OpenTracing and Extraction of Architectural Information

Seminar "Advanced Software Engineering"

Marcus Rottschäfer

University of Stuttgart
Institute of Software Technology (ISTE)
70569 Stuttgart, Germany

Abstract. Traces in software systems are representations of the control flow of an execution, typically captured in the context of a dynamic program analysis. These can contain, next to other data, information about the method execution, timing, and the hierarchy of called methods. Information provided by execution traces may help application performance monitoring (APM) tools and software performance engineering (SPE) approaches to extract the software architecture of a system, measure the performance of an application or identify problems. Unfortunately, most APM tools rely on different tracing formats, which makes it difficult to achieve compatibility among different APM tools. The OpenTracing format tries to solve this by offering an open source tracing format that can be used throughout multiple concrete tracing implementations. One of these tracing implementations is given by the Kieker framework, a dynamic analysis tool enabling APM. Kieker can be used by the SLAstic framework, an approach for online capacity management in distributed software systems allowing continuous monitoring and analyzing, to enable the extraction of architectural information based on tracing data. This paper provides a comparison of OpenTracing with different tracing formats, including Kieker. We present a way to transform the OpenTracing format into Kieker, which allows to analyze traces from OpenTracing with the Kieker framework and the SLAstic framework.

1 Introduction

A common way of looking into software at runtime is offered by so called traces. A trace, in this case an execution trace, can be seen as a specific path going through an application. This path is determined by an execution of the application - depending on the current input and states. That said, traces generated from different executions do not necessarily have to be the same (and indeed are not the same in most cases). Normally, the execution itself does not provide much more information than the calculated result of the program. With tracing, however, the execution can give valuable information about multiple aspects of a system. Especially when regarding big software systems, tracing data can prove to be of high value for performance measurement, and by that, for business success [2].

The data collected by traces during execution typically originate from a variety of layers and locations. These can include data about executed methods, states, timing information about CPU, location data like hosts (for example in highly distributed systems), error messages, database information and many more [8]. In combination with strong analysis methods, tracing data can help for software performance improvement and even allows to extract architectural models from the observed system [3, 4, 14].

Dynamic analysis of programs is an approach to better understand the behavior of a system [1], for example by collecting traces. However, a second method of analyzing a program is available, the static analysis. This form of analysis focuses on gaining insight into a system by looking at the source code rather than executing the application. This approach can work well for procedural software systems, as stated by Hamou-Lhadj et al. [1], but fails for large object oriented systems, as polymorphism and dynamic binding begin to bring limitations to source code analysis. Due to this reason, we focus on dynamic analysis APM tools for collecting traces, like Kieker [15].

As stated by Heger et al. [2], modern development paradigms like DevOps with its frequent releases and new architectural styles like microservices and server-less architectures present new challenges to APM tools. Microservices for example provide extremely fine grained services, deployed independently among a high number of servers worldwide. The advantages are high scalability and separation of concerns, while increasing resilience and maintainability [7]. However, as applications developed with this architectural style are deployed on many different nodes, the need for cross-platform monitoring increases. This means that modern APM tools should allow for tracing in highly distributed environments in order to capture the control flow in, for example, microservice architectures.

There is another aspect to microservices that is relevant for APM tools. The different microservices that make up an application are normally written in different programming languages, also using different platforms and frameworks [7]. In order to still be able to monitor the application with traces, APM tools need a huge support for different languages and technologies.

All these requirements on APM tools for modern software need good monitoring frameworks for collecting traces and good analysis parts to analyze the traces and to extract the information needed to prevent performance-related incidents. One important part to achieve this is the APM tool interoperability [2]. In contrast to every APM tool having its own format, it could help if these tools share a common, open tracing format. Such open formats would help, for example, applications in gaining independence from concrete tracing implementations or prevent analysis approaches to be reimplemented for multiple APM tool formats.

Ongoing work in this field exists [2]. In this paper, three approaches will be introduced that try to achieve this interoperability, which are OpenTracing [10], the OPEN.xtrace format [8] and the Open Trace Format (OTF) [6].

The OpenTracing format offers a set of APIs allowing developers to abstract from a concrete tracing implementation [10]. The OpenTracing framework functions as a layer, encapsulating the application from the used tracing and logging framework. A more detailed description of OpenTracing can be found in Section 2.

Typically, APM tools collect more than just the names of method calls and their execution duration. As mentioned by Okanovic et al., information collected by these tools typically contains timing data (for responses, for CPU), data on variables, location data like the host and runtime environment, as well as information about possible exceptions and errors [8]. But these information can differ widely depending on which APM tool has been chosen [8]. In order to overcome this incompatibility among different APM tools and tracing formats, the OPEN.xtrace format has been proposed [8]. It works as an open and extensible format for representing execution traces. The format includes a default implementation and adapters for different APM tool formats. Details can be found in Section 2.

The aim of this paper is not only to compare different open data trace formats, but also to assess whether the OpenTracing framework [10] can be used for Kieker. The Kieker framework [14, 15] provides an open source APM tool that allows for monitoring of distributed software systems. It is possible to transform the tracing data collected in Kieker [14] into a representation in the OPEN.xtrace format, as stated in [8].

The rest of the paper is structured as follows. Section 2 is about the foundations, explaining the mentioned tracing formats and APM tools in more detail. Afterwards, in Section 3, a comparison of the different tracing formats will be presented, followed by Section 4 which gives insight into the extraction of architectural models from tracing data. Section 5 explains the results obtained by the comparison, leading to an answer whether OpenTracing offers a format that can be used by Kieker and SLAstic to extract architectural information or not. This section will also provide a small concept on how the transformation between the two formats can be achieved. Finally, Section 6 sums up the results and provides an outlook into future work.

2 Foundations

This section deals with the different tracing formats from multiple perspectives. Firstly there will be a deeper introduction to tracing in general, followed by introducing the aforementioned tracing formats and APM tools (OpenTracing project [10], OPEN.xtrace [8], OTF [6] and Kieker [14]).

Tracing can be seen as an activity of a dynamic analysis application. It consists of collecting one or multiple traces from executing the application one wants to observe. There are multiple descriptions of what exactly a trace is; however, to the best of our knowledge, no formal definition exists. For this paper, traces will be regarded as a form of representation of the control flow of an application. This representation is enriched with multiple raw data, collected during the execution of an application [8].

In the following, traces will be introduced in the context of different tracing formats. These tracing formats include OpenTracing [10], OPEN.xtrace[8] and the tracing used in the APM tool Kieker [14]. This introduction of the different formats lays the foundation for a comparison of the tracing formats in Section 3.

OpenTracing OpenTracing [10] can be seen as a collection of open-source APIs enabling a concrete tracing-implementation independent way of monitoring the performance of an application. When using OpenTracing , developers can change the concrete tracing implementation (for example Jaeger [5]) in constant time [10]. This means that when a different tracing implementation should be used as an APM tool, the developer does not need to change every line of code (correcting the tracer), but instead just needs another tracing tool to implement the OpenTracing interface.

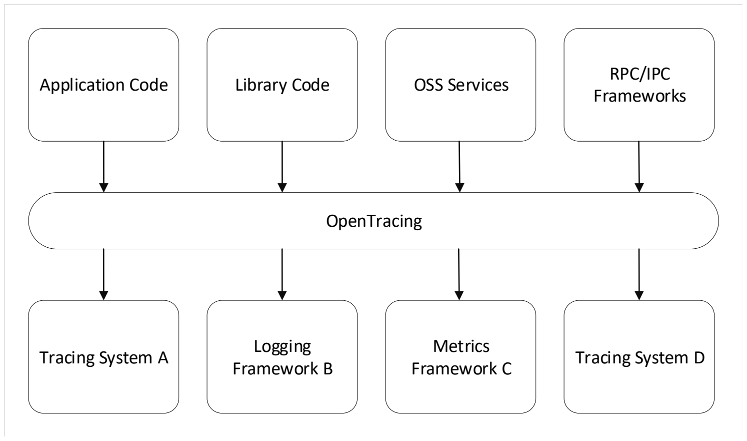

Fig. 1. The OpenTracing framework seen as a layer between the system and tracing implementations [9].

OpenTracing can be defined in a more general way, as stated in [9]: “Open-Tracing is a thin standardization layer that sits between application/library code and various systems that consume tracing and causality data”. This definition is visualized in Figure 1. It shows the OpenTracing framework as described in [9]. The frameworks builds a layer used by different concrete tracing and logging implementations. In order to connect, for example, some application code to a tracing or logging framework, the application code needs to use the OpenTracing API. It does not need to know about the concrete tracing or logging implementation, even if the implementation changes, which leads to more independence regarding the tracing and logging.

Traces in OpenTracing are seen as directed acyclic graphs (DAGs) that consist of different spans building their nodes [10]. A span is a logical representation of some work that is done by the application. It consists at least of an operation name, a start time and a finish time [11, 12]. The formal OpenTracing format specification [11] further provides additional features that a span should support. These are, for example, a set of span tags, which allow to store various information about the context of a span like database information, IP addresses, hostnames, errors, etc. in key-value pairs. The specification further provides so called span logs, a set of key-value pairs with timestamps for additional logging information. A span always has a span context, containing OpenTracing implementation-dependent span id's [11] to identify specific spans, and a set of baggage items, key-value pairs that can store information. To model traces in distributed systems, spans can have multiple children spans and parent spans, which are started either in serial or in parallel. The baggage items are bound to the references of a span to its children spans, thereby enabling all children of a span to access its baggage items. This is a powerful feature, because lower spans can access data from higher application levels, but is also a cost-intensive feature as the key-value pairs are copied into every child [11].

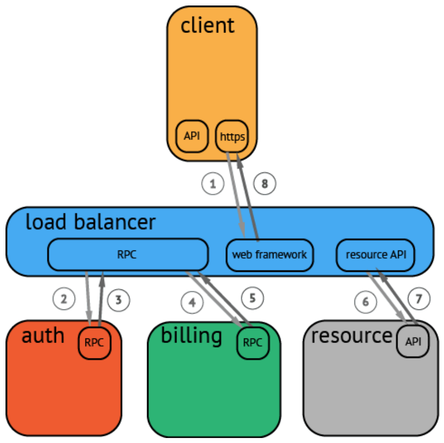

Fig. 2. Visualization of an OpenTracing sample trace [10].

Figure 2 shows a simple trace visualization taken from [10]. This trace consists of four method invocations or method calls. Each invocation or call corresponds to an edge connecting two spans. For the first invocation for example, a client makes a https-call to a web framework guarded by a load balancer (1). This span then starts multiple child spans, beginning with a remote procedure call to an authentication component (2). The visualization shows the DAG-like structure of a trace in OpenTracing but is not suited for visualizing parallelism or durations of calls/invocations.

```
import opentracing

tracer = opentracing.tracer

def say_hello(hello_to):
span = tracer.start_span('say-hello')
hello_str = 'Hello, {}!'.format(hello_to)
print(hello_str)
span.finish()
```

Listing 1: Sample script for OpenTracing, showing a trace with just one span [12].

The OpenTracing APIs are currently available in nine different programming languages [10]. Python is supported as well [12]. The small python script in Listing 1 shows a simple implementation of tracing a “hello-name” program. In this example, just one span is created, which is also the root span. The concrete tracing implementation (line 2, tracer = OpenTracing .tracer) can be changed to whatever tracer is preferred. Note that only the minimal attributes of a span are regarded: ”say-hello”(line 4) correlates to the operation name of the span, start time and finish time measurement can be found likewise.

OPEN.xtrace The OPEN.xtrace format is an open and extensible format for representing execution traces. It overcomes the problem of incompatibility between different tracing formats used by different APM tools. This is achieved by providing a meta-model for traces, as well as a default implementation of the format [8]. OPEN.xtrace further offers multiple adapters to transform in the most cases unique tracing formats of some APM tools into the OPEN.xtrace format. These adapters are publicly available [13], currently there are four APM tools supported.

The support also includes Kieker [15]. Its adapter is implemented as a Kieker plugin, allowing to export traces collected by Kieker into the OPEN.xtrace format while also enabling to import traces from OPEN.xtrace.

Open Trace Format The Open Trace Format (OTF) [6] provides a trace format library that is especially suited for High Performance Computing (HPC). As collecting traces in this environment requires some special properties like good storage efficiency, parallel access and a high access speed for the trace [6], OTF has some features making it well suited for parallel super computers.

For example, a trace in OTF is split up into multiple streams. Each stream is an independently accessible file; by this, different processes of the application can read and write in disjoint parts of the trace simultaneously.

Although being well suited for massive parallel applications, OTF can be used on single processor workstations as well [6]. The format is available under an open source license.

Kieker The Kieker framework [14], an APM tool enabling dynamic analysis of concurrent and distributed software systems, consists of two main parts. The first part, Kieker.Monitoring, is responsible for recording and tracking of the control flow of the application, while the second part, Kieker.Analysis, provides the possible mechanisms to analyze the collected data.

All the measurements obtained from Kieker.Monitoring are collected in the form of Monitoring Records. This abstract data structure comprises all data that has been recorded during a single measurement [14]. In order to enable analysis of the obtained Monitoring Records, said records are being serialized and written into a Monitoring Log/Stream. The records contained in the stream are then being deserialized in the Kieker.Analysis part of the application, allowing different analysis plugins to access the Monitoring Records.

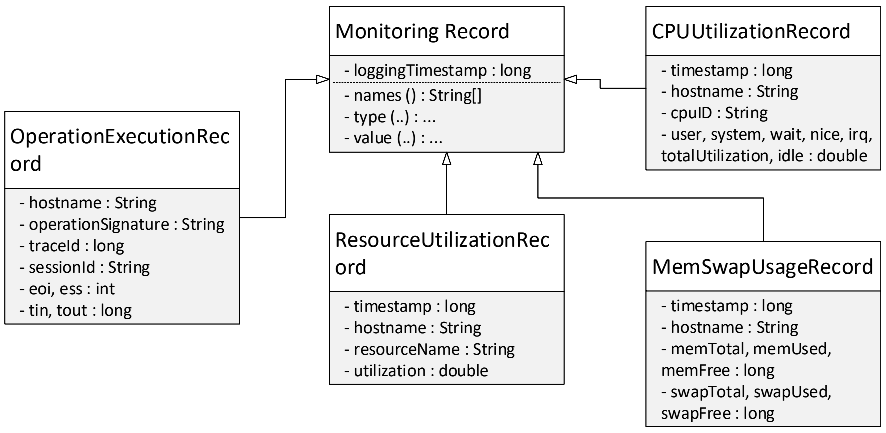

Fig. 3. Monitoring Record meta classes for storing measurements of resource utilization and control flow. Concrete example meta classes are derived from the Monitoring Record type [14, page 109].

Figure 3 shows both concrete examples and abstract meta classes of Monitoring Records. Types of meta classes are defined by extending from the abstract Monitoring Record meta class. Each Monitoring Record type has at least a timestamp storing the time that record has been logged [14]. The four concrete Monitoring Records can be further divided into records for resource utilization (CPUUtilizationRecord, MemSwapUsageRecord, ResourceUtilizationRecord) and records for control flow information (OperationExecutionRecord). Of special interest in our case is the OperationExecutionRecord, as this record stores information about the control flow of the software in form of executed software operations. This record allows the Kieker framework to collect execution traces.

Next to attributes concerning the location, like where the execution was performed (hostname) or which operation was executed (operationSignature), OperationExecutionRecord provides attributes specially suited for traces. These are, next to the storage of timestamps in nanosecond resolution (*tin*, *tout*) [14], the *EOI* and *ESS* values. Said values are needed for concurrent and distributed tracing. Without them, one would need to assume that the start of an execution (*tin*) and the end of an execution (*tout*) within different OperationExecutionsRecords of a trace will never have the same value, as well as that clocks in distributed systems are always running synchronized [14].

Of course, these assumptions can not hold in a real environment. Therefore, the *EOI* and *ESS* values have been introduced to be able to extract a trace from collected OperationExecutionRecords. The execution order index (*EOI*) of an OperationExecutionRecord denotes the number of execution (for this record) within the trace specified by the attribute traceId. It is unique among all other records within that trace.

The execution stack size (*ESS*) for a OperationExecutionRecord corresponds to the size of the execution stack as the execution of the recorded operation started [14]. It must not be unique in that trace.

3 Comparison of Tracing Formats

This section will provide a comparison of the tracing formats mentioned in Section 2. The goal of this comparison is to get an overview of the differences between the tracing formats.

On the first look, approaches to develop open trace formats and implementations like OpenTracing [10], the OPEN.xtrace format [8, 13] or the Open Trace Format (OTF) [6] seem to all aim for exactly the same goal: to create an open format for representing traces that can be used throughout multiple APM tools. This creates the question, whether the formats are somehow all equal, or if there is a good reason, why they can't be merged into one format.

Okanovic et al. [8] provide a small comparison of some open tracing formats, as well as a comparison of the OPEN.xtrace format with some APM tools. While searching for related work, they found the Open Trace Format (OTF [6]). It enables tracing on both standard workstations as well as on (high performance) super computers, with a focus on good performance for parallel I/O.

The OPEN.xtrace format is similar to OTF in regards of an open data format for traces, but it also differs in a detail. OPEN.xtrace is focused on representing execution traces, while OTF is about "[...] collection of arbitrary system events" [8], as stated by Okanovic et al., which makes both formats not redundant.

As stated in Section 2, the OpenTracing framework treats traces as graphs consisting of one or more spans. This representation is an abstraction of the method-wise tracing, because spans are representations of logical units of work [12]. This makes a difference in comparison with OPEN.xtrace, as this format represents traces consisting of method executions [8].

The comparison which is the most interesting in this context is between Kieker's traces and the OpenTracing format, as this comparison helps to find out whether traces in OpenTracing can be used in Kieker (and in the SLAstic framework) to extract architectural information, or not. As mentioned in Section 2, Kieker extracts traces from a set of OperationExecutionRecords obtained from a Monitoring Log/Stream. The structure of an OperationExecutionRecord is shown in Figure 3. This meta class provides some attributes that will be compared with the attributes of an OpenTracing span as specified in the official OpenTracing format specification [11].

OperationExecutionRecord attributes	**OpenTracing attributes**	**OpenTracing span category**
operationSignature: String	operation name: String	Span attribute
tin: long	start timestamp: long	Span attribute
tout: long	finish timestamp: long	Span attribute
sessionId : String	sessionId : String	Custom span tag
hostname: String	peer.hostname: String	Standard span tag
ESS : int	-	-
EOI: int	spanId: long	Span context
EOI: int	parentSpanId: long	References via Span context
traceId: long	traceId: long	Span context

Table 1. Juxtaposition between attributes of an OperationExecutionRecord from Kieker and a span from OpenTracing .

Table 1 shows a contrasting juxtaposition between attributes of an OperationExecutionRecord and a span in OpenTracing . It makes sense to compare the two types OperationExecutionRecord and span with another, as instances of both types are the parts that compose a trace in Kieker and OpenTracing , respectively [10, 14]. The juxtaposition shows which attributes of both types are alike; the hostname of an OperationExecutionRecord for example corresponds to the attribute peer.hostname, which is a standard span tag specified by the OpenTracing semantic specification [11]. Time measurement attributes like *tin* and *tout* [14] correspond to the start timestamp and finish timestamp of a span [10, 11] in the OpenTracing format. The operation signature of an OperationExecutionRecord can be described with the attribute operation name of a span; identifiers for traces and sessions can also be found in the OpenTracing format [11].

The first real difference between both types is created by the execution order index (*EOI*) and the execution stack size (*ESS*) values of the OperationExecutionRecords. The Kieker framework uses these values to model distributed traces [14]; in OpenTracing , however, there are no such attributes [11]. As one can see, the EOI value has an equivalent attribute in the OpenTracing format, which is the spanId. This correlation is explained in more detail in Section 5, where a concept is provided of how to transform a Kieker trace into an OpenTracing trace and vice versa.

Okanovic et al. provide a list of the data available in OPEN.xtrace in comparison with a selection of APM tools [8]. This selection includes Kieker; their comparison comprises 35 different attributes that are captured by the OPEN.xtrace format. According to the comparison, Kieker fulfills 12 of these attributes, although the authors state that especially Kieker has extension mechanisms that allow Kieker to collect measurements about more than the said 12 attributes [8]. In order to use the OpenTracing API in Kieker, OpenTracing must of course support the attributes that are already supported by Kieker.

It needs to be noted that the OpenTracing format provides more attributes than only those mentioned in Table 1. By using span tags, an arbitrary number of attributes can be measured and stored. The OpenTracing semantic convention [11] provides a list of 18 standard span tags. These tags are attributes that the OpenTracing API supports; the concept of span tags of course allows to further specify custom span tags, as for example done in Table 1 with sessionId. If there is no appropriate attribute in the OpenTracing standard span tag convention, one can create a custom span tag to measure and store a specific attribute.

4 Extraction of Architectural Information from Tracing Data

This section deals with the issue of how to extract architectural information about a software system from collected traces. As mentioned previously, dynamic analysis programs try to capture the behaviour of a software system [1] in order to gain insight into the programs properties. One important property of a software system is, of course, its architecture.

Traces collected about a software system are describing the behaviour of object interaction, for example in terms of method invocations and method calls. Analyzing these traces can therefore help to understand the connections and relationships among different entities of the system. Also, collected traces often contain additional data like location data. This allows tools to extract information about the architecture of a system from a set of collected traces. Still, one issue needs to be made clear: architecture models based on tracing data can only include the components and invocations that occur in the trace data itself. This means that sufficiently enough tracing data during the execution of the software needs to be collected.

Different techniques for extracting architectural models from tracing data exist. One approach has been proposed by Israr et al. [4] which is called SAMEtech. SAMEtech stands for *System Architecture and Model Extraction Technique* and provides a way to extract a so called Layered Architecture (LArch) model from traces. The LArch model is able to show interactions between architectural objects, as well as the type of interaction, that means whether it is synchronous, asynchronous or forwarding.

In contrast to, for example, the ANGIOTRACE approach [3], SAMEtech does not require a special format for the tracing data. It is able to create an architectural model from "conventional trace data" [4], meaning trace formats that are used by common tracing tools. Listing 2 shows the attributes of a generic trace format, which represents attributes used by common tracing tools and which works for the SAMEtech approach [4]. Note that in case of procedures or methods, *send* and *receive* correlate to the call and return.

```
[timestamp, \{send | receive | otherType\}, component identifier,
 message-information (optional)]
```

Listing 2: Textual representation of the attributes of a generic trace format, which can be used by the SAMEtech approach [4].

However, the SAMEtech approach has some limitations regarding the trace data. The send and receive timestamps must allow to refer to a single message, for example by globally ordered timestamps. If the process is concurrent and runs on multiple threads, each trace record must identify the thread. [4].

The underlying Interaction Tree Algorithm scales well for large numbers of trace events ("hundreds of thousands of trace events" [4]), but the component can't have internal concurrency for the model extraction to work well.

The rest of this section will look further into the creation of architectural models for the tracing formats introduced in Section 2.

Extraction of Architectural Models in Kieker Kieker is able to transform traces (collected as sets of OperationExecutionRecords) into special architectural views, for example UML sequence diagrams or dependency graph representations [14]. This is achieved by a Kieker.Analysis plugin, a Kieker visualization component. Figure 4 shows a dependency graph visualization generated with Kieker. Basis for this visualization was a bookstore application, analyzed in 1635 traces collected by Kieker.Monitoring [14]. The diagram shows the existence of different deployment-level entities and their relationship. The different deployment components are grouped into two execution containers, called *SRV0* and *SRV1*. Two entities are connected by an edge if the first entity uses a service provided by the second entity. Based on the collected traces, the edges can even be weighted according to the total number of times, an action of a providing service has been called.

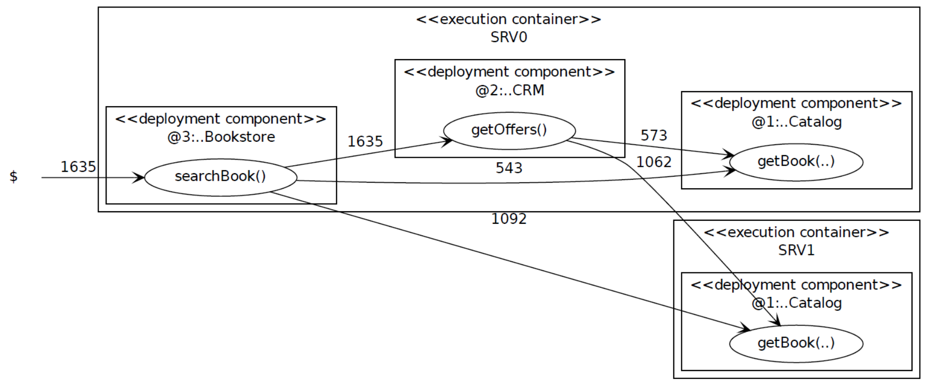

Fig. 4. Visualization of a deployment-level dependency graph, generated by Kieker from a bookstore application with 1635 traces [14].

As an adapter allows Kieker traces to be transformed into the OPEN.xtrace format (and the other way around) [14], execution traces in the format of OPEN.xtrace can be used in Kieker as well. Due to this, the visualization component will work for OPEN.xtrace traces, allowing Kieker and the SLAstic framework [14] to extract information about the architecture of the monitored application.

5 Transformation of OpenTracing into the Kieker Format

In this section, a concept will be developed of how to transform a trace in Kieker into a trace in OpenTracing (and the other way around). The basis for this concept is laid by the comparison of the tracing formats in Section 3.

The juxtaposition between different attributes of an OperationExecutionRecord and a span in OpenTracing (Table 1) revealed some features that are similar to both formats.

In order to transform one format into another, these likewise features need to be copied into the other format, respectively. These features comprise all the attributes in Table 1 that are having a similar attribute in the other format, namely the hostname, the operation name, trace- and session IDs, start- and finish measurements.

To create a trace in the Kieker format from an existing trace in the OpenTracing format, the set of spans that make up the OpenTracing trace must be iterated. For each span, an OperationExecutionRecord object has to be created, with its attributes being filled with the values of the corresponding OpenTracing attributes. To convert the formats vice versa, the procedure needs to be done the other way around.

As mentioned in Section 3, Table 1 comprises only a small part of all possible attributes. The given attributes for the Kieker format can be extended [8, 14], allowing to capture much more data than available in the table. OpenTracing allows for a similar extension of the collected attributes, enabled by its standard- and custom span tags [11]. The idea to transform these attributes as well is to create a custom span tag (or a standard span tag, see semantic convention [11]) for an attribute available in an extended OperationExecutionRecord.

With only the attributed mentioned so far, the requirements to distributed tracing would be the same as to Kieker without the EOI and ESS values: no two spans within a trace could have the same start timestamp or finish timestamp and clocks in a distributed system must run perfectly synchronized [14]. As stated by van Hoorn, these assumptions can not hold in a real environment [14].

This is the reason why there is a need to model the EOI and ESS values in OpenTracing as well. The EOI value of an OperationExecutionRecord correlates to the index of the execution of that specific record within the trace. This index is unique within the trace, as the execution counter is incremented with each executed OperationExecutionRecord. The EOI value can therefore be seen as an attribute uniquely identifying a record in a set of records that make up a trace. This is the reason why its equivalent in the OpenTracing format is the spanId (see Table 1), the integer value uniquely identifying a single span.

Another important attribute to the OpenTracing format is the set of references to and from other spans - parent and child spans. The references towards one or multiple other spans needs to be made via the spanIds of the parents and children. These references therefore have their equivalent in the EOI values of the referenced OperationExecutionRecords, as shown in Table 1.

The ESS value from the Kieker format does not have a corresponding attribute in the OpenTracing format. It counts the execution stack size for each OperationExecutionRecord during its execution. In order to model the ESS value with the OpenTracing format for a transformation, one must count the depth of the corresponding span in the DAG building the trace as seen from the root span. This measurement can then be used as the ESS value in the correlating OperationExecutionRecord.

For an implementation of this transformation of attributes in a distributed environment, it is important to know the *Inject* and *Extract* capabilities of OpenTracing . *Inject* and *Extract* are mechanisms provided by OpenTracing that enable “inter-process trace propagation without tightly coupling the programmer to a particular OpenTracing implementation” [10], as stated in the OpenTracing documentation. This means that when using the concept, a developer can propagate different spans through multiple processes of the application. The main idea is that a process can *Extract* a span context instance that is, for example, the child of a different span in a different process. The spans of course belong to the same trace. This whole concepts enables a concrete implementation-independent mechanism to support cross-process tracing, a key feature to distributed tracing.

6 Conclusion

Execution traces collected from (distributed) software systems can help to understand the interaction between different entities of the system and provide some of the data needed for APM tools. Approaches exist to extract specific information from a set of collected traces, like architectural models [4, 3, 14]. One specific APM tool we looked into a bit more deeply was the Kieker [14, 15] framework. As Kieker is currently using its own tracing format for representing traces, compatibility between Kieker and other APM tools is limited to some few adapters.

In order to overcome this dependency from a specific APM tool tracing format, for example when a developer wants to change the tracing framework that is used for monitoring a software system, open source tracing formats have been proposed. These formats offer an interface for the representation of traces that can be used by different APM tool. If multiple APM tools support such a format for traces, compatibility among these tools will increase and analysis applications will not need to be reimplemented for multiple tool.

In this paper we presented some open formats for representing traces. These formats were the Open Trace Format [6] introduced by Knuepfer et al., the OPEN.xtrace format [8, 13] and the OpenTracing format [10]. For the OPEN.-xtrace format, an adapter enabling transformation of an OPEN.xtrace-trace into the Kieker format already exists.

We would like to find out whether the OpenTracing format provides the attributes that are needed to transform an OpenTracing trace into a trace for the Kieker framework. The intention behind this was to evaluate whether the OpenTracing format can be used for the extraction of architectural models from the Kieker framework, or not. As a result, we provided a concept of how this transformation can be achieved. This concept showed that some attributes of both formats are already alike. Some attributes, however, are only available in one of each formats.

The concept further provided a way of how to overcome these differences in order to achieve a transformation between both tracing formats. Different attributes of an OpenTracing span need to be copied into the corresponding attributes of an OperationExecutionRecord.

To sum up the results, one can say that the transformation between a trace in OpenTracing and a trace in Kieker is possible. This transformation can theoretically be accomplished by converting the given spans into a set of OperationExecutionRecords, as described in detail in Section 5.

Future work could include a proof-of-concept implementation of the concept. This implementation would help to evaluate, if the concept works and can be used for a transformation of the OpenTracing format into the Kieker format.

References

[1] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

[2] C. Heger, A. van Hoorn, M. Mann, and D. Okanović. Application performance management: State of the art and challenges for the future. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 429–432. ACM, 2017.

[3] C. E. Hrischuk, C. M. Woodside, and J. A. Rolia. Trace-based load characterization for generating performance software models. *IEEE Transactions on Software Engineering*, 25(1):122–135, Jan 1999. ISSN 0098-5589.

[4] T. Israr, M. Woodside, and G. Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *J. Syst. Softw.*, 80(4):474–492, Apr. 2007. ISSN 0164-1212.

[5] jaeger.io. Jaeger: open source, end-to-end distributed tracing. `https://www.jaegertracing.io/`. Accessed: 2018-05-14.

[6] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the open trace format (otf). In *Proceedings of the 6th International Conference on Computational Science - Volume Part II*, ICCS'06, pages 526–533, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-34381-4, 978-3-540-34381-3.

[7] S. Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015.

[8] D. Okanović, A. van Hoorn, C. Heger, A. Wert, and S. Siegl. Towards performance tooling interoperability: An open format for representing execution traces. In *European Workshop on Performance Engineering*, pages 94–108. Springer, 2016.

[9] Opentracing.io. Common use cases. `http://opentracing.io/documentation/pages/instrumentation/common-use-cases.html`, . Accessed: 2018-05-26.

[10] Opentracing.io. Vendor-neutral APIs and instrumentation for distributed tracing. `http://opentracing.io/`, . Accessed: 2018-04-25.

[11] Opentracing.io. Concets and terminology. `http://opentracing.io/documentation/pages/spec.html`, . Accessed: 2018-05-26.

[12] Opentracing.io. opentracing-tutorial. `https://github.com/yurishkuro/opentracing-tutorial/tree/master/python/lesson01`, . Accessed: 2018-05-14.

[13] OPEN.XTRACE. OPEN|APM interoperability initiative. `https://spec-rgdevops.atlassian.net/wiki/spaces/OPENXTRACE/overview`. Accessed: 2018-05-15.

[14] A. van Hoorn. *Model-Driven Online Capacity Management for Component-Based Software Systems*. Number 2014/6 in Kiel Computer Science Series. Department of Computer Science, Kiel University, Kiel, Germany, 2014. ISBN 978-3-7357-5118-8. Dissertation, Faculty of Engineering, Kiel University.

[15] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 247–248. ACM, 2012.