Benefits and Drawbacks of Representing and Analyzing Source Code and Software Engineering Artifacts with Graph Databases

Rudolf Ramler[1(✉)], Georg Buchgeher[1], Claus Klammer[1], Michael Pfeiffer[1], Christian Salomon[1], Hannes Thaller[2], and Lukas Linsbauer[2]

[1] Software Competence Center Hagenberg GmbH, Softwarepark 21, 4232 Hagenberg, Austria
{rudolf.ramler,georg.buchgeher,claus.klammer, michael.pfeiffer,christian.salomon}@scch.at
[2] Johannes Kepler University Linz, Altenberger Street 69, 4040 Linz, Austria
{hannes.thaller,lukas.linsbauer}@jku.at
http://www.scch.at
https://www.jku.at/isse

Abstract. Source code and related artifacts of software systems encode valuable expert knowledge accumulated over many person-years of development. Analyzing software systems and extracting this knowledge requires processing the source code and reconstructing structure and dependency information. In analysis projects over the last years, we have created tools and services using graph databases for representing and analyzing source code and other software engineering artifacts as well as their dependencies. Graph databases such as Neo4j are optimized for storing, traversing, and manipulating data in the form of nodes and relationships. They are scalable, extendable, and can quickly be adapted for different application scenarios. In this paper, we share our insights and experience from five different cases where graph databases have been used as a common solution concept for analyzing source code and related artifacts. They cover a broad spectrum of use cases from industry and research, ranging from lightweight dependency analysis to analyzing the architecture of a large-scale software system with 44 million lines of code. We discuss the benefits and drawbacks of using graph databases in the reported cases. The benefits are related to representing dependencies between source code elements and other artifacts, the support for rapid prototyping of analysis solutions, and the power and flexibility of the graph query language. The drawbacks concern the generic frontends of graph databases and the lack of support for time series data. A summary of application scenarios for using graph databases concludes the paper.

Keywords: Static analysis · Dependency analysis
Knowledge extraction · Graph database · Neo4j · Experience report

D. Winkler et al. (Eds.): SWQD 2019, LNBIP 338, pp. 125–148, 2019.
https://doi.org/10.1007/978-3-030-05767-1_9

1 Introduction

The size and complexity of real-world software systems are continuously increasing. Today, many companies develop and maintain software systems containing hundreds or thousands of source code files encompassing up to several million lines of code. They often consist of a mix of various technologies and, additionally, a wide range of related software engineering artifacts such as tests, documentation, change requests, bug reports, and execution logs. Source code and artifacts encode valuable expert knowledge accumulated over decades of development. They represent complex structures and related information about various parts of the software system.

Nevertheless, the source code and a large part of the artifacts are stored in text files organized in conventional directory structures. Analyzing software systems requires parsing these files and reconstructing the structure and relationship information. This first step is typically the precondition for a further, more advanced analysis aiming at software understanding, interactive exploration, fault detection, visualization, and documentation. Advanced analysis applications benefit from infrastructure for processing and representing the structure of software systems in a scalable and extensible way.

In several projects developing tools and services for software analysis, we have created different implementations of such infrastructures using graph databases for representing source code, software engineering artifacts and their relationships. Graph databases [18] are NoSQL databases that support graph data models, i.e., data represented in the form of nodes connected via edges with each other. Graph databases are particularly useful if relationships between nodes are a central characteristic of the stored data. They are optimized for storing, querying, and manipulating vast amounts of highly connected data by native support for relationships and enhanced traversal capabilities. Hence, they are frequently used in a wide range of applications such as recommendation engines, social networks, collaboration platforms, and medical research systems.

The objective of this paper is to collect and share our experiences with graph databases in representing and analyzing source code and software engineering artifacts. We describe five different cases related to different application scenarios and project contexts. Across all cases, the use of graph databases has emerged as a common element in the implemented solutions – yet with variations in how the data is modeled, stored and accessed. By comparing and discussing the different approaches, data models, and underlying design decisions, we provide insights into the advantages and disadvantages of graph databases for building analysis tools and services.

The remainder of the paper is structured as follows. Section 2 provides an introduction to graph databases and outlines related work. The research design of our experience report is described in Sect. 3. Details about the five presented cases are described in Sect. 4. The discussion of identified advantages and disadvantages follows in Sect. 5. Finally, Sect. 6 concludes the paper by summarizing the key findings and suggestions for future work.

2 Background and Related Work

A graph is composed of nodes and relationships. A node represents an entity (e.g., `class`, `method` or `variable`) and a relationship represents how two nodes are associated (e.g., a class `contains` a method, a method `calls` a method or `reads` a variable). Nodes and relationships can have properties that are usually specified in form of key-value-pairs (e.g., `name='foo'`). This general-purpose concept enables creating arbitrary connected structures that closely match the modeled characteristics of the problem domain [4].

A graph database is an online database management system with `create`, `read`, `update` and `delete` operations optimized for graph data models [18]. Graph databases are specifically designed to support fast and scalable management, storage, and traversal of nodes and relationships. This support allows to specify all relationships representing connections between entities at the time the data is created, and storing them persistently in the database. When the database is queried, these relations can be quickly traversed without the need to compute them dynamically via foreign keys and costly join operations.

The widespread adoption of NoSQL databases for many problem domains also led to the development of several databases specialized on graph data. Examples include AllegroGraph (Franz Inc.), InfiniteGraph (Objectivity Inc.), Neo4j (Neo4j Technology Inc.), and OrientDB (Callidus Software Inc.). A review and comparison of contemporary graph databases can be found in [3].

The cases we describe in this paper use the graph database *Neo4j*[1]. It is one of the most popular graph databases, also offered open source. Neo4j is based on a native graph storage and processing engine. It comes with the declarative graph query language *Cypher* that supports the definition, manipulation, and querying of graphs. Cypher queries can be issued using programming language specific drivers or the Web-based user interface *Neo4j browser*. In addition, Neo4j supports a programming language independent REST API and a low-level Java driver that can directly access database search facilities.

Many approaches and tools exist for analyzing software systems [13] and for performing queries on source code [1]. Most of these approaches and tools rely on database technologies to store structure and dependency information. Although the use of relational databases is still prevalent, NoSQL databases are receiving more and more attention. Zhang et al. [24] implemented a framework for querying heterogeneous code repositories using the document-oriented database MongoDB.

Graph databases have been applied in a few instances, by Yamaguchi et al. [22] for analyzing code to discover vulnerabilities, by Urma and Mycroft [21] for querying source code, by Goonetilleke et al. [8] to implement the tool Frappe for code comprehension, and in the open source tool jQAssistant[2] for ensuring code quality of Java programs. These applications are related to the cases described

[1] https://neo4j.com.
[2] https://jqassistant.org.

in our study. However, an analysis of large-scale software systems similar to what we present in our case study has only been described for Frappe [9].

3 Research Design

This paper reports our insights and experiences gained from using graph databases in the form of a collection of individual cases. Each case provides a first-hand account by the authors, who have been personally involved in the reported cases and the development of the associated tools and services.

Research goals and questions: Besides describing *how graph databases are used*, the paper explores *what are the advantages* and *what are the disadvantage* of using graph databases for representing and analyzing source code and related artifacts.

We conducted the following steps to provide answers to these questions.

1. **Case selection:** The key criteria for selecting a case to be included in our report were the use of a graph database, its application for supporting software analysis tasks, and the development of related tools or services.
2. **Case description:** We used a template with a uniform structure related to a set of open questions to describe the cases. The descriptions were prepared by the authors involved in the cases.
3. **Review of descriptions:** The case descriptions were reviewed by co-authors not involved in the case to assure that the descriptions are complete and consistent. Variations and extensions to the structure of the descriptions were introduced to capture individual aspects of the reported cases.
4. **Compile overview:** A table showing the essential characteristics of all reported cases was prepared for comparing the cases and for identifying their commonalities and individualities.
5. **Exploration of individual cases:** The cases were discussed and explored further w.r.t. design decisions, encountered challenges, open issues, and feedback from users. The findings were the basis for deriving a list of advantages, disadvantages, and lessons learned for each case.
6. **Synthesis of findings:** The findings from the individual cases were aggregated to high-level advantages and disadvantages as well as general lessons learned for presentation in the paper.

Several measures were taken to mitigate threats to validity. We decided to select multiple cases with different characteristics to support generalization. The information about the cases was provided by authors who were personally involved. To reduce the resulting bias, we used a shared template for preparing case descriptions, which were then reviewed and discussed with authors not directly involved in the cases.

Nevertheless, analyzing cases does not always allow to accurately identify the boundary between the observed phenomenon and the context [19]. The advantages, disadvantages and lessons learned we identified in our work may therefore

still depend on influence factors rooted in the specific project or application context. To counter this threat, we decided to put the primary focus of this paper on the presentation of the individual cases to support the reader in transferring the insights and experiences to his/her own context.

4 Description of Cases

The five presented cases (*Case 1–5*) cover a variety of different applications scenarios related to the analysis of source code and related artifacts. Each of them is based on a unique goal and motivation, derived from its application in industry projects[3] or research. The common theme shared by all cases is the implementation of tool support and services for which graph databases have been applied. Table 1 provides an overview of the key characteristics of the five cases, summarizing the spectrum of different applications realized with graph databases.

In the subsections below, the descriptions of the cases are based on the following structure.

- **Project context:** In what application scenarios is the graph database used?
- **Data model:** What data model is used to represent the structure and relationships of the analyzed source code or software engineering artifacts?
- **Data sources:** How is the data created and imported into the database?
- **Access and usage:** How is the data in the database accessed and used?
- **Status and ongoing work:** What is the current state of the work and what are the next steps?

4.1 Case 1: AutoDoc for Lightweight Dependency Analysis

AutoDoc is a lightweight and flexible dependency analysis tool based on static code analysis for various programming languages.

Project Context: The tool has been developed in a project with industry partners from the domain of embedded systems. These systems have to fulfill high quality demands. The project supported the industry partners in improving software quality by providing unit testing and source code analysis technology. AutoDoc statically analyzes the source code of the software system and generates information about components and their interrelations. Furthermore, a set of code-related metrics is calculated including Halstead, McCabe complexity, and the Microsoft maintainability index. The results are used by developers in various ways, for example, to gain an overview of interdependencies between the units of a software system, to determine areas for refactoring and testability improvements, or to estimate the impact of architectural changes. By using a language agnostic parsing frontend, the tool is applicable for systems implemented in various different programming languages; currently, it is used to analyze systems implemented in C.

[3] If not already revealed in previous publications, details about involved industry partners have been omitted due to confidentiality obligations.

Table 1. Overview of the Reported Cases.

	AutoDoc	SCoRe	eKNOWS CMS	Sherlock	Gradient
Goal and motivation	Lightweight, flexible, customizable code analysis	Analyzing industrial PLC software	Service for reusable static code analysis	Dependency analysis for regression test selection	Probabilistic software modeling
Application context	Industry projects	Industry projects	Industry projects	Industry projects	Research prototype
Represented source code or artifacts	Call, read, and write dependencies in various languages (currently C)	PLC programs written in IEC 61131-3 languages	Java systems (from high-level config to source code statements)	Source code structure (C++, C#), code changes, system tests, test coverage	Static code structure and models of behavior (Java VM languages)
Users	Developers	Developers, software architects	Developers of software engineering tools and services	Software testers, quality and release managers	Researchers, developers
Usage scenarios	Analysis of dependencies enriched with metrics	Design and architecture review, support for refactoring	Analysis of large-scale systems, evolution analysis	Selecting regression tests for source code changes	Program comprehen-sion and behavioral analysis
Data access and user interfaces	Neo4j Web interface	Neo4j Web interface	Different (3rd party) tools, REST API	Custom client, export to test tool	Custom Web interface
Data sources	Source code files	Source code files	Version control systems (VCS)	VCS, task management, coverage analysis, test management	Source code, execution traces from run-time
Schema size (distinct elements)	2 node types, 6 relationship types, 9 metric values	24 node types, 15 relationship types	83 node types, 88 relationship types	9 node types, 15 relationship types	11 node types, 20 relationship types
Size of analyzed system	72,000 LOC C code	742,000 LOC IEC61131-3	44 million LOC Java	2.5 million LOC C++ and C#	n.a.
Data import/update strategy	Import, existing data replaced	Bulk import, existing data replaced	Cyclic builds, data partially updated	Nightly and manual updates, data partially replaced	Triggered builds with incremental updates

Data Model: AutoDoc stores dependency analysis results as graph into the Neo4j database. The data model is depicted in Fig. 1. It models dependencies due to function calls and field access. The nodes are of type `function` or `field`. Relations of type `calls` exist between functions; `reads`, `writes`, `read_writes`, or `indirect_writes` relations between functions and fields. The more abstract indirect_writes relations show that a member of a field is written by a function. Metrics values, e.g., complexity measures, are stored as attributes for function nodes.

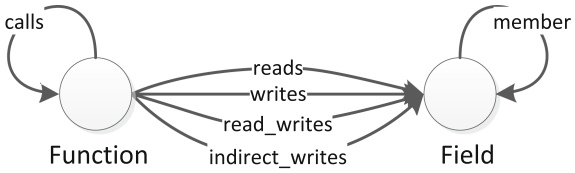

Fig. 1. Data model of AutoDoc.

Data Sources: The foundation of the tool is the general-purpose code analysis platform part of SCCH's eKNOWS tool family [7], which is used in different projects in the field of static code analysis, domain knowledge extraction, and re-documentation. The platform provides parser frontends for numerous programming languages to transform the code into the Generic Abstract Syntax Tree Model (GASTM). The generic representation allows to make downstream transformations and additional analyses available for any of the supported programming languages. AutoDoc utilizes the C code parser frontend of eKNOWS to transform the code into its GASTM representation (Fig. 2). Call graph and dependencies are calculated from this model. The resulting graph is stored in the Neo4j graph database. Analysis runs are triggered either manually on demand or periodically by integrating the tool into the build process.

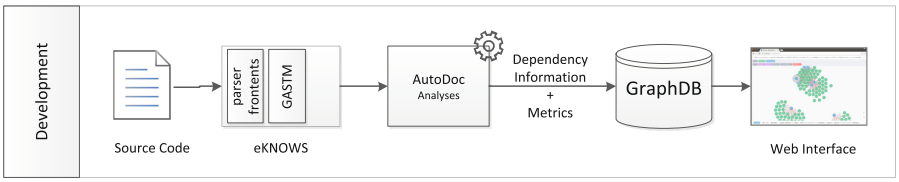

Fig. 2. System overview AutoDoc.

Access and Usage: The data stored in the graph database is accessed via the standard Neo4j Web interface, which supports users in exploring the analysis results by writing queries or executing previously saved queries. We provide predefined queries to answer common questions with respect to system components, dependencies, and accessed variables, for example, *"How often are global*

variables accessed?", *"What global variables are written by different functions?"*, *"What functions access many different global variables?"*, *"What are the most complex functions?"*, or *"Which functions have a high maintainability index?"*. The typical target audience of AutoDoc are developers performing a detailed analysis of the system under development. We found that these users are comfortable with formulating ad-hoc queries using the easy to understand Cypher query language.

Status and Ongoing Work: AutoDoc is currently used by our industry partners in two main ways, first, for exploring and refactoring of legacy code and, second, for continuous code quality assurance. While the Neo4j browser seems sufficient for the first use case, the generation of an analysis report is considered for the later. The currently analyzed software systems contain up to 72,000 lines of embedded C code. Nevertheless, it is planned to expand the application of AutoDoc to analyzing a system of systems with more than a million lines of code.

4.2 Case 2: SCoRe for PLC Programs

SCoRe is a static analysis tool developed explicitly for analyzing PLC programs written in IEC 61131-3 programming languages for industrial automation and production systems.

Project Context: The programming languages defined by the IEC 61131-3 standard [11] are used in industry to implement the control software of real-time systems. The software runs on dedicated hardware, i.e., programmable logic controllers (PLCs). Due to the focus on a relatively small niche, these software systems have received little attention in the past. Only a few software engineering tools are available, mainly proprietary programming environments tied to PLCs of specific vendors. However, the size and complexity of today's industrial control systems increased the demand for additional tools supporting quality assurance, testing, software architecture, and design tasks.

Together with our industry partners, we developed a tool for automated static code analysis of large-scale PLC programs (c.f. [2,15,16]). The tool *SCoRe* (for *Source Code Review*) supports detecting a range of problematic code constructs, violations of programming conventions, and potential defects. In addition, we implemented support for analyzing the software design and architecture of control systems by exporting the structure and dependency information extracted in static code analysis available to the graph database Neo4j. This solution allows exploring and examining the various program elements and their dependencies via custom queries and by browsing the graph visualization in the Neo4j Web interface. Figure 3 provides an overview of the tool chain. It distinguishes the tool's application in *implementation* for automated static code analysis based on a set of predefined rules from *design and architecture analysis*, which requires support for interactively exploring and reviewing the software system.

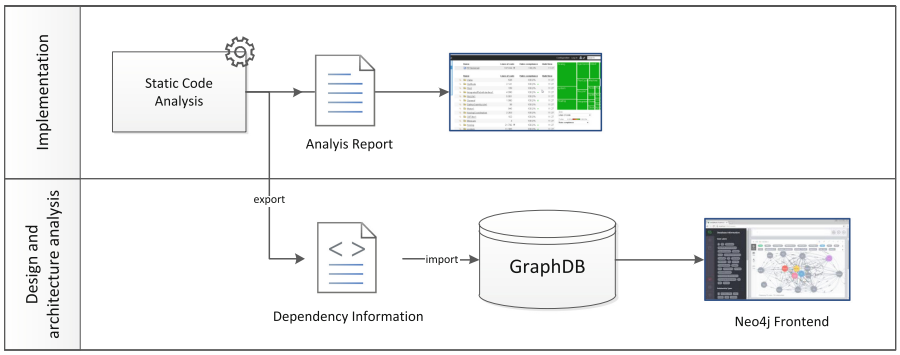

Fig. 3. Overview and usage of SCoRe.

Data Model: The Neo4j database contains 24 distinct node types that represent the various building blocks commonly used in PLC programs [11]. They are structured into `functional units` that contain `program units` (e.g., function blocks or functions) and define `system variables`. Program units can `call` other program units, `read` and `write` system variables, and to `send events` or `listen for events`. Instances of program units are assigned to process `tasks` for execution in cyclic intervals. Figure 4 provides an overview of the most important building blocks (nodes) and their dependencies (relations). Nodes can contain additional attributes such as the path to the source code files or optional metric values (e.g., number of imports, or fan-in and fan-out).

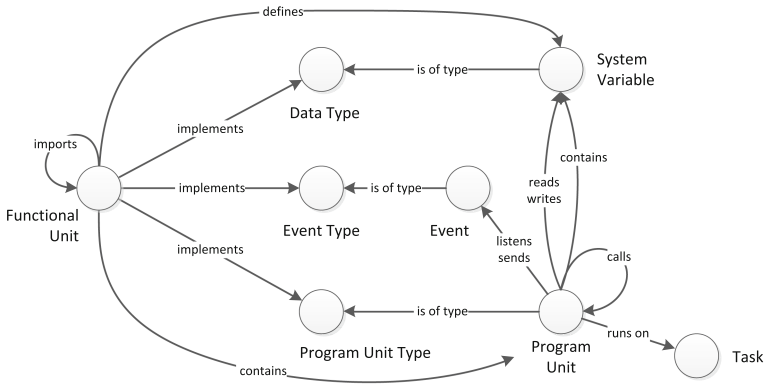

Fig. 4. Data model of SCoRe.

Data Sources: The SCoRe tool parses the PLC program, constructs the abstract syntax tree, the control flow graph, and the data flow graph. The predefined rules are used to detect issues in the source code, which are then listed

as report and in SonarQube's quality dashboard. As an add-on, the tool also exports the information about program elements and dependencies to CSV files, which can be imported into the Neo4j graph database. The imported files represent the snapshot of the system at the analysis time. The database is wiped before an import, as updates or incremental changes are currently not supported.

However, the data in the database can be enhanced after the import by computing additional attributes and relations not present in the initial dataset. Metric values are a typical example. They are computed with queries on imported data, and the results are inserted as additional attributes to existing nodes. In that way, additional abstraction layers can be created, which can again be used in querying and exploring the system.

Access and Usage: The generic Neo4j Web frontend is used to for submitting custom Cypher queries and interactive exploration of result graphs. It can be easily accessed by all members of the development team as only a Web browser is required. Together with our industry partners we identified the following application scenarios for SCoRe: Evaluating the compliance of the implementation with design decisions and guidelines (e.g., use of global variables), computing metrics (e.g., coupling and cohesion), support for refactoring (e.g., identifying large program units), and analyzing the potential impact of changes (e.g., dependencies on changed elements).

Status and Ongoing Work: The tool SCoRe is currently applied by two industry partners developing industrial automation systems. It has been used to analyze systems up to 742 KLOC implemented in the IEC 61131-3 programming languages, which resulted in about 450,000 nodes and 2,500,000 relationships in the graph database. The import into Neo4j required 21.4 seconds on a standard desktop computer. We are currently working on extending the tool to support the simultaneous analysis of IEC 61131-3 and C/C++ as our industry partners are also using a combination of both technologies in PLC programs.

4.3 Case 3: eKNOWS Code Model Service

The *eKNOWS Code Model Service* (eKNOWS CMS) is a service that provides reusable static code analysis functionality for Java programs via a dedicated REST API.

Project Context: eKNOWS CMS has been developed as the foundation of a microservice-based system for extracting architectural information from large-scale service-oriented software systems via static code analysis [6]. The system has been developed in close cooperation with Raiffeisen Software GmbH (RSG), a provider of IT solutions for the finance domain in Austria.

System Overview: Figure 5 depicts an overview of the eKNOWS CMS. As shown in the figure, eKNOWS CMS is implemented as a microservice that provides static code analysis functionality to a set of other microservices and tools. These analyses are provided via a dedicated Representational State Transfer (REST) API. Static code analysis in the eKNOWS CMS differs from many other code analysis approaches where analysis is performed via abstract syntax tree (AST) visitors for deriving information from the system implementation. Instead, we have implemented static code analysis by means of Cypher queries. We provide the following kinds of analyses:

- Search for type and interface declarations of a specified module.
- Search for type, field, and method declarations with specified metadata.
- Search for extended types and implemented interfaces of a specified type declaration, search for all type declarations derived from a specified type, and search for all type declarations implementing a specified interface.
- Search for import relationships of a specified module, and search for modules importing a specified module.
- Search for method declarations of types and interfaces.
- Calculation of call graphs and caller graphs for specified method declarations
- Search for XML documents and elements and attributes of XML documents
- Search for MANIFEST files and their attributes
- Calculation of type dependency relationships.

On top of the eKNOWS CMS we have developed a set of services that use the provided code analysis functionality. These services then provide information to different tools used at RSG. A detailed description of the developed services and used tools can be found in [6].

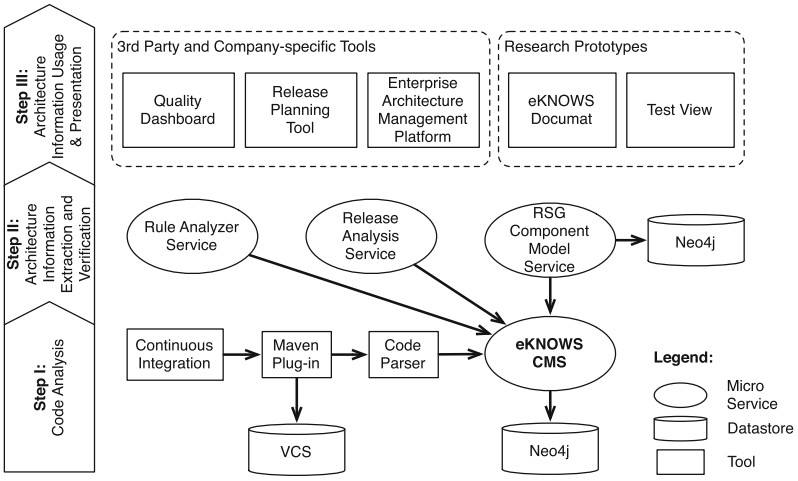

Fig. 5. eKNOWS code model service - System overview

Data Model: Figure 6 depicts an excerpt of the data model of the eKNOWS CMS. The data model consists of 83 different node and 88 relationship types that are used for storing complete `implementation artifacts` (i.e., source code, XML, and Manifest files) in Neo4j by converting these artifacts into graph structures. All implementation artifacts are assigned to a dedicated `module`, which is the unit of versioning and deployment. Modules define dependencies to other modules via `import relationships`. The eKNOWS CMS can store multiple `versions` of the system implementation in Neo4j, i.e., we store all released versions of a module along with the version of the current development iteration that is overwritten whenever a cyclic build process is triggered. Modules can be aggregated to `applications` to describe modular systems. Finally, we also store the results of resource-intensive analyses, i.e., call graphs and dependencies between types in Neo4j to avoid redundant analyses.

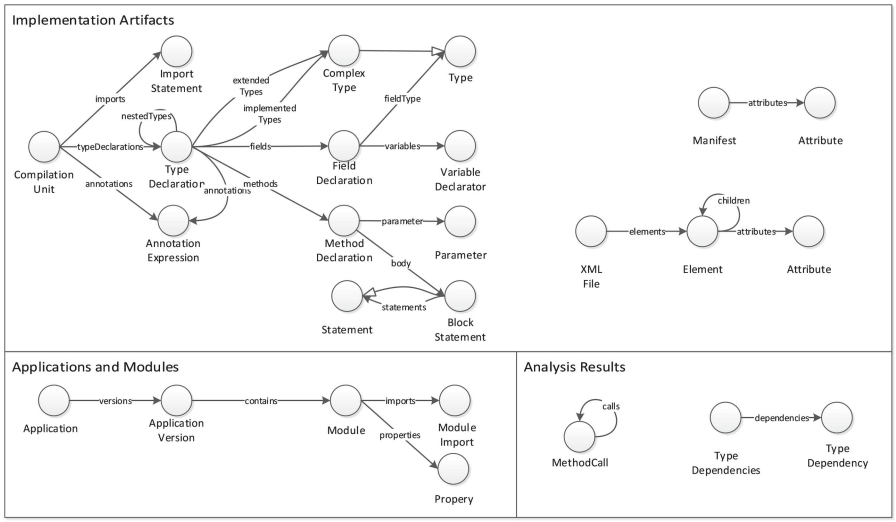

Fig. 6. eKNOWS CMS data model (Excerpt).

Data Sources: The eKNOWS CMS operates on implementation artifacts that are written to Neo4j as part of cyclic build processes. A dedicated Maven Plug-in fetches the system implementation from version control systems (VCS) and stores this data in Neo4j (see Fig. 5).

Status and Ongoing Work: eKNOWS CMS has been successfully evaluated in an industrial case study at RSG in which we have analyzed the entire codebase of RSG's latest online banking solution (see [6]). We have analyzed over 44 million lines of code, which were stored as 138,595,573 nodes and 138,141,947

relationships in Neo4j. The correlation between the number of nodes and relationships results from the fact that we store implementation structures 1:1 in the database without relationships between implementation artifacts. Such relationships are either calculated on demand (e.g., inheritance and implemented interfaces relationships), or they are stored as dedicated data structures (e.g., call graphs and type dependencies) where references are defined via node IDs.

Currently, we are working on supporting additional use cases of stakeholders at RSG by providing corresponding architecture information using the analysis functionality of eKNOWS CMS. We will further extend the eKNOWS CMS with additional kinds of analyses to improve our support for automated generation of viewpoint-based software architecture documentation.

4.4 Case 4: Sherlock for Regression Test Case Selection

Sherlock is a tool that supports regression test case selection in manual system testing based on test coverage and code changes.

Project Context: Regression testing [23] is performed after making changes to an existing software system to ensure that these changes do not have unexpected adverse side effects on the behavior of existing, unchanged parts of the software system. The straightforward approach to regression testing is to re-execute all existing test cases to make sure they still pass. However, many software projects have a large number of test cases, and it is often impossible to re-execute all of them every time a change has been made. Regression test case selection aims at selecting a reasonably small subset of the existing test cases, which still has a high chance of detecting any issues introduced by changes.

We developed the tool *Sherlock* for selecting regression test cases based on a list of locations in the source code where changes have been made and the information which test cases cover these source code locations [5]. Sherlock specifically supports interactively selecting test cases for manual regression testing in the context of a large-scale software product by OMICRON electronics GmbH. This software product encompasses more than 30 modules (about 2.5 MLOC in total, mostly implemented in C++) that interact with each other and share a common framework as well as various base libraries and hardware drivers. The system has grown to its current size over a time span of more than two decades. Engineers in different roles (i.e., developers, architects) have contributed over time, creating a large and sophisticated software system with complex dependencies between application modules, framework components, custom interfaces, and various third-party libraries. Thus, today, one of the foremost challenges of effective and efficient regression testing lies in acquiring and managing the knowledge about the huge amount of dependencies in the software system.

Data Sources: As regression testing is a time-consuming activity [12], Sherlock helps to reduce required efforts and costs by concentrating on those tests, which exercise the parts of the system that are affected by changes. For identifying and

selecting the relevant test cases, Sherlock incorporates information from three data sources (as shown in Fig. 7):

1. Information about source code and changes (e.g., check-ins) is extracted from the *version control system* of Microsoft's Team Foundation Server (TFS).
2. The list of available test cases and their properties are retrieved from the *test management system* SilkCentral Test Manager.
3. The relationship between test cases and code changes is determined from coverage analysis results produced by the *profiler* SmartBear AQtime Pro.

Custom implemented adapters based on Neo4jClient binding for .NET are used to extract, transform and load (via a bulk Cypher import) this information into a *Neo4j graph database* that acts as Sherlock's central data store.

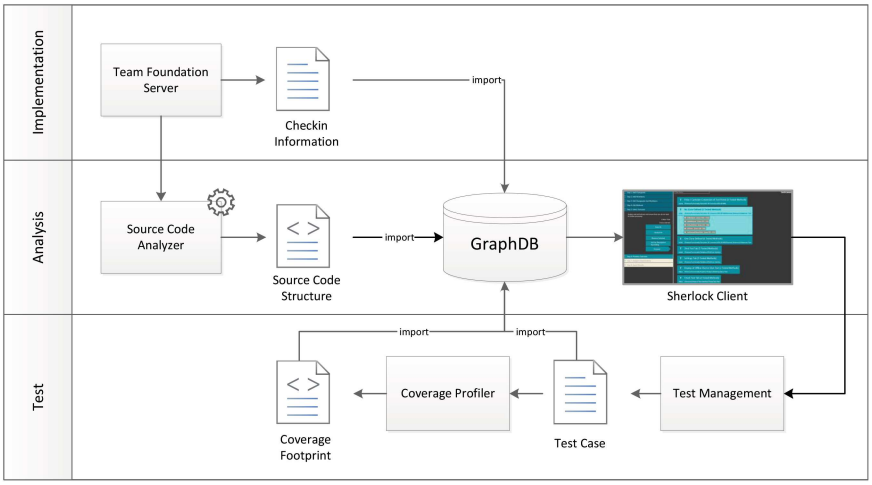

Fig. 7. Sherlock system overview.

Data Model: Sherlock stores the data in a tool-agnostic graph format (Fig. 8). Central information in the Sherlock data model are nodes holding information about all `methods` in the system under test, including details such as name or line numbers stored as attributes. Methods are grouped by `files` and are part of one or more `change-sets` (check-ins performed by a `developer`). A change-set in TFS may be connected to a `work item` that contains either a task, bug, or feature description, and which is assigned to a `software release`. A method is furthermore connected to a `regression test case` if it is part of the test's coverage footprint, i.e., if the method is called during the execution of this test case.

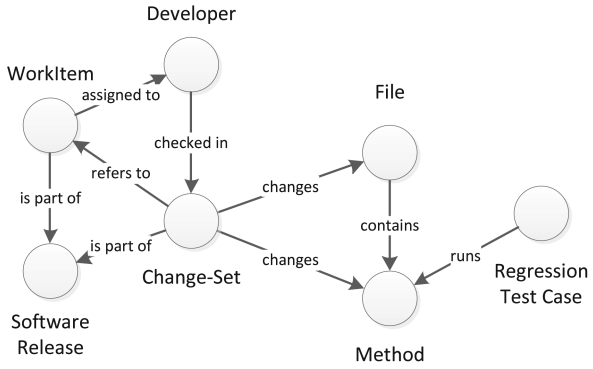

Fig. 8. Sherlock data model.

Access and Usage: Sherlock supports the testers in selecting the minimal set of regression tests to target the changes made by the developers. Therefore the testers access the aggregated information stored in the graph database using a rich client application. For compiling a regression test suite, the client application implements several Cypher queries so that testers can identify test cases related to an individual change (e.g., a bug fix), for all changes within a specified time range (e.g., all fixes and enhancements combined in a maintenance release), or all changes made on a branch before it is merged back into the trunk (e.g., all changes made while implementing a new feature). The resulting set of test cases is the basis for subsequent regression test runs scheduled from within the test management system. Sherlock provides an export interface to update the test plans in the test management system accordingly. In addition, the tool also supports the analysis of the available test cases in general. For example, it indicates coverage gaps when changed source code is not covered by the set of selected test cases or any test at all.

Status and Ongoing Work: Initially, the project started with test case selection based on static code analysis results [5], but experiments showed that in this case, static analysis approaches are not able to reveal enough dependencies relevant for test case selection, e.g., because of multi-language interoperability or reflection mechanisms. For this reason, code coverage information was collected to expose dynamic dependencies as well. Today, check-in information and source code structure are kept up-to-date by a Windows service that runs on a daily basis. This service uses custom adapters to import 21,000 check-ins and more than 200,000 methods that are grouped in 20,000 files. Coverage foot-prints for currently 400 tests are imported and updated manually after a test case was successfully profiled. Sherlock is a valuable aid for providing guidance in selecting appropriate regression test cases for testers who lack detailed knowledge of the structure and dependencies of the system under test. In an evaluation, we found that a junior tester using Sherlock was able to produce test suites with less or

equal effort and at the same level of accuracy as highly experienced testers who accomplished the same tasks manually [17].

4.5 Case 5: Gradient for Probabilistic Software Modeling

Gradient is a Probabilistic Software Modeling (PSM) [20] system prototype that uses static and dynamic analysis to model the structure and behavior of a program.

Project Context: PSM systems allow engineers to inspect a program's structure (Types, Properties, Executables) *and* behavior (runtime objects) using statistical models. These statistical models can be used in applications such as visualization of runtime behavior (e.g., possible values of property `age` form a `Person` class), finding the most likely value combination of the parameters of an executable or test-case generation. The program structure is extracted via static code analysis while the behavior observations are extracted via dynamic code analysis. The static and dynamic information is then used to build a network of probabilistic models with similar behavior as the original program. The objective of Gradient is to empower software engineers with the possibility of behavioral analysis of programs without switching the level of abstraction (Types, Properties, Executables) or to content themselves with a single execution trace (e.g., debugging).

System Overview: Gradient leverages static and dynamic code analysis and builds a network of models that mirror the system under inspection. Naturally, it needs multiple stages, components, and technologies to work.

Figure 9 shows an overview of the Gradient system that is split into two parts, the client- and server-side, operating on three levels: Development, Runtime, and Modeling. First, ① the program structure is extracted from the *Source Code* and stored directly into the *Graph Database* (Neo4j) ②. Then the source code is compiled and patched with monitoring aspects that execute the monitoring logic. The *Patched Byte code* ④, containing additional monitoring logic, directly stores *Runtime Events* into the *Document Database* (MongoDB). This entire process is handled by the Gradient client which in addition reports (not shown in Fig. 9) the analysis progress to the Gradient server. This client is provided to users in the form of Gradle [10] (build tool) plugin that handles the entire tool-chain in a non-intrusive and transparent fashion. The Gradient Server than retrieves the structural and behavioral data from the databases and combines them into statistical models ⑥. The *Structure* of the program is retrieved from Neo4j along with the respective *Runtime Events* from MongoDB. The resulting models are stored back in the databases for later use where Neo4j stores the *Model Metadata* and MongoDB the raw model data. At last, the *Gradient Frontend*, hosted on the Gradient Server as a web application, can be used to access the statistical models to inspect the behavior of the analyzed program.

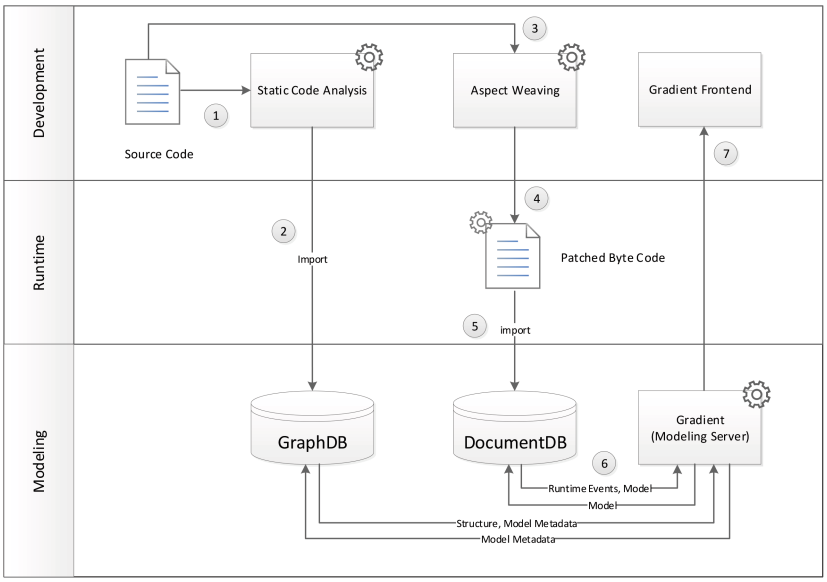

Fig. 9. Source code is statically analyzed and stored in the graph database. Furthermore, it is compiled, and the resulting byte code is patched with monitoring code that reports runtime events to the document database. Both, structure and runtime events are combined into a statistical model that is written back to the database for later analysis.

Data Model: Gradient uses a generic high-level data model that can be split into roughly three categories 1. *Code Elements*, 2. *Project Elements* and, 3. *Model Elements*, where the digest of it is shown in Fig. 10. Code Elements are related to the source code data model. Project Elements give code elements a project context and enable model versioning. Model Elements capture additional concepts related to the statistical models.

The *Code Elements* section in Fig. 10 shows that the Gradient data model has a higher level of abstraction than the traditional Abstract Syntax Tree (AST) as it only considers *Types, Properties, and Executables*. In compensation, the data model introduces *Invocation* and *Access* nodes as explicit relationship concepts that are only implicitly captured on a statement level in an AST. Another addition is the *ElementType* that allows direct access to typing information of typed elements, which cannot be straightforwardly retrieved from an AST.

Project Elements section contains *Project* nodes and *Version* nodes used to manage different projects registered on the same Gradient server. Type, Property, Executable are also *Versionables* containing a version hash that, along with their qualified name, uniquely identifies them within a project and its versions.

This also enables the database to reuse *Versionables*, along their associated statistical models, with the same qualified names and version hash across different versions. *Modeling Elements* attach model specific information to code

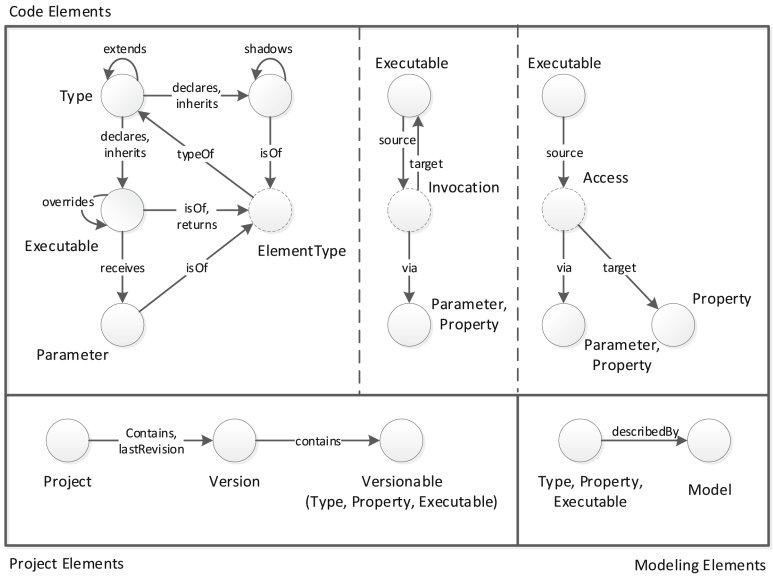

Fig. 10. The graph data model is split into 3 categories: Code Elements, Project Elements, Model Elements. Code Elements model source code concepts, Project Elements model project and versioning concepts, and Modeling Elements model concepts related to the statistical models built by Gradient.

elements that are modeled. For example, *Model* contains the id of the statistical model stored in the MongoDB.

Data Sources: Gradient has two sources of data, 1. Static code analysis on the source code, 2. Dynamic code analysis executed by the patched byte code. The static analysis parses the source code via Spoon [14] and transforms it into the Gradient model. This graph is then written into Neo4j providing the structure. The dynamic analysis is executed by the patched byte code that contains instructions to write monitoring events into the MongoDB. Neo4j can also be thought of an index database where each interaction starts by retrieving specific nodes that point to raw data in MongoDB. This raw data can be millions of runtime events, each being a JSON document or binary data of the statistical models.

Access and Usage: Gradient allows engineers to interact with their source code by inspecting the behavior of types, properties or executables. Neo4j in this setting is used as a persistent data structure of the source code that reflects the parts that are exposed to the user, and as index database for binary data stored in MongoDB. Engineers that use Gradient interact mainly via a graph in list or visual form with the statistical models, or via pre-configured queries and tasks that fully abstract the structure.

Status and Ongoing Work: Gradient is an ongoing research prototype for Java to demonstrate the feasibility of Probabilistic Software Modeling. It currently implements the static and dynamic analysis as most parts of the statistical modeling and simple views to view structure, models, and the raw data. High priority features for the future are tools for test-case generation, anomaly detection, and a frontend that allows simple interaction with the models for software developers unfamiliar with statistical modeling. Also, interactions between statistical models of different versions of the source systems are part of the future work.

5 Discussion

This section summarizes the insights and lessons learned from the five reported cases via a discussion of advantages and disadvantages of using graph databases. In each case, slightly different aspects of often the same advantage or disadvantage were observed. We therefore aggregated the individual findings collected from the different cases into high-level statements. For each of these statements, examples describing the experienced benefits and drawbacks are given, including references to the cases where they have been found.

5.1 Advantages

Graph databases (e.g., Neo4j in our case) are a suitable choice for *storing and querying the data extracted from source code and related artifacts.*

+ Graphs are a natural way to represent the manifold dependencies that are omnipresent in software systems. Working with dependencies has been an essential motivation and was often the central aspect of the tools we developed (*Case 1–5*).
+ Graph databases can handle data from large-scale software systems up to several millions of lines of code as demonstrated by *Case 3*. The limit of Graph databases rather lies in the type of data that has to be processed. For log-like execution data as in *Case 5* a document-centered database (e.g., MongoDB) is preferable; in this case, both databases were used in combination.

Graph databases provide excellent *support for rapid prototyping and exploring different options* for working with artifacts from software engineering. This advantage derives from NoSQL databases being schema-less and highly extensible.

+ Building tools on top of graph databases allow to start using them early, while still under development, and to advance the tools and the underlying data model whenever new requirements or usage scenarios are encountered (*Case 1–5*).
+ In *Case 4* the Neoclipse plugin for Eclipse has been used in a first prototype to demonstrate the integrating of the tool Sherlock into the development environment.

+ In all five cases presented above the implemented tools emerged out of research projects, where the initial versions of the tools were repeatedly revised and successively extended, e.g., to match the diverse needs of our industry partners (*Case 1–4*).

Graph query languages (e.g., Cypher in our case) provide a powerful and simple way to understand, retrieve, and manipulate graphs representing source code or related artifacts.

+ Structured processing of code and other artifacts was found to be easier when using the expressiveness of graph queries than with conventional, programmatic approaches (*Case 2* and *3*). For instance, searching the AST for all method call expressions of a particular method declaration can be achieved by a simple query instead of implementing an AST visitor that requires visiting all method call expressions of a compilation unit and determining for each expression if it belongs to the specified method declaration by checking all parent elements until a method declaration expression is found.
+ In *Case 1* and *Case 2*, the users of our tools were developers. For them, it was straightforward to write queries after a short introduction to the Cypher query language. Dozens of queries have been created so far, supporting a wide range of common analysis tasks. The queries are stored as scripts that can be easily shared and adapted to new analysis tasks.
+ The standard Neo4j Web frontend was used for querying the graph structure and to retrieve the required information about the analyzed software system. Therefore it was not necessary to develop a dedicated client or user interface in *Case 1* and *Case 2*.

5.2 Disadvantages

The *generic frontends* available for graph databases (e.g., Web-based Neo4j browser) are often not adequate for supporting end users in performing the specific tasks involved in the studied cases. Custom user interfaces had to be implemented for several of our tools.

– The standard Web interface of Neo4j provides a convenient way to submit queries to the database and to review the results using a visualization of the graph. Nevertheless, the dynamic visualization makes it difficult to maintain the overview when working with large result sets containing dependency data (*Case 1* and *Case 2*). Alternative clients (e.g., yFiles Neo4j Explorer) offer improved layouts and comfortable filtering, but the inherent weaknesses of a generic solution remain.
– Support for specific graphical representations cannot always be provided. In *Case 2*, for example, company partners suggested to display all program elements implemented in the same unit grouped using visual containers (e.g., boxes) representing these units. In contrast, a generic visualization will show the "implemented in" relationship as lines connecting each of the program elements with nodes representing the units.

- The generic user interface does not provide any guidance for users to perform tasks step by step, e.g., in selecting test cases based on a previous selection of a set of code changes (*Case 4*).
- Available frontends are usually restricted to explore data of only one graph database. It is not possible to connect data from two or more databases running in parallel or from an external data source, e.g., when combining the static structure of a software system with code changes (*Case 4*) or its dynamic behavior (*Case 5*).

Graph databases show a *lack of support for time series data*. This deficiency can be attributed to their specialization on graph data and the philosophy followed by many NoSQL databases, which is "do one thing and do it well". Neo4j, as we used it in our cases, does not offer any features specific for storing or querying time series data.

- In modeling software engineering artifacts, however, time-related dependencies play a major role due to the evolutionary and incremental approach in which software is developed. Thus, we had to develop graph models that can represent a specific combination of code, artifacts, and dependencies at a particular point in time, e.g., by relating them to dedicated nodes representing software releases or versions (*Case 3–5*).
- In *Case 4*, some of the nodes (e.g., work items) also contain timestamps as attributes, which were required to formulate queries with an additional `where` clause to retrieve all elements in a specific time span.
- In *Case 5*, log-like time series data from execution is stored in a separate database. The characteristic property of such data is the sequential ordering of the entries, which are recorded over time.
- Time points are also relevant in *Case 1* and *Case 2*. However, in these cases, the pragmatic solution was to store only a snapshot of the software system at a particular point in time in the graph database. For analyzing another snapshot, e.g., a new build or version, the entire data set has to be replaced. Managing the dependencies to builds, releases, versions, etc. is left to the users applying the tools.

6 Summary and Conclusions

In this paper, we described our experiences and lessons learned from building software analysis tools and services based on graph databases. We presented five different cases related to different application scenarios and project contexts. Each of the five cases (*Case 1–5*) is an example showing that graph databases can be effectively used for representing and analyzing source code and software engineering artifacts. The diversity as well as the size and complexity of the reported cases underpin this finding.

A *broad range of program elements* is stored in the form nodes and relationships in the graph databases. They range from dependencies such as function calls and read/write access to variables (*Case 1*) to the entire AST of large-scale

software systems (*Case 3*). Software written in *various programming languages* has been represented as graphs, e.g., C, C++, C#, Java, and IEC 61131-3 languages (*Case 1–5*). The resulting *size and complexity of the graph structures* range from only two distinct node types and 6 distinct relationship types (*Case 1*) to 83 node types and 88 different relationship types (*Case 3*).

The graph databases showed a high level of *scalability* when used for analyzing up to 44 million lines of code at the level of individual syntax elements. The resulting graph contained more than 138 million nodes and about the same amount of relationships, which were stored in the database (*Case 3*).

The graph models were used for capturing the data extracted from various *different artifacts*: Source code files (*Case 1–5*); system configurations in form of XML files (*Case 3*); work items, check-ins, software tests, and coverage information (*Case 4*); project and version information (*Case 3–5*).

The databases are populated in a single initial import replacing existing data (*Case 1–2*) or they are updated in increments partially extending the data in the database (*Case 3–5*). The various approaches for accessing the data include Neo4j's standard Web interface (*Case 1–2*), custom built client applications (*Case 3–5*), a REST API (*Case 3*), and an export interface (*Case 4*).

The insights and lessons learned we collected from using graph databases have been compiled into a list of advantages and disadvantages to support decisions in related and future applications.

The key *advantages*, relevant for choosing graph databases as storage option in software analysis, are related to the *versatility of the graph data model*. It was found suitable to represent all kind of structures and relations usually encountered in software systems. It can be used to represent dependencies between individual program elements as well as for links across technology boundaries. In addition, the flexibility and scalability of graph databases provide an ideal basis for *prototyping and evolving analysis solutions*. Finally, *specialized graph query languages* are a powerful yet easy to use means for traversing the huge amounts of nodes and relations required to represent large and complex software systems.

The identified *disadvantages* concern, first, the *limited usefulness of standard database frontends* for end users. The issue with highly generic clients such as the Neo4j browser and similar tools is not a limitation in their functionality. On the contrary, they provide too many options. Custom interfaces built for end users offer only a fraction of their functionality, but they are meaningful in context of a specific usage scenario. Second, graph databases provide *no support for time series data*. Although this is natural consequence of the specialization of graph databases, there is nevertheless the need to represent time-related aspects in all kind of data produced in software development processes.

In future we expect to see a rising number of projects using graph databases for source code analysis and related software engineering tasks. As contribution we plan to investigate ways to combine storage approaches specialized for graph and time series data for building a new tool and service infrastructure.

Acknowledgements. The research reported in this paper was supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry for Digital and Economic Affairs, and the Province of Upper Austria in the frame of the COMET center SCCH.

References

1. Alves, T.L., Hage, J., Rademaker, P.: A comparative study of code query technologies. In: 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM) 2011, pp. 145–154. IEEE (2011)
2. Angerer, F., Prähofer, H., Ramler, R., Grillenberger, F.: Points-to analysis of IEC 61131–3 programs: Implementation and application. In: IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA) 2013, pp. 1–8. IEEE (2013)
3. Angles, R.: A comparison of current graph database models. In: IEEE 28th International Conference on Data Engineering Workshops (ICDEW) 2012. pp. 171–177. IEEE (2012)
4. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. (CSUR) **40**(1), 1 (2008)
5. Buchgeher, G., Ernstbrunner, C., Ramler, R., Lusser, M.: Towards tool-support for test case selection in manual regression testing. In: IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2013, pp. 74–79. IEEE (2013)
6. Buchgeher, G., Weinreich, R., Huber, H.: A platform for the automated provisioning of architecture information for large-scale service-oriented software systems. In: European Conference on Software Architecture. Springer (2018) (to appear)
7. Fleck, G., Kirchmayr, W., Moser, M., Nocke, L., Pichler, J., Tober, R., Witlatschil, M.: Experience report on building ASTM based tools for multi-language reverse engineering. In:IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) 2016, vol. 1, pp. 683–687. IEEE (2016)
8. Goonetilleke, O., Meibusch, D., Barham, B.: Graph data management of evolving dependency graphs for multi-versioned codebases. In: IEEE International Conference on Software Maintenance and Evolution (ICSME) 2017, pp. 574–583. IEEE (2017)
9. Hawes, N., Barham, B., Cifuentes, C.: Frappé: Querying the Linux kernel dependency graph. In: Proceedings of the GRADES 2015, p. 4. ACM (2015)
10. Ikkink, H.K.: Gradle Dependency Management. Packt Publishing, Birmingham (2015)
11. John, K.H., Tiegelkamp, M.: IEC 61131–3: Programming Industrial Automation Systems. Concepts and Programming Languages, Requirements for Programming Systems Decision-making Aids. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12015-2
12. Juergens, E., Hummel, B., Deissenboeck, F., Feilkas, M., Schlogel, C., Wubbeke, A.: Regression test selection of manual system tests in practice. In: 15th European Conference on Software Maintenance and Reengineering, pp. 309–312, March 2011
13. Passos, L., Terra, R., Valente, M.T., Diniz, R., das Mendonca, N.C.: Static architecture-conformance checking: an illustrative overview. IEEE Softw. **27**(5), 82–89 (2010)
14. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: SPOON: A library for implementing analyses and transformations of Java source code. Softw. Pract. Exp. **46**(9), 1155–1179 (2015)

15. Prähofer, H., Angerer, F., Ramler, R., Grillenberger, F.: Static code analysis of iec 61131–3 programs: Comprehensive tool support and experiences from large-scale industrial application. IEEE Trans. Ind. Inform. **13**(1), 37–47 (2017)
16. Prähofer, H., Angerer, F., Ramler, R., Lacheiner, H., Grillenberger, F.: Opportunities and challenges of static code analysis of iec 61131–3 programs. In: IEEE 17th Conference on Emerging Technologies & Factory Automation (ETFA), pp. 1–8. IEEE (2012)
17. Ramler, R., Salomon, C., Buchgeher, G., Lusser, M.: Tool support for change-based regression testing: an industry experience report. In: Winkler, D., Biffl, S., Bergsmann, J. (eds.) SWQD 2017. LNBIP, vol. 269, pp. 133–152. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-49421-0_10
18. Robinson, I., Webber, J., Eifrem, E.: Graph Databases: New Opportunities for Connected Data. O'Reilly. Media Inc., Sebastopol (2015)
19. Runeson, P., Host, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering. Guidelines and Examples. Wiley, Hoboken (2012)
20. Thaller, H.: Probabilistic Software Modeling, Jun 2018. arXiv:1806.08942 [cs]
21. Urma, R.G., Mycroft, A.: Source-code queries with graph databases-with application to programming language usage and evolution. Sci. Comput. Program. **97**, 127–134 (2015)
22. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: IEEE Symposium on Security and Privacy (SP), pp. 590–604. IEEE (2014)
23. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. **22**(2), 67–120 (2012)
24. Zhang, T., Pan, M., Zhao, J., Yu, Y., Li, X.: An open framework for semantic code queries on heterogeneous repositories. In: International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 39–46. IEEE (2015)