lines_of_code": { "low": 21, "high": 0 } }`

Figure 3.16: Example of `WitnessFile` and `Witness` Nodes

The following node types serve as normalized enumerations within the data model and are each uniquely identified by a single scalar value:

- `DataModel`. It encodes abstract machine models such as LP64 and ILP32
- `Language`. Currently there is only one node of this type representing the C programming language, as all verification artifacts are C-based
- `Version`. Represents the semantic version of the witness format specification, currently limited to a single instance (2.0)

Overview Previous Page 1 of 1 Next

#	n - Labels	n - Properties
1	DataModel	`{ "identifier": "LP64" }`
2	Language	`{ "identifier": "C" }`
3	Version	`{ "identifier": "2.0", "file_format": "yml" }`
4	DataModel	`{ "identifier": "ILP32" }`

Figure 3.17: Overview of Metadata Nodes Content

The `Specification` nodes are identified by the formal property that defines the verification objective. This modeling approach effectively builds a repository of specifications within the graph.

Overview Previous Page 1 of 1 Next

#	n - Labels	n - Properties
1	Specification	{ "identifier": "CHECK(init(main()), LTL(G ! overflow))" }
2	Specification	{ "identifier": "CHECK(init(main()), LTL(G ! call(reach_error())))" }
3	Specification	{ "identifier": "CHECK(init(main()), LTL(G valid-free))\nCHECK(init(main()), LTL(G valid-deref))\nCHECK(init(main()), LTL(G valid-memtrack))" }
4	Specification	{ "identifier": "CHECK(init(main()), LTL(F end))" }
5	Specification	{ "identifier": "CHECK(init(main()), LTL(G ! data-race))" }
6	Specification	{ "identifier": "" }
7	Specification	{ "identifier": "CHECK(init(main()), LTL(G valid-memcleanup))" }

Figure 3.18: Overview of Specification Nodes

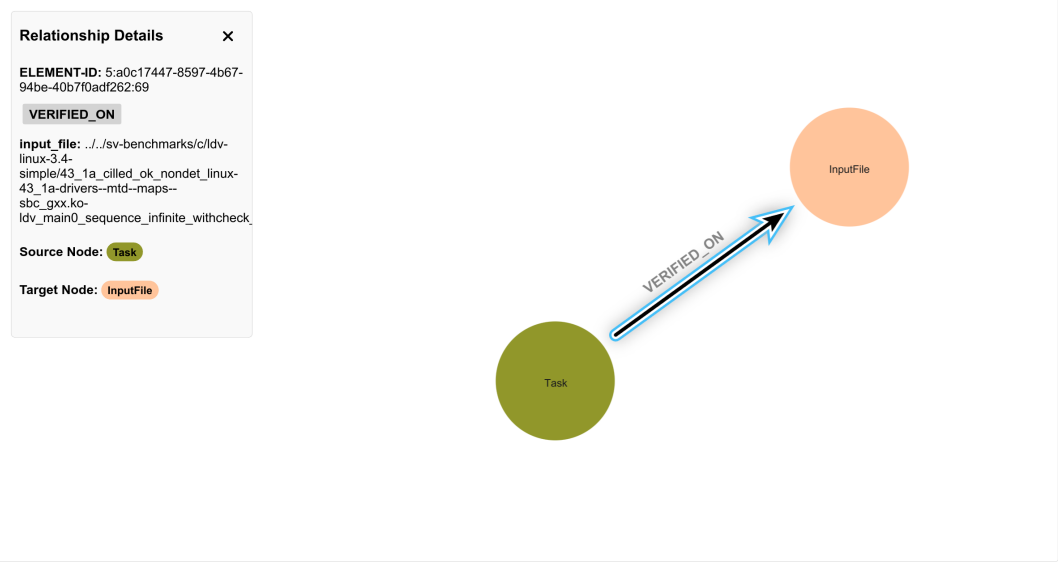


Figure 3.19: Example of `VERIFIED_ON` Relationship

The `InputFile` node represents the program being verified and is uniquely identified by the hash value of the program file's content. In the YAML witness file, a program is specified by both its storage path and its content hash. However, the

same program can appear under different file paths in different verification contexts. To prevent redundant nodes representing the same underlying program, the file path is not stored as a property on the `InputFile` node itself. Instead, the path is stored as a property of the relationship connecting the `InputFile` node to a `Task` node.

The `Producer` node type represents the verification tool responsible for generating a witness. Each `Producer` node is uniquely identified by the combination of its name and version properties. This design allows multiple versions of the same verification tool to be explicitly distinguished within the graph. Optional metadata associated with a specific invocation of the tool such as command-line parameters or configuration settings are not stored as properties on the `Producer` node itself but are instead modeled as properties of the relationship connecting a `Producer` node to a Witness node. This separation ensures that tool-level attributes are normalized globally, while run-specific metadata remains contextualized at the level of individual verification executions.

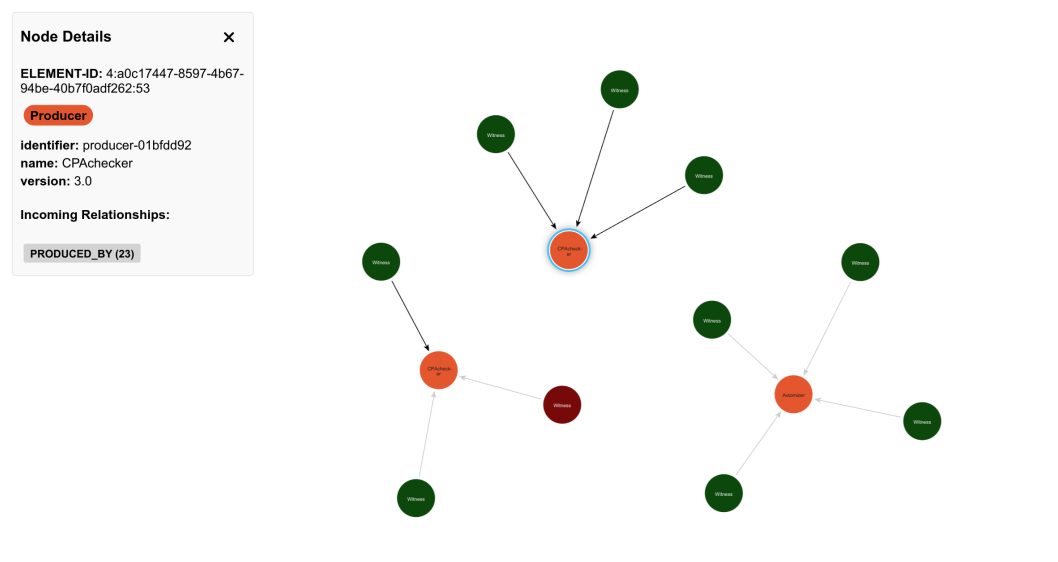


Figure 3.20: Example of `Producer` related Subgraphs

The `Task` node acts as an intermediate node that connects the `Witness` node to related metadata nodes: `Specification`, `DataModel`, `InputFile`, and `Language`.

3.4.2.2 Content related Nodes

Like the file-based data model, this model also includes a node called `Content`, which serves as an intermediate node and the entry point to the actual verification witness. The `Content` node always carries a second label: either `InvariantSet` or `ViolationSequence`, depending on the type of witness. Similar to the file-based

model, this node points to nodes of type `Segment` or `Invariant`, reflecting the structure of the verification content.

Similar to the metadata context, this model also defines several globally shared nodes with an enumerated character. These nodes were introduced to explicitly model the commonalities across verification artifacts. The `ExpressionFormat` node represents the format of invariants and constraints. Both invariants and waypoints in error path segments are limited to a predefined set of types. These enumerated values are represented by the nodes `InvariantType` and `WaypointType`.

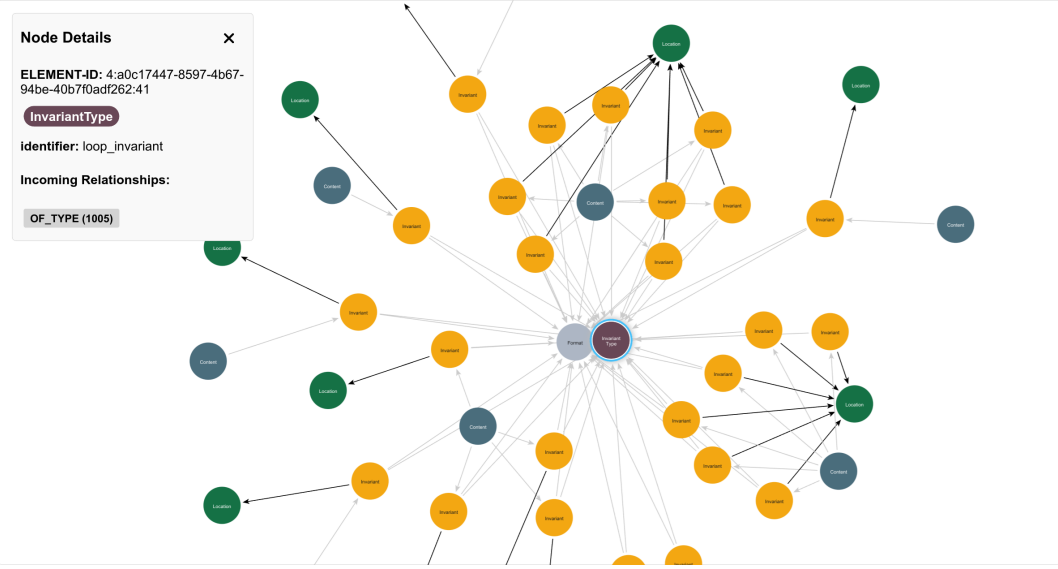


Figure 3.21: Example of `InvariantSet` related Subgraphs

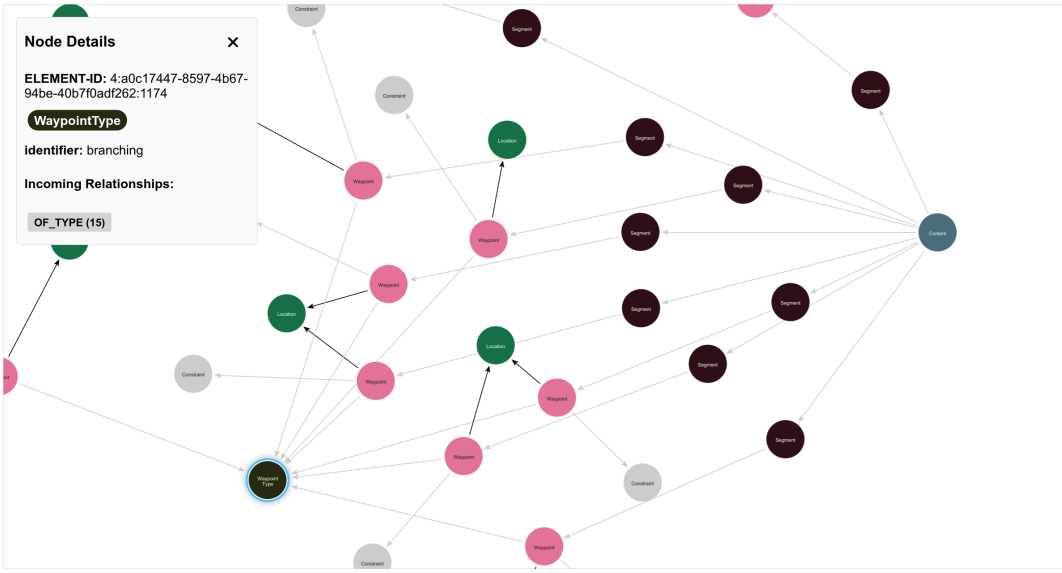
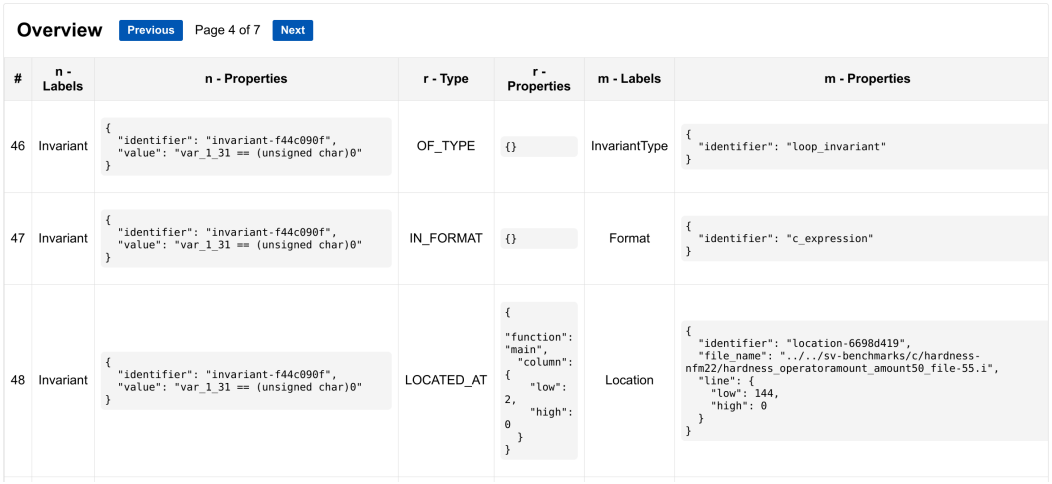


Figure 3.22: Example of `ViolationSequence` related Subgraphs

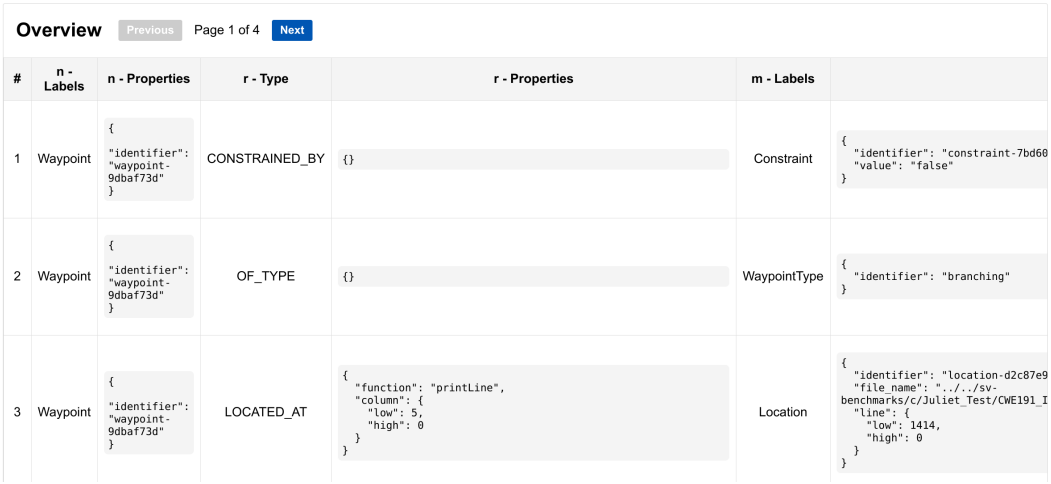
Both invariants and waypoints in error path segments are associated with a location in the program. Locations are represented by `Location` nodes, which are uniquely identified by the program file path and the line number within the file. There also can be optional attributes related to a location in the YAML file such as `column` and `function`. Since these attributes are optional and not required for identifying a unique location, they are not stored as properties of the `Location` node. Instead, they are modeled as properties of the relationship connecting an `Invariant` or `Waypoint` node to its corresponding `Location` node.



Overview Previous Page 4 of 7 Next

#	n - Labels	n - Properties	r - Type	r - Properties	m - Labels	m - Properties
46	Invariant	{ "identifier": "invariant-f44c090f", "value": "var_1_31 == (unsigned char)0" }	OF_TYPE	{}	InvariantType	{ "identifier": "loop_invariant" }
47	Invariant	{ "identifier": "invariant-f44c090f", "value": "var_1_31 == (unsigned char)0" }	IN_FORMAT	{}	Format	{ "identifier": "c_expression" }
48	Invariant	{ "identifier": "invariant-f44c090f", "value": "var_1_31 == (unsigned char)0" }	LOCATED_AT	{ "function": "main", "column": { "low": 2, "high": 0 } }	Location	{ "identifier": "location-6698d419", "file_name": "../../sv-benchmarks/c/hardness-nfm22/hardness_operatoramount_amount50_file-55.i", "line": { "low": 144, "high": 0 } }

Figure 3.23: Example of `Invariant` related Nodes and Relationships



Overview Previous Page 1 of 4 Next

#	n - Labels	n - Properties	r - Type	r - Properties	m - Labels	
1	Waypoint	{ "identifier": "waypoint-9dbaf73d" }	CONSTRAINED_BY	{}	Constraint	{ "identifier": "constraint-7bd60 "value": "false" }
2	Waypoint	{ "identifier": "waypoint-9dbaf73d" }	OF_TYPE	{}	WaypointType	{ "identifier": "branching" }
3	Waypoint	{ "identifier": "waypoint-9dbaf73d" }	LOCATED_AT	{ "function": "printLine", "column": { "low": 5, "high": 0 } }	Location	{ "identifier": "location-d2c87e9 "file_name": "../../sv-benchmarks/c/Juliet_Test/CWE191_I "line": { "low": 1414, "high": 0 } }

Figure 3.24: Example of `Waypoint` related Nodes and Relationships

There is one more detail in the modeling of `Segment` nodes within violation witnesses, that is worth mentioning. Segments are composed of `Waypoint` nodes that represent program locations which should either be followed or avoided during execution. The Witness Format 2.0 specifies an attribute in the waypoint mapping that indicates the role of each waypoint. To represent this in the graph, the type of

waypoint is modeled not as a property on the node, but as the type of relationship connecting a `Segment` to its `Waypoints`. Each `Segment` node must have exactly one outgoing `FOLLOW` relationship to a `Waypoint` node and may have zero, one, or multiple outgoing `AVOID` relationships to other `Waypoint` nodes.

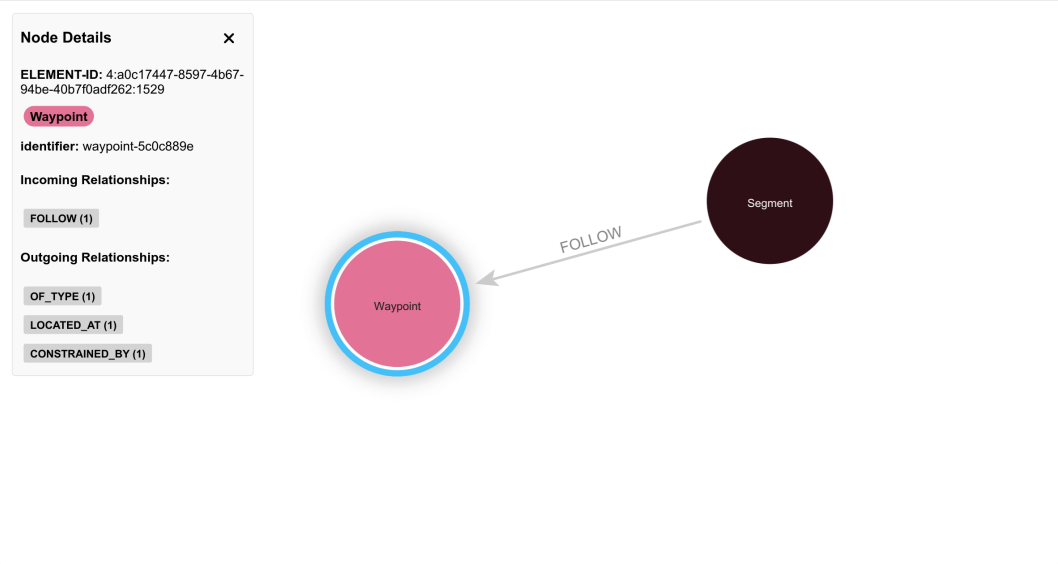


Figure 3.25: Example of `Segment` to `Waypoint` Relationship

3.4.2.3 Implementation of the Persistence Process

The persistence of the data in this model was implemented using the features of Spring Data Neo4j. However, due to the complexity and granularity of the modeling primitives, the persistence process required extensive use of custom queries to iteratively construct the interconnected structure of the model. In each case, nodes were created sequentially, followed by the explicit creation of the relationships linking them. For globally defined nodes each instance was created only once and then queried whenever needed to establish further connections. Both the persistence process and the reconstruction process (retrieving and reassembling witness data from the database) involved several manual steps to correctly resolve relationships and ensure integration of each witness into the broader graph.

3.5 Data Visualization

The WitnessDB UI component also includes an interface for executing custom queries and interactively exploring the data stored in the database. This interface is accessible under the `/explore` route.

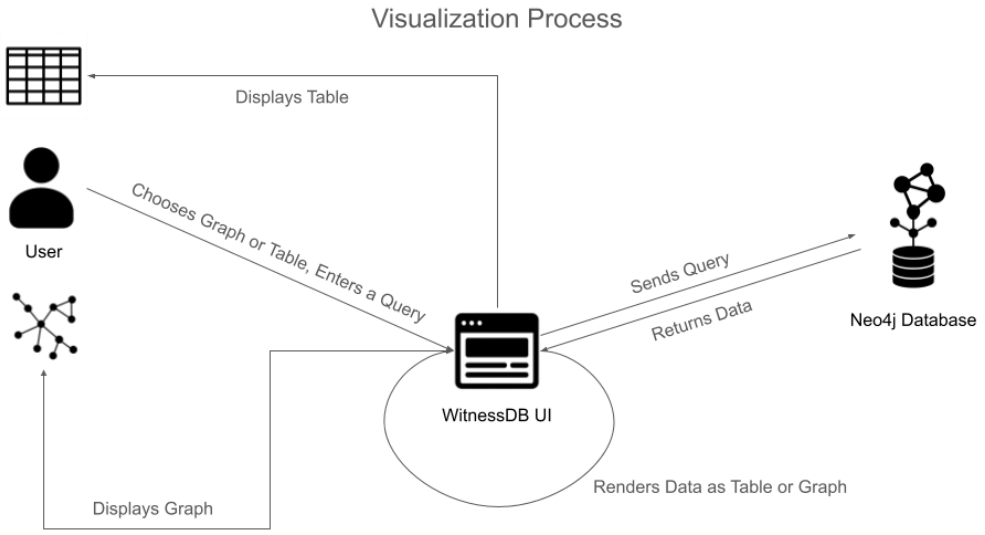


Figure 3.26: Visualization Process Overview

In the intial part of the project several visualization tools for Neo4j graph databases were evaluated, including enterprise solutions such as *Neo4j Bloom* and open-source alternatives like *Pototo.js*. However, the enterprise solutions require a commercial license and the available open-source tools are either insufficiently maintained or require complex integration efforts. To provide a complete and fully integrated system, a minimal visualization interface was implemented directly within the WitnessDB UI component. This interface was developed using the NVL, which is also used in Neo4j's enterprise visualization products. For more information see the 3.2.3 section.

A direct connection between the WitnessDB UI component and the Neo4j database was established to support responsive interaction. The visualization interface integrates features inspired by enterprise solutions as well as custom features.

3.5.1 Shared Components

The Data Explorer is structured around five core components that are shared by both the graph visualization and the tabular view. These include a dropdown section displaying database metadata, a toggle button that switches the view mode between graph and table, an input field for submitting custom queries, a container that renders either the graph or the table depending on the active mode and a download button for exporting the currently displayed data.

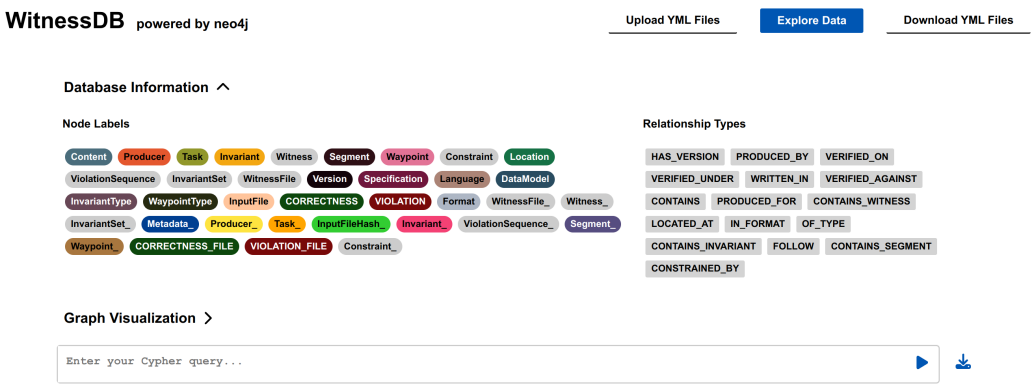


Figure 3.27: Shared Components of the Visualization View

The *Database Information* dropdown provides a dynamically generated set of available node labels and relationship types in the database. Each entry in the list is rendered as a clickable button that automatically populates the query input area with a corresponding query and immediately executes it. This allows users to quickly explore specific node or relationship types without manually writing queries. The contents of the dropdown are refreshed automatically whenever the page is reloaded or the route is changed.

```
1 MATCH (n:Producer) RETURN n LIMIT 50
```

Listing 5: Executed Query when the `Producer` Label is clicked

```
1 MATCH p=()-[r:VERIFIED_ON]->() RETURN p LIMIT 50
```

Listing 6: Executed Query when the `VERIFIED_ON` Relationship Type is clicked

The *Query Input Area* is automatically populated with a default query each time the page is reloaded or the route is changed to ensure that users always start with a valid, executable query. The Data Explorer enforces a read-only queries to maintain the integrity of the underlying data. Any attempt to execute a schema-modifying or write query triggers a validation mechanism that dynamically renders an error message directly under the input area. Any other execution errors, such as syntax errors or runtime query failures, are caught and displayed as error messages in the same location. The input area is designed to dynamically expand in height as the content grows.

```
1 MATCH (n)-[r]-(m) RETURN n, r, m LIMIT 50
```

Listing 7: Default Query

Figure 3.28: Write Operation Error Message

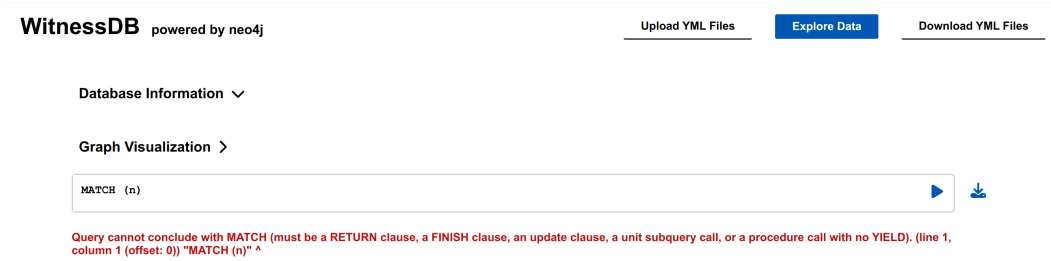


Figure 3.29: Syntax Error Message

Figure 3.30: Long Query

The *Download* button captures the currently displayed data and initiates a download process. In graph mode, it generates a PNG image of the graph container.It captures a visual snapshot of the rendered graph at the moment of download. In table mode, it exports the displayed data as a CSV file. Both export functions operate client-side so no additional server-side processing is required.

The behavior and implementation of the *Content Container* will be discussed in detail in the following subsections.

3.5.2 Graph Visualization

The graph within the content container is rendered using the NVL. To render a graph, the NVL requires a DOM element as the target container and collections of nodes and relationships as input data. For successful visualization, the underlying queries must return complete node and relationship objects. Queries that return only aggregated data or individual scalar attributes are cannot be visualized. In such cases, the system generates an appropriate error message to inform the user that the query result cannot be visualized as a graph.

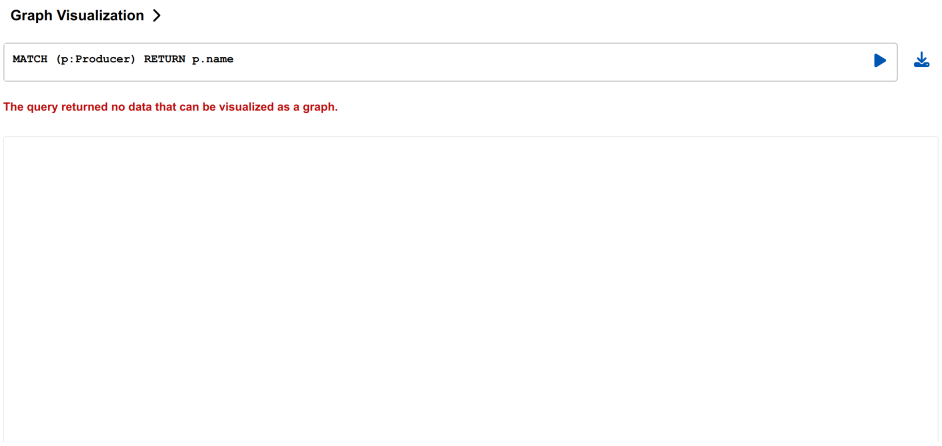


Figure 3.31: Example for Data that cannot be visualized as a Graph

After the data is fetched, the `nvlResultTransformer` reformats the raw query results into a structure compatible with the NVL.

The graph container supports multiple interactive features, including zooming in and out, dragging around nodes and relationships, panning across the canvas and clicking on nodes or relationships to display detailed information.

When a node or relationship is clicked, a details container is displayed on top of the graph container. The details window includes a close button that allows the user to hide the information panel at any time. When the graph container is exported as an image, the details window is captured as well in the exported PNG if visible as seen in Figures 3.25, 3.22, 3.21, 3.20, 3.19

The *Node Details* window displays the internal element ID of the selected node, as well as all associated labels and properties. It also lists its incoming and outgoing relationships. These relationships are grouped by type and presented as interactive buttons, each labeled with the relationship type and the count of incoming or outgoing connections of that type. Clicking one of these relationship buttons triggers a global query that retrieves the corresponding related nodes and populates the graph visualization with both the selected node and its newly fetched neighbors. Information about a node's incoming and outgoing relationships is not preloaded. Instead, this data is fetched dynamically only when a node is clicked.

```
1 MATCH p=()-[r:PRODUCED_BY]->(n) WHERE n.identifier = "producer
    -49df1859" RETURN p
```

Listing 8: Example of executed Query after a button in Node Details is clicked

The *Relationship Details* window also displays the internal element ID of the selected relationship and a list of its properties when any are defined. There are also references to the relationship's source and target nodes, each rendered as a clickable button. Clicking either of these node buttons updates the details window to display

the metadata and properties of the corresponding node. In the graph visualization container relationships that have properties are visually distinguished by being colored in black, while relationships without properties by default gray.
The interactive capabilities of the graph container and details panel, together with the *Database Information* buttons, make it possible for users to navigate and explore the data without the need to manually execute queries. This combination of features enables open-ended, exploratory analysis of the graph, users can discover relationships and insights without being constrained by predefined query paths.

3.5.3 Table Representation

The *Table Representation* is capable of displaying a wider variety of data types than the graph visualization. When queries return complete nodes or relationships, the table organizes their data into structured columns, including their labels or types and all associated properties as it can be seen in Figures 3.24, 3.23. The table can also represent scalars, lists, and paths. By comparison, the *Graph Visualization* can only render complete nodes and relationships, and relationships are visualized only if they are connected to nodes included in the result. This fundamental difference makes the table view more flexible for exploring data that does not conform strictly to a graph topology.

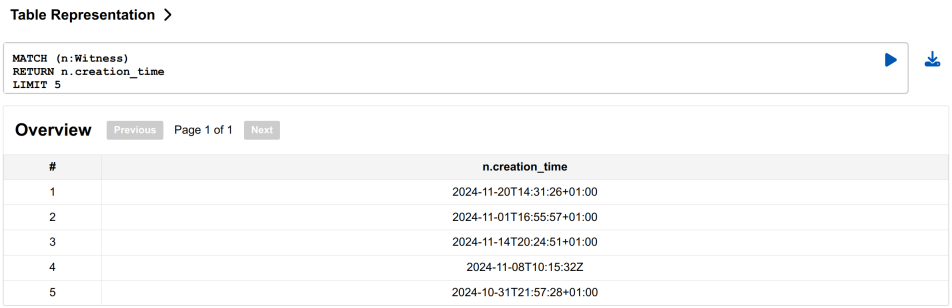


Figure 3.32: Table Representation of a single Node Property

Figure 3.33: Table Representation of Relationships

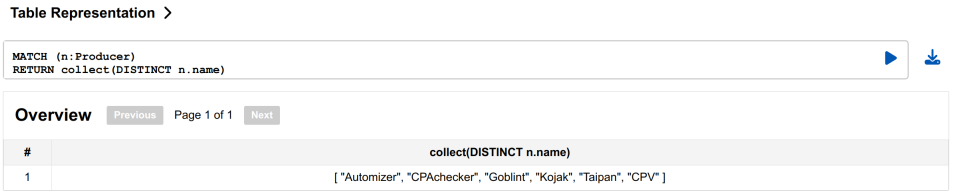


Figure 3.34: Table Representation of a List

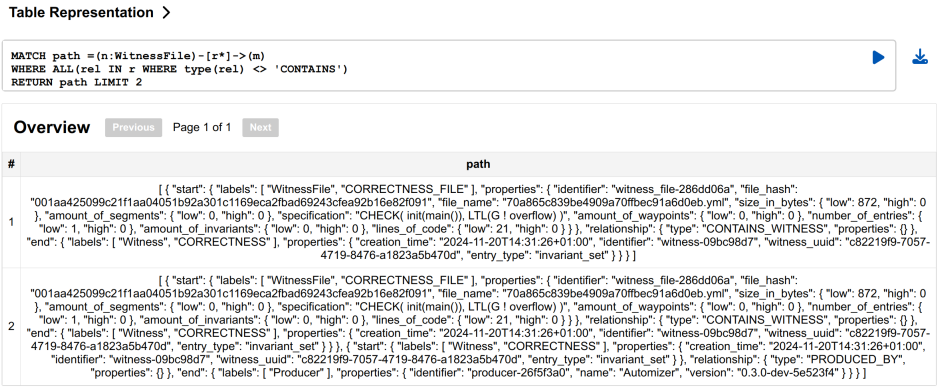


Figure 3.35: Table Representation of Paths

The Table also includes a simple paginator that enables users to navigate through the data to prevent the entire dataset from being rendered at once. Since the number of columns can grow unpredictably depending on the structure of the executed query, the content area within the table is designed to expand horizontally. However, the container itself maintains a fixed size and does not grow with the content, instead users can scroll the content inside the container. When downloading the table, the system exports the entire query result, not just the currently visible portion.

3.6 Reproducing and Downloading YAML Files

The ability to reproduce and download stored witness files is a key feature of WitnessDB. It ensures that verification data can be retrieved in its original form, which ensures consistent reproduction of results and integration with other verification tools. This feature not only allows the user to download a previously uploaded witness exactly as it was received, but also supports generating composed files that combine multiple stored verification witnesses into a single YAML document.

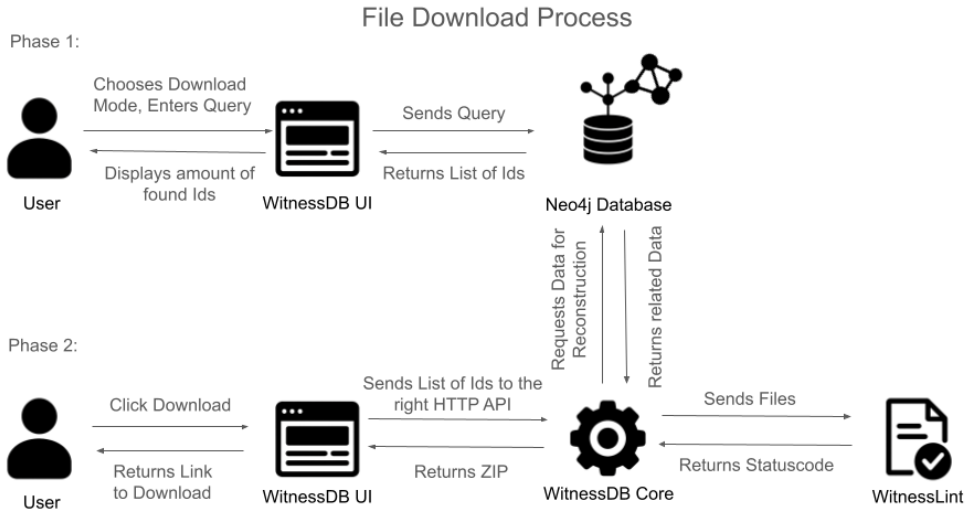


Figure 3.36: Download Process Overview

3.6.1 User Interface for File Downloads

The file download interface is exposed via the `/download` route of the UI component. Users must first choose between two modes: downloading an existing witness file in its original form or composing a new YAML file from selected verification witnesses stored in the database. The interface provides a toggle to switch between these two modes, with the default mode set to downloading previously uploaded files.

To specify which file(s) to download, the user can enter a query in the provided input area. This query is executed directly against the graph database and determines which nodes are selected as entry points for the reconstruction process. Once the query is submitted, the WitnessDB UI component immediately evaluates whether it yields valid nodes. The user receives feedback in the form of either an error message or a notification displaying the number of valid nodes that match the query criteria. After confirming the query results, the user can initiate the download by clicking the *Download* button.

The UI component sends an HTTP request to the core component's download API, transmitting a list of identifiers corresponding to the selected nodes. This triggers the reconstruction process within the WitnessDB Core component. Once the process is complete, the core component returns a ZIP archive containing the reconstructed files, which the user can then download.

3.6.1.1 Downloading modes

If the user wants to reproduce an uploaded witness file, they need to construct a query that returns nodes of type `WitnessFile` or `WitnessFile_`, or both. As explained in section 3.4, these nodes represent the metadata information generated during the validation process by WitnessLint and correspond to the two data schemata used in WitnessDB. The YAML files will be reconstructed and bundled into a ZIP archive, regardless of the data schema used for storage in the database. For user convenience, both node types are tagged with two additional common labels, indicating whether the file contains violation witnesses or correctness witnesses. Users are not restricted to selecting only one type of witness, both types can be included together in the same ZIP archive.

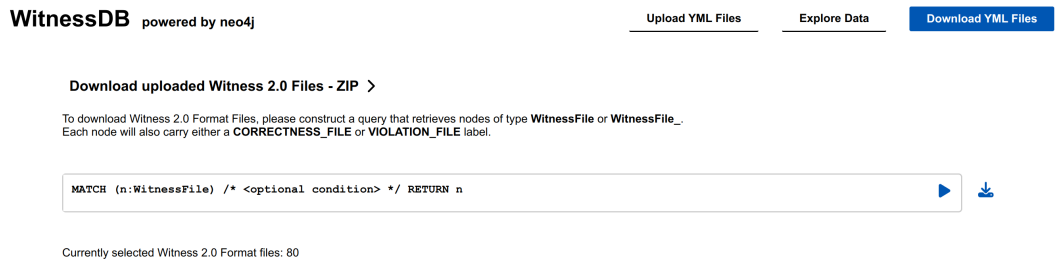


Figure 3.37: Download uploaded Files Interface

If the user chooses to compose a new YAML file containing one or multiple verification witnesses, they need to construct a query that returns nodes of type Witness or `Witness_`, or both, which correspond to the two data schemata. It is very important that the user selects only correctness or violation witnesses, but not both, as described. Since the newly generated files are validated by WitnessLint, the composition will be rejected if the included witnesses are of mixed types. This is currently the only content-level constraint enforced by WitnessLint without available program source code, aside from requiring a valid YAML structure. If the user ignores this rule, the system will return an empty ZIP archive to indicate the invalid composition. The UI component ensures that only nodes of type `Witness` or `Witness_` are selected, but it does not verify whether all selected witnesses are of the same type. For user convenience, both node types carry additional labels that indicate whether they represent correctness or violation witnesses.

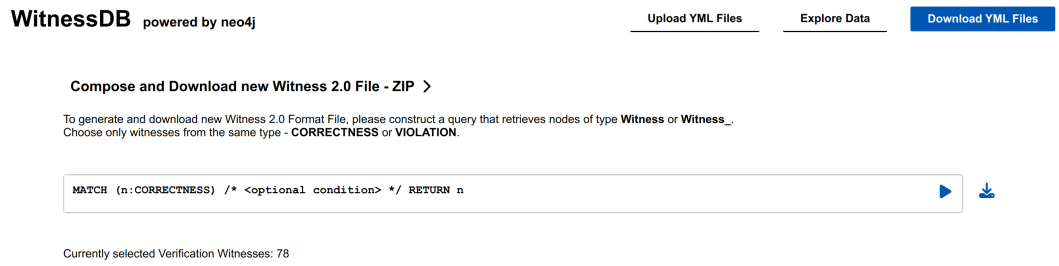


Figure 3.38: Download composed File Interface

3.6.2 Processing Logic in the WitnessDB Core Component

The WitnessDB Core component provides two dedicated HTTP APIs to support file downloading functionality: one for reconstructing originally uploaded witness files, and another for constructing new YAML files composed of selected verification witnesses. Both endpoints accept a list of node identifiers sent by the UI component as input.

To handle files from both supported data schemata (subgraph-oriented and interconnected), the Core component differentiates between the two by using a naming convention: identifiers associated with the subgraph-oriented schema have special prefix. This allows the system to quickly determine the appropriate reconstruction logic to apply without additional queries.

```
1 "identifier": "_witness-b7c562a6"
2 "identifier": "_witness_file-ebeb4131"
```

Listing 9: Example of subgraph-oriented schema node identifiers

```
1 "identifier": "witness-7356efca"
2 "identifier": "witness_file-7356efca"
```

Listing 10: Example of interconnected schema node identifiers

If, during the processing of either download endpoint, any of the provided identifiers cannot be resolved to a valid node in the database,the system collects it and appends a `errors.txt` file to the resulting ZIP archive.

3.6.2.1 Reconstructing Originally Uploaded Witness Files

When witness files are uploaded to WitnessDB, the system stores not only the witness data but also a related metadata node, which includes the original file name as explain in section 3.4. During the reconstruction process, the database is queried using the identifiers of `WitnessFile` or `WitnessFile_` nodes. These nodes reference the stored verification witnesses, which are then retrieved and reconstructed in their respective domain-specific data classes according to the original data schema used at upload time.

Each reconstructed entity is then mapped back to the intermediate data representation used for serialization. This intermediate structure is serialized into a YAML document and the serialized output is once again validated using WitnessLint. The output files are named using their original hash-based file names, which are retrieved from the metadata nodes.

3.6.2.2 Composing New Witness Files from Stored Entities

During this process, the WitnessDB Core component receives a list of identifiers corresponding to `Witness` or `Witness_` nodes. These nodes represent verification witnesses stored in the two supported data schemata. Each node is resolved and reconstructed into its respective domain-specific data class and then mapped to the intermediate data structure used for YAML serialization. The resulting list of intermediate data instances is serialized into a single YAML file, representing a composed witness file. A new hash value is computed from the contents of the serialized file to serve as its file name. Before the file is returned to the user, it is passed through the WitnessLint validation process. Only if the file passes this validation is it included in the ZIP archive sent back to the UI component. If the validation fails the resulting archive will be empty.

4 Discussion

This chapter discusses key aspects of WitnessDB. The goal is to evaluate the system's design choices, identify limitations and outline opportunities for further development.

4.1 Validation and Data Integrity

Graph databases differ fundamentally from traditional relational databases, both in their intended purpose and in the way they handle data. In SQL databases, data is inherently structured as independent records within tables, and connections between data points must be explicitly established through constructs such as foreign keys, join tables, or database views. These mechanisms are necessary to generate relationships and gain insight across tables that are otherwise isolated by design.

In contrast, graph databases operate under the assumption that data is inherently connected. This makes them especially suitable for domains such as social networks, recommendation systems and, in the context of this project, the interconnected structure of witness files. Given that the system is built on an interconnected data model, it was critical to ensure that any data introduced to the system meet the expected schema and format. Verification tools are themselves under active development, and as a result, many generated witness files fail to conform fully to the specification. To address this, an external validator was integrated into the upload pipeline.

Throughout the development process, I uploaded various batches of witness files and observed that **approximately 25% of the files were rejected**. This rejection rate underlines the importance of the validation step: it ensures the integrity of the graph by preventing malformed or incomplete data from being introduced.

One trade-off of using an external validation tool is that it introduces latency into the upload workflow, resulting in a relatively slow upload process. However, since witness uploads are not expected to be a frequent operation in typical usage, this performance cost was considered acceptable in favor of maintaining the overall integrity and consistency of the graph data.

Another limitation arising from this strong focus on data integrity is that all write operations and schema modifications are disabled in the main user interface. In this first version of the system, users cannot create or delete nodes, add or remove relationships, or modify existing node properties through the user interface. While this design choice was intentional and has its advantages, it also restricts the flexibility of the system. There are certainly additional relationships and properties that could be meaningfully added to the data schema and allowing users to define such extensions

interactively would open new opportunities for data enrichment. Although deleting witness data is not expected to be a routine operation, any data storage system should support controlled data deletion as a core capability. Introducing user-driven schema evolution and selective data editing represents a clear direction for future development.

4.2 Interaction Design and Usability

The user interaction model in WitnessDB relies heavily on writing Cypher queries. While the system includes basic filtering functionality through buttons in the Database Information dropdown and within the Node/Relationship Details panel, these options are primarily intended for quick exploration. For more complex use cases or targeted analyses, users are expected to manually construct custom Cypher queries.

In practice, I found the Cypher query language to be relatively user-friendly with an intuitive syntax and comprehensive documentation. The official Neo4j Cypher Manual [18] offers a lot of examples and interactive demonstrations, while more formal discussions of Cypher's syntax, semantics, and ongoing development can be found in academic work such as Francis et al. [13]. Still, Cypher is less widely adopted than SQL, and users unfamiliar with graph databases may require a learning period to take full advantage of its capabilities.

The decision to expose a flexible, query-driven interface instead of constraining users with rigid UI elements was intentional. Since this initial version of the system is designed for general-purpose exploration of verification witnesses and the specific use cases are not predetermined, prioritizing flexibility over strict guidance ensures that users can adapt their queries and views to their individual needs.

4.3 Comparison of Data Models

4.3.1 File Structure-Based Model

The file-based data model puts more emphasis on storing data inside nodes as properties and makes less use of relationships. This modeling approach resembles a SQL-oriented way of thinking, where all the data is embedded within nodes and each node can be seen as equivalent to a row in a table, with the properties corresponding to columns. In this model, relationships play a role similar to foreign keys rather than representing meaningful domain-inspired connections. This focus on nodes over relationships is also reflected in the schema and naming conventions used in the model.

One advantage of this approach is that users with a strong SQL background may find it easier to work with, since the structure aligns more closely with relational

data models and because Cypher is partly inspired by SQL.

However, one main disadvantage of this model is that it ignores the inherent strengths of graph databases and the Cypher language. Connecting data across files or witnesses requires writing more complex queries, similar to what would be needed in SQL to join multiple tables. Another drawback is that data gets duplicated - properties with the same values must be stored separately for each node, leading to redundancy and potential inconsistencies.

Still, the development of this model was significantly easier compared to the interconnected model and modifications to the witness format can be integrated more easily in this approach.

For quick exploration of the data, the file-based model offers both advantages and disadvantages. When witnesses are explored using the *Graph Visualization* interface, there is relatively little to observe beyond the overall structure of a witness file. Since all witness subgraphs follow a similar pattern, the visualization primarily provides high-level aspects such as the type of witness, which is indirectly reflected in the subgraph's structure or the number of invariants or segments associated with it. However, the actual details and meaningful information are embedded within node properties, which requires users to open the *Node Details* panel to inspect specific data points.

An example query that retrieves the subgraphs of the first three `WitnessFile_` nodes is shown in Listing 11, and the corresponding subgraphs are visualized in Figure 4.1.

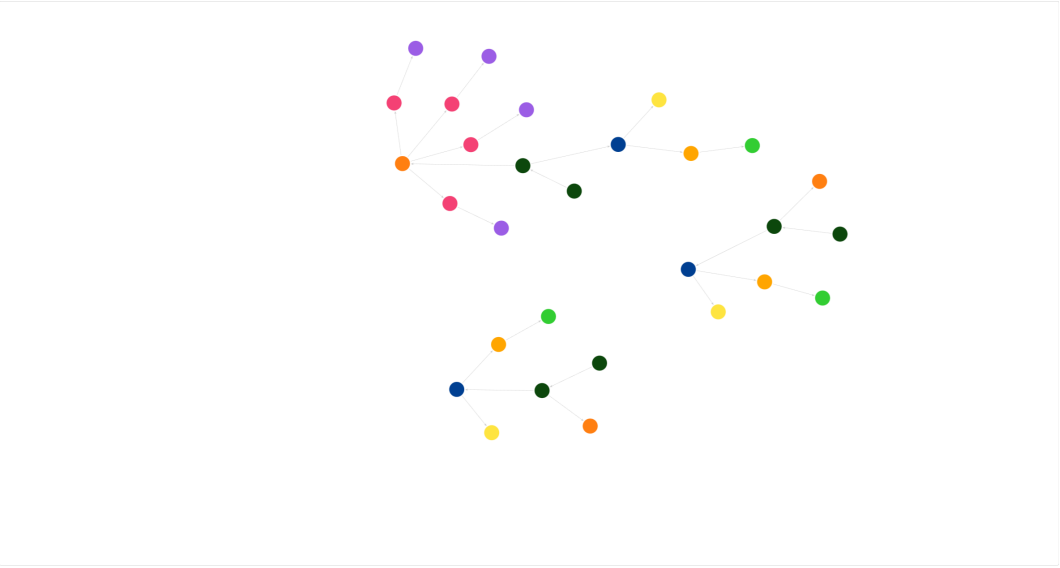


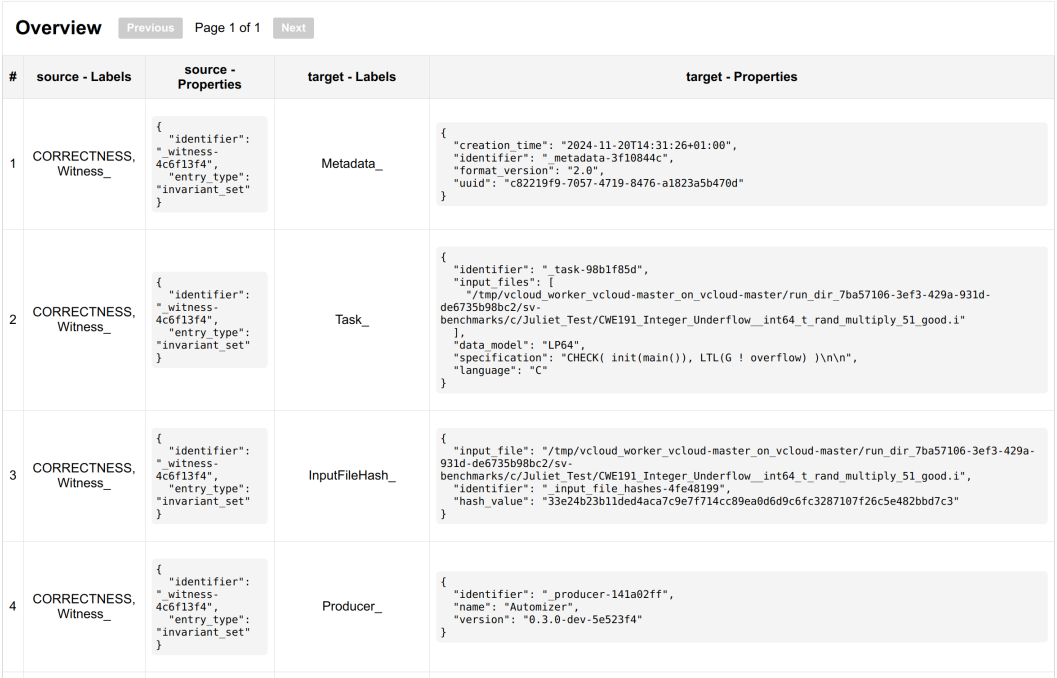
Figure 4.1: Three subgraphs corresponding to diffrent `WitnessFile_` nodes

```
MATCH (n:WitnessFile_) WITH n LIMIT 3
MATCH path=(n)-[*]-(m) RETURN path
```

Listing 11: Query to return subgraphs of first three `WitnessFile_` nodes

By contrast, the *Table Representation* is far more suitable for both quick and in-depth exploration of the data in this model. Because the details are stored inside the nodes as properties rather than expressed through the graph's structure, a tabular format provides more direct access to the information, allowing users to view, filter, and analyze the data more efficiently without navigating the graph visually.

As illustrated in Figure 4.2, the transition relationships between nodes can be examined in a more structured format. The query used to generate these transitions is shown in Listing 12.



Overview Previous Page 1 of 1 Next

#	source - Labels	source - Properties	target - Labels	target - Properties
1	CORRECTNESS, Witness_	{ "identifier": "_witness-4c6f13f4", "entry_type": "invariant_set" }	Metadata_	{ "creation_time": "2024-11-20T14:31:26+01:00", "identifier": "_metadata-3f10844c", "format_version": "2.0", "uuid": "c82219f9-7057-4719-8476-a1823a5b470d" }
2	CORRECTNESS, Witness_	{ "identifier": "_witness-4c6f13f4", "entry_type": "invariant_set" }	Task_	{ "identifier": "_task-98b1f85d", "input_files": ["/tmp/vcloud_worker_vcloud-master_on_vcloud-master/run_dir_7ba57106-3ef3-429a-931d-de6735b98bc2/sv-benchmarks/c/Juliet_Test/CWE191_Integer_Underflow__int64_t_rand_multiply_51_good.i"], "data_model": "LP64", "specification": "CHECK(init(main()), LTL(G ! overflow))\n\n", "language": "C" }
3	CORRECTNESS, Witness_	{ "identifier": "_witness-4c6f13f4", "entry_type": "invariant_set" }	InputFileHash_	{ "input_file": "/tmp/vcloud_worker_vcloud-master_on_vcloud-master/run_dir_7ba57106-3ef3-429a-931d-de6735b98bc2/sv-benchmarks/c/Juliet_Test/CWE191_Integer_Underflow__int64_t_rand_multiply_51_good.i", "identifier": "_input_file_hashes-4fe48199", "hash_value": "33e24b23b11ded4aca7c9e7f714cc89ea0d6d9c6fc3287107f26c5e482bbd7c3" }
4	CORRECTNESS, Witness_	{ "identifier": "_witness-4c6f13f4", "entry_type": "invariant_set" }	Producer_	{ "identifier": "_producer-141a02ff", "name": "Automizer", "version": "0.3.0-dev-5e523f4" }

Figure 4.2: Nodes connected via transition relationships

```
MATCH (n:Witness_) WITH n LIMIT 1
MATCH path=(n)-[*]-(m)
RETURN n as source, m as target
```

Listing 12: Query to return transitions from a `Witness_` node

This model is also better suited for using transition relationships as the subgraphs are smaller and more isolated, whereas in the interconnected data model such relationships can lead to performance issues due to the increased graph complexity and traversal cost.

4.3.2 Domain-Based Model

The domain-based model places greater focus on the interconnected nature of verification data. Similarities and shared elements across witnesses are externalized into separate nodes, which are connected through semantically meaningful relationships. This design makes it possible for users to traverse the data model along intuitive, sentence-like paths, aligning closely with the natural structure of graph databases. This model encourages a more native interaction with interconnected data, making it easier to focus on specific subgraphs that correspond to particular use cases or research questions. It also makes it possible to integrate additional data that is not included in the witness file format but plays an important role in the verification process. For example, the metadata subgraph representing the context of a SV-COMP competition can be easily extended with details, such as computational and storage resource information. Similarly, the verification witness itself can be enriched by linking it to a graph representation of the program being verified, which is currently only represented by its file path.

```
1 MATCH path =(n:WitnessFile)-[r*]->(m)
2 WHERE ALL(rel IN r WHERE type(rel) <> 'CONTAINS')
3 RETURN path
```

Listing 13: Query to create a subgraph of all metadata-related nodes and relationships across multiple witnesses

This model also proves well-suited for exploring the data using the *Graph Visualization* interface, since a significant portion of the information is stored directly in the graph's structure rather than solely in node properties. The interface can render subgraphs relevant to a specific use case while filtering out unnecessary details. Through the built-in navigation features, users can traverse the entire model as needed, following meaningful relationships to explore connected data. For example, Figure 4.3 illustrates a subgraph of violation witnesses, including their segments and associated waypoints that are classified as *branching*. The corresponding Cypher query used to extract this subgraph is shown in Listing 14.

```
1 MATCH path=(m:Witness)-[]-(n:ViolationSequence)-[]-(s:Segment)
     -[:FOLLOW]-(w:Waypoint)-[:OF_TYPE]-(t)
2 WHERE t.identifier = 'branching'
3 RETURN path
```

Listing 14: Query to extract subgraphs with branching waypoints for Graph Visualization

```
1 MATCH(witness:Witness)-[]-(n:ViolationSequence)-[]-(segment:
     Segment)-[f:FOLLOW]-(waypoint:Waypoint)-[o:OF_TYPE]-(type)
2 WHERE type.identifier = 'branching'
3 RETURN witness, segment, f, waypoint, o, type
```

Listing 15: Query to extract subgraphs with branching waypoints for Table Representation

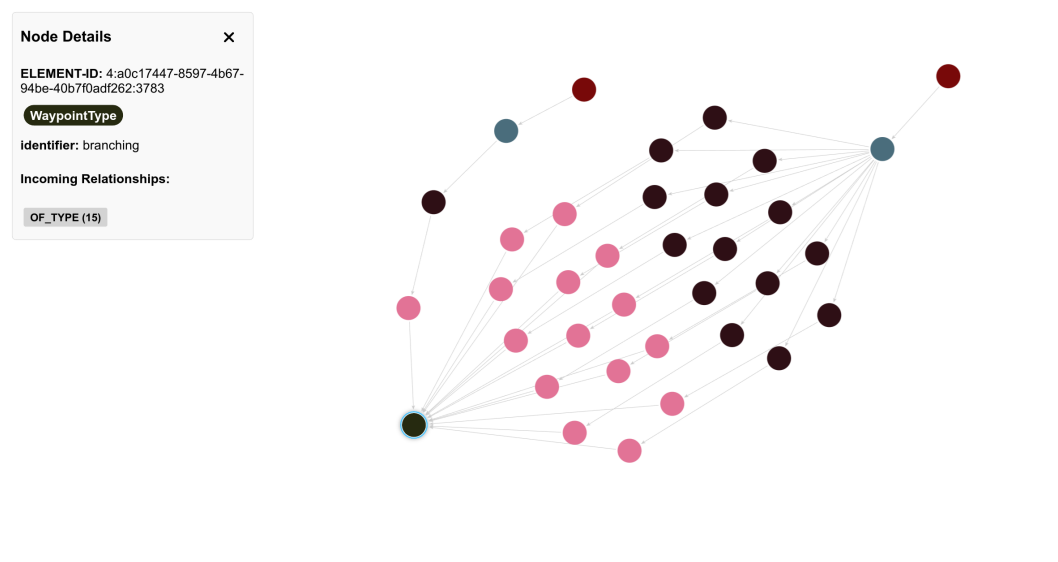


Figure 4.3: Subgraph showing violation witnesses with segments containing waypoints of type `branching`

However, if a user prefers summarized information about a single witness in a table format, it becomes necessary to assemble data from multiple interconnected nodes and relationships, introducing additional query complexity compared to the file-based model. Another drawback is the higher development effort required to construct this model. Much of this complexity is related to the need to translate witness data from YAML format into a graph structure, this transformation required explicit modeling of nested mappings and sequences. Using graph-native formats like GraphML for further use cases would reduce the effort required for such integration. The table view shown in Figure 4.4 is generated using the query in Listing 15.

witness - Labels	witness - Properties	segment - Labels	segment - Properties	f - Type	f - Properties	waypoint - Labels	waypoint - Properties	o - Type	o - Properties	type - Labels	type - Properties
Witness, VIOLATION	{ "creation_time": "2024-11-14T11:48:47+01:00", "identifier": "witness-fa8e4efe", "witness_uuid": "b85f660f-3603-3578-9a89-bf2b23d7d111", "entry_type": "violation_sequence" }	Segment	{ "identifier": "segment-87263435" }	FOLLOW	{}	Waypoint	{ "identifier": "waypoint-dca879be" }	OF_TYPE	{}	WaypointType	{ "identifier" "branching" }
Witness, VIOLATION	{ "creation_time": "2024-11-01T17:03:53+01:00", "identifier": "witness-2f5dcea8", "witness_uuid": "99ffc6f0-5c47-3659-9296-9b07a8ed60f5", "entry_type": "violation_sequence" }	Segment	{ "identifier": "segment-d4356ef2" }	FOLLOW	{}	Waypoint	{ "identifier": "waypoint-1a5e5d5e" }	OF_TYPE	{}	WaypointType	{ "identifier" "branching" }

Figure 4.4: Table snippit showing violation witnesses with segments containing waypoints of type `branching`

4.4 Reconstructing Witness Format 2.0 YAML Files

Data extraction and download are fundamental features of any storage system. However, it was essential not only to provide the ability to export custom-composed data (in graph or table formats) but also to make it possible for users to retrieve the original witnesses in YAML format. This is important because verification tools consume witnesses in their standardized YAML representation.

The reconstructed YAML file may not exactly match the original input in terms of formatting or serialization details, as different serialization tools are used in the reconstruction process. Still, the system guarantees that the reconstructed YAML file is syntactically valid and semantically equivalent, since every generated file undergoes validation by the external validator at the end of the reconstruction process. While this validation step ensures correctness, users should be aware that downloading a large number of files may take longer, as each file is individually validated. The reconstruction process unifies the syntax of witnesses produced by different verification tools. Because the Witness 2.0 format defines a witness file as a sequence of one or more entries, the system supports combining multiple compatible witness entries into a single YAML file.

5 Future Work

Future improvements can be grouped into three main directions: enhancing existing system features, extending the data model and exploring advanced analytical use cases.

5.1 Feature Enhancements

Several improvements could extend the functionality of the current system. Export functionality, for example, could include support for formats such as `.graphml`, which is compatible with Neo4j but relies on procedures, which to a degree are only available in the Enterprise Edition. For more information see: Neo4j - Export to GraphML.

Likewise, advanced user management such as role-based access control is also not supported in the Community Edition, as discussed in section 3.2.1.1, which limits the deployment of the system in multi-user or security-critical environments. Addressing these limitations could be a valuable focus in future development efforts.

Further enhancements may also target Cypher-related features. Supporting the upload and execution of complete Cypher script files would make it easier to perform complex graph transformations and analytical workflows. A query management system could allow users to store, organize, and reuse frequently used queries more efficiently. As discussed in section 4.1 controlled, user-driven modifications to the schema could increase flexibility when adapting the data model to new use cases.

5.2 Data Model Extensions

An important direction for extending the graph model could be the integration of program-level data as first-class graph structures. Programs themselves can be naturally represented as graphs and are interpreted as automata by verification tools. This structural similarity is reflected in the fact that the original witness format was based on GraphML [6]. Since witnesses are inherently tied to the semantics of a specific program, integrating program structure into the graph would create a more complete and semantically grounded view of the verification process. Neo4j provides native support for importing `.graphml` data for direct integration without requiring external transformation pipelines. For more information see Neo4j - Import GraphML

Information regarding SV-COMP such as computational resources details, benchmark categories or competition results could also be added to the graph. Such data

could contribute to building a comprehensive, graph-based knowledge representation of the verification ecosystem.

5.3 Analytical Use Cases

Beyond data storage and visualization, a graph-based representation of verification artifacts opens up new opportunities for analysis-driven use cases, for example through the application of machine learning techniques.

In the context of tool selection, Czech et al. [10] propose a method for predicting the relative performance of verification tools using machine learning. Their approach models program structure as a graph that combines elements of control flow, program dependence, and abstract syntax trees. By applying label ranking algorithms with graph-based kernels, they are able to estimate which tools are likely to perform best on a given verification task, without executing them. This example illustrates how structural properties of programs, when represented as graphs, can serve as input to effective learning models.

Another example comes from the domain of hardware verification, where graph-structured data also plays a key role. According to Wu et al. [25], machine learning techniques are increasingly used to improve both formal and simulation-based hardware verification workflows. Graph-based learning approaches are applied to optimize SAT solving and equivalence checking, while other ML techniques support test generation, bug localization, and coverage analysis. The underlying artifacts in these processes are naturally represented as graphs, which makes them compatible with a wide range of ML models.

These examples show that graph-based representations are not only suitable for modeling verification data, but also beneficial for integrating intelligent analysis methods. The system developed in this thesis already stores data, which could serve as a foundation for similar ML-based applications.

6 Conclusion

This thesis presented the design and implementation of *WitnessDB*, a graph-based system for storing, querying and exploring valid software verification witnesses. The system addresses a key limitation in the current handling of witness artifacts, which are distributed as files within archive bundles.

By leveraging Neo4j, the system represents Witness Format 2.0 data as structured and semantically meaningful graphs. This includes a formalized schema design, a conversion pipeline from YAML to graph structures and a mechanism for reconstructing valid witness files from the graph. WitnessDB follows a modular architecture with distinct components for processing, validation, storage and user interaction. A web-based interface makes file upload, query execution and interactive data exploration possible. The system supports both a file-structured and a normalized representation of witness contents, offering flexibility for use cases that require either minimal structural overhead or deeper semantic modeling.

The approach demonstrated that graph-based modeling of verification artifacts allows more flexible interaction with the data than traditional formats such as files or relational databases. The system establishes a general foundation that can be extended to support a wide range of future developments in verification infrastructure, analysis and tool integration. While certain features remain beyond the scope of this work, the architecture is designed to handle such extensions with minimal structural changes.

In summary, WitnessDB demonstrates how graph-based modeling can serve as a robust and extensible platform that bridges data representation, validation workflows and interactive analysis within a unified framework.

Appendix A: Project Setup and Deployment Instructions for WitnessDB

This appendix documents the setup and deployment requirements for the WitnessDB system developed in this thesis. The information reflects the tested configuration and provides guidance for running the application.

Prerequisites

WitnessDB was developed and tested using the following software versions:

- Maven 3.8.8
- Java 22.0.1
- Docker 28.1.1

Other compatible versions may work, but the above versions are known to be stable with the project.

All other tools and dependencies required by the system are provided within the Docker container environments. No additional installations of supporting tools (e.g., Node.js, npm, Python, pip, Neo4j) are needed.

Build and Deployment

Detailed instructions for building and deploying WitnessDB can be found in the project's `README` file, located in the root directory of the repository.

Configuration

Database credentials and related configuration files are located in the `resources` directory of the main project folder.

By default, WitnessDB assumes that all components are running on `localhost`. If deployed on a different host or distributed setup, the configuration files must be updated accordingly.

Bibliography

[1] P. Ayaziová, D. Beyer, M. Lingsch-Rosenfeld, M. Spiessl, and J. Strejček. Software verification witnesses 2.0. In T. Neele and A. Wijs, editors, *Model Checking Software*, pages 184–203, Cham, 2025. Springer Nature Switzerland.

[2] D. Beyer. A data set of program invariants and error paths. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 111–115, 2019.

[3] D. Beyer. Competition on software verification and witness validation: Sv-comp 2023. In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 495–522, Cham, 2023. Springer Nature Switzerland.

[4] D. Beyer and M. Dangl. Verification-aided debugging: An interactive web-service for exploring error witnesses. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, pages 502–509, Cham, 2016. Springer International Publishing.

[5] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig. Verification witnesses. *ACM Transactions on Software Engineering and Methodology*, (4), 2022.

[6] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 721–733, New York, NY, USA, 2015. Association for Computing Machinery.

[7] D. Beyer and J. Strejček. Improvements in software verification and witness validation: SV-COMP 2025. In *Proc. TACAS (3)*, LNCS 15698. Springer, 2025.

[8] L. Bschor. Modern architecture and improved UI for tables of BENCHEXEC. Bachelor's Thesis, LMU Munich, Software Systems Lab, 2019.

[9] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, Nov. 2009.

[10] M. Czech, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim. Predicting rankings of software verification competitions, 2017.

[11] B. Dirk and S. Jan. Results of the 14th intl. competition on software verification (sv-comp 2025), Mar. 2025.

[12] T.-T.-T. Do, T.-B. Mai-Hoang, V.-Q. Nguyen, and Q.-T. Huynh. Query-based performance comparison of graph database and relational database. In *Proceedings of the 11th International Symposium on Information and Communication*

Technology, SoICT '22, page 375–381, New York, NY, USA, 2022. Association for Computing Machinery.

[13] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. SIGMOD '18, page 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.

[14] M. Hailer. New approaches and visualization for verification coverage. Master's Thesis, LMU Munich, Software Systems Lab, 2022.

[15] Kotlin. Kotlin introduction. `https://kotlinlang.org/`, 2025. Accessed: 2025-05-07.

[16] S. Münchow. A web frontend for visualization of computation steps and their results in cpachecker. Bachelor's Thesis, LMU Munich, Software Systems Lab, 2020.

[17] Neo4j. Connect to a neo4j dbms. `https://neo4j.com/docs/browser-manual/current/operations/dbms-connection/`, 2025. Accessed: 2025-05-07.

[18] Neo4j. Cypher query language manual. `https://neo4j.com/docs/cypher-manual/current/introduction/`, 2025. Accessed: 2025-05-07.

[19] Neo4j. Neo4j browser. `https://neo4j.com/docs/browser-manual/current/`, 2025. Accessed: 2025-05-07.

[20] Neo4j. Neo4j visualization library. `https://neo4j.com/docs/nvl/current/`, 2025. Accessed: 2025-05-07.

[21] Neo4j. Neo4j visualization library - nvlresulttransformer. `https://neo4j.com/docs/api/nvl/current/functions/_neo4j_nvl_base.nvlResultTransformer.html`, 2025. Accessed: 2025-05-07.

[22] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, Inc., 2nd edition, 2015.

[23] Spring. Spring overview. `https://spring.io/`, 2025. Accessed: 2025-05-07.

[24] J. Stegeman. Native vs. non-native graph database. `https://neo4j.com/blog/cypher-and-gql/native-vs-non-native-graph-technology/`, 2023. Accessed: 2025-05-07.

[25] N. Wu, Y. Li, H. Yang, H. Chen, S. Dai, C. Hao, C. Yu, and Y. Xie. Survey of machine learning for software-assisted hardware design verification: Past, present, and prospect. *ACM Trans. Des. Autom. Electron. Syst.*, 29(4), June 2024.