software

MDPI

Article

Analysing and Transforming Graph Structures: The Graph Transformation Framework

Andreas H. Schuler [1,2,*,†], Christoph Praschl [2,†] and Andreas Pointner [2,†]

[1] Department of Telecooperation, Johannes Kepler University Linz, 4040 Linz, Austria
[2] Research Group Advanced Information Systems and Technology, Research and Development Department, University of Applied Sciences Upper Austria, 4232 Hagenberg, Austria; christoph.praschl@fh-hagenberg.at (C.P.); andreas.pointner@fh-hagenberg.at (A.P.)
* Correspondence: andreas.schuler@fh-hagenberg.at
† These authors contributed equally to this work.

Abstract: Interconnected data or, in particular, graph structures are a valuable source of information. Gaining insights and knowledge from graph structures is applied throughout a wide range of application areas, for which efficient tools are desired. In this work we present an open source Java graph transformation framework. The framework provides a simple fluent Application Programming Interface (API) to transform a provided graph structure to a desired target format and, in turn, allow further analysis. First, we provide an overview on the architecture of the framework and its core components. Second, we provide an illustrative example which shows how to use the framework's core API for transforming and verifying graph structures. Next to that, we present an instantiation of the framework in the context of analyzing the third-party dependencies amongst open source libraries on the Android platform. The example scenario provides insights on a typical scenario in which the graph transformation framework is applied to efficiently process complex graph structures. The framework is open-source and actively developed, and we further provide information on how to obtain it from its official GitHub page.

Keywords: graph analysis; framework; transformation; verification; network analysis

Citation: Schuler, A.H.; Praschl, C.; Pointner, A. Analysing and Transforming Graph Structures: The Graph Transformation Framework. *Software* **2023**, 2, 218–233. https://doi.org/10.3390/software2020010

Academic Editor: Andreas L. Symeonidis

Received: 24 February 2023
Revised: 21 March 2023
Accepted: 4 April 2023
Published: 6 April 2023

1. Introduction

In recent years, the significance of network analysis and graph structures has grown substantially in both academia and industry. This is because they offer the capability to represent and analyze massive amounts of interconnected data. In this way, it is possible to gain valuable insights across a broad spectrum in the area of application domains, including but not limited to planning, vulnerability analysis, social networks, modeling, and also gene expression, protein interactions, Web graph structure, Internet traffic analysis, and disease spread through contact networks [1–7]. For example, Jo et al. [8] used graph representations for the visualization of online bibliographic datasets, together with an interactive interface for performing visual exploration of the information network, to get an in-depth analysis of, for example, understanding the history and flow of research, its current status, and ongoing trends. Ureña et al. [9] proposed a completely different use-case for utilizing graphs allowing to model the relationships and the history of reputations such as likes of social media users as a source of trustworthiness. The growth of network theory is driven by its cross-disciplinary significance and it serves as a crucial tool in a systematic method of understanding complex systems. To meet the growing demand for efficient graph analysis tools, several libraries have been proposed that offer a comprehensive and user-friendly approach for defining, representing, and analyzing graph structures [10–13].

Creating a strong, efficient, and versatile graph library is a challenging process with numerous design decisions and performance compromises. While existing libraries primarily focus on the representation and analysis of graph structures, we introduce the

Graph Transformation Framework (GTF), an open-source Java library that offers the means to separate graph representation and analysis. This article outlines the design of GTF, discussing key considerations and showcasing its key features and supported algorithms. GTF provides a simple Java-based API to define and represent a graph, and also includes a mechanism to transform the graph structure to an arbitrary output format, allowing for subsequent analysis and verifying its consistency. This makes GTF suitable for scenarios where an input network structure needs to be transformed into a desired target structure.

GTF does not replace existing contributions for graph and network processing in Java, but rather complements them [10–13]. It can be seamlessly integrated into existing workflows for graph and network analysis, providing users with an additional tool for transforming and verifying graph structures. In summary, this paper makes the following contributions:

- An open source framework for the definition, transformation and verification of graph structures. The framework is hosted on GitHub, including a wide variety of examples and possible application domains (https://github.com/FHOOEAIST/GTF) (accessed on 3 April 2023).
- An instantiation of the framework to show how it can be applied in a study of the library dependencies in Android open source applications, for which we provide a replication package available under (https://zenodo.org/record/6077478) (accessed on 3 April 2023).

2. Related Work

Representing and analyzing network structures has seen an upsurge in recent years. As a result, several software libraries have been proposed to *define*, *analyze*, and *visualize* graph structures. Such libraries are deployed among a wide range of different application and research domains, e.g., malware detection [2], software performance analysis [3], and social networking and navigation [9]. With GTF we contributed a software package in Java to aid developers in academia and industry alike to define, represent, and transform typed graph structures. Our approach shares some design considerations with JGraph-T [10]. A library offers efficient and generic graph structures, together with a collection of graph-based algorithms. The library is implemented in Java, which, due to its object-oriented nature, allows JGraph-T to model edges and vertices as arbitrary objects. It was first introduced in 2003 and has since been updated and extended regularly. However, GTF has a different focus, as JGraph-T offers a wide range of algorithms towards analyzing a network structure. In contrast to that, in our library GTF, the primary goal lies in graph definition and its subsequent transformation to a desired output format. Besides JGraph-T, the Java ecosystem is very limited when it comes to graph libraries and software packages. Since Michail et al. [10] first stated these circumstances, not much has changed. The list of graph packages for the Java platform remains limited, with the exception of JGraph-T [10], Google Guava [11], JUNG [12], and JgraphX [13].

While Google Guava [11] is mainly a set of core libraries for Java with a focus on additional collection types and utility functionality for concurreny, I/O hashing, and more, it also contains a basic module for graph-structured data. It supports three structural types that are not compatible with each other: simple *graphs* with arbitrary values in the nodes, *value-graphs* with values in nodes and edges, and *networks* picking up the features of a value-graph which are also allowed to have parallel edges. The graph module does not provide any functionality for I/O or visualization.

The Java Universal Network/Graph Framework (JUNG) [12] offers a unified and customizable framework for modeling, analyzing, and visualizing data that can be depicted as a graph or network. Its architecture allows us to support a high variety for graph-structured data, and also supports multi-modal graphs, multi-edge graphs, and hypergraphs. In addition to structural variety, it also supports many algorithms from the field of graph theory, data mining, and social network analysis, including, for example, processes for clustering, decomposition, and optimization. The most recent version of the JUNG framework 2.1.1

was published in September 2016. Even if there is no official announcement, this indicates that the framework has reached the end of its life.

JgraphX [13] is a graph library that focuses on the visualization of graphs. It is mainly used for visualization and interaction with node-edge graphs. It provides a wide range of features for visualizing and editing graphs, including layout algorithms, drag-and-drop support, and customizable styles. The framework has been developed for multiple years, but reached its end of life at the end of 2020, and thus active development and maintenance was stopped.

The NetworkX [14] python package offers capabilities to explore and analyze network structures. Next to the basic data structures for representing networks, NetworkX offers a wide range of graph algorithms. For ease of exchange, NetworkX supports reading and writing various graph formats with existing data.

NetworKit [15] is a software package written in C++ with python bindings for the comprehensive structural analysis of massive complex networks. NetworKit shares a modular software architecture to aid code reuse and extensibility, and therefore allows to efficiently contribute new functionality to the library. NetworKit is constantly extended with new functionality and algorithms which allow for exploring and characterizing large network data sets [15].

Two more software packages, implemented in C++ and well-known for their efficiency, are LEDA [16] and the Boost Graph Library [17]. Both libraries provide implementations of a generic graph data structure together with a set of basic graph algorithms.

The Stanford Network Analysis Platform (SNAP) [18] is described as a general purpose, high performance system for analysis and manipulation of large networks. The authors further describe SNAP as capable of handling hundreds of millions of vertices and edges. Furthermore, SNAP provides over 140 graph algorithms for network analysis.

The igraph [19] software package offers a wide range of tools for network science. The package is characterized by its capabilities to handle large graphs with millions of vertices efficiently. Furthermore, it provides functionality to create, manipulate, and visualize networks.

A lightweight graph processing framework specific for multicore architectures, LIGRA, was presented by Shun and Blelloch [20]. LIGRA defines a set of operations that efficiently create graph traversal algorithms, which makes it particularly useful for algorithms that operate on sub-graphs.

Gunrock [21] is a framework for graph-processing which is specifically designed to run on the GPU. The framework offers a high-level programming model to reduce GPU programming knowledge when used.

Visualizing complex networks and graph structures is often a crucial step to gain insights on the distribution of nodes and edges of a graph structure. Throughout the years with GraphViz [22], Gephi [23], Cytoscape [24], and the Graph Visualisation Toolkit [25], several approaches have been described that allow for visual inspection of complex networks, even for cases where the network is considered to be large (e.g., $>$20,000 nodes [23]). Features of these available approaches cover visualization, filtering, manipulating or layouting networks of different structures, size, and complexity.

GraphViz [22] may be one of the most well-known open source software suites when it comes to visualizing networks or hierarchical data. As described by Ellson et al. [22], it is often the weapon of choice when preparing graph visualizations that are included in papers. GraphViz offers a wide variety of algorithms and tools to visualize graphs.

Gephi [23] is an open source software for exploring and manipulating networks and is specialized at visualizing and manipulating large networks. Using a 3D-engine, Gephi allows for rendering large networks in real time. With a wide range of layout algorithms, it allows for exploring dynamic networks.

Graphia [26] is an open source visual analytics applications written in C++ with a focus on large, connected datasets. Comparable to other solutions, such as Gephi, it is intended for exploring and filtering graphs using an out-of-the-box application.

Finally, with Cytoscape [24] and the Graph visualization toolkit [25], there are software suites that allow for visualizing complex network structures independent of the type of data or domain.

In contrast to the existing work, our GTF approach is primarily designed to offer an unobtrusive and intuitive way in network structure representation. Once a particular network structure is defined, GTF offers an easy and extensible API to further process the network to a desired output representation. At the time of writing, GTF itself does not support any form of visualization of network structures. This is due to the fact that we believe that there is already very good software and tools available for this purpose, with the related work in this paper outlining a variety of visualization approaches. Furthermore, we believe that the GTF could be used in combination with the approaches listed above, where a defined graph structure is to be preprocessed and transformed to a desired output format which is supported by either one of the described tools and approaches.

3. The Graph Transformation Framework

In this section we present the Graph Transformation Framework [27] and describe its basic features, which include an API for (1) the definition of graph structures, (2) the specification of rules to transform a graph structure to a desired output format, and (3) the definition of constraints to ensure consistency in relation to a graph transformation. These modules and their interaction are shown in Figure 1.

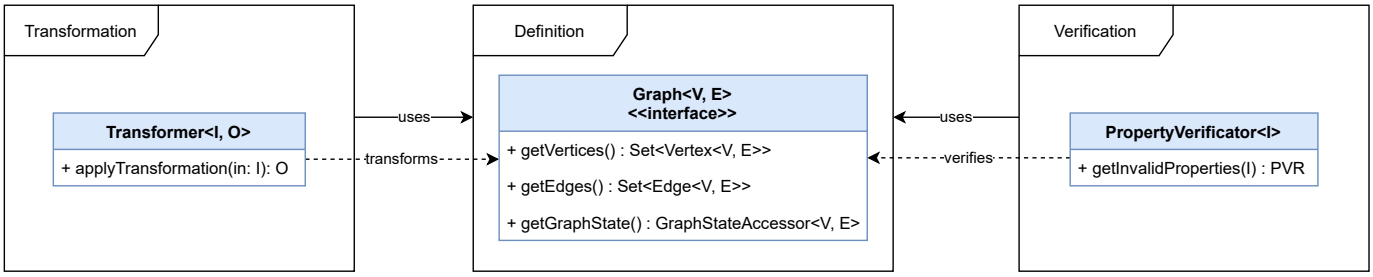

Figure 1. High level overview showing the interaction of the main modules for *Graph Definition*, *Transformation*, and *Verification* based on the corresponding classes.

3.1. Graph Definition

In this work, we rely on a basic graph definition, where a graph G consists of a set of vertices V and a set of edges E, which allows for defining a graph as $G = (V, E)$. In addition, an edge consists of a source and a target vertex, which allows for deriving the two functions $s, t : E \to V$ from it. This formal graph definition builds the foundation for a set of core interfaces defined in GTF and shown in Figure 2. We differentiate between three concepts: ① The graph, with vertices, edges and its traversal strategy, shown in blue. ② A graph state, which holds the internal state, shown in green. ③ A builder concept, to create a new graph, shown in yellow.

The most central component is the generic `Graph` ① interface. This interface is used to represent a graph-based object structure. Furthermore, a `Graph` provides the required means to interact with its components, the vertices, and edges. To access and manage these components a `GraphStateAccessor` ② was used, which represents a read-only view of the state of a graph. Depending on the implementation, this state is either represented as an adjacency matrix or any other means to efficiently store vertices and associated edges. In the context of the `GraphStateAccessor`, there is another extension of this interface called `GraphState`, which not only allows to read the vertices and edges of a graph, but also gives the possibility of manipulating the graph by adding or removing parts. Every vertex and edge in the GTF is used as a container and is decorated by another element. These elements contain the actual information of the graph.

Since the basic `GraphStateAccessor` represents a read-only view of the graph, GTF provides a builder pattern API ③ to create the graph, but is not limited to this way of

creation. For this reason, the `GraphBuilder` interface allows creating vertices and edges via method chaining with "addVertex", "from", and "to" definitions that are delegated to specialized vertex and edge builders called `VertexWithBuilder` and `EdgeBuilder`.

Based on these concepts, we were able to create a weighted, directed graph structure. The structure also supports creating self-referencing vertices. Multiple edges between two nodes are not yet supported. All these limitations are given for the current implementation, and based on the concept it is possible to implement other `Builders` and `GraphStates` that allow for different graphs to be represented. The example shown in Listing 1 shows how the API can be used to create a graph representation. In this example, a graph of persons is built, with relations between them.

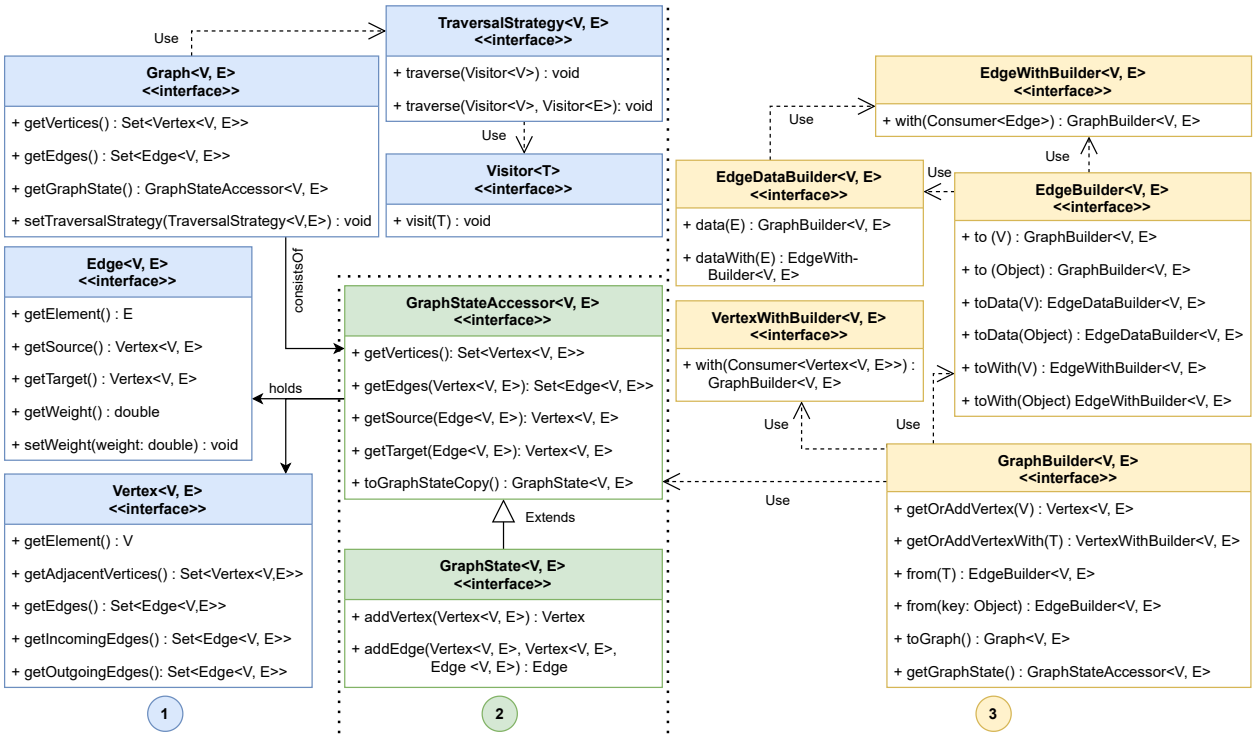

Figure 2. The core interfaces of the GTF with `Graph` as a central component, consisting of a `GraphStateAccessor`, which is responsible for a graph's components with `Edge` and `Vertex` objects. While the accessor represents a read-only view of the graph, it is extended by the write-able variant `GraphState`. Next to the actual graph-related domain interfaces, GTF provides multiple builder concepts to create a graph from scratch using given objects, which should be decorated.

3.2. Graph Transformation

Once a graph instance is defined, the framework provides an API to transform the input graph into a desired output. The format or type of the output, however, is not predetermined. To achieve this, GTF introduces two fundamental concepts; a *Transformer* and a *Renderer*, which are depicted in Figure 3.

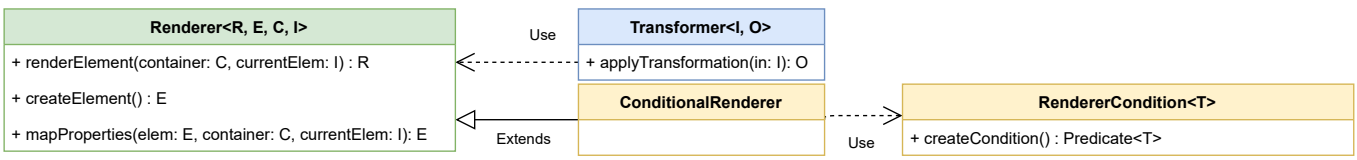

Figure 3. The base transformation classes, consisting of a `Transformer`, which is responsible for base level transformation, and a `TransformationRenderer`, which renders single elements in the transformation process. Additionally, a `ConditionalRenderer` and its `RendererCondition` is shown, which is used to process different elements in the transformation.

Listing 1. Example of how a graph can be generated using the methods from the GTF.

```
var graph = GraphBuilderImpl.<String, String>create()
  .from("Markus").toData("Max")
     .data("is_father")
  .from("Marianne").toData("Max")
     .data("is_mother")
  .from("Max").toData("Erika")
     .data("are_married")
  .from("Erika").toData("John")
     .data("work_together")
  .from("John").toData("Richard")
     .data("are_friends")
  .from("Max").toData("Richard")
     .data("are_best_friends")
  .toGraph();
```

3.2.1. Transformer

Specifying a transformation from a source graph structure requires defining transformation rules. We define that as a function t which transforms a graph G into a target model T, expressed in $t : G(V, E) \rightarrow T$. One thing that has to be highlighted here is that the target model can be of any structure and does not need to be a graph again. Like this, it is possible to transform the graph, for example, to a textual representation. In GTF, such transformation rules are combined using a *Transformer*. GTF provides several *Transformers* out of the box, e.g., the *GraphVizTransformer* which enables transformations from a GTF defined graph structure to the .dot-format. The *Transformer* is further responsible to traverse the source graph structure and apply the available transformation rules on a per-element basis.

3.2.2. Renderer

For each single element traversed, a *Renderer* is used to transform it from its source representation to the desired target representation. Henceforth, a `Renderer` is responsible for rendering a single element as part of the whole transformation process. Given a graph structure as the source input for the transformation process, a *Renderer* is responsible to transform a single `Vertex` or `Edge` from that particular graph instance. A *Transfomer* usually contains a series of *Renderer* implementations which provide three consecutive steps that are executed during the transformation. First, a *Renderer* is responsible for creating the desired target element representation of the current source element it is being applied to. This is achieved using the `createElement` method depicted in Figure 3. Second, the values and the desired properties are mapped from the source element to the target element using the `mapProperties` method. Finally, the `renderElement` method combines those two together and creates the final result. To allow for processing of different types of source elements, a special renderer the `ConditionalRenderer` qA introduced, which decides based on a condition using `RendererCondition` instances if a specific element should be processed by a particular `Renderer` or not.

3.2.3. Illustrative Example

Given the provided abstract description of the two fundamental concepts for transforming a graph structure, we further provided an illustrative example that shows the interplay between the described components and further shed light on how the transformation is applied. The example shows how the graph structure defined in Listing 1 is transformed to a GraphViz representation for visualization. The initial step is to create a transformer, and Algorithm 1 shows the individual steps.

Algorithm 1: Specifying a `Transformer` for a graph to GraphViz transformation.

input : *renderer*—the element renderer
input : dfs—the graph traversal strategy
output: *result*—the transformed result

```
1 transformer = g →
2     g.setTraversalStrategy(dfs);
3     result +="digraph G {";
4     g.traverse (
5         v → {},
6         e → result += renderer.renderElement(g,e));
7     result +="}";
8     return result;
```

A `Transformer` requires a traversal strategy for processing the underlying source graph structure. Furthermore, creating a `Transformer` requires specifying which elements of the graph should be traversed. GTF allows both—a transformer which operates on vertices or on edges. For the transformer in Algorithm 1 we only defined an edge-based traversal which allows for an efficient transformation into valid GraphViz .dot-file statements. The responsibility for the actual transformation of an edge to the desired target element, however, lies within a `Renderer`. Algorithm 2 shows the implementation of the `mapProperties` method, which maps an edge from a traversed graph to its string representation in the GraphViz .dot-file format.

Algorithm 2: Mapping properties from source to target element for simple graph to GraphViz transformation.

```
1 mapProperties(g: Graph, r: Str, e: Edge): Str
2     return ( r +=
3         e.getSource().getElement() + "→" +
4         e.getTarget().getElement() +
5         "[label=" + e.getElement() + "];" );
```

For the case at hand, the mapping is a simple graph to text transformation. However, advanced transformation scenarios, e.g., graph to model transformations, are also possible. Once the `Transformer` and the `Renderer` are defined, graph instances, e.g., the sample graph from Listing 1, can be transformed to a GraphViz representation which results in the rendered graph visualization in Figure 4.

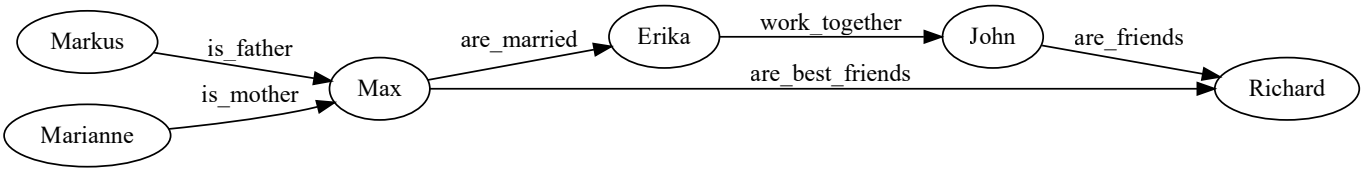

Figure 4. The rendered GraphViz representation that was created with Algorithms 1 and 2 from the graph created in Listing 1. Rendered with https://dreampuf.github.io/GraphvizOnline (accessed on 3 April 2023).

3.3. Graph Verification

A crucial part in the GTF when transforming a source graph into a target representation is to ensure that the input source satisfies all requirements to ensure the validity of the resulting target output. Therefore, in the GTF, we provide a mechanism to verify if a source input graph is syntactically sound according to given constraints [28]. We define a constraint r that takes any input I and results in a boolean result, expressed as $r : I \rightarrow boolean$.

The verification v takes a set of these constraints rs and individually applies them with the given input I, expressed as $v(rs, I) : \forall r \in rs \rightarrow r(I)$. This verification process is loosened from the actual graph structure used in GTF to contribute a generally usable method, as shown in Figure 5. To achieve this, a `PropertyVerificator` was used which returns invalid properties in the form of a `PropertyVerificatorResult` for a given object and its referenced child objects. Invalid properties are defined via constraints, which are managed by a `PropertyRestrictor` object on either a class or field level. The traversal of the underlying graph was done via a visitor pattern based on the `AbstractVisitor` class, which is extended by different implementations which are built in descending order from object down to field and primitive class level. Likewise, an object is disassembled by its components and every part is visited and checked by the verification mechanism. The different visitor implementations are managed by a `ConstraintVisitorFactory` class.

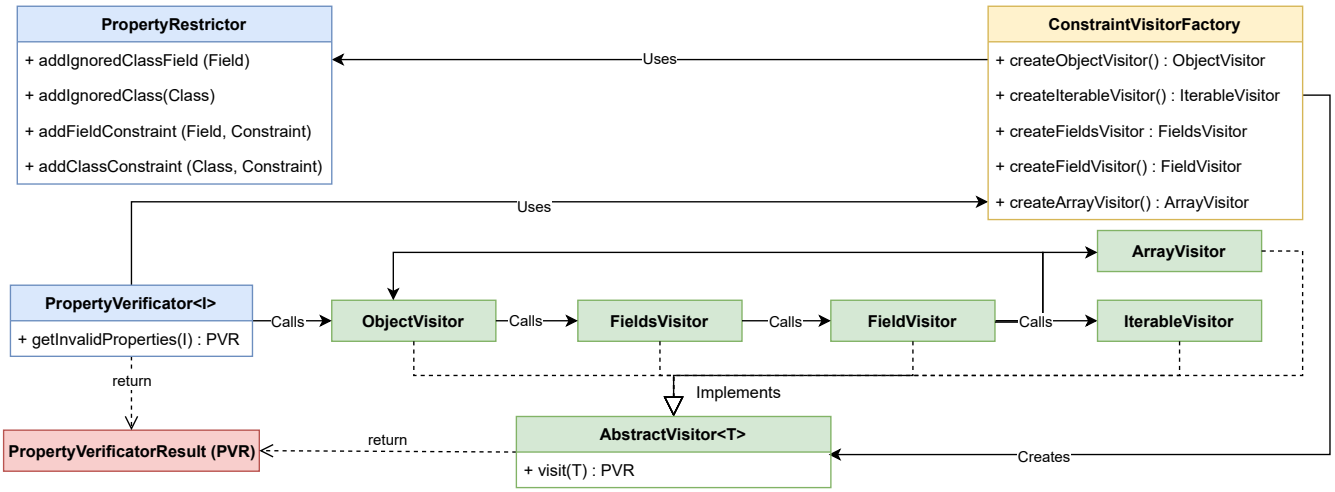

Figure 5. The syntactic verification process of the GTF is based on the `PropertyVerificator` that results in a `PropertyVerificatorResult` and uses an `ObjectVisitor` to check given constraints defined in the `PropertyRestrictor` class. An `ObjectVisitor`, in turn, uses a `FieldsVisitor` to check its fields, where every individual field is again checked by a `Fieldvisitor`. Depending on the type of the field, this visitor checks the defined constraints or repeats the process via a `ObjectVisitor`, `ArrayVisitor`, or `IterableVisitor` for the referenced child objects.

4. Example Scenario

Within this section, an example scenario [29] used for the evaluation of the Graph Transformation Framework is presented. This scenario is based on the graph-based representation of the library dependencies of open source Android applications within the F-Droid repository. Next to the general description of the example scenario, this section also states the process of crawling the data and building an analyzable dependency graph.

4.1. Library Dependencies in Android

This scenario focuses on analyzing the dependencies amongst open source library usage in Android. In particular, we applied the presented GTF to determine the most used third-party libraries in Android open-source apps. We further provided the following guiding questions which we aimed to answer by applying the presented GTF:

- GQ_1**:** *What are the most used non-platform third-party libraries in Android open source applications?* When implementing Android applications, developers usually base their implementations on a selection of third-party APIs for particular purposes. This question aims to shed light on how developers design their applications through the use of well-established libraries and APIs.
- GQ_2**:** *Are there any conflicting dependencies according to the version in the given project?* Conflicting library versions can lead to significant problems at runtime of an applica-

tion. An application may further encounter undesired behavior or even threats to its security if it depends on multiple versions of the same library at the same time.

4.2. Studied Sample

Free and Open Source Android App Repository (F-Droid (https://f-droid.org) (accessed on 3 April 2023)) is a public repository that hosts thousands of open source Android projects. It provides an open API endpoint (https://f-droid.org/repo/index-v1.jar (accessed on 9 August 2021)) with an index file listing all hosted projects, which in turn contains various meta information according to these projects as e.g., the source repository. Using these data, we are able to get further project insights by crawling the individual repositories and likewise are able to determine the individual project's dependencies. Based on this analysis, we can in turn retrieve the transitive dependencies. Summarizing all this information allows us to create a transitive dependency graph of projects hosted on F-Droid.

Due to the heterogeneous project structures, we limited the selected projects based on the following three constraints: (1) We only used GitHub (https://github.com/ (accessed on 3 April 2023)) as a source for projects as it offers an open API to list all project files and to download specific ones using its Universal Resource Identifier (URI), (2) we restricted the build tool to the Android Studio standard Gradle and for support projects containing at least one `.gradle` file, and finally (3) we only included one version per project, which was identified either by `master` or `main` as branch name. Considering these restrictions, we were able to extract 15.074 version-unique dependencies for 1.722 of 2.632 projects, which are hosted on GitHub, from a total of 3.470 F-Droid apps. This procedure is shown as step (1) "Gradle Crawler" of the process in Figure 6.

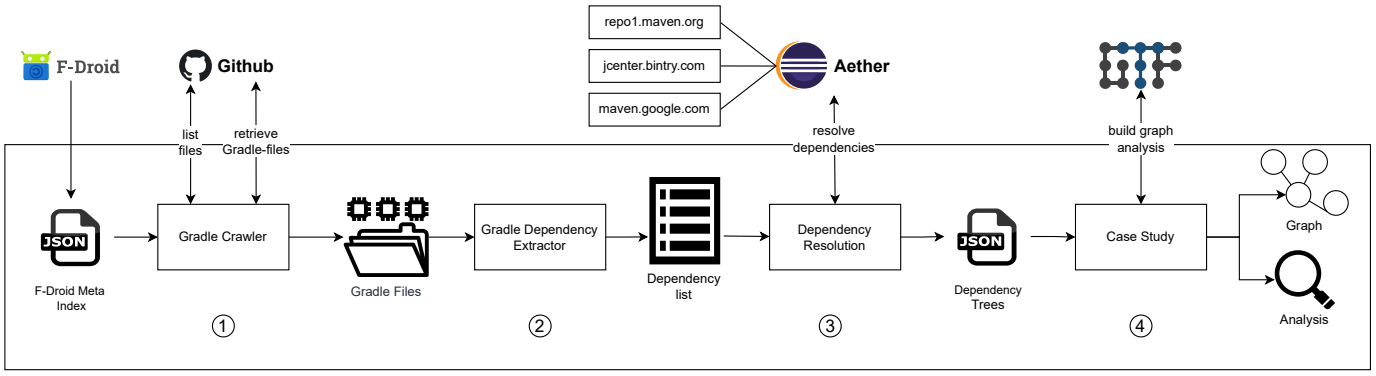

Figure 6. The process of using Android projects from F-Droid to create a library dependency graph and perform analysis on it. The image shows the various steps that are performed, with: (1) Crawling the Gradle files from GitHub, (2) extracting dependencies from the Gradle files, (3) running a dependency resolution to obtain transitive dependency information, and (4) creating and analyzing a dependency graph.

4.3. Obtain Dependency Information

Building up on the retrieved files from the various GitHub-hosted projects as described in Section 4.2 leads to two Gradle-related problems: (1) The actual Gradle build tool cannot be used to retrieve a project's dependency, since it requires the complete project to extract this information. Additionally, (2) due to the high variety of the projects, not only was one Android or one Gradle version used, but multiple ones were. This leads to the situation in which all these software versions are required to build a dependency tree using Gradle for the different F-Droid applications.

To overcome these issues, a parser was used to extract a project's direct dependencies based on the plain Gradle-files. For this, Gradle files are not distinguished according to their locations in the project, but are used as one combined source of information for the associated project to collect all dependencies. This is done by extracting the dependencies with their versions using a pattern matching approach in a first step. Since placeholders

for versions are a common approach in Gradle files, these placeholders were resolved and replaced again using pattern-matching in a second step. This process is shown in Figure 6 as (2) "Gradle Dependency Extractor".

Based on the extracted direct dependencies, the Eclipse Aether (https://projects.eclipse.org/projects/technology.aether (accessed on 3 April 2023)) library can be used to retrieve transitive dependency information. For this, Aether resolves a given Maven or Gradle dependency and results into a dependency tree with all transitive dependencies. Combining the information of direct and indirect dependencies allows for building up a complete dependency tree for a given project, and is shown in step (3) "Dependency Resolution" of Figure 6. These dependency trees are saved using the JSON exchange format in the presented process. Analyzing the results of the Dependency Resolution points out a problem with the dependencies "androidx.appcompat:appcompat:1.3.-beta01" and "androidx.fragment:fragment-ktx:1.3.0-rc01". Both dependencies have the same transitive child dependency, namely "androidx.fragment:fragment:1.3.0-rc01", whereas one time the version was specified with a soft requirement of the version "1.3.0-rc01" and was specified the other time with a hard requirement of the version "[1.3.0-rc01]" as defined for version ranges in Maven (https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html (accessed on 3 April 2023)). Even if the same artifact is retrieved both times, a different dependency tree was created for these scenarios. This problem has to be considered in the creation of the dependency graph.

4.4. Build Graph Structure

Taking into account the structure of the resulting dependencies trees as described in Section 4.3, a specialized graph structure was used. This graph structure contains two different types of elements decorating a vertex in the graph. These two types are `ProjectNode` and `DependencyNode`. The first one contains project-related information that allow us to uniquely identify the project on GitHub as the username of the project owner and the corresponding repository name. The latter one defines the Maven coordinate of the artifact using the `groupId`, the `artifactId` as well as the `version`. A dependency is uniquely identified by the combination of the three individual ones, called the "artifactQualifier". The structure of these two types is shown in Figure 7.

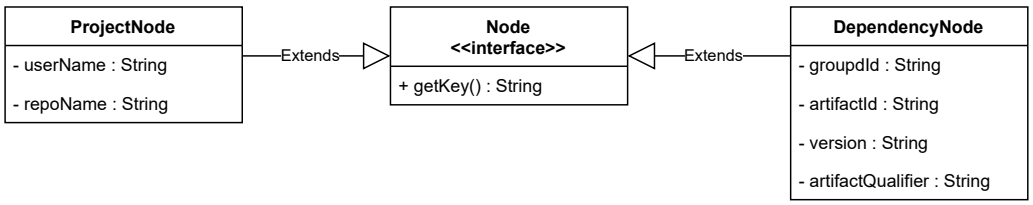

Figure 7. The domain class structure for the elements that are decorated in the graph. The base interface Node defines a function to extract a key which identifies the Node in the graph. The two derived classes `ProjectNode` and `DependencyNode` define the necessary domain data for a building on a dependency graph.

As part of the final step of the presented process shown in Figure 6, the described graph structure was used as a basis to create a dependency graph over all `F-Droid` projects using the Graph Transformation Framework and used the dependency tree files from step (3) Dependency Resolution as input. To achieve the graph creation, every crawled project was represented as `ProjectNode` vertex. Starting from these root nodes, the graph was extended by iterating the dependency trees in a recursive breadth first manner. If a dependency with the same key is already contained in the graph, the used `GraphBuilder` reuses the existing vertex, or otherwise a new vertex is created. Each `DependencyNode` is also connected with its parent, a `DependencyNode` object for indirect project dependencies and a `ProjectNode` for the direct dependencies. If a dependency, which is already part of the graph, contains different transitive child dependencies, the existing branch is extended. This allows to us overcome the dependency issues with hard and soft version constraints,

as described in Section 4.3. The process of creating the graph out of the dependency trees is shown as Java-code in Listing 2.

Listing 2. `makeGraph()` function that defines on how to create the graph from the list of json dependency files. `addDependencies()` function defines the recursion step to iterate through the dependency tree and adds the nodes to the graph by utilizing `graphBuilder` functions.

```
Graph<Node,Void> makeGraph(String[] f){
 GraphBuilder<Node, Void> gb =
 GraphBuilderImpl.create(Node::getKey);

 for (String file : f) {
   var repo = loadFile(file);
   var projectNode =
     ProjectNode.from(repo);
   gb.addVertex(projectNode);
   addDep(gb, projectNode,
     repo.getDependencies());
  }

 return gb.toGraph();
}

void addDep(GraphBuilder<Node, Void> builder, Node parent,
    Collection<Dependency> dependencyList) {
 if (dependencyList.isEmpty()) return;
 for (Dependency dep : dependencyList){
   var node = DependencyNode.from(dep);
   builder.from(parent).to(node);
   addDep(builder, node,
     dep.getChildren());
 }
}
```

4.5. Analyze

The graph created with GTF allows us to execute different programmatic analysis as the usage of individual dependencies over all projects. However, to get a first overview of the graph, the programmatic representation is not that ideal. For this reason, we provided an approach to transform the graph into any other representation, such as a visual one based on GraphViz. Since the obtained dependency graph consists of thousands of nodes, this visual representation is not ideal. As a consequence, GTF also provides the possibility to extract a sub graph based on a given root node, which is a restricted, read-only view of the original graph. In doing so, we are able to reconstruct the project's dependency graph (shown in Listing 3).

Based on these graph structures, we can answer different questions, such as "What are the top ten transitive non-platform dependencies used by the crawled projects?" or "Are there any conflicting dependencies according to the version in the given project?" which we try to answer using transformation and verification approaches.

The Listing 4 allows us to generate a CSV-based representation which can be used to answer GQ_1 "What are the most used non-platform third-party libraries in Android open source applications?". The top ten non-platform dependencies of the crawled projects are shown in Figure 8. The count of these dependencies also includes transitive occurrences, where, for example, a project and one of its direct dependencies use a specific dependency.

Listing 3. Java-code that creates a subgraph starting from a `ProjectNode` `Vertex`.

```
Vertex<Node, Void> projectNode = ...;

// Select vertices for the subgraph
var depVertices = new ArrayList<...>();
var stack = new ArrayDeque<...>();
stack.add(projectNode);
while (!stack.isEmpty()) {
  var node = stack.pop();
  depVertices.add(node);
  node.getOutgoingEdges()
      .stream()
      .map(Edge::getTarget)
      .forEach(stack::push);
}

// Create a subgraph out of vertices
return depVertices.stream().collect(GraphCollector.toSubGraph());
```

Listing 4. Java-code that creates a human-readable dependency information for a provided graph structure. It results in a list of key values where each key contains the dependency coordinates and each value contains the number of occurrences.

```
Map<...> result = new HashMap<>();
graph.setVertexTraversalStrategy(new
    DepthFirstSearchTraversalStrategy<>(graph));

// Iterate the graph and create a dependency map
graph.traverseVertices(v -> {
  if (v.getElement() instanceof DependencyNode)
    result.put(
        v.getElement().getKey(),
        v.getIncomingEdges()
          .stream()
          .mapToDouble(Edge::getWeight)
          .sum()
    );
});

return "Dependency;Count\n"+
  result.entrySet().stream()
    .sorted(Comparator.comparingInt(
       Map.Entry::getValue).reversed())
    .map(e -> e.getKey()
         + ";"
         + e.getValue()
    )
    .collect(Collectors.joining("\n"));
```

To answer the GQ_2 "Are there any conflicting dependencies according to the version in the given project?", the verification module (shown in Listing 5) can be used by defining a constraint for an automatic analysis. A randomly chosen project, namely, "antenna pod", showed that there are 65 dependencies that occur in at least two different versions in the whole dependency graph.

Listing 5. Java-code that shows the verification example, which finds dependencies with different versions.

```
class DependencyConstraint implements Constraint<String> {
  public static final ConstraintError DDCE = new ConstraintError("
      DD");

  Set<String> deps = new HashSet<>();

  ConstraintError apply(String dependencyNode, Field field) {
      if (!deps.add(dependencyNode))
          return DDCE;
      return ConstraintError.NoError;
  }
}

var pr = new PropertyRestrictor(false);
pr.addFieldConstraint(DependencyNode.class, "artifactQualifier",
    new DependencyConstraint());

var pv = new GraphPropertyVerificator<Node, String>();
pv.setRestrictor(pr);

var stat = pv.getInvalidProperties(graph).createStatistic();
return stat
  .getConstraintViolators(DDCE)
  .keySet();
```

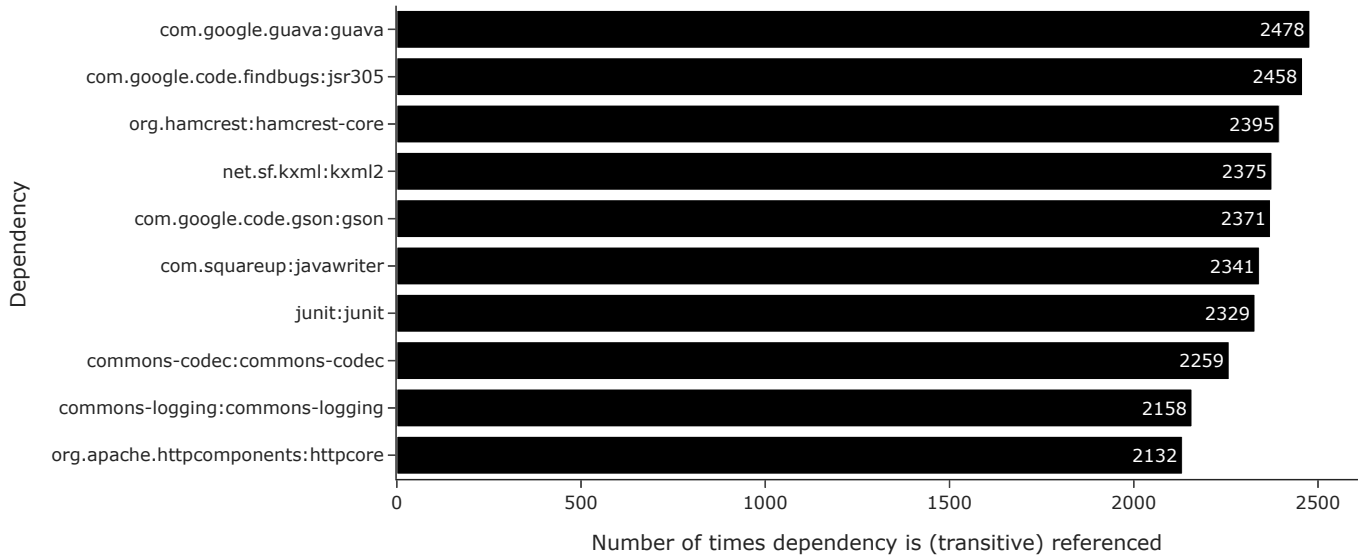

Figure 8. The top ten transitive non-platform dependencies that were used inside the crawled projects with the accumulated number of occurrences.

5. Discussion

Within the example scenario, we have shown one sample use case for the proposed Graph Transformation Framework. This example is based on an analysis of around 15,000 nodes. Compared to other examples, such as the one presented by Cauteruccio et al. [1] with 272,062 nodes or our previous publication on verifications using graph databases [30] with 17,000,000 nodes, this is a relatively small number. While the framework is based on loose interfaces that allow us to use, for example, any persistence layer, the available implementations currently only support in-memory persistence. This leads to limitations regarding the size of the analyzed graph. However, due to the presented architecture, it is possible to extend the framework by additional implementations allowing

alternative persistence strategies such as file or database persistence, which in turn allow for overcoming this limitation.

Additionally, the framework is based on the graph definition $G = (V, E)$, with the edge definition as functions $s, t : E \to V$. Sticking to this definition, our framework is only intended for regular graphs, where edges connect two vertices. Similarly, the definition of hypergraphs with edges that can connect more than two vertices or multi-edge graphs with an arbitrary number of edges between the same two nodes are not possible.

One of the key advantages of our framework is its implementation of all graph concepts through strictly defined interfaces, combined with loose composition between individual components. This design enables the easy extension of the framework with new graph concepts and algorithms without the need to modify the original code elements.

This design approach also promotes modularity and flexibility in the framework, as it enables users to easily swap out individual components without disrupting the entire system. Furthermore, the use of well-defined interfaces enhances the clarity and readability of the codebase, making it easier for developers to understand and work with the framework and to interweave it with other frameworks as shown on the example of GraphViz.

By leveraging this design approach, our framework provides a powerful tool for graph analysis and algorithm development that can adapt to evolving needs and requirements. Users can easily incorporate new graph concepts and algorithms as they become available, allowing them to stay at the forefront of research and innovation in the field.

6. Conclusions

In this paper, we presented the Graph Transformation Framework, a framework for the definition and subsequent transformation of graph and network structures. We further shared insights on GTF's architecture and how a graph structure consisting of vertices and edges can be easily defined using the provided domain specific language. We see GTF not as a supplement to existing approaches to graph analysis, but rather as a complement to well established tools and workflows. We further present an example application of GTF to analyze library dependencies in Android open source applications. For future work, we seek to further improve GTF by including an additional domain specific language, which would allow for defining queries on a graph structure more efficiently. GTF is actively developed and maintained, is publicly available, and can be obtained from [27]. The documentation is hosted via GitHub pages (https://github.aist.science/GTF/ (accessed on 3 April 2023)).

Author Contributions: Conceptualization, A.H.S., C.P. and A.P.; methodology, A.H.S., C.P. and A.P.; software, A.H.S., C.P. and A.P.; validation, A.H.S., C.P. and A.P.; formal analysis, A.H.S.; investigation, A.H.S.; resources, C.P. and A.P.; data curation, C.P. and A.P.; writing—original draft preparation, A.H.S., C.P. and A.P.; writing—review and editing, C.P.; visualization, C.P. and A.P.; supervision, A.H.S.; project administration, A.H.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The *Graph Transformation Framework* is available on GitHub (https://github.com/FHOOEAIST/GTF) (accessed on 3 April 2023). The replication package of this publication is hosted on Zenodo (https://zenodo.org/record/6077478) (accessed on 3 April 2023).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Cauteruccio, F.; Corradini, E.; Terracina, G.; Ursino, D.; Virgili, L. Extraction and analysis of text patterns from NSFW adult content in Reddit. *Data Knowl. Eng.* **2022**, *138*, 101979. [CrossRef]
2. Hashemi, H.; Azmoodeh, A.; Hamzeh, A.; Hashemi, S. Graph embedding as a new approach for unknown malware detection. *J. Comput. Virol. Hacking Tech.* **2016**, *13*, 153–166. [CrossRef]
3. Petriu, D.C.; Wang, X. Deriving Software Performance Models from Architectural Patterns by Graph Transformations. In *Theory and Application of Graph Transformations, Proceedings of the 6th International Workshop, TAGT'98, Paderborn, Germany, 16–20 November 1998*; Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2000; pp. 475–488.
4. Fleder, M.; Kester, M.S.; Pillai, S. Bitcoin Transaction Graph Analysis. *arXiv* **2015**. arXiv:1502.01657. [CrossRef].
5. Kleinberg, J.M.; Kumar, R.; Raghavan, P.; Rajagopalan, S.; Tomkins, A.S. The web as a graph: Measurements, models, and methods. In Proceedings of the Computing and Combinatorics: 5th Annual International Conference, COCOON'99, Tokyo, Japan, 26–28 July 1999; Proceedings 5; Springer: Berlin/Heidelberg, Germany, 1999; pp. 1–17.
6. Roy, A.; Kumbhar, F.H.; Dhillon, H.S.; Saxena, N.; Shin, S.Y.; Singh, S. Efficient monitoring and contact tracing for COVID-19: A smart IoT-based framework. *IEEE Internet Things Mag.* **2020**, *3*, 17–23. [CrossRef]
7. Theocharidis, A.; Van Dongen, S.; Enright, A.J.; Freeman, T.C. Network visualization and analysis of gene expression data using BioLayout Express 3D. *Nat. Protoc.* **2009**, *4*, 1535–1550. [CrossRef] [PubMed]
8. Jo, S.; Park, B.; Lee, S.; Kim, J. OLGAVis: On-Line Graph Analysis and Visualization for Bibliographic Information Network. *Appl. Sci.* **2021**, *11*, 3862. [CrossRef]
9. Urena, R.; Chiclana, F.; Herrera-Viedma, E. DeciTrustNET: A graph based trust and reputation framework for social networks. *Inf. Fusion* **2020**, *61*, 101–112. [CrossRef]
10. Michail, D.; Kinable, J.; Naveh, B.; Sichi, J.V. JGraphT—A Java Library for Graph Data Structures and Algorithms. *ACM Trans. Math. Softw.* **2020**, *46*, 16. [CrossRef]
11. Bejeck, B. *Getting Started with Google Guava*; Packt Publishing Ltd.: Birmingham, UK, 2013.
12. O'Madadhain, J.; Fisher, D.; White, S.; Boey, Y. *The Jung (Java Universal Network/Graph) Framework*; University of California: Irvine, CA, USA, 2003.
13. Tamassia, R. *Handbook of Graph Drawing and Visualization*, 1st ed.; Chapman & Hall/CRC: Boca Raton, FL, USA, 2016.
14. Hagberg, A.; Swart, P.; Chult, D. Exploring Network Structure, Dynamics, and Function Using NetworkX. In Proceedings of the 7th Python in Science Conference, Pasadena, CA, USA, 21 August 2008.
15. Staudt, C.; Sazonovs, A.; Meyerhenke, H. NetworKit: An Interactive Tool Suite for High-Performance Network Analysis. *arXiv* **2014**, arXiv:1403.3005.
16. Mehlhorn, K.; Näher, S. *LEDA—A Platform for Combinatorial and Geometric Computing*; Cambridge University Press: Cambridge, UK, 1999; Volume 38. [CrossRef]
17. Siek, J.; Lumsdaine, A.; Lee, L.Q. *The Boost Graph Library: User Guide and Reference Manual*; Addison-Wesley: Boston, MA, USA, 2002.
18. Leskovec, J.; Sosic, R. SNAP: A General Purpose Network Analysis and Graph Mining Library. *arXiv* **2016**, arXiv:1606.07550.
19. Csardi, G.; Nepusz, T. The igraph software package for complex network research. *Int. J. Complex Syst.* **2006**, *1695*, 1–9.
20. Shun, J.; Blelloch, G.E. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Shenzhen, China, 23–27 February 2013; Volume 48, pp. 135–146. [CrossRef]
21. Wang, Y.; Davidson, A.; Pan, Y.; Wu, Y.; Riffel, A.; Owens, J.D. Gunrock. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Barcelona, Spain, 12–16 March 2016. [CrossRef]
22. Ellson, J.; Gansner, E.; Koutsofios, L.; North, S.C.; Woodhull, G. Graphviz—Open Source Graph Drawing Tools. In Proceedings of the Graph Drawing, Irvine, CA, USA, 26–28 August 2002; Mutzel, P., Jünger, M., Leipert, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2002; pp. 483–484.
23. Bastian, M.; Heymann, S.; Jacomy, M. Gephi: An Open Source Software for Exploring and Manipulating Networks. In Proceedings of the International AAAI Conference on Web and Social Media, San Jose, CA, USA, 17–20 May 2009; Volume 3.
24. Shannon, P.; Markiel, A.; Ozier, O.; Baliga, N.S.; Wang, J.T.; Ramage, D.; Amin, N.; Schwikowski, B.; Ideker, T. Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Res.* **2003**, *13*, 2498–2504. [CrossRef] [PubMed]
25. Gansner, E.R.; North, S.C. An Open Graph Visualization System and Its Applications to Software Engineering. *Softw. Pract. Exper.* **2000**, *30*, 1203–1233. [CrossRef]
26. Freeman, T.C.; Horsewell, S.; Patir, A.; Harling-Lee, J.; Regan, T.; Shih, B.B.; Prendergast, J.; Hume, D.A.; Angus, T. Graphia: A platform for the graph-based visualisation and analysis of high dimensional data. *PLoS Comput. Biol.* **2022**, *18*, e1010310. [CrossRef] [PubMed]
27. Schuler, A.; Pointner, A.; Praschl, C. Graph Transformation Framework, 2022. If You Use This Software, Please Cite It as Below. Available online: https://zenodo.org/record/7185323#.ZC4-nPkzbIU (accessed date on 3 April 2023) . [CrossRef]
28. Calegari, D.; Szasz, N. Verification of model transformations: A survey of the state-of-the-art. *Electron. Notes Theor. Comput. Sci.* **2013**, *292*, 5–25. [CrossRef]

29. Schuler, A.; Pointner, A.; Praschl, C. Graph Transformation Framework Paper Example. 2023. Available online: https://zenodo.org/record/7752904#.ZC4_NPlBzIU (accessed date on 3 April 2023). [CrossRef]
30. Praschl, C.; Pointner, A.; Krauss, O.; Helm, E.; Schuler, A. Model Verification in Graph Databases and its Application in Neo4j. In Proceedings of the European Modeling & Simulation Symposium, EMSS, Rome, Italy, 19–21 September 2022. [CrossRef]