EasyChair Preprint

№ 1662

Graph-Based Analysis and Visualization of Kieker Traces

Richard Müller and Matteo Fischer

October 14, 2019

Graph-Based Analysis and Visualization of Software Traces

Richard Müller
rmueller@wifa.uni-leipzig.de
Leipzig University, Leipzig, Germany

Matteo Fischer
Leipzig University, Leipzig, Germany

Abstract

Graphs are a suitable representation of software artifacts' data created during development and maintenance activities. Software traces monitored with Kieker are one example of such data. We present a jQAssistant plugin that scans event-based Kieker traces and stores them in a Neo4j graph database. This opens up new possibilities for analyzing and visualizing these traces with respect to application performance monitoring and architecture discovery. We illustrate the feasibility and usefulness of the plugin with the Bookstore application example.

1 Introduction

Software development and maintenance activities create different software artifacts, such as source code, test results, error reports, and traces. These data are usually stored in separate files. However, to analyze and visualize these heterogeneous data it is advantageous to have a unified data source. Therefore, we have developed an extensible open source stack [6]. As software data maps to a multivariate, compound, attributed, and time-dependent graph [2] this stack uses jQAssistant plugins that scan software artifacts and store them in a Neo4j graph database. On this basis, the structured and connected software data can be easily analyzed and visualized.

Kieker is a framework to monitor, analyze, and visualize software behavior [1]. It provides capabilities for event-based as well as state-based monitoring [3, p. 67]. It primarily supports the programming language Java but additionally offers adapters for .NET, Cobol, and Visual Basic 6. The framework comes up with several tools to inspect and analyze traces as well as to visualize them with static images of UML sequence diagrams, markov chains, dependency graphs, and trace timing diagrams. Furthermore, there are output writers that persistently save the traces at the file system or in a relational database. Currently, no writer stores the traces in a graph database.

The contribution of this paper is a jQAssistant plugin that scans event-based Kieker traces and stores them as a graph in a Neo4j database for further analysis and visualization to support application performance monitoring and architecture discovery. The plugin complements the existing Kieker tools and integrates with the open source stack for software analysis and visualization [6]. Hence, the graph query language Cypher can be used to inspect and analyze the traces as well as interactive visualizations instead of static images can be generated.

2 Technical Background

Next, the tools used with the Kieker plugin are introduced including Neo4j, Cypher, and jQAssistant.

2.1 Neo4j and Cypher

Neo4j[1] is a native graph database that is used to store, manage, and query large amounts of connected data. It models graph data with a *labeled property graph* [7, p. 15]. *Labels* are used to classify *nodes*. *Relationships* connect nodes and have a *type* and a *direction*. *Properties* are attributes of nodes and relationships. They are stored as *key-value pairs*.

Cypher[2] is the graph query language of Neo4j [5]. It matches given patterns in the graph using a visual, ASCII art-based syntax.

2.2 jQAssistant

The jQAssistant[3] framework is based on Neo4j and provides interfaces for scanner, rule, and report plugins [6]. Scanner plugins extract data from software artifacts and store them as a graphs in the Neo4j database. There are several scanner plugins[4], for example for Java source code, Jira, and GitHub-Issues. The scanned graph data can be analyzed by applying rules defined in Cypher. There are two types of rules. *Concepts* are used for data enrichment and *constraints* for detecting violations. Report plugins automatically generate a documentation based on the scanned and analyzed data. All jQAssistant plugins can be used with the build tool Apache Maven or executed independently from the command line.

3 Kieker Plugin

The flowchart in Figure 1 shows how the Kieker plugin works. Its complete implementation is available on GitHub[5].

[1] https://neo4j.com/
[2] https://www.opencypher.org
[3] https://jqassistant.org
[4] https://softvis-research.github.io/jqassistant-plugins/
[5] https://github.com/softvis-research/jqa-kieker-plugin

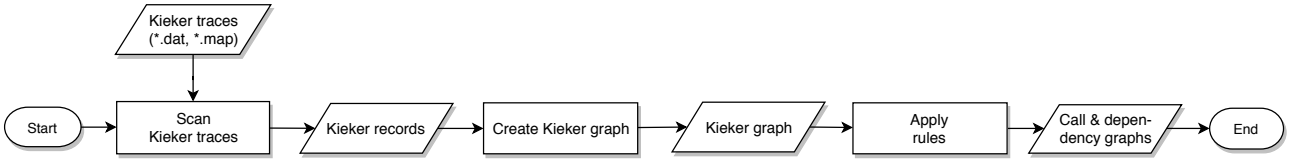

Figure 1: Flowchart of the Kieker plugin for jQAssistant

First, the `FSDirectoryReader` provided by the Kieker framework reads the traces. The interface `IMonitoringRecordReceiver` is used to get the records from the trace files. Currently, the record types `KiekerMetadataRecord`, `TraceMetadata`, `BeforeOperationEvent`, `AfterOperationEvent`, and `CallOperationEvent` are supported.

Second, the Kieker graph is created based on the schema depicted in Figure 2. Here, the records are mapped to the nodes `Record`, `Trace`, `Execution:Event`, and `Call:Event`. The records' attributes are mapped to the corresponding node properties. Then, the `CONTAINS` relationships are set for these records. Note that the two records `BeforeOperationEvent` and `AfterOperationEvent` are merged in one node labeled `Execution:Event`. Furthermore, the information about methods and types is extracted from the records resulting in the nodes `Method` and `Type` with their properties. Both node types are connected by a `DECLARES` relationship. The relationship from the event nodes to a method node is defined by `CALLED_BY` or `CALLS` if it is a call event and by `EXECUTES` if it is an execution event.

Third, rules in the form of concepts are applied to create the call graph with the relationship `CALLS` between methods and the dependency graph with the relationship `DEPENDS_ON` between types. Finally, another rule calculates the property `duration` of each method based on its related execution events.

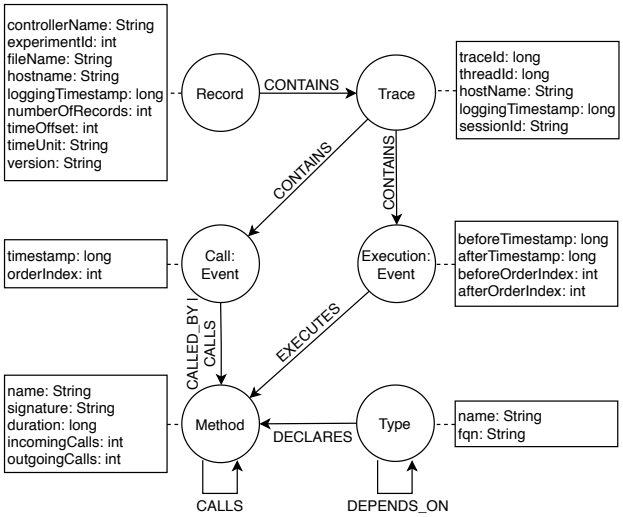

Figure 2: The Kieker graph schema

4 Application Example

As an application example, we refer to the Bookstore application described in the Kieker user guide [4, p. 9]. We have used AspectJ-based instrumentation where the monitoring probes are weaved into the byte code of the Bookstore application. We have activated the aspects `OperationExecution` and `OperationCall`. Then, we have scanned the monitored traces with the jQAssistant command line tool using the Kieker plugin. Based on the structured and connected software data we have analyzed and visualized the traces as explained subsequently.

4.1 Analysis

Cypher queries are useful to inspect and analyze the traces. Listing 1 contains an example query for aggregated method calls and Table 1 summarizes its result ordered by the method property `duration`. It is a similar view for aggregated method calls as provided by the Kieker trace diagnosis tool. It indicates that most time was spent in the `main` method and that the method `getBook` declared by the type `Catalog` was called 10 times.

Listing 1: Cypher query for aggregated method calls

```
MATCH (t:Type)-[:DECLARES]->(m:Method)
WHERE t.fqn STARTS WITH "kieker"
RETURN t.name as Type, m.name AS Method,
m.incomingCalls AS Calls, m.duration AS Duration
ORDER BY Duration DESC
```

Type	Method	Calls	Duration [ns]
BookstoreStarter	main	1	55498700
BookstoreStarter$1	run	5	33558300
Bookstore	searchBook	5	32389100
Catalog	getBook	10	30357600
CRM	getOffers	5	19180500
BookstoreStarter	spawnAsyncRequest	5	12639600
BookstoreStarter	extractNumRequestsFromArgs	1	1280600

Table 1: Result of the Cypher query in Listing 1

4.2 Visualization

The interactive call and dependency graphs are generated with the yFiles Neo4j explorer[6].

The call graph in Figure 3 shows all methods of the Bookstore application with their `CALLS` relationships. The property `duration` of each method node is mapped to a color gradient from green (short) to red (long).

[6] `https://www.yworks.com/neo4j-explorer/`

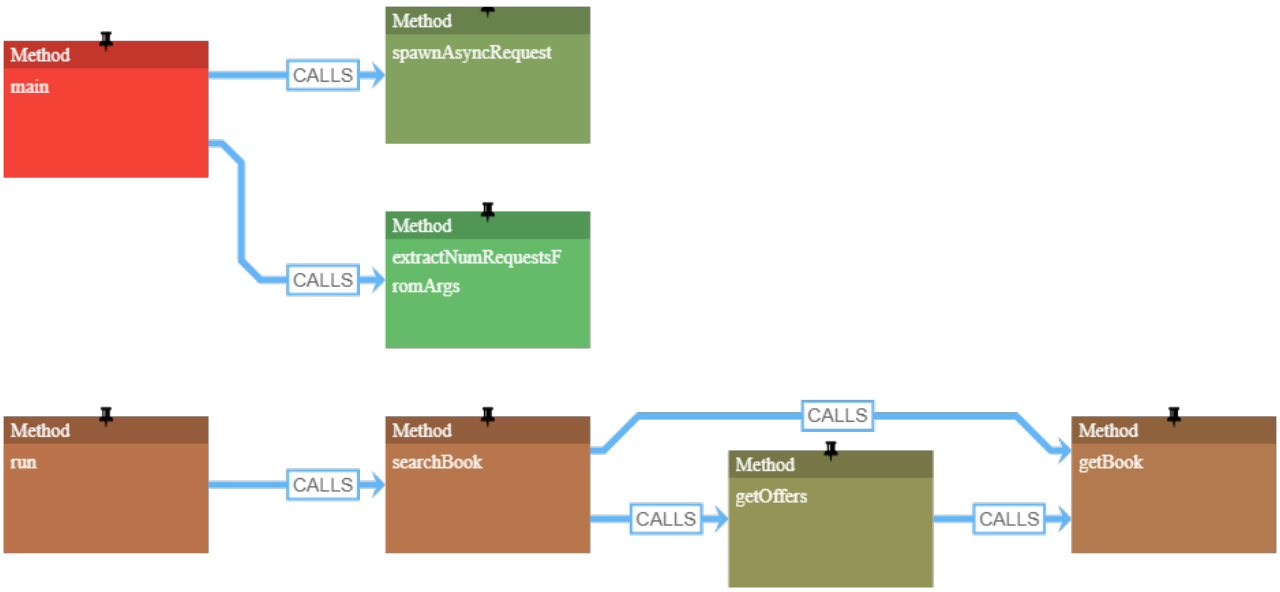

Figure 3: The call graph of the Bookstore application

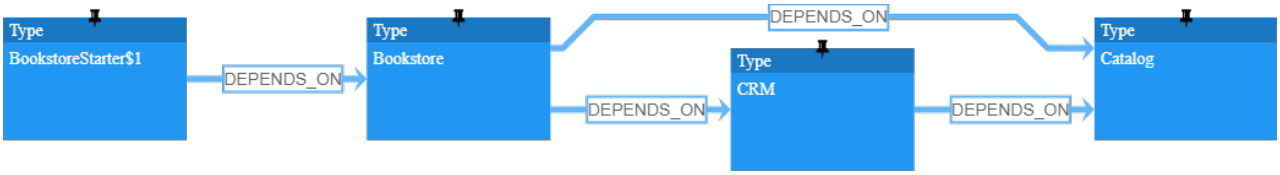

Figure 4: The dependency graph of the Bookstore application

The dependency graph in Figure 4 shows all types of the Bookstore application and their `DEPENDS_ON` relationships. The dependency is created based on method calls between different types.

5 Conclusion and Future Work

We have presented a Kieker plugin that scans event-based Kieker traces and stores them as a graph in a Neo4j database. Furthermore, we have illustrated the feasibility and usefulness of the plugin with an application example. We have analyzed the Bookstore traces with an example Cypher query for aggregated method calls and have visualized the call and dependency graphs with the yFiles Neo4j explorer.

For future work, we can imagine extending the plugin to scan further record types, for example, state-based records. Moreover, a special Kieker writer for graph databases could be developed as a contribution to the Kieker framework using this plugin as a blueprint.

References

[1] A. van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A framework for application performance monitoring and dynamic software analysis". In: *3rd ACM/SPEC Int. Conf. Perform. Eng. (ICPE 2012)*. ACM, 2012, pp. 247–248.

[2] S. Diehl and A. C. Telea. "Multivariate Graphs in Software Engineering". In: *Multivar. Netw. Vis. Dagstuhl Semin. #13201 Dagstuhl Castle, Ger. May 12-17, 2013 Revis. Discuss.* Ed. by A. Kerren, H. C. Purchase, and M. O. Ward. Vol. 8380. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014. Chap. 2, pp. 13–36.

[3] J. Waller. *Performance Benchmarking of Application Monitoring Frameworks*. Kiel Computer Science Series 2014/5. Department of Computer Science, Kiel University, 2014.

[4] Kieker Project. *Kieker 1.13 User Guide*. `http://kieker-monitoring.net/documentation/`. 2017.

[5] N. Francis et al. "Cypher: An Evolving Query Language for Property Graphs". In: *ACM SIGMOD Int. Conf. Manag. Data*. 2018, p. 13.

[6] R. Müller et al. "Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization". In: *Proc. 6th IEEE Work. Conf. Softw. Vis.* Madrid, Spain: IEEE, 2018.

[7] M. Needham and A. E. Hodler. *Graph Algorithms - Practical Examples in Apache Spark & Neo4j*. 1st ed. O'Reilly, 2019.