University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master's thesis

Framework for dependency analysis of software artifacts

Plzeň 2023 Miroslav Krýsl

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Miroslav KRÝSL**
Osobní číslo:	**A20N0092P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Framework pro analýzu závislostí softwarových artefaktů**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s principy komponentově orientovaných softwarových systémů, způsoby reprezentace a analýzy grafových dat.
2. Seznamte se s metodami a nástroji používanými pro statickou analýzu komponent a jejich závislostí na Katedře informatiky a výpočetní techniky FAV ZČU.
3. Analyzujte možnosti rozšíření těchto nástrojů s důrazem na podporu zpracování rozsáhlých datových sad a podporu vývoje experimentů v různých programovacích jazycích.
4. Navrhněte a implementujte sadu rozšíření a demonstrujte jejich užití implementací experimentu založeného na některé z metod z bodu 2.
5. Ověřte funkčnost a kvalitu vytvořených nástrojů, kriticky zhodnoťte jejich použití a výsledky provedených analýz.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná/elektronická**
Jazyk zpracování: **Angličtina**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Ing. Jakub Daněk**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **9. září 2022**
Termín odevzdání diplomové práce: **18. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 11. října 2022

Declaration

I hereby declare that this master's thesis is completely my own work and that I used only the cited sources.

Plzeň, 22nd June 2023

Miroslav Krýsl

Abstract

This thesis aims to familiarize with the component-based systems, graph data representation and analysis and with existing methods and tools for static analysis of component-based systems which are being developed at the Department of Computer Science at the University of West Bohemia in Pilsen, Czech Republic. Based on the findings, the result of this thesis is a framework design and implementation with emphasis on support for development in multiple programming languages and on the ability to process large datasets. The created framework then can serve to support the research of the component-based systems. The author of this thesis proposes generalization and extension of the framework for software artifacts dependency analysis which has been created as a part of M. Hotovec's master's thesis. The framework data storage model has also been analyzed with emphasis on graph databases. ArangoDB database has been eventually chosen as a storage solution and a core library in Java has been implemented to allow the development of framework tools. The resulting design decisions allows the framework to be used in broader range of use cases such as components compatibility extraction and verification, which has been demonstrated by replicating this functionality in a framework tool created as a part of this thesis.

Abstrakt

Cílem této práce je seznámit se s komponentově orientovanými systémy, s reprezentací a analýzou grafových dat a s existujícími metodami a nástroji pro statickou analýzu komponentově orientovaných systémů, které jsou vyvíjeny na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Na základě zjištěných poznatků je výsledkem této práce návrh a implementace frameworku s důrazem na podporu vývoje ve více programovacích jazycích a na schopnost zpracovávat velké datové sady. Vytvořený framework pak může sloužit pro podporu výzkumu komponentově orientovaných systémů. Autor této práce navrhuje zobecnění a rozšíření frameworku pro analýzu závislostí softwarových artefaktů, který byl vytvořen v rámci diplomové práce M. Hotovce. Model ukládání dat frameworku byl rovněž analyzován s důrazem na grafové databáze. Jako řešení pro ukládání dat byla nakonec zvolena databáze ArangoDB. Dále byla implementována knihovna s jádrem frameworku v jazyce Java, které umožňuje vývoj nástrojů frameworku. Výsledná návrhová rozhodnutí umožňují využití frameworku v širší škále případů použití, jako je například extrakce a verifikace kompatibility komponent, což bylo demonstrováno replikací této funkcionality v nástroji frameworku vytvořeném v rámci této práce.

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor, Ing. Jakub Daňek, for his invaluable guidance and support throughout my master's program and elaboration of this thesis.

I am grateful to my colleagues and the management of Yoso Czech s.r.o for support and for providing me with a pleasant working environment to work on this thesis.

I would also like to thank my friends, my family and especially my fiancée for their love and support during this process. Without them, this journey would not have been possible.

Contents

1 Itroduction

Component-based development emphasizes the creation of modular software using reusable components. Each component functions independently and communicates with others through defined interfaces. This design approach enables the separation of functionality into smaller units, enhancing code reusability. Components can be integrated and shared across multiple different projects, promoting modularity and reducing maintenance costs.

Essential part of a component's development lifecycle are modifications of its functionality complemented by its interface modifications. Incorrect usage of incompatible components versions could lead to a break-down of the whole system. This leads to a need for a robust instrument for components versioning and tracking of their mutual compatibility to ensure their smooth composition and integration.

Researchers at the Department of Computer Science at the University of West Bohemia in Pilsen, Czech Republic, are working on tools for automatic verification of component compatibility specifically targeting the OSGi component framework. They introduced the project of experimental Component Repository supporting Compatibility Evaluation (CRCE) to provide means of compatibility information extraction and storage for later components compatibility verification.

M. Hotovec has presented foundations for a modular framework intended for dependency analysis of software artifacts in his thesis[1]. The main idea is to split rather complex and diverse features of a single monolithic system into multiple individual tools, each focusing on a simple task.

This thesis aims to familiarize with the component-based systems, graph data representation and analysis and with the mentioned existing methods and tools for static analysis of component-based systems. Based on the findings, the result of this thesis will be a framework design and implementation with emphasis on support for development in multiple programming languages and on the ability to process large datasets. The created framework then can serve to support the research of the component-based systems. The author of this thesis proposes generalization and extension of the modular framework created as a part of M. Hotovec's master's thesis. The aim is to provide support for broader and more generic use cases including not only software artifacts dependency analysis, but such use cases as the compatibility extraction and verification introduced by CRCE.

The first two chapters of this thesis are devoted to the introduction into

component-based software systems (chapter 2) and graph data representation and analysis (chapter 3). The following chapter 4 summarizes the tools and methods which are being researched and developed at the Department of Computer Science at the University of West Bohemia. The chapter 5 deals with the framework design analysis while proposing generalizations and changes. The next chapter 6 deals with the framework storage model analysis to be able to support large datasets processing. The two penultimate chapters present the implementation part of this thesis – the implementation of a core Java library and two framework tools (chapter 7), and how the framework design and implementation have been verified (chapter 8). The final thesis conclusion is in the last chapter 9.

2 Component-based software systems

This chapter is based on the work of F. Bachman et al. [2].

A component-based system is a software system that is designed and built using a component-based design strategy including products and concepts supporting this strategy. The system is composed of individual software components that are executed on physical or logical devices.

Components in a component-based system implement one or more interfaces and satisfy contracts. These contracts ensure that independently developed components follow certain rules for predictable interactions and can be deployed into standardized environments.

The motivation behind a component-based system lies in the following factors.

1. Predictability is achieved through uniform design rules enforced across components, allowing for the prediction of system properties such as scalability and security.

2. Components provide flexibility, as they can be independently developed and deployed without unexpected interactions.

3. Component model facilitate the deployment of independently developed components into a common standardized environment, ensuring predictable interactions and avoiding resource contention.

4. Reusing existing components and making key architectural decisions upfront (they are part of the component model and framework) allows the system's time-to-market reduction.

2.1 Software component

The concept of a software component can vary depending on different perspectives. It can be seen as a reusable entity, a commercial off-the-shelf (COTS) product, a unit of project management, or a design abstraction.

In the referenced work [2], the component is defined as follows.

A component is:

- an opaque implementation of functionality

- subject to third-party composition
- conforming with a component model

Components are capable of being composed by third parties thanks to their opaqueness, which allows for abstraction and information hiding. They can be used independently of the tools or knowledge possessed by the component provider. Components must also adhere to specific rules that define how components can interact and comply with architectural constraints.

2.2 Interface

Interfaces are crucial for integrating components into a common environment. They provide a mechanism for controlling dependencies between components. An interface acts as a specification of the properties that clients can rely on when interacting with a component, while other properties remain hidden.

On of the classification schemes categorizes properties into four kinds.

syntactic – conventional APIs, interface specifications written in programming languages. These are the signatures of services the components provide – the types and order of arguments to the service and the manner of returning results and their types.

behavioral – define the outcome of operations, for example, by defining pre-conditions and corresponding post-conditions.

synchronization – deal with the components synchronization aspects distributed and concurrent systems.

quality of service – deal with the qualitative attributes, such as response delay, precision and portability.

While the methods for specifying syntactic properties are commonly available in the vast majority of programming languages, the ways of expressing the other three kinds of properties remain limited.

2.3 Contract

Contracts establish mutual liabilities and agreements between components and their clients – the dependencies and expectations that both can rely

upon for effective interaction. Contracts can range from simple, single-component interactions to complex patterns involving multiple components and multiple invocations. They can also specify both functional and behavioral properties of components and their interactions.

Contracts can be seen from two perspectives: component contracts and interaction contracts.

Component contracts – focus on individual components and describe the services they provide along with any additional properties and conditions necessary for proper functioning. These contracts go further than the interfaces, as they involve implementation-specific details and requirements on the environment and clients.

Interaction contracts – outline the patterns of interaction among multiple components in the system.

Overall, contracts play a crucial role as they formalize the dependencies, requirements, and interaction patterns between components, thus ensuring effective communication and reliable system behavior.

2.4 Component model

The purpose of component models is to enable uniform composition, ensure appropriate quality attributes, and facilitate the deployment of components and applications.

Component models aim to enable interaction between components by standardizing a way to specify each component's requirements and capabilities. They also contribute to achieving desired quality attributes in a system by standardizing the types of components used and their patterns of interaction (the architectural style).

The deployment process is addressed by providing standard compose-time and runtime infrastructure by concrete component model implementation. This simplifies the deployment lifecycle starting from the development, through composition environment, and to the final production environment.

Component models impose standards regarding components in the following areas:

1. **Component types:** a component's type is defined by the interfaces it implements.

2. **Interaction schemes:** specification how components are located, their lifecycle, topology constraints, the means of communication used and, for example, how security and transactions are achieved.

3. **Resource binding:** management of available resources and how their provision to components.

An example of a component model is OSGi.

2.5 Component framework

Component frameworks generally provide coordination mechanisms that enforce a specific model of component interactions. They manage the component lifecycle, controlling resources and facilitating communication between components.

An example of a component framework is Felix which is an implementation of OSGi component model.

2.6 Composition

In component-based development, systems are assembled through composition rather than integration. Composition is a process of combining components to enable interactions defined by the component model. It is essential for a component model to define contracts for compositions.

The major classes of compositions are:

Component deployment within framework – components must be deployed into frameworks prior to be composed or executed. The composition contract describes the interface that components must implement in order to allow the framework to manage their lifecycle and resources.

Framework deployment within framework – frameworks may be deployed into other, higher order frameworks. It has an analogical composition contract to component deployment.

Simple composition – components deployed within the same framework can be composed. The composition contract specifies the application-specific functionality.

Heterogeneous Composition – components be composed also within multiple frameworks. In such case, bridging contracts between the framework environments are needed in addition to composition contracts.

Framework extension – frameworks itself may be treated as components and be composed (extended) with other components. These components are often called "plug-ins". The plu-gin contract specifies the means to extend the framework behavior.

Component assembly – components can be sub-assembled (nested), thus the composition contract involves only the top-level components.

3 Graph data representation and analysis

Graphs serve as powerful models for representing and analyzing complex relationships and structures in various domains, ranging from social networks to transportation systems. The discipline dealing with graphs is called graph theory.

One of the significant domains in the context of software engineering is the analysis of software components and their dependencies. Software components and their mutual relations naturally form graph-like structures. Graph algorithms and visualization tools can serve as a powerful means to perform a comprehensive analysis for this domain.

This chapter is devoted to the ways of representing, managing and analyzing graph data. In the first section (3.1), multiple fundamental terms of graph theory are outlined, as well as basic data structures for representing graphs. The section that follows deals with the graph data storage and management (3.2).

3.1 Graph theory fundamental terms

This section introduces fundamental terms from the graph theory. Information in this section is based on the work of Md. S. Rahman [3] (where not stated otherwise).

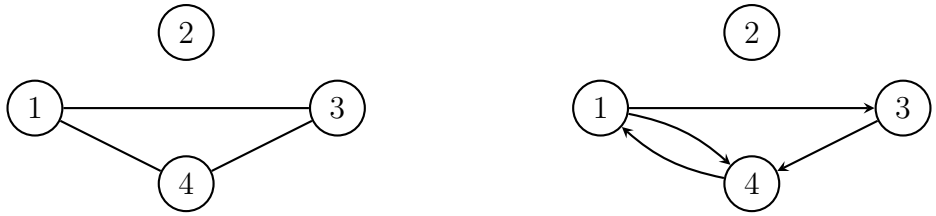
3.1.1 Graph

A graph G is a tuple consisting of a finite set V of vertices (they can be called nodes in different terminology) and a finite set E of edges. Each edge uv (or e_k) is a pair of vertices u and v where both u and v belong to the set of graph vertices. These two vertices associated with an edge are called the *endpoints* (or *end-vertices*) of the edge. The set of vertices of a graph G is often denoted as $V(G)$ and the set of edges as $E(G)$. The number of vertices $|V(G)|$ is denoted as n, number of edges $|E(G)|$ as m.

Graphical depiction of a graph is usually realized by representing each vertex by a point or a small circle and each edge by a line segment or a curve between its two end-vertices.

When the pair of edge vertices is unordered, the graph is called *undirected*. The edge is in this case often denoted as (u, v). An example of an undirected graph is on the figure 3.1a.

When the pair of edge vertices u and v is ordered, the graph is called *directed*. The edge is in this case often denoted as $\{u, v\}$. The vertex u is called the *tail* (or *start*, *source*) of the edge uv and the v is called the *head* (or *end*, *target*) of the edge uv. On a graphical representation, the edge direction is usually depicted using an arrow pointing from the start vertex to the end vertex. An example of a directed graph is on the figure 3.1b.



(a) Undirected graph. (b) Directed graph.

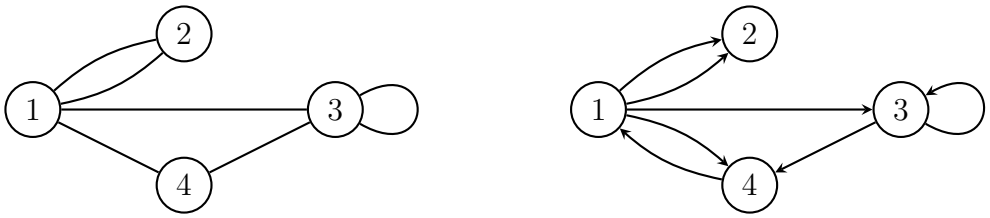
Figure 3.1: Examples of simple graphs.

A *loop* is an edge whose end-vertices are the same (edges (v_3, v_3) and $\{v_3, v_3\}$ on the figures 3.2b and 3.2a respectively).

Multiple edges are edges with the same pair of end-vertices (edges $\{v_1, v_2\}$ and (v_1, v_2) on the figures 3.2b and 3.2a respectively).

A graph is called *simple graph* if it does not have any loop or multiple edge, otherwise, it is called a *multigraph*.

Examples of simple graphs are on the figure 3.1, multigraphs on the figure 3.2.



(a) Undirected graph. (b) Directed graph.

Figure 3.2: Examples of multigraphs.

A graph D is called the *underlying graph* of graph D if D is constructed from D such that an edge (u, v) belongs to D if and only if $\{u, v\}$ or $\{v, u\}$ or both belong to D.

A directed graph D is called *corresponding* to G if D is constructed from G such that an edge $\{u, v\}$ belong to D if and only if (u, v) (or alternatively written (v, u) belongs to G.

A directed graph obtained from an undirected graph G by replacing each edge (u, v) belonging to G by a directed edge uv or vu, but not both is called an *orientation* of G.

3.1.2 Adjacency

For every edge uv in graph G, the two vertices u and v are said to be *adjacent* in G and the vertex v is called a *neighbor* of vertex u. The edge uv is said to be *incident* to the vertices u and v. In undirected graph, the adjacency applies both directions, thus the vertex u is also a neighbor of v in G.

For example, on the undirected graph on the figure 3.2a, adjacent vertices of the vertex v_4 are v_1 and v_3. On the directed graph on the figure 3.2b, adjacent vertex of the vertex v_4 is only v_1.

3.1.3 Vertex degree

The degree of a vertex v in an undirected graph G, is the number of edges incident to v. Each loop is counted twice for a vertex v. The degree of the vertex v is denoted as $\deg(v)$.

Regarding the directed graphs, *indegree* of a vertex v in a directed graph is the number of edges terminating at v, and the *outdegree* of v is the number of edges leaving v. The degree of v is then the sum of its *indegree* and *outdegree*.

For example, on the undirected graph on the figure 3.2a, the value for vertex v_3 is $\deg(v_3) = 4$. On the directed graph on the figure 3.2b, the values for vertex v_3 are $\text{indeg}(v_3) = 2$, $\text{outdeg}(v_3) = 2$ and $\deg(v_3) = 4$.

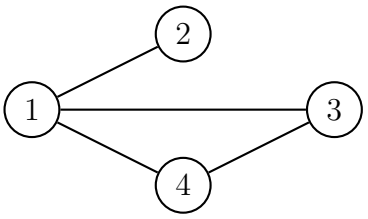
3.1.4 Subgraph

A subgraph of a graph G is a graph G', such that the set of vertices $V(G')$ is a subset of $V(G)$ and set of edges $E(G')$ is a subset of $E(G)$.

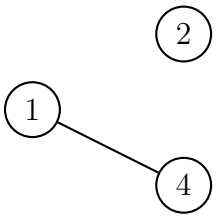
For example, on the figure 3.3, the graph G' is a subgraph of G.

3.1.5 Walk, trail, path and cycle

A *walk* in a graph G is a nonempty sequence $W = v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k$ consisting of alternating vertices and edges of graph G where v_i and v_{i+1} are end-vertices of the edge e_i. The vertices v_0 and v_k are called the *end-vertices*



(a) A graph G.



(b) A graph G' – subgraph of G.

Figure 3.3: Example of graph and its subgraph.

of W, the others are called *internal vertices*. The number k is the length of the walk W, which corresponds to the number of edges on the walk.

A *trail* of a graph G is a walk in G with no repeated edges. A trail with the end-vertices being the same is called a *circuit*.

A *path* of a graph G is a walk in G with no repeated vertex, except end-vertices. A path with the end-vertices being the same is called a *cycle*.

3.1.6 Connectivity

A vertex u is said to be *reachable* from a vertex v, if there is a path from v to u.

An undirected graph G is said to be *connected* if every pair of vertices is reachable from each other. A directed graph G is said to be *weakly connected* or *connected* if its underlying graph is connected. In addition, a directed graph, where every pair of vertices is reachable from each other, is called *strongly connected*. Otherwise, both undirected and directed, are called *disconnected*.

3.1.7 Graph representation

The first – graphical – representation has been outlined in the previous sections. Although convenient for visualization, the graphical representation is not suitable for storage and later processing in a computer.

There are generally three main approaches how to represent a graph in a computer. These are adjacency matrix, incidence matrix and adjacency list.

Adjacency matrix

The adjacency matrix $A(G)$ of graph G is a matrix $A(G) = [a_{ij}]$ with dimensions $n \times n$ where a_{ij} is the number of edges between the two vertices v_i and v_j. In other words, every row and every column corresponds to a single

vertex. The corresponding cell then contains the number of edges between these two vertices. An example of a graph and its corresponding adjacency matrix is on the figure 3.4.

Testing for vertices adjacency requires constant time for the adjacency matrix. Scanning for neighbors has however $O(n)$ time complexity.

This representation is suitable for representing dense graphs, i.e., graphs with a relatively large number of edges between most of the vertices. For sparse graphs, it wastes a lot of memory due to most of the values in a matrix being zero.

Incidence matrix

The incidence matrix $I(G)$ of graph G is a matrix $I(G) = [a_{ij}]$ with dimensions $n \times m$. For undirected graphs, a_{ij} is 1 if the edge e_j is incident to the vertex v_i and 0 otherwise. For directed graphs, a_{ij} is -1 if the edge e_j leaves the vertex v_i, 1 if the edge enters the vertex, and 0 otherwise. An example of a graph and its corresponding incidence matrix is on the figure 3.4.

In other words, every row corresponds to a single vertex and every column to a single edge. The corresponding cell contains the information whether this vertex is an end-vertex of the edge, or alternatively for directed graphs, whether the edge leaves or enters the vertex.

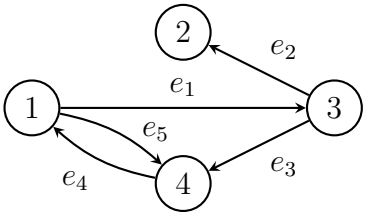
This representation is suitable for use cases, in which it is important to know whether a vertex is incident to an edge or not in constant time. Otherwise, it is not economical, especially for a graph which contains a relatively high number of edges compared to n and for use cases, where the performance of neighbors scanning is also important, since the time complexity is similarly to the adjacency matrix also O(n).

Adjacency list

The adjacency list Adj(G) of a graph G is an array of n lists. There is a list for every vertex v of graph G and each of these lists contains a record for each neighbor of the corresponding v. An example of a graph and its corresponding adjacency list is on the figure 3.4.

Both, checking adjacency of vertices and scanning for neighbors, have time complexity $O(\deg(v))$.

This representation is much more economical than the adjacency matrix and the incidence matrix, particularly if the number of graph edges is relatively low.



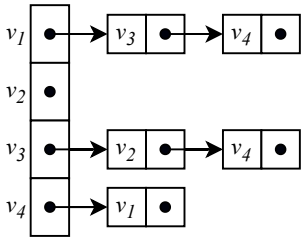
(a) A graph G.

$$A(G) = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

(b) Adjacency matrix of the graf G.

$$I(G) = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} -1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 1 \end{bmatrix} \end{array}$$

(c) Incidence matrix of the graf G.



(d) Adjacency list of the graf G.

Figure 3.4: Examples of adjacency matrix, incidence matrix and adjacency list.

Other graph representations

The three introduced graph representations are the basis for other more advanced ones. The matrix representations can be efficiently used for mathematical analysis of the graphs, such as determining whether the graph is connected in case of adjacency matrix, and counting the number of spanning trees in case of incidence matrix [4]. On the other hand, an adjacency list and its modifications provide good solutions for storing large volumes of graph-structured data [5].

The sole definition of vertices and their interconnecting edges is not sufficient for most of the use cases. The vertices and edges are therefore often accompanied by additional information [6]. This topic is further elaborated in the following section 3.2.

3.2 Graph data management

Graph data management involves the technologies for storing, processing, and analyzing large volumes of graph data. These technologies have seen significant advancements in recent years. There are two main areas of focus within graph data management. The first area is graph database models, which involves the design of graph structure with emphasis on graph-oriented

operations and integrity constraints to ensure data consistency. The second area is graph data management systems which include graph databases and graph-processing frameworks. Graph databases are designed for persistent data storage, while graph-processing frameworks focus on batch processing and analysis of large graphs.[6]

The next three sections are devoted to the basic overview of graph data models (section 3.2.1), methods for graph data analysis (section 3.2.2) and graph databases (section 3.2.3). NoSQL graph databases and their implementations are also described more in detail in the sections 6.3.3 and 6.4.

3.2.1 Graph data models

A graph database model represents data structures as graphs, allowing data manipulation through graph-oriented operations. Data manipulation involves graph transformations and operations based on graph features such as paths, neighborhoods, and connectivity. Graph query languages play a significant role in defining the flavor of a database model, while integrity constraints ensure data consistency.[6]

There are generally the following basic models.[6]

Labeled graph – a directed graph where vertices and edges are assigned labels from a vocabulary. Vertices are labeled with types, representing a domain of values, and edges have labels representing relations between these types.

Hypergraph – an extension of a graph where edges are generalized as hyperedges and can connect more than two vertices. It allows for complex object definitions, functional dependencies, and multiple structural inheritance.

Hypernode – a directed graph where vertices can themselves be graphs, allowing nested graphs and encapsulation of information. It represents both simple and complex objects, mappings, records, and hierarchies.

Property graph – a directed graph where vertices and edges can have labels and properties. It allows representing metadata bound to the vertices and edges using key/value pairs.

RDF model – The Resource Description Framework (RDF) [7] was originally designed for a metadata representation. The ability to interconnect

resources gives an advantage of an extensible way using graph-like structure for data. The RDF data model structures can be viewed as triples representing relationships between subjects, predicates, and objects, or as a general representation of a graph where edges are also considered vertices.

3.2.2 Graph data analysis

Graph data manipulation and analysis can be expressed by graph operations and graph transformations based on features like neighborhood, paths and graph patterns. Another approach is to enclose all operations into a graph query language. An examples of such languages are SPARQL – a standard query language for RDF model, Cypher – used by Neo4j graph database, AQL – used by ArangoDB graph database and GraphQL – for declarative data fetching and manipulation.[6]

The queries can be classified into the following categories.[6]

Adjacency queries – determine vertex/edge relationships in a graph. They are used to find neighbors of a vertex, check adjacency between vertices or to check incidence of an edge and a vertex.

Pattern matching queries – involve finding subgraphs that match a given graph pattern. It can involve complex patterns, semantic matching or approximate matching. These types of queries are important in various domains like pattern recognition and social network analysis.

Reachability queries – determine whether a path connects two vertices in a graph database. They serve as the foundation for other graph queries, such as calculating the shortest-path distance in road networks.

Analytical queries – analytical queries measure and aggregate quantitative and topological graph features. Examples include calculating vertices degrees, path lengths, and applying complex algorithms for graph analysis such as PageRank.

3.2.3 Graph databases

Graph databases are specialized systems for managing graph-like data. They have a long history, but recent technological advancements have made practical systems possible. Graph databases include major components, such as

storage engines, database languages, indexes, query optimizers and transaction support.

They can be classified into two categories – native or nonnative – depending on their internal implementation. Native graph databases utilize data structures and indexes built specifically for storing graph-like data, while nonnative graph databases use other database structures supplemented by interfaces for graph operation queries. Examples of native graph databases are Neo4j and AllegroGraph, examples of nonnative are ArangoDB and OrientDB.[6]

Although the native graph databases may perform better for graph operations, it is not always a rule. Example could be benchmark comparison of ArangoDB and Neo4j where ArangoDB outperformed Neo4j in both memory comsuption and processing time.[8]

Graph databases are also described more in detail in the section 6.3.3 and a comparison of three of them is in section 6.4.

Special case of databases which can be used for managing graph data is RDF databases. Data in RDF is a directed graph composed of triple statements, where each statement is a triple object-predicate-subject represented by 1) a subject (a start vertice), 2) a predicate connecting the subject with the object (an edge), and 3) object (end vertice), thus forming a graph-like structure. SPARQL is used as a standard query language. Examples include Jena, Virtuoso, and Blazegraph.[6]

3.2.4 Graph-processing frameworks

Graph databases are primarily used for efficient storage and retrieval of graph data, while a graph-processing frameworks are designed for performing intensive computations and analysis on graph data, however, the functionality of both can overlap. Graph-processing frameworks utilize in-memory batch processing, distributed and parallel strategies. Examples of such systems are Hadoop and MapReduce which are adapted for graph processing, while Pregel or Apache Giraph are graph-specific platforms.[6]

4 Tools for static analysis of components

There are multiple existing methods and tools used for static analysis of components and their dependencies which are developed at the Department of Computer Science and Engineering Faculty of Applied Sciences, University of West Bohemia. They are described as a part of this work in the following sections.

The tools include **CRCE** – an experimental repository system used for research of component-based systems (section 4.1), **JaCC** – a Java library for extraction and comparison of Java types using bytecode inspection (section 4.2), **OBCC** – a Java library for checking compatibility of OSGi bundles (section 4.3), and the last is M. Hotovec's master's thesis work on modular framework for dependency analysis (section 4.4).

4.1 CRCE

The Component Repository and Compatibility Evaluation (CRCE) is an experimental repository system. It has been designed to support research into component-based and modular systems undertaken by ReliSA research group at the Faculty of Applied Sciences, University of West Bohemia. The source code and information are available at GitHub [9]. The idea behind the CRCE project is described in detail in [10].

CRCE manages various types of resources, including component distribution packages and metadata files that describe component properties and their relations. Additionally, CRCE handles auxiliary information called "results", which are generated by indexing and verification operations. These results include detailed reports of tests performed, configuration properties, logs of simulation runs, and component interface comparison results. The volume of results can be substantial. Furthermore, a single result resource can be related to multiple components and their properties.

The results often originate from external tools invoked by CRCE verification plugins. These plugins include OSGi Bundle Compatibility Checker (OBCC) which is a compatibility verification tool implementing type-based compatibility checks of OSGi bundles and packages. It checks compatibility in terms of interface and method signatures and automatically converts the

results to version numbers that strictly follow the OSGi semantic versioning scheme. The compatibility verification functionality has been implemented into CRCE as a part of the J. Daňek's work in his master's thesis [11].

CRCE offers advantages over other solutions in terms of performance and analysis costs. Key reasons are:

Pre-computed results – CRCE stores and retrieves compatibility results, eliminating the need for repeated compatibility checks. This improves performance by reducing computational overhead.

Metadata-based analysis – CRCE uses metadata storage to provide summarized and detailed component information. This reduces analysis costs by avoiding repeated analysis of components.

Credibility and verification – CRCE's results provide detailed information, enhancing credibility and enabling informed decision-making.

Scalability and resource constraints – CRCE is designed for resource-constrained devices by utilizing pre-computed results.

CRCE includes a web interface for interacting with the repository, browsing its contents, and querying information. It also provides REST API for applications.

4.1.1 Metadata

Another CRCE related article [12] comprehensibly discusses the metadata structures and compatibility verification in CRCE. The metadata structure is based on OSGi Bundle Repository (OBR) requirements and capabilities, allowing the description of various interface features and properties. It follows a generic structure that allows it to represent various components beyond a specific component model. The generic structure may be cumbersome to work with, but it enables CRCE to support different component models without modifying the data structure. It comprises five core elements: Resource, Capability, Requirement, Filter, and Property. A diagram of these metadata elements is on the figure 4.1;

Resource – represents auxiliary information, data, or assets that are managed and associated with a component in the repository. Resource is an additional piece of information created during indexing, verification, and

analysis operations. The data associated with a resource are divided into capabilities, requirements, and properties.

Capability – represents a characteristic or feature of a component that describes its ability to perform a certain function or provide a specific service. A capability itself can be described by its properties, its own requirements and possible child capabilities, thus forming a tree like structure.

Requirement – represents a specification or condition that a component needs – prerequisites that must be fulfilled – in order to function properly or interact with other components. A requirement itself can be described by its possible child requirements, thus also forming a tree like structure.

Property – represents to a characteristic, attribute, or feature associated with an entity.

Attribute – attributes are the actual lowest level bearers of information about entities. They can be textual, numerical, boolean, enumerated values, or even complex data structures. All entities – resources, capabilities, requirements and properties can directly provide information about themselves using attributes.

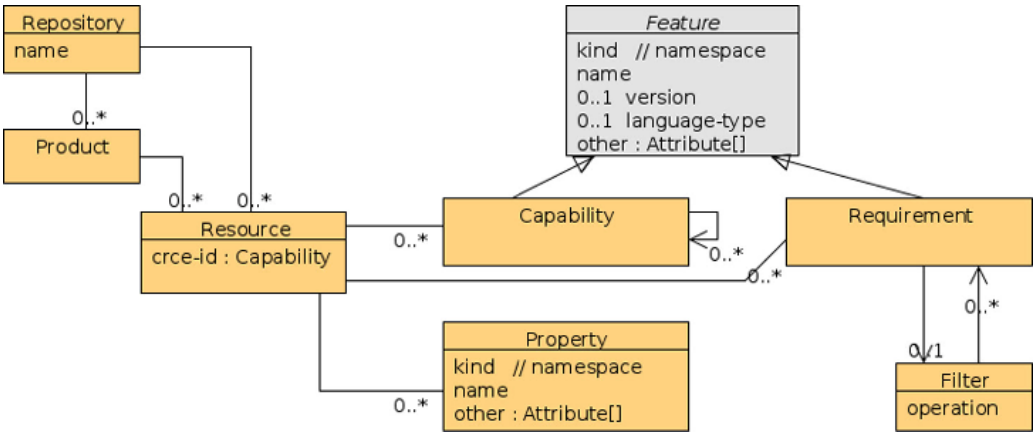


Figure 4.1: Core elements of the CRCE metadata model (taken from [12])

This metadata is accessible internally for CRCE modules by using metadata API or by previously mentioned REST API.

In the context of compatibility verification, the metadata hierarchical structure is used to represent compatibility information at various levels, such as individual features, aggregates (packages or the whole component),

and their relationships. It provides a systematic way to capture and analyze differences between different versions of components.

4.2 JaCC

The Java Class Comparator (JaCC) is a Java library developed at the Department of Computer Science, the University of West Bohemia. Information about the JaCC library is taken from the work [11].

JaCC offers a capability to extract Java type information representation and perform type comparison on the extracted data. This is achieved through Java bytecode inspection, allowing the retrieval of type representation. The process of reconstructing a component's complete public API from its binary form is extensively detailed in [13].

The workflow of JaCC involves a recursive approach, starting from the top level (package) and progressing down to the lowest levels (field types, method arguments, etc.). At the lowest levels, the resulting difference class for each element is obtained through direct comparison. At higher levels, the difference is computed as an aggregation from the element's children.

To address the limitations of the Reflection API, the library presents its own Java type system representation, which closely resembles the classes provided by the `java.lang.reflect` package. This custom representation was introduced to enable compatibility checks, allowing the acquisition of the representation through means other than the classloader, such as bytecode inspection.

4.3 OBCC

The OSGi Bundle Compatibility Checker (OBCC) is also a Java library developed at the Department of Computer Science, the University of West Bohemia. Information about the OBCC library is taken from the work [11].

OBCC offers a capability to check the compatibility of OSGi bundles based on subtype relations. It serves as a foundation for tools focused on automated versioning and safe component updates.

Internally, it uses the JaCC library for loading OSGI bundles, extracting Java type representation from bytecode, and to perform compatibility checks.

4.4 Dependency analysis of software artifacts

In his master's thesis work [1], M. Hotovec proposed and implemented a foundation of a framework for dependency analysis of software artifacts. The two basic ideas of the framework design are:

1. The framework is modular – it consists of multiple tools that operate independently.
2. Framework tools are able to generate datasets, to contribute to existing ones or to infer outcomes based on a dataset state.
3. The dataset data model is standardized for all tools and can be represented by a standardized intermediate format.
4. The framework workflow consists of chaining multiple tools in a row. Each tool contributes to the desired result by modifying the dataset and handing it over to the next tool in the pipeline.
5. The tools are not restricted to using a specific programming language.

4.4.1 Common JSON data model

One of the requirements for the framework is that it must be modular with support for multiple programming languages. The individual framework tools can depend on the results made by the preceding ones. They also must be able to contribute to the common objective by adding their own results to the dataset and pass it to the following tools. This results in the necessity for a common, language-independent data representation which must be followed by all the framework tools.

As a part of the author's work, a common JSON format has been designed for the purpose of intermediate data representation. A diagram of the format's data model is on diagram 4.2, which is taken from the referenced work. The main data objects are described further on in this section.

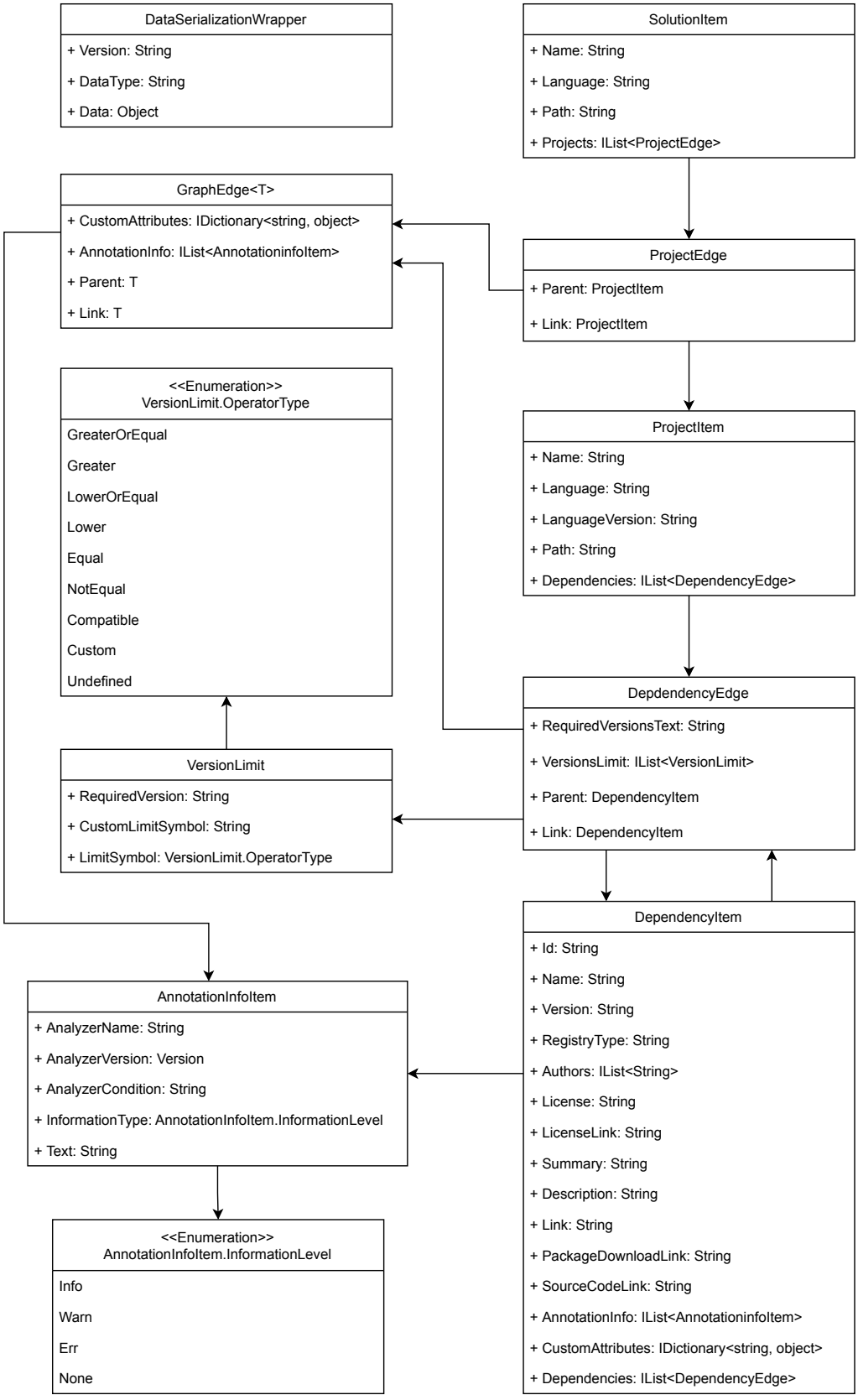


Figure 4.2: Data model of the universal JSON data format (taken from [1]).

Following core types of objects are introduced for the JSON format. All of them have predefined set of attributes, which are listed in the mentioned diagram 4.2.

Data wrapper – the top level wrapper object with field dedicated for the format version information, type of the contained data, and the data itself. (Depicted as *DataSerializationWrapper* in the diagram.) The determination of the contained data type is realized using the attribute *DataType* which is the type's textual representation in the used programming language. This solution makes it difficult to ensure the data format compatibility among multiple programming languages. This issue is addressed further in the section 5.2.

Dependency – a representation of a single software package. (*DependencyItem* in the diagram.)

Dependency edge – a representation of a single dependency relation between two packages or between a project and a package. (*DependencyEdge* in the diagram.)

Project – a representation of a project – a collection of jointly developed software artifacts. (*ProjectItem* in the diagram.)

Project edge – an association of a project with a solution. (*ProjectEdge* in the diagram.)

Solution – a representation of a solution – a collection of jointly developed projects. (*SolutionItem* in the diagram.)

Other data objects like annotations (*AnnotationInfoItem* in the diagram) and versioning limits are also available. An important feature of the data format is a possibility to include non-predefined attributes in a special attribute *CustomAttribute*. This gives freedom for the framework tools to include arbitrary data in the dataset.

An example of a serialized dataset is in the listing 1. The example consists of one dependency with ID `pipgrip` which itself has a single dependency with ID `anytree`. Some fields and sub-dependencies have been omitted for brevity.

The complete description of the format is available in chapter 6 of the referenced thesis.

```
{
  "$id": "1",
  "Version": "0.1.0.0",
  "DataType": "System.Collections.Generic.List`1[Janus.Core.Models.DependencyItem]",
  "Data": [
    {
      "$id": "2",
      "Id": "pipgrip",
      "Name": "pipgrip",
      "Version": "0.1.0",
      "RegistryType": "PyPI",
      "Authors": [
        "ddelange"
      ],
      "License": "BSD-3-Clause",
      "Summary": "Composable command line interface toolkit",
      "Link": "https://pypi.org/project/pipgrip/0.1.0/",
      "AnnotationInfo": [],
      "CustomAttributes": {
        "$id": "3"
      },
      "Dependencies": [
        {
          "$id": "4",
          "AnnotationInfo": [],
          "CustomAttributes": {
            "$id": "5"
          },
          "Parent": {
            "$ref": "2"
          },
          "Link": {
            "$id": "6",
            "Id": "anytree",
            "Name": "anytree",
            "Version": "2.8.0",
            "RegistryType": "PyPI",
            "Authors": [
              "c0fec0de"
            ],
            "License": "Apache 2.0",
            "Summary": "Powerful and Lightweight Python Tree Data Structure..",
            "AnnotationInfo": [],
            "CustomAttributes": {
              "$id": "7"
            },
            "Dependencies": []
          }
        }
      ]
    }
  ]
}
```


Listing 1: Example of a dataset serialized into the universal JSON format.

4.4.2 Tooling concepts

As part of the original thesis [1], three main concepts have been introduced for the framework – parsers (section 4.4.3), analyzers (section 4.4.3) and importers/exporters (section 4.4.3).

4.4.3 Parsers

Parsers are framework tools used to create a dependency graph by querying software artifacts repositories, such as Maven Central Repository[1], Python Package Index[2] or NuGET[3], and obtaining metadata about dependencies. The output is a dependency graph represented in the universal data model. In addition to the information about dependency relations, arbitrary metadata about dependencies, dependency edges or the artifact repository itself can be obtained and saved.

The author of the work has implemented two parsers. The first one is PyPI parser – which parses dependency information for a given package from Python Package Index (PyPI), a repository for python artifacts. The second one is NuGet parser – which likewise parses dependency information, but from NuGet, a package manager for .NET platform.

Analyzers

Analyzers are framework tools designed to analyze a dependency graph produced by parsers and provide analysis results. The output of analyzers is an annotated dependency graph, where the annotations of relevant elements provide information about the analysis results. The result can be further processed by other analyzers by exporting the modified dataset into the universal data model.

Some analyzers may not be able to use the universal data format due to their nature. To this category belong, for example, the analyzers which collect and print metrics.

The author of the work has implemented several analyzers. These analyzers include Circular Dependency analyzer – for detecting circular dependencies in a dependency graph, Version Conflict analyzer – for detecting version conflicts, Key Value analyzer – for filtering dependencies by attribute value and Counter analyzer – for counting annotated dependencies.

[1] `https://mvnrepository.com/repos/central`
[2] `https://pypi.org`
[3] `https://www.nuget.org`

Importers and Exporters

To be able to hand over the dataset created/modified by parsers and analyzers to other tools, the dataset needs to be transformed into an intermediate format. Moreover, the framework tools may provide a capability to export the dataset into various specialized formats, for example, a graph description language or a simple text with better human readability. Converting the data to these formats may cause an information loss, but that is acceptable for use cases which require to extract or visualize only a subset of the whole dataset. Importers and exporters serve exactly for this purpose. Importers import the data from an intermediate format, exporters convert the tools individual data representation into an intermediate format or another representation.

They must not be confused with framework tools, they more like a libraries, components or services inside every tool for handling the required data transformation functionality.

The author of the work has implemented importers for the common JSON data model (section 4.4.1), DOT format for graph representation, and a simple textual format where the dependencies are individually listed each on one line.

5 Framework design analysis

A lot of work has already been done in the field of software dependency analysis at the University of West Bohemia as described in the previous section 4. This thesis aims to build upon that work and create foundations for a universal software dependency analysis framework (SDAF).

The first section 5.1 is devoted to the overall framework concept and the next section 5.2 outlines the common data model. The analysis of the framework data storage model is quite complex, so a separate chapter 6 is devoted to this topic.

5.1 Framework concept

The overall framework concept is based on the work of M. Hotovec[1] and it is practically unchanged. His concept is already described in this thesis in the section 4.4. To recap:

1. The framework is modular – it consists of multiple tools that operate independently.
2. Framework tools are able to generate datasets, to contribute to existing ones or to infer outcomes based on a dataset state.
3. The dataset data model is standardized for all tools and can be represented by a standardized intermediate format.
4. The framework workflow consists of chaining multiple tools in a row. Each tool contributes to the desired result by modifying the dataset and handing it over to the next tool in the pipeline.
5. The tools are not restricted to using a specific programming language.

As part of his work, the framework tools have been divided into two categories – parsers (this thesis section 4.4.3) and analyzers (this thesis section 4.4.3). The author of this thesis decided not to differentiate between various types of tools as the definition from the point 2 is sufficient, and framework tools can have much more responsibilities than the ones outlined for parsers and analyzers.

The diagram of the execution pipeline flow within the framework is visualized on the figure 5.1.

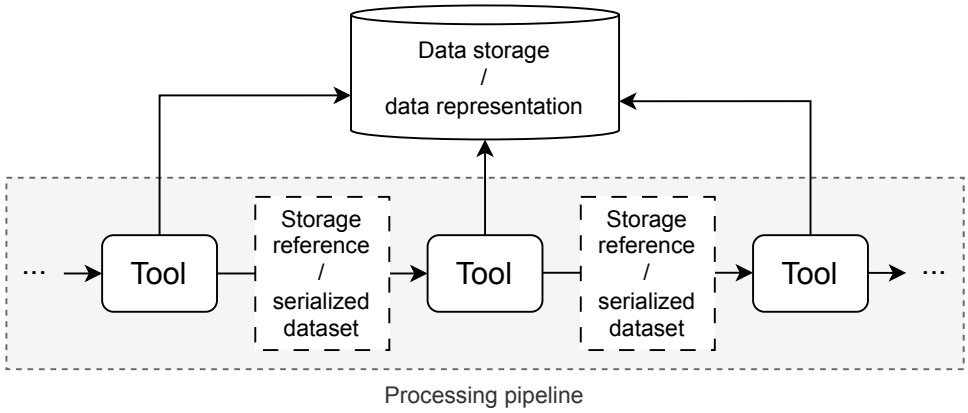


Figure 5.1: The framework execution pipeline.

5.2 Common data model

In his work [1], M. Hotovec has proposed a data model for the purpose of software dependency analysis. The data model is briefly described in this thesis in the section 4.4.1. In short, the two main types of data objects are *dependencies* and *dependency edges*. A dependency represents a software package – a jointly developed, versioned, published and deployed collection of software resources. This software package definition is equivalent to the definition used by package managers such as Pip for Python or NuGet for .NET platform, for which the data model has been designed primarily. In the graph terminology, a dependency can be interpreted as a vertex. A dependency edge is a representation of the one-way relation between two dependencies – the first one needs the second one (to be successfully compiled, deployed, or another type of the relation which can be identified as a need). It can also be used as a relation of a dependency association with a project (discussed later in this paragraph). In the graph terminology, a dependency edge can be interpreted as an edge in a directed graph. The two other, vertex-like data object types are a *project*, which serves basically as a container for a set of dependencies, and a *solution*, which serves a container for a set of projects. One other, edge-like data object type is a *project edge*, which represents a relation of project association with a solution.

The proposed data model also includes predefined data attributes for the data objects. These were designed on the research, which is part of the same work, about common metadata information provided by the mentioned package managers.

The data model intermediate format clearly defines a way, how to include arbitrary types of data with unconstrained structure in the top-level data

wrapper object. The framework tools then have freedom of choice what data types they support. The determination of the contained data type is, however, realized using the data wrapper attribute *DataType* which is the type's textual representation in the used programming language and therefore the solution it is tightly coupled to this language. It can be difficult to ensure the data format compatibility among multiple programming languages.

Although the data model conforms to the use cases for analysis of software packages dependency graph, the focus on these use cases causes the model to be insufficient for broader use cases. The example of such use case is CRCE project (described in the section 4.1) which deals with software components compatibility evaluation. Among others, the CRCE data model deals with a number of different types of data objects, for example, the compatibility information which is a relation between two components. This compatibility information can be comprised from a large volume of hierarchical metadata. The proposed data model can handle such information by mapping the different types of data objects to the types defined by the format, and by storing the additional data into the attribute `CustomAttributes` (mentioned in the section 4.4). This is impractical, as the semantic meaning of the original object type is "bent" to fit another use case and the attributes with relevant information are "stuffed" in a single top-level attribute. The mapping mechanism is, however, used for the purpose of intermediate data transfer format. The intermediate data format is further elaborated in the section 6.6.2.

5.2.1 Data model generalization

Due to the presented reasons, the author of this thesis proposes a generalization of this data model to be truly universal. The generalization of dependency, project and solution is called an *artifact*, and the generalization of dependency edge and project edge is called a *relation*.

Artifact – a graph vertex. It represents a collection of information that is treated as a whole. It can be a representation of various concepts, for example, a software package, component, or a project.

Relation – a graph edge. It represents a relation between two artifacts, accompanied by a collection of information. It can be a representation of any relation kind which is sensible for the given context. It can vary from a dependency relation or compatibility relation with a large volume of associated information (metadata), to simple relations like membership of a

package in a project.

Both of the two object types artifact and relation can be further differentiated by the concrete type of the concepts they represent. The type is specified using a textual name of the type. The default type for artifact is "`artifact`" and for relation is "`relation`". The resulting graph created from individual artifacts and relations can be very complex and contain a large amount of interconnected information. Example of how this graph structure may look like is on the figure 5.2.

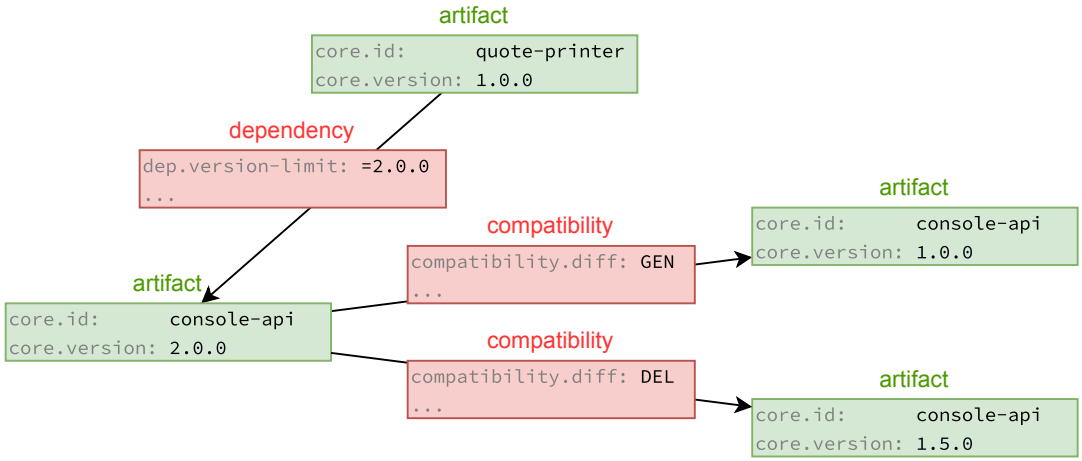


Figure 5.2: The example of the framework dataset graph structure.

The information which the artifacts and relations possess is not limited to the predefined attributes. The attributes form a hierarchical structure and each attribute or a subtree of attributes can be referenced using a predefined namespace. The namespace can be language-specific, tool-specific any other depending on the use case.

Since M. Hotovec already identified the most commonly used attributes, they form the core namespace. The prefix (or the first name of the hierarchy) of the core namespace is simple "`core`".

M. Hotovec performed an analysis of a suitable data format for the information representation [1, section 3.1 and chapter 6]. Based on his analysis, he chose JSON data format as it provides sufficient means to store any type of structured data using number attributes, text attributes, etc., and compound data types like arrays and objects. The author of this thesis decided to use the same approach and represents the artifacts and relations information using the JSON or JSON-like format. Example of such representation is on the listing 2.

```
{
  "core": {
    "id": "asm-all-repackaged",
    "version": "1.1.13"
  },
  "java": {
    "file": {
      "jar": "/dip/experiment/data/asm-all-repackaged/1.1.13/asm-all-repackaged-1.1.13.jar"
    }
  }
}
```

Listing 2: Example of an artifact data representation.

6 Framework data storage model analysis

The design of the dependency analysis framework must be able to support development in multiple programming languages and also consider the possibility of processing large data sets. This affects the two main framework concepts, which serve as a medium for direct communication/cooperation between individual framework tools. The two concepts are:

- inter-tool data transfer,
- data storage and access.

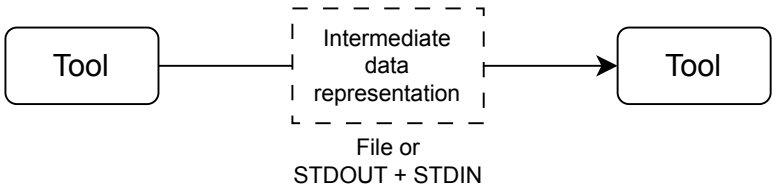
This chapter deals with an analysis of the possible framework design decisions involving these two concepts.

6.1 Inter-tool data transfer analysis

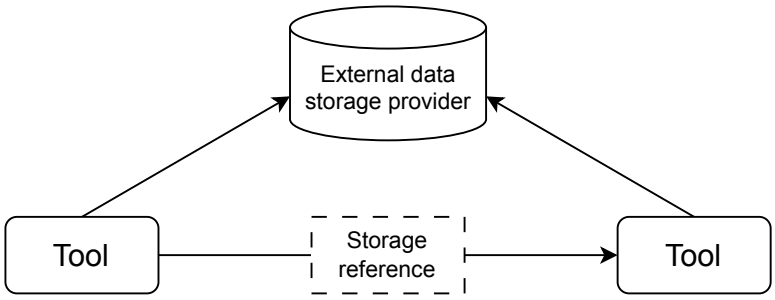
The form of data transfer between individual tools can have a very large effect on the resulting performance. There are generally the following two main ways to transfer the whole dataset.

- Serializing and deserializing the dataset before and after tool logic execution respectively (figure 6.1a).
- Using external storage provider (independent of the framework tools) shared between the individual tools (figure 6.1b).

The first case is the one proposed in the original framework design [1], where the process of data transfer is following. When a tool is done with its processing logic, it fully writes the dataset into its output, and the following tool in the pipeline reads it from its output. Input/output can be either a file or STDIN/STDOUT. The intermediate data representation can be anything like an originally proposed standardized JSON document, DOT graph representation or plain text. While working with small data set or chaining reasonably small number of tools, the performance cost of the fully serializing/deserializing the data might be negligible compared to the performance cost of the tool logic itself. For larger datasets or multi-tool chaining,



(a) Inter-tool data transfer using whole dataset serialization and deserialization.



(b) Inter-tool data transfer using external storage provider.

Figure 6.1: Inter-tool data transfer approaches.

the data transfer process can become a serious bottleneck of the execution pipeline. There is also a memory constraint aside from the time costs. When not using any external storage provider or temporary disk storage, the dataset size is limited by the size of the computer's main memory

The second way of data transfer solves the problems of the first one. The dataset is stored using an external provider independent of the individual framework tools. The framework tools can therefore just pass a reference to the used storage instead of full (de)serialization of the dataset. Individual tools do not have to completely load the whole dataset, but they can access only the smallest subset of data which they need for immediate use.

As has been said in this chapter introduction, the choice of the data transfer way is tightly coupled with the choice of the storage solution. When adopting the first solution i.e. serializing and deserializing the whole dataset, any type of storage can be used depending on the needs of individual tools. For the second solution, a multi-tool storage with standardized data structure specification has to be implemented or used.

6.2 Data storage types analysis

Both problems – how to store data and how to transfer it between tools – are tightly coupled. In some cases, the storage itself could be used as a

mean to transfer the data. The chosen storage solution then could affect the possibilities for the data transfer.

This section analyzes the possible types of data storage. For the framework purposes, the types can be categorized into groups by three individual criteria. A subsection is dedicated to each criterion and respective advantages and disadvantages of each storage type.

6.2.1 By implementation origin

One of the criteria by which the storage can be categorized is the implementation origin. The possible categories are:

- Custom implementation
- Existing third-party solution

By using a custom implementation the storage can be tailored exactly for the specific use case. However, the more specialized the implementation is, the more difficult it will be to reuse it (and possibly modify) for other tools. Also, it would be cumbersome or even impossible to reuse it for other programming languages. It could be made in a more generic manner, but it would require a lot of work for development and testing, and the requirements for the tools developed in the future may not always be known. Other requirements like support for large amount of data, concurrent access, optimizations (data size in memory, searching, indexes) could then be non-trivial to implement. There is also a possibility that the required custom solution will be just 'reinventing the wheel'.

On the other hand, using an existing third party solution would overcome the mentioned problems. There are a lot of projects providing robust storage solutions which are years-proven and have a large community of developers and users. Most of them support multiple programming languages, optimizations like compression and indexes, a query language for efficient data retrieval, etc. Possible third party solutions are described in the section 6.3.

The advantages and disadvantages are summarized in the tables 6.1 and 6.2.

Custom implementation

Advantages	Disadvantages
• Optimal for single-tool adhoc usage • Smaller development effort for simple use cases	• Complex implementation for multi-tool usage • Bigger development effort for complex use case requirements

Table 6.1: Advantages and disadvantages of custom storage implementation

Third-party solution

Advantages	Disadvantages
• Ready to use • Standardized ecosystem • Multi-language support • Multi-tool usage support	• May be non-optimal for some use cases

Table 6.2: Advantages and disadvantages of third-party storage solution

6.2.2 By usage scope

One of the criteria by which the storage can be categorized is the usage scope in terms of whether the storage will be used in a single tool or reused by multiple tools. The possible categories are:

- Single-tool usage
- Multi-tool usage

Leaving the responsibility to create/use a dedicated storage solution on the developers of each tool allows the framework to be truly versatile. Every tool could have different requirements which can be fulfilled by using a storage solution optimal for a specific use case. This, however, leads to an additional development effort which needs to be done for every tool. A non-standardized solution (within the framework) additionally requires the whole dataset to be transferred/transformed between individual tools in the execution pipeline. Problems which arise from that are described in the section dealing with problems of data transfer (section 6.1.

Standardizing a single storage solution (or set of solutions) within the framework gives an advantage to reuse data structure specifications and data

access libraries developed for the framework. It comes with the downside that not all possible use-cases can be efficiently applied with the chosen solution. The custom solution design or the choice of a third party solution must be reasoned with taking all possible use-cases into account.

The advantages and disadvantages are summarized in the tables 6.5 and 6.6.

Single-tool

Advantages	Disadvantages
• Optimal solution for single use cases	• Bigger development effort • Duplication of data access logic • Need of a full data transfer/transformation between individual tools

Table 6.3: Advantages and disadvantages of single-tool storage solution.

Multi-tool

Advantages	Disadvantages
• Reuse of data structure specifications and data access libraries • Inter-tool data transfer may be without cost	• May be non-optimal for some use cases

Table 6.4: Advantages and disadvantages of multi-tool storage solution.

6.2.3 By data location

The last criterion for categorizing the data storage types is the data location. From the perspective of the individual tools, the storage can be categorized into two following groups.

- In-memory (with computer main memory as primary location)
- On-disk (with computer secondary memory as primary location)

In-memory data storages store the data exclusively in the volatile primary computer memory. Some implementations may use secondary memory as a persistent backup.

On-disk storages store the data in the non-volatile secondary computer memory – on a disk or other persistent media. Similar to in-memory storages, their storage location is not exclusively limited to secondary memory, but they can maintain a cache located in the main memory consisting of preloaded data, indexes or other information.

The main trait distinguishing the in-memory data storages from the on-disk data storage is that the whole dataset must fit into the main memory which can have generally smaller capacity than the secondary. Another trait of the in-memory storage is that the access to it is generally faster than to the on-disk storage [14].

The advantages and disadvantages are summarized in the tables 6.5 and 6.6.

In-memory

Advantages	Disadvantages
• Faster data access.	• Limited capacity • Non-persistent in some cases.

Table 6.5: Advantages and disadvantages of in-memory storage solution.

On-disk

Advantages	Disadvantages
• Always persistent • Generally large capacity	• Slower data access • May require an additional software

Table 6.6: Advantages and disadvantages of on-disk storage solution.

6.3 Data storage solutions

Multiple data storage solutions are possible for the framework. Each of them is categorized into groups by using the criteria introduced in the previous section 6.3. The advantages and disadvantages of these groups are already mentioned in the same section. This chapter summarizes the data storage solutions and their respective solutions for inter-tool data transfer.

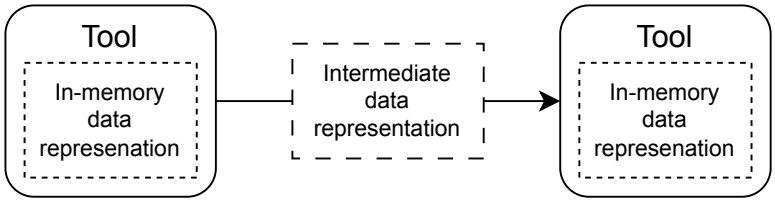


Figure 6.2: Process in-memory data storage solution.

6.3.1 Process memory storage

The simplest solution is to let the developers of each tool create their own data representation using the programming language constructs (or other structures). This data representation is located solely in the process memory. This approach belongs to the groups of *custom implemented* and *in-memory* solutions. As mentioned in the advantages of these groups, the implementation for simple use cases can be straightforward and optimal. However, using this approach requires the dataset to be fully deserialized into standardized intermediate representation for the purposes of the inter-tool transfer.

A graphical depiction of this approach is on the figure 6.2.

6.3.2 File-based storage

File-based storage belongs to the groups of *on-disk* and *multi-tool* solutions. By defining a standardized file schema or using the existing file formats like JSON or XML, the same data storage can be reused by multiple tools just by passing a reference to the file/directory. Therefore, it can be used directly as a mean to transfer the dataset between tools. The storage is persistent and independent of the individual tools and programming languages.

This solution has already been introduced in the original framework design [1], where the whole dataset is being serialized to a JSON format and optionally stored as a single file. It is a valid and simple solution, especially for small dataset and simple use cases. However, the serialization and deserialization of a single file can easily become too much resource demanding for bigger datasets. Also, special use cases like searching, inserting and updating can be non-trivial to be efficiently implemented, even when the dataset is split and organized into multiple files. Nevertheless, this approach has been already implemented by several projects providing a complete database storage solutions. These are summarized in the following section 6.3.3.

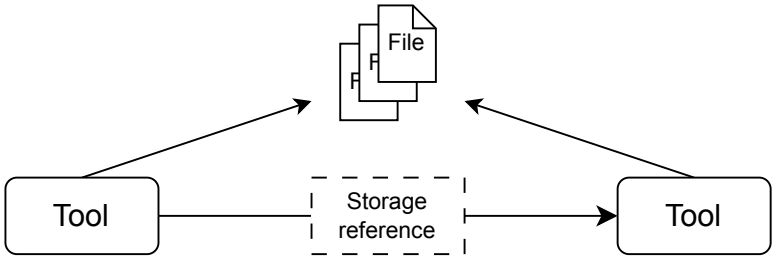


Figure 6.3: File-based data storage solution.

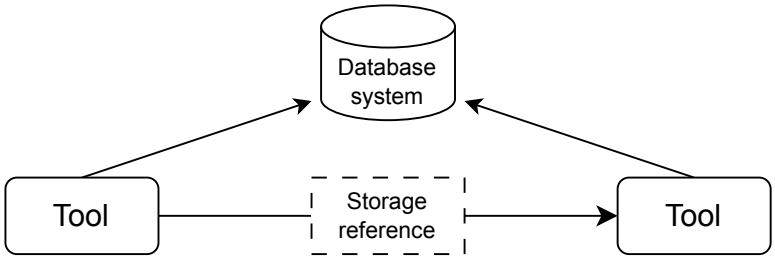


Figure 6.4: Database data storage solution.

6.3.3 Database storage

This section is devoted to a storage solutions using database systems. All information about relational and NoSQL databases is taken from the book "SQL & NoSQL Databases" [15].

A database system is a software that allows for the storage, organization, and retrieval of data. A database management system (DBMS) is software used to describe, store, and query data independently of specific applications. It includes a storage component for organized data and a management component for querying and manipulating data. DBMS also manages access and editing permissions for users.[15]

Relational databases are commonly used, but real-time web-based services and big data present challenges that may require NoSQL approaches.

From the framework perspective, a solution using a database storage belongs to groups of *multi-tool* and *third-party* solutions. It would require to adapt/map the framework data model to the data model provided by the chosen database system. However, the database system would manage features such as data consistency, relational integrity, transactional access, data organization and query optimization.

A graphical depiction of this approach is on the figure 6.4.

Relational database storage

A relational database is a structured collection of data organized and presented in tabular form. The data is stored in tables, where each table represents a specific entity or concept. Tables consist of rows and columns. Each row represents a unique record, while each column represents a specific piece of information associated with the records.

The tables in a relational database are interconnected through relationships, which define how the data in one table is related to the data in another table. This allows for efficient retrieval and manipulation of data based on these relationships.

Structured Query Language (SQL) is the primary language used to interact with relational databases. It provides a way to query, manipulate, and manage the data stored in the tables. SQL allows users to retrieve specific data based on conditions, perform calculations, join tables together, and more.

SQL databases emphasize ACID (Atomicity, Consistency, Isolation, Durability) properties to ensure data integrity, transactional consistency, and reliability. They traditionally also allow vertical scaling by upgrading hardware resources such as CPU, RAM, or storage. Some SQL databases also support horizontal scaling through sharding or replication.

Relational databases are widely used in systems that require centralized, structured, and persistent data storage.[15]

Example of relational databases are MySQL, PostgreSQL or Oracle Database.

NoSQL databases storage

NoSQL (Not Only SQL) database management systems provide a non-relational approach to data storage and retrieval. They have been developed to address the limitations of SQL databases in terms of scalability, performance, and handling unstructured or semi-structured data.

There are multiple differences between SQL and NoSQL database management systems, although they are not strict, and in some cases the boundaries are blurred. The key differences of NoSQL databases are:

1. Instead of relational data model, they employ various other models such as key-value, document, columnar, and graph.

2. They are schema-less or have a flexible schema.

3. They are designed for horizontal scalability, allowing them to distribute data across multiple servers and split the traffic load.

4. Other query language, specific to the corresponding data model, is used instead of SQL.

5. NoSQL databases often relax some ACID properties to achieve better scalability and performance. They follow BASE (Basically Available, Soft state, Eventual consistency) model which prioritizes availability, partition tolerance, and eventual consistency over strict consistency.

NoSQL databases are well-suited for systems which deal with large volumes of unstructured or semi-structured data, real-time applications or other applications with data structure which does not map well to the relational model.

The NoSQL databases model can be categorized into the following four groups: *key-value store*, *column store*, *document store* and *graph database*. A single database can support multiple of these models.[15]

Key-value store Key-value stores store data as pairs of keys and associated values. Each key corresponds to a specific value in the database. By specifying a key, the corresponding value can be retrieved from the database.

Key-value stores are popular due to their scalability for large amounts of data. The absence of checking referential integrity allows fast and efficient reading and writing. In-memory databases further enhance processing speed by buffering key-value pairs in the main memory. Key-value stores offer schema flexibility, as they do not require pre-defined schemas and can store data under arbitrary keys. However, the lack of a structured schema can present challenges in data management.[15]

Examples of key-value stores are Amazon DynamoDB or Redis.

Column store In column stores, data is stored in columns instead of rows. Storing data in relational tables per column instead of per row has proven to be more efficient for read operations. This structure allows for efficient data compression and high query performance, especially for analytical and aggregative operations. Similar to key-value stores, the data can be accessed by keys. Within the key there is another structure, dividing the row into several columns, which are also addressed with keys. The storage unit addressed with a certain combination of row key and column key is called a cell. The columns itself are grouped into column families which are used as units for access control.

Column stores are well-suited for read-intensive applications. While they provide fast query performance, they may have slower write operations due to the need for data reorganization and compression.[15]

Examples of column store databases are Apache Cassandra, Amazon Redshift or Google BigQuery.

Document store Document stores function as key-value stores, where the values are called documents and each key represents a document ID. These documents have their own internal structure, typically represented in JSON format, consisting of attribute-value pairs that can recursively contain additional pairs. The structure is schema-free – there is no need to define a schema before inserting data. Instead, the user or processing application takes on the responsibility of schema management. While this offers flexibility in storing diverse data types, it also lacks referential integrity and normalization.

They are highly scalable and excel in processing large amounts of heterogeneous data, where constant data consistency is not essential.[15]

Examples of document store databases are MongoDB or CouchDB.

Graph databases Graph databases represent data and schemas as graphs or graph-like structures. Data manipulations in graph databases involve graph transformations that address typical graph properties such as paths, adjacency, and subgraphs. Integrity constraints are supported to ensure data consistency, with consistency defined in relation to graph structures and referential integrity of edges. Graph databases excel when data is organized in networks, focusing on the connections between records. They offer the advantage of index-free adjacency, allowing for quick retrieval of neighboring vertices without considering all edges. Indexes, such as balanced trees (B-trees), are used to ensure efficient access to vertices and edges based on their properties. The graph database can utilize its own graph as an index, which is advantageous for query performance.[15]

Examples of graph databases are Neo4j, ArangoDB or OrientDB. The main consideration was to use a graph database as a storage solution. Existing graph databases are therefore described further on in section 6.4.

6.4 Graph databases

All analyzed storage solutions are possible to use for the framework. However, a graph database is the most advantageous solution. The reasons are

summarized in the storage analysis conclusion (section 6.5).

Three main graph database candidates have been considered for the framework: Neo4j, ArangoDB and OrientDB.

6.4.1 Neo4j

Neo4j is an open-source graph database implemented in Java. It uses a property graph model – vertices can be connected by relations, both can carry additional information called properties. It is highly regarded as the most popular and widely used graph database globally. It uses a Cypher query language with SQL-like syntax. Neo4j provides features like flexible schema, scalability, reliability, drivers support in multiple programming languages and high performance.[16]

Whilst the Neo4j offers a flexible data schema, the objects are structured as a flat JSON document i.e., the first level properties can not be another JSON document [17]. Although the complex data structures can be serialized as a JSON and be represented as text fields, this introduces a limitation for framework data model.

6.4.2 ArangoDB

ArangoDB is a multi-model open-source database system implemented in C++. It supports storing data as key/value pairs, documents, and graphs, all accessible through its own query language – AQL. ArangoDB offers simplified performance scaling, increased flexibility, fault tolerance, a large storage memory, and cost advantages compared to other databases.[16]

Flexible document schema is well-suited for the framework the rich-structured data model as it allows objects to be stored as JSON objects without any structural restrictions. The big advantage is also the fact that the storage engine is aware of this structuring. As a result, there is, among others, a possibility to create indexes for nested fields or combination of fields.[18]

6.4.3 OrientDB

OrientDB is a multi-model open-source database system supporting documents, graphs, key/value, and objects. It is implemented in Java. It offers transactional support and distributed architecture with replication. OrientDB allows manipulation of the database using Java, SQL, or Gremlin. It supports various modes including schema-less, full and mixed. Similar to

ArangoDB, its flexible document schema is eligible for the framework data model.[16]

6.4.4 Summary of graph databases

ArangoDB stands out as a multi-model database supporting multiple data models, while Neo4j and OrientDB have a primary focus on graph databases. Neo4j is specifically designed for graph data, emphasizing relationships, and has a rich set of graph-specific features. ArangoDB provides its own AQL query language, Neo4j uses the Cypher query language, and OrientDB supports SQL, Java, and Gremlin. Neo4j has a larger and more active graph community worldwide compared to ArangoDB and OrientDB.[16]

According to the benchmark test [8] which monitored processing time and memory consumption for common data access operations as well as graph-related operations, from the three candidate databases, ArangoDB performed the best in all tasks such as reading, writing, finding neighboring vertices and finding the shortest path in a graph. OrientDB has the worst time results among all tested databases, while Neo4j has the highest memory consumption. Other tested databases were MongoDB and PostgreSQL.

The table 6.7 summarizes described features and differences.

	Neo4j	**OrientDB**	**ArangoDB**
Implementation language	Java	Java	C++, JavaScript
Query Language	Cypher	SQL, Gremlin, ...	AQL
Database model	graph	graph, document, key-value	graph, document, key-value, object
Data schema	schema-less, flexible	schema-less, flexible	schema-less, flexible
Data modeling	limited, flat document	full JSON document	full JSON document
Drivers support	multi-language	multi-language	multi-language
Community, (free) edition	yes	yes	yes
Overall performance	good	worse	the best

Table 6.7: Features and difference of graph databases Neo4j, OrientDB and ArangoDB.

6.5 Analysis conclusion

The analysis has shown that multiple solutions are viable for the framework data storage and data transfer.

It has been decided to use a third-party NoSQL graph database. The major decisive factors were following.

1. The framework must support processing large amounts of data. A database solution offers a way for individual framework tools to operate only on the required data without a need to load the whole data model into memory. Also, the database itself can serve as medium to

transfer the dataset between multiple independent framework tools; this approach is practically cost-free.

2. Since a database system is independent of the framework tools, it allows for easy use of multiple programming languages.

3. Due to the graph-like nature of the framework data model, a NoSQL graph database has been shown as the most eligible solution. Graph databases also support efficient implementation for typical graph operations like graph search.

4. Schema-free nature of graph databases corresponds to the flexibility of the framework data model entity attributes.

5. Implementing an own database solution has turned out to be meaningless, since there exist multiple mature and full-fledged database solutions.

ArangoDB has been chosen eventually from the three graph database candidates. The main reason was support for flexible and rich-structured data schema. OrientDB would be, however, an equally suitable choice.

In addition to a graph database solution, it has been decided to also support the originally proposed JSON data format [1] for intermediate data representation. Framework tools will therefore have freedom of choice to support either the database data storage solution, its own solution or both. The mentioned intermediate data format can continue to serve its original purpose – transferring data between framework tools – and also act as a medium for exporting and importing the data from and into the database respectively.

6.6 Framework data management

ArangoDB has been chosen as the main storage solution. It has also been decided to support the JSON intermediate data transfer format as a complementary way to transfer data between tools or for exporting/importing data from/into a database. The developers of the framework tools then can choose whether the tool will support the intermediate format, since some features provided by the graph databases could be complicated to mimic using a custom in memory representation.

The following section 6.6.1 contains a brief description of the ArangoDB features, and how they are used within the framework, the section 6.6.2

describes the usage of the intermediate data format and the last section 6.6.3 describes the way to pass database reference between framework tools.

6.6.1 ArangoDB features

ArnagoDB is an opensource NoSQL graph database. It supports multiple data store models. The first is the document store – it stores individual records as structured JSON documents. Every record is also identifiable by a unique key – therefore, it can be used as a key value store. To support the graph database functionality, ArangoDB adds a layer on-top of the base document store for efficient implementation of graph structure and operations [5].

Collections

Collections allow to store documents and can be used to group together records of similar types. Documents stored in the same collection do not have to have the same data structure. Each collection is identifiable by its unique name.[18]

ArangoDB has two types of collections – vertex collections and edge collections. Documents in both of them are identifiable by a textual key (attribute `_key` in the document) which is unique within the parent collection. Another way to identify a document is by its ID (attribute `_id` in the document) which is composed of the parent collection name, a slash character, and the document key (for example, "artifacts/artifact_10"). IDs are unique across the whole database.[18]

Vertex collections are for vertex-like documents. Edge collections are used to store directed connections between vertices in a graph. The documents inside edge collections have two additional special attributes `_from` and `_to` to reference start and end vertex documents by their ID.[18]

The framework data model is easily representable by the concept of ArangoDB collections. Artifacts correspond to documents in vertex collections and relations to documents in edge collections. Individual types of artifacts and relations are expressed through membership in different collections, thus the types correspond to ArangoDB collections.

Graphs

A graph consists of vertices – documents from vertex collections and edges – documents from edge collections. It is also possible to use a document from edge collection as a vertex.

Graphs are either named or anonymous. Named graphs are fully managed by ArangoDB with transactional operations and integrity checks. They are also visible in the ArangoDB web interface. Anonymous graphs are on the other hand defined per-query on the client side and can be used ad hoc. The functionality for both types is the same. All graph operations, such as traversal, are performed on these graphs.[18]

While the named graphs provide better readability and maintainability since they can be used with multiple queries, there is a potential performance decrease for large graphs traversing. On the other hand, anonymous graphs can improve query performance by traversing only the specified collections.

The framework dataset can gain large volumes of data and complex graph structure with several types of artifact and relations. It is therefore advantageous to define a subgraph of the whole-database graph consisting only of the required artifact and relation types (document collections). The operations are optimized for the specified subgraph thanks to ArangoDB graph storage engine.

Indexes

Besides the ArangoDB built-in indexes, which cover the document attributes `_key`, `_id`, `_from` and `_to`, ArangoDB supports user-defined indexes. ArangoDB supports multiple index types. The simple value indexes have basic support for equality and comparison operations, sorting and filtering and can consist of multiple attributes. Indexing of single array items is also supported. Other types of indexes include TTL (time-to-live) index – for automatic removal of expired data, multidimensional index – for efficient range intersect queries, and geo index – for geospatial coordinates.[18]

Web interface

A built-in web interface is included in the ArangoDB server. It allows the management of databases, collections, documents, users, graphs and more. Queries can be created, edited and executed through a command interface.[18]

One of the most valuable features is the ability to visualize and browse graphs using a graphical interface. The example view of the graphical interface is in the figure 6.5.

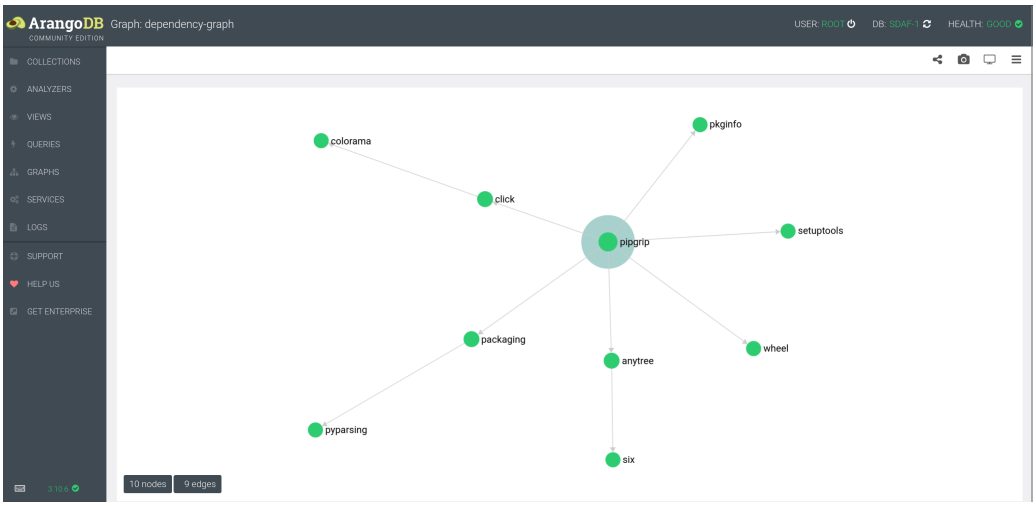


Figure 6.5: Graph visualization in ArangoDB web interface.

6.6.2 Intermediate data transfer format

As discussed in the framework data model introduction section (5.2), the original intermediate data transfer JSON format is not sufficient to the full coverage of possible framework use cases, it can still be used as a mean for inter-tool data transfer.

During converting to the intermediate JSON data format, each artifact, regardless of its type, is converted into a *DependencyItem* and each relation into a *DependencyEdge*. All attributes from the `core` namespace are stored into matching *DependencyItem* attributes. All other attributes are stored as sub-attributes of the attribute `CustomAttributes`, while preserving the hierarchical structure. The procedure is the same for relations. There is also a special textual sub-attribute `type` inside the `CustomAttributes` attribute, which preserves the type of the artifact or relation.

The original *DataSerializationWrapper* attribute *DataType* for indicating the contained data type is omitted, and the is only one implicitly expected data type is array of *DependencyItem*s.

Example of the serialized dataset using the outlined algorithm is on the listing 3.

JSON structure formalization

Since the intermediate data transfer format is standardized, it can also be formalized using a schema description language. For JSON, there exist two main projects dealing with this task: JSONSchema [19] and OpenAPI [20].

The former provides a way to formalize only the JSON structure. The latter adds on top of the structure formalization a support for complete web service API specification.

While the OpenAPI might seem to be too complex for the framework, it provides a relatively simple way to define a JSON schema structure. It has as massive support for countless programming languages and even for specific use cases compared to JSONSchema. It provides a tool for generating data classes or structures in the desired programming language with a lot of customization options. Due to the presented features, OpenAPI has been chosen in favor of JSONSchema.

The intermediate JSON format formalization can be used by the developers of the framework tools to easily generate DTO (data transfers objects) for the specific language used for the tool development.

6.6.3 Database reference passing

The passing of the database reference has been realized using a simple JSON file with included database information. The path to the file is then passed to the framework tools. The database reference JSON file has the structure shown in the listing 4. In consists of the database server host name and port number to specify the server location, username and password for access rights, and the name of the database to be used.

```
{
  "Data" : [ {
    "Id" : "core",
    "Version" : "6.0.0",
    "CustomAttributes" : {
      "type" : "artifact",
      "uid" : "core=6.0.0"
    },
    "$id" : "1"
  }, {
    "Id" : "core",
    "Version" : "7.0.0",
    "CustomAttributes" : {
      "type" : "artifact",
      "uid" : "core=7.0.0"
    },
    "Dependencies" : [ {
      "Parent" : {
        "$ref" : "2"
      },
      "Link" : {
        "$ref" : "1"
      },
      "CustomAttributes" : {
        "compatibility" : {
          "difference" : "DELETION"
        },
        "type" : "compatibility"
      },
      "$id" : "3"
    } ],
    "$id" : "2"
  } ]
}
```

Listing 3: Example of a dataset serialized into intermediate data format.

```
{
  "host": "localhost",
  "port": 8529,
  "user": "root",
  "password": "password",
  "database": "sdaf"
}
```

Listing 4: Example of a database reference file

7 Framework implementation

As a part of this thesis, the following has been implemented.

- **Framework core Java library** – the Java library with core domain objects, attribute namespace, (de)serialization from/into intermediate JSON format and default implementations of DAOs (data access objects) for simple in-memory storage and ArangoDB storage (section 7.1).
- **Framework tools**:
  - **Data importer** – simple tool for straightforward uploading of data into framework storage (section 7.2.1).
  - **Java compatibility extractor** – tool for extracting Java components compatibility info (section 7.2.2).

7.1 Framework core Java library

The framework core Java library provides basic building blocks for development of framework tools in the Java programming language. The core library consists of core domain classes (section 7.1.1), basic intermediate format (de)serialization functionality (section 7.1.2) and a set of basic DAOs (data access objects) for interaction with in-memory and ArangoDB storage (section 7.1.3).

7.1.1 Domain classes

Artifact

Class *Artifact* is the most basic representation of an artifact. It consists only of an artifact type and unique ID. As discussed in the section 6.6.1, the artifact type is mapped into an ArangoDB collection and artifact unique ID into document's `_key` attribute.

The absence of any other information associated with the artifact is intentional, every other information required for a specific use case, such as concrete attributes and adjacent relations or artifacts, can be subsequently queried from DAOs. There is no unnecessary data being loaded. This class's main purpose is, therefore, to serve as a reference to an artifact and to be used for the subsequent queries.

Relation

Class *Relation* is the most basic representation of a relation. It consists of a relation type, unique ID and source and target (start and end) vertex (via *Artifact* class). Similarly to the class *Artifact*, the relation type is mapped into an ArangoDB edge collection and artifact unique ID into document's `_key` attribute. In addition, the unique IDs of the start and target artifacts are mapped into `_from` and `_to` attributes respectively. Also, it serves similarly only as a reference to relation for subsequent queries.

Name

Class *Name* is an attribute hierarchical name consisting of a name segment for each hierarchical level. It is used for the attribute name representation across the whole core library.

AttributeSpec

Class *AttributeSpec* specification of an attribute consisting of an attribute name and class. This class is primarily intended for the tool developers to specify the attribute names and their data types in attribute namespaces. The specifications can be subsequently used to query these attributes from DAOs.

Attribute

Attribute is a representation of a single named value with an associated Java type. It can vary from simple single values like string or number, to more complex compound types forming a JSON-like tree structure.

It is required that attribute values can be easily serialized as JSON. Since the core library uses Jackson internally for attributes serialization and deserialization, all Java primitive types (including their boxed versions), such as integer, double, string and boolean, collections, such as lists, sets and maps are supported by default. Serialization of custom classes is also supported with a default behavior provided by Jackson, with a possibility to use Jackson annotations for (de)serialization customization, or even to use polymorphism and inheritance. That is, however, not recommended, since the JSON representation should be as less complex as possible for good multi-language support. It is advised to use plain DTOs (data transfer objects) for attributes representation and simple tasks. Additional logic should be moved into domain objects, created for that purpose, wrapper objects or services.

It is also presumed that the attribute classes are immutable because the in-memory storage solution stores the exact instances without copying them. Any subsequent modification of the provided instances would also change their representation in the storage, which is not desirable.

The representation of attributes in JSON format, together with strict detachment from their representation in Java types, makes it possible for the information carried by attributes to be retrieved and used easily from any other programming language.

AttributeMap `AttributeMap` is a wrapper very a tree hierarchy of attributes. Primarily, it is used for inserting or querying multiple attributes of an artifact or a relation. It is also used as a part of the in-memory dataset storage for representing the attribute hierarchy.

Internally it is a map of maps forming a tree. Every node in the tree is a hashmap having keys of type String and associated values of type Map, List or a primitive values, thus mimicking the JSON structure.

Individual attributes can be inserted, and queried from the attribute map using an *AttributeSpec*.

7.1.2 Serialization and deserialization

For the purposes of data serialization and deserialization into/from the intermediate data format, two Java interfaces have been defined: *DataSerializer* and *DataDeserializer*. There is one default implementation of both, which handles data in the intermediate JSON data format designed by M. Hotovec and modified by the author of this thesis (section 6.6.2).

The intermediate JSON format has been formalized by an OpenAPI specification, which is a part of this thesis. Using this specification, all DTOs (data transfers objects) are generated for (de)serialization) simplification. The DTOs are than deserialized using the Jackson library. The created OpenAPI specification can be used to generate DTOs for any other programming language which is supported by the OpenAPI generator tool, thus, making the framework to be easily extended with other tools.

The deserialization has been tested on several original files from [1], however, since the format used for this work is a subset of the original one, some files were incompatible. To test the specification suitability for other languages, DTOs for Python has been generated and are available as a part of this work.

The created OpenAPI specification is available in appendices (appendix A).

7.1.3 DAO

A set of interfaces for commonly used basic DAOs (data access objects) has been defined:

- `ArtifactDao` – a DAO for basic artifact CRUD (create, read, update, delete) operations.
- `RelationsDao` – a DAO for basic relation CRUD operations and for fetching artifacts adjacent to the relations.
- `AttributeDao` – a DAO for basic CRUD operations of artifact and relation attributes.

The core library includes default implementation of these three DAO interfaces for in-memory storage and for ArangoDB storage.

7.2 Framework tools

Two command-line tools has been implemented as a part of this thesis. The first is Data importer (section 7.2.1) and the second is Java compatibility extractor (section 7.2.2).

Both tools are implemented using the Spring Boot Framework[1], which is a framework for creating stand-alone applications with strong emphasis on dependency injection and autoconfiguration of application components. For a user-friendly commandline interface, Picocli[2] library has been used.

7.2.1 Data Importer

Data Importer is a simple Java utility command-line program for uploading data into a framework dataset, whether it is located in memory or in database. It supports two ways of importing data.

The primary way is to use the intermediate JSON data format by accepting it as an input (file or STDIN) and importing it into the storage. This behavior should by default for all tools with support for in-memory storage, since the dataset has to be uploaded somehow into the storage, in this case by using the intermediate data format.

The other way to import the data is to use a simple JSON file with an array of artifact definitions and an array of relation definitions. This

[1] `https://spring.io/projects/spring-boot`
[2] `https://picocli.info/`

additional way of importing data has been introduced for manual data generation, because writing the intermediate JSON data format with tree-like structure in hand could be challenging for humans. Example of such JSON is on the listing 5

```
{
  "artifacts": [
    {
      "uid": "core=7.0.0",
      "type": "artifact",
      "attributes": {
        "core": {
          "id": "core",
          "version": "7.0.0"
        }
      }
    },
    {
      "uid": "core=6.0.0",
      "type": "artifact",
      "attributes": {
        "core": {
          "id": "core",
          "version": "6.0.0"
        }
      }
    }
  ],
  "relations": [
    {
      "uid": "(core=6.0.0)-(core=7.0.0)",
      "type": "compatibility",
      "source": {
        "type": "artifact",
        "uid": "core=7.0.0"
      },
      "target": {
        "type": "artifact",
        "uid": "core=6.0.0"
      },
      "attributes": {
        "compatibility": {
          "difference": "DELETION"
        }
      }
    }
  ]
}
```

Listing 5: Example of data importer simple input JSON.

A usage help can be printed by running the program with option `-h`. The full usage help is in the listing 6.

```
Usage: data-importer [-h] [-d=DB_FILE] [--stdin | -f=INPUT_FILE] [--stdout |
                     -o=OUTPUT_FILE] [-j=DATA_FILE | DATA]
Simple utility program for importing/exporting data into/from SDAF dataset.
      DATA                The string with JSON data to load into the storage.
  -d, --db-file=DB_FILE   The file with ArangoDB connection options. If not
                            given, the data are handled in-memory.
  -f, --input-file=INPUT_FILE
                          The input file to read the storage data from.
  -h, --help              Display a help message.
  -j, --data-file=DATA_FILE
                          The file with JSON data to load into the storage.
  -o, --output-file=OUTPUT_FILE
                          The output file to write the storage data into.
      --stdin             Read the storage data from stdin.
      --stdout            Write the storage data into stdout.
```

Listing 6: Data importer tool usage help.

7.2.2 Java Compatibility Extractor

Java Compatibility Extractor is a simple Java utility command-line program for running compatibility information extraction on the framework dataset. The main purpose of this program is to demonstrate the ability of the designed framework to handle such tasks, as is the compatibility extraction performed by the experimental CRCE repository[9] (described in this thesis in section 4.1).

The flow of the program is following.

1. In the framework dataset, find Java artifacts.
2. For each artifact, find possible target artifacts – with the same ID (`core.id`), different version (`core.version`) and which has not already been compared.
3. Load JAR files for both (attribute `java.file.jar`), run comparison and compatibility information extraction on these JARs.
4. Create and save a relation of type "compatibility" into framework dataset and save all the related compatibility information as the relation attributes.

The result of this program is, that a relation edge with compatibility information is added for every pair of artifact with the same ID and different version.

The algorithm of compatibility extraction is similar to the one implemented by J. Daněk [11] for CRCE as a part of his work. Internally, the Java Compatibility Extractor uses JaCC library (described in this thesis in the section 4.1). The extracted information is parsed and mapped into *CompatibilityInfo* classes representing the full compatibility tree structure with the root *ArtifactCompatibilityInfo* denoting the top-level compatibility of the artifact pair. The example of extracted compatibility information is in the listing 7.


```
{
    "compatibility": {
      "difference": "INSERTION",
      "children": [
        {
          "difference": "INSERTION",
          "code": "java.package",
          "before": "org.jvnet.hk2.config.types",
          "after": "org.jvnet.hk2.config.types",
          "children": [
            {
              "difference": "INSERTION",
              "code": "java.class",
              "before": "org.jvnet.hk2.config.types.Property",
              "after": "org.jvnet.hk2.config.types.Property",
              "children": [
                {
                  "difference": "INSERTION",
                  "code": "java.interfaces",
                  "children": [
                    {
                      "difference": "INSERTION",
                      "code": "java.class",
                      "after": "org.jvnet.hk2.component.Injectable",
                      "note": "Inserted"
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  }
```


Listing 7: Example of a relation with extracted compatibility info.

Usage

A usage help can be printed by running the program with option `-h`. The full usage help is in the listing 8.

```
Usage: java-compatibility [-h] [-d=DB_FILE] [--stdin | -f=INPUT_FILE] [--stdout
                          | -o=OUTPUT_FILE] [ARTIFACT_UIDS...]
Utility program for running Java compatibility extraction over an SDAF dataset.
      [ARTIFACT_UIDS...]   The unique IDs of all the artifact for which to
                             extract the compatibility information.
                           Default is for all.
  -d, --db-file=DB_FILE    The file with ArangoDB connection options. If not
                             given, the data are handled in-memory.
  -f, --input-file=INPUT_FILE
                           The input file to read the storage data from.
  -h, --help               Display a help message.
  -o, --output-file=OUTPUT_FILE
                           The output file to write the storage data into.
      --stdin              Read the storage data from stdin.
      --stdout             Write the storage data into stdout.
```

Listing 8: Java compatibility tool usage help.

8 Testing

Testing of the designed framework and implemented core library has been realized by demonstrating the ability to handle such tasks, as is the compatibility extraction performed by the experimental CRCE repository[9]. For this purpose, the Java Compatibility Extractor tool 7.2.2 has been implemented. Since the tool mimics the same functionality as the compatibility extraction plugin created by J. Daněk [11] for CRCE, the testing for this thesis has been realized using the same performance test scenario and test data. The correctness testing of the created tool is not relevant in the context of this thesis, since the main purpose of the created tool was to demonstrate the ability to handle the specified task.

Unlike the CRCE plugin which uses the OBCC library for compatibility extraction, the Java Compatibility Extractor uses the JaCC library. However, OBCC uses internally JaCC and performs the compatibility extraction only for OSGi exported packages. Thus, the Java Compatibility Extractor will do more computation compared to CRCE plugin, but it will contain the same results as a subset.

The testing scenario and results are in the section 8.1.

Test machine specifications

The specifications of the test machine and ArangoDB database server used for the test scenario are in the following table 8.1.

Computer Model	Lenovo Yoga S740-15IRH
CPU	Intel i7-9750H (12) @ 4.500GHz
OS	Fedora Linux 38 (Workstation Edition) x86_64
Linux kernel	6.3.4-201.fc38.x86_64
Java JRE	OpenJDK Runtime Environment 17.0.7 Red_Hat-17.0.7.0.7-4.fc38
ArangoDB	ArangoDB 3.10.6 Docker image `arangodb:3.10.6`[1]

Table 8.1: Test machine specifications.

[1] `https://hub.docker.com/_/arangodb`

8.1 Performance

Performance testing has been conducted in order to evaluate the time it takes for compatibility data to be fully extracted. The same testing dataset has been used as in the work of J. Daňek [11]. The testing involved six different sets of components and their revisions[2] from the Glassfish Application Server. The components were initially chosen to have variations in size and to have a substantial number of available revisions [11].

The details of the components in the testing set are in the table 8.2.

The tests have been run against an ArangoDB database storage.

8.2 Test scenario

The test scenario consists of two steps.

The first step involves uploading information about all components into the framework data storage at once. Data Importer tool is used for this task. An artifact record is created for each component revision with attributes `core.id` and `core.version` for the component identification, and attribute `java.file.jar` for denoting the component's JAR archive location in the local filesystem.

The second step is the compatibility extraction itself. The Java Compatibility Extractor is used for this task together with a modified version of the same tool for running in multi-threaded environment.

For each artifact, the compatibility information is extracted for every pair of the artifact and its revision which yet not have been processed. Start and end times were recorded using application logging messages directly inside the service which performs the extraction. The times were recorded for the whole extraction of artifact and its previous revisions, and also for each single pair extraction. The duration was computed as a difference between the times recorded before and after extraction. The following times were measured:

- The duration of a single artifact pair compatibility extraction.
- The total extraction duration for a single artifact and all its previous revisions.
- The total extraction duration for all artifacts.

[2] `http://relisa-dev.kiv.zcu.cz/data/experiments/crce-2013-12/`

Name	Number of revisions	Average size [kB]
config-types	139	6.8
osgi-adapter	140	75.7
dataprovider	90	103.9
asm-all-repackaged	141	293.8
bean-validator	142	575.1
webservices-osgi	44	11186.5

Table 8.2: List of components used for the performance test.

8.2.1 Test results

The results are analyzed from three perspectives:

1. Time to compute compatibility information for a single artifact pair.
2. Time to compute compatibility information for artifact's all previous revisions.
3. Total time to compute compatibility information for all artifacts and their revisions.

8.2.2 Single-pair extraction duration

The table 8.3 shows the mean times and the standard deviation required for compatibility extraction of single pair of components.

The standard deviation is rather high for smaller components, but as the component size increases and the extraction time grows, the standard deviation is getting smaller relatively to the extraction time. This applies for both single-threaded and multi-threaded environments. From that fact can be stated that the compatibility extraction times are predictable, and the higher standard deviation for smaller components is likely caused by external causes, such as operating system thread scheduler.

Since the concurrent access to the database in the multi-threaded does not need to be transactional, because all read and write operation in this use case do not affect each other, there is no time penalty apart from the possible waiting times due to increased disk IO read/write operations. On the other hand, the multi-threaded environment introduces time overhead which is needed for splitting the work between threads and for the thread context switches. This is projected into the slightly higher times required for single-pair extraction.

Component	**Mean time [ms]**	**Std. deviation [ms]**	**Mean time [ms]**	**Std. deviation [ms]**
	1 threads		4 threads	
config-types	3.9	7.22	4.45	6.87
osgi-adapter	19.61	14.76	23.42	8.59
dataprovider	31.82	6.52	50.41	14.14
asm-all-repackaged	101.85	11.15	144.16	25.54
bean-validator	187.78	107.79	266.5	170.33
webservices-osgi	3532.75	151.74	7582.5	776.3

Table 8.3: The times needed to extract compatibility information for a single pair of components.

8.2.3 Previous revisions extraction duration

Time duration required for compatibility extraction between a component and all of its previous revisions depends only on two factors:

1. Size of the component.
2. Number of previous revisions.

From the figures 8.1 and 8.2 is clearly visible, that the required time grows linearly in direct proportion to the number of previous revisions for both single-threaded and multi-threaded environments.

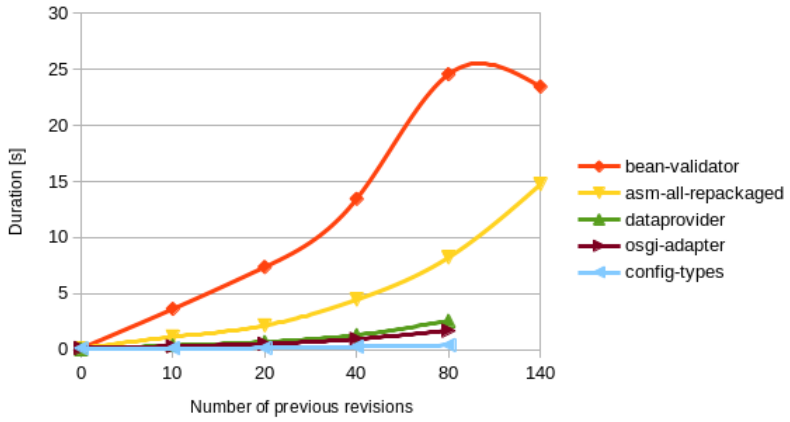


Figure 8.1: Extraction times based on number of previous revisions – single-threaded.

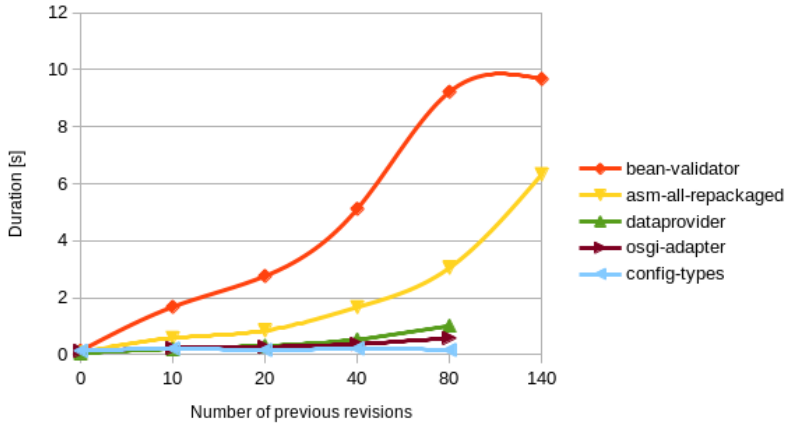


Figure 8.2: Extraction times based on number of previous revisions – multi-threaded.

8.2.4 Total extraction duration

Similar to [11], running the extraction in multi-threaded environment resulted in longer mean times for single-pair compatibility information extraction. The overall extraction time is, however, shorter than in the single-threaded environment. The total extraction times for both environments are in the table 8.4.

	Total duration [s]	
	1 threads	4 threads
Whole extraction scenario	6595.72	3080.78

Table 8.4: The times neede in total for the whole compatibility extraction scenario.

8.3 Test conclusion

The measured times for both single-threaded and multi-threaded environments are similar to the ones determined by J. Daňek [11] for the CRCE compatibility plugin. As a result, it can be concluded that the framework design is suitable for more advanced tasks such as the compatibility extraction.

9 Conclusion

The author of this thesis has created a universal and modular framework for analysis of software artifacts and their relations.

The framework design analysis has been performed based on the initial study of component-based software systems (chapter 2), graph data representation and analysis (chapter 3), and into existing tools and methods which are being researched and developed at the Department of Computer Science at the University of West Bohemia (chapter 4). The analysis of the framework overall concept and data model has been performed in the chapter 5 and the subsequent analysis of the framework storage model in the chapter 6. The analyses itself are built on existing work done by researches at the Department of Computer Science at the University of West Bohemia and on the thesis of M. Hotovec [1].

As a part of the analysis, framework data model generalization has been proposed to support a broader set of use cases, such as the compatibility extraction and verification introduced by CRCE project. For the framework data storage model, the ArangoDB database has been chosen as the main data storage provider.

The framework data and storage models have been designed with emphasis on multi-language development support. This has been achieved thanks to the ArangoDB which provides an application and language-independent data storage. In addition to this, the framework also utilizes the intermediate JSON data format for dataset transfer between individual framework tools, which has been originally proposed by M. Hotovec [1]. An OpenAPI formalization of a common intermediate JSON format has been created as a part of this thesis for easy multi-language development.

The requirement for the framework to be able to process large amounts of data has been fulfilled by designing the data model to support rich-structured and hierarchical records and by using the ArangoDB database solution, whose data model conforms to the designed framework data model.

A core framework library for Java programming language has been developed as a part of this thesis. To demonstrate the framework suitability for component-based systems research, two tools have been developed (chapter 7) on top of the created core library. The first tool is Data Importer — a tool for framework dataset initialization, import, and export. The second is Java Compatibility Exctractor — which mimics the functionality of a java compatibility extraction plugin developed for the CRCE

experimental repository.

The same performance test, which has been done in the work of J. Daňek [11] to verify the CRCE compatibility extraction plugin, has been cunducted. It has been eventually determined that the created tool performs similarly as the CRCE plugin. The results clearly state that the framework can be used for more advanced tasks such as the compatibility extraction.

The created framework can be used as is as a platform for development of additional tools in various programming languages. The tools created in the future can be used to support the research into component-oriented software system systems or into other fields since the framework provides a flexible data model and storage solution.

Bibliography

[1] M. Hotovec, 'Analýza závislostí softwarových artefaktů,' M.S. thesis, The University of West Bohemia, 2022.

[2] F. Bachmann, L. Bass, C. Buhman *et al.*, 'Volume ii: Technical concepts of component-based software engineering,' Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon Software Engineering ..., Tech. Rep., 2000.

[3] M. S. Rahman *et al.*, *Basic graph theory*. Springer, 2017, vol. 9.

[4] R. Čada, T. Kaiser and Z. Ryjáček, *Diskrétní matematika*. Západočeská univerzita v Plzni, 2004.

[5] ArangoDB GmbH. 'Index free adjacency or hybrid indexes for graph databases.' (2016), [Online]. Available: `https://www.arangodb.com/2016/04/index-free-adjacency-hybrid-indexes-graph-databases/` (visited on 08/05/2023).

[6] R. Angles and C. Gutierrez, 'An introduction to graph data management,' *Graph Data Management: Fundamental Issues and Recent Developments*, pp. 1–32, 2018.

[7] G. Klyne and J. J. Carroll. 'Resource description framework (rdf): Concepts and abstract syntax.' (2004), [Online]. Available: `https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/` (visited on 08/05/2023).

[8] ArangoDB GmbH. 'Nosql performance benchmark 2018 – mongodb, postgresql, orientdb, neo4j and arangodb.' (2018), [Online]. Available: `https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb` (visited on 17/05/2023).

[9] ReliSA. 'Crce - component repository supporting compatibility evaluation.' (2019), [Online]. Available: `https://github.com/ReliSA/CRCE` (visited on 15/05/2023).

[10] P. Brada and K. Jezek, 'Ensuring component application consistency on small devices: A repository-based approach,' in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, IEEE, 2012, pp. 109–116.

[11] J. Daňek, 'Determining and publishing component compatibility information,' M.S. thesis, The University of West Bohemia, 2014.

[12] P. Brada and K. Jezek, 'Repository and meta-data design for efficient component consistency verification,' *Science of Computer Programming*, vol. 97, pp. 349–365, 2015.

[13] J. Bauml and P. Brada, 'Reconstruction of type information from java bytecode for component compatibility,' *Electronic Notes in Theoretical Computer Science*, vol. 264, no. 4, pp. 3–18, 2011.

[14] M. Roohitavaf. 'In-memory vs. on-disk databases.' (2020), [Online]. Available: `https://www.mydistributed.systems/2020/07/an-overview-of-storage-engines.html` (visited on 08/05/2023).

[15] A. Meier and M. Kaufmann, *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management*. Springer, 2019.

[16] D. Fernandes and J. Bernardino, 'Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb.,' in *Data*, 2018, pp. 373–380.

[17] Neo4j, Inc. 'Neo4j documentation.' (2023), [Online]. Available: `https://neo4j.com/docs/` (visited on 08/05/2023).

[18] ArangoDB GmbH. 'Arangodb documentation.' (2023), [Online]. Available: `%5Curl%7Bhttps://www.arangodb.com/docs/stable/%7D` (visited on 08/05/2023).

[19] OpenJS Foundation. 'Json schema.' (2022), [Online]. Available: `https://json-schema.org/` (visited on 08/05/2023).

[20] The Linux Foundation. 'Openapi.' (2023), [Online]. Available: `https://json-schema.org/` (visited on 08/05/2023).

List of Abreviations

ACID Atomicity, Consistency, Isolation, Durability

API Application Interface

AQL Arango Query Language

BASE Basically Available, Soft State, Eventual Consistency

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

CRCE Component Repository and Compatibility Evaluation

DAO Data Access Object

DBMS Database Management System

DTO Data Transfer Object

IO Input/Output

JAR Java Archive

JSON JavaScript Object Notation

JaCC Java Class Comparator

OBCC OSGi bundle comparator

OBR OSGi Bundle Repository

OSGi Open Services Gateway Initiative

PyPI Python Package Index

RAM Random Access Memory

RDF Resource Description Framework

REST Representational State Transfer

SDAF Software Dependency Analysis Framework

SQL Structured Query Language

STDIN Standard Input

STDOUT Standard Output

List of Figures

List of Tables

List of Listings

Appendices

A. OpenAPI intermediate JSON format specification

```yaml
openapi: 3.0.3

info:
  version: 1.0.0
  title: SDAF data JSON schema.
  description: SDAF.

paths: { }

components:

  schemas:


    DataSerializationWrapper:
      type: object
      description: Root wrapper object.
      properties:
        Version:
          description: Version of the data format.
          type: string
        Data:
          description: The data.
          type: array
          items:
            $ref: "#/components/schemas/DependencyItem"


    DependencyItem:
      description: Dependency item information.
      type: object
      properties:
        $id:
          description: ID of the serialized dependency item to be used in references.
          type: string
        $ref:
          description: Reference to a serialized dependency item that is already present in the file.
          type: string
        Id:
          description: The unique ID of the artifact in the registry.
          type: string
        Name:
          description: The dependency name.
          type: string
        Version:
          description: The dependency version.
          type: string
        RegistryType:
          description: The type of package registry.
          type: string
        Authors:
          description: The list of authors.
```

```
      type: array
      items:
        description: Author name, email and/or other identifier.
        type: string
    License:
      description: The license identifier.
      type: string
    LicenseLink:
      description: The link to license.
      type: string
    Summary:
      description: The summary text of the package.
      type: string
    Description:
      description: The description text of the package.
      type: string
    Link:
      description: The link to package info.
      type: string
    PackageDownloadLink:
      description: The link to download the package.
      type: string
    SourceCodeLink:
      description: The link to the source code.
      type: string
    AnnotationInfo:
      description: Annotation info added by analyzers.
      type: array
      items:
        $ref: "#/components/schemas/AnnotationInfoItem"
    CustomAttributes:
      description: Custom attributes based on needs of the dependency registry or analyzer.
      type: object
      additionalProperties: true
    Dependencies:
      description: List of sub-dependencies for this dependency.
      type: array
      items:
        $ref: "#/components/schemas/DependencyEdge"


DependencyEdge:
  description: Specific edge for the dependency.
  type: object
  properties:
    $id:
      description: ID of the serialized dependency edge.
      type: string
    RequiredVersionsText:
      description: Required versions of the dependency - textual representation.
      type: string
    VersionsLimit:
      description: List of the limits for the specific version required.
      type: array
      items:
        $ref: "#/components/schemas/VersionLimit"
    Parent:
      $ref: "#/components/schemas/DependencyItem"
    Link:
      $ref: "#/components/schemas/DependencyItem"
```

```
    AnnotationInfo:
      description: Annotation info added by analyzers.
      type: array
      items:
        $ref: "#/components/schemas/AnnotationInfoItem"
    CustomAttributes:
      description: Custom attributes based on needs of the dependency registry or analyzer.
      type: object
      additionalProperties: true


VersionLimit:
  description: Package version limitation.
  type: object
  properties:
    RequiredVersion:
      description: Required version of the dependency.
      type: string
    CustomLimitSymbol:
      description: Optional, custom symbol of the operation.
      type: string
    LimitSymbol:
      description: Type of operation of the version.
      type: string
      enum:
        - GREATER_OR_EQUAL
        - GREATER
        - LOWER_OR_EQUAL
        - LOWER
        - EQUAL
        - NOT_EQUAL
        - COMPATIBLE
        - CUSTOM
        - UNDEFINED


AnnotationInfoItem:
  description: Annotation information added by analyzers.
  type: object
  properties:
    AnalyzerName:
      description: Name of the analyzer, which created this item.
      type: string
    AnalyzerVersion:
      description: Analyzer version.
      type: string
    AnalyzerCondition:
      description: Analyzer condition, which was used to match this report.
      type: string
    InformationType:
      description: Type of information provided.
      type: string
      enum:
        - Info
        - Warn
        - Err
        - None
    Text:
      description: Textual representation of the information.
      type: string
```