A Generic Measurement Model for Service-based Systems

Andreas Hausotter, Arne Koschel
University of Applied Sciences & Arts Hannover
Faculty IV, Department of Computer Science,
Hannover, Germany
email: Andreas.Hausotter@hs-hannover.de
email: Arne.Koschel@hs-hannover.de

Johannes Busch, Malte Zuch
University of Applied Sciences & Arts Hannover
Faculty IV, Department of Computer Science,
Hannover, Germany
email: Johannes.Busch@stud.hs-hannover.de
email: Malte.Zuch@hs-hannover.de

Abstract—**The transfer of historically grown monolithic software architectures into modern service-oriented architectures creates a lot of loose coupling points. This can lead to an unforeseen system behavior and can significantly impede those continuous modernization processes, since it is not clear where bottlenecks in a system arise. It is therefore necessary to monitor such modernization processes with an adaptive monitoring concept in order to be able to correctly record and interpret unpredictable system dynamics. For this purpose, a general measurement methodology and a specific implementation concept are presented in this work.**

Keywords–Quality of Service; Indicator Measurement; XML-Model; Service-orientation; SOA

I. INTRODUCTION

The background of this work is cooperation with a partner of the German insurance industry and their IT-Architecture. Many IT-driven and data-driven companies face the challenge of continually modernizing their infrastructure, technologies, systems and processes. The insurance industry in particular is characterized by the fact that extensive digitization of processes took place very early. This was done well before researching modern service-based approaches, such as 'traditional' service-oriented architectures (SOA) or even microservices (MS) and without the use of distributed infrastructures such as cloud computing. Historically grown software monoliths were state of the art. The modernization of such monoliths in the direction of service-based architectures is a major challenge. This conversion process is the main motivation of this work and will be explained in more detail below.

A. Motivation

Systems cannot be abruptly switched off and replaced by new architectures, but must be continuously transformed into modern architectural forms. In this continuous modernization process, monolithic structures are broken down and distributed into services. This gives companies more agility and adaptability to changing business requirements. However, a decentralized and service-oriented system architecture is usually quite fine-granular and loosely coupled. Generally, this provokes an unpredictable dynamic system behavior. This also applies to our partner in the insurance industry. In order to remain competitive, the insurance industry has to respond quickly to customer information portals, such as check24.de, where different insurance companies competetively can offer, e.g., car insurances. This scenario motivates the need for a holistic measurement concept and defines the general application scenario of this work.

So there is a fundamental need for information about the system behavior. Relevant information is collected in the 'Information Product', which represents the output of the 'Core Measurement Process' (cf. Fig. 1). The 'Information Need' provides the input for the subprocess 'Plan the Measurement Process', the subprocess 'Perform the Measurement Process' generates the Information Product'. The process goal is to satisfy the 'Information Need'.

Nowadays it is normal that customers are demanding online services unpredictably and with high volatility. These volatile demands may lead to bottlenecks in distributed service-oriented architectures. Therefore, a reliable measurement of the whole system behavior is necessary in order to eliminate any bottlenecks. Such a measurement concept and its prototypical implementation are the core contributions of our work.

B. Contribution

In order to monitor individual system components with respect to time behavior, fixed time limits have so far been used. These fixed time limits are often used in historically grown software systems of the German insurance industry. If a system component (service) could not respond within these time limits, this was interpreted as a bad quality feature. However, with these static limits, a dynamic system behavior can be poorly monitored and interpreted. The challenge is to determine, when dynamic systems are overloaded. In this

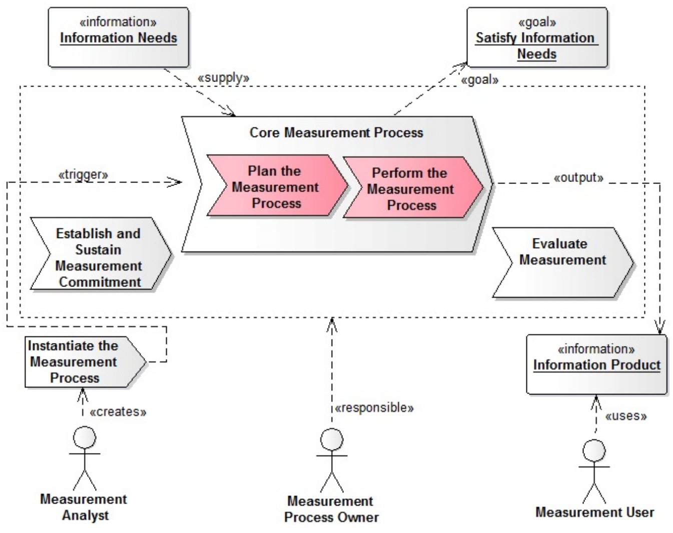

Figure 1. The Core Measurement Process

respect, a partner company of the insurance industry demands to integrate a metric, which could replace their static time limits in the future with a more dynamic metric. The general requirements lead to the following questions:

- How could static rules and timeouts be supplemented by a dynamic measurement metric?
- How could the measuring system be build on existing XML-Standards?

In previous work [1] [2], we have already developed a framework for dynamically measuring the service response time as a Quality of Service (QoS) Parameter within service-oriented architectures. As a significant enhancement to this previous work, our core contributions here are implementation details of the dynamic measuring system. Our measuring system considers existing XML standards and can flexibly record the load behavior of a software system. This measuring system should measure the response time as a particular QoS parameter as an example.

The measuring system should be able to consider both dynamic limits as well as static limits (optional). Normally, only the dynamic limits should be considered. But, if a service exceeds a fixed limit of, e.g., 5 seconds, then this should also be recognized. Another requirement is that the measuring system should 'inject' measurement agents into a software system as flexible and automated as possible. Implementation details hereof are also contributions of this article.

In Section II, related work concerning the topic of measurement models of service-based systems is explained. The measurement process with its core concepts and the information model are described in Section III and more implementation details in Section IV. Some mathematical equation explains the general measurement concept. After clarifying the general measurement plan, Section V shows how the planned measurement concept can be applied for detecting the so called 'Spikes', situations of high system-loads. The final Section VI will summarize this work. The different advantages and disadvantages of the described measurement model will be discussed. Also, an outlook to future work will show how the results of this work will be used in upcoming work in Section VI.

II. Prior and Related Work

In prior work, we already discussed several aspects of the combination of SOA, Business Process Management (BPM), Workflow Management Systems (WfMS), Business Rules Management (BRM), and Business Activity Monitoring (BAM) [3][4][5] as well as Distributed Event Monitoring and Distributed Event-Condition-Action (ECA) rule processing [6][7]. Building on this experience, we now address the area of QoS measurement for combined BRM, BPM, and SOA environments, mainly but not limited to, within the (German) insurance domain.

Work related to our research falls into several categories. We will discuss these categories in sequence.

General work on (event) monitoring has a long history (cf. [8][9] or the ACM DEBS conference series for overviews). Monitoring techniques in such (distributed) event based systems are well understood, thus such work can well contribute general monitoring principles to the work presented here. This also includes commercial solutions, such as the Dynatrace [10] system or open source monitoring software like, for example, the NAGIOS [11] solution. In these systems there is, however, generally no focus on QoS measurement within SOAs. Also, they usually do not take application domain specific requirements into account (as we do with the insurance domain).

Active Database Management Systems (ADBMS) offer some elements for use in our work (see [12][13] for overviews). Event monitoring techniques in ADBMSs are partially useful, but concentrate mostly on monitoring ADBMS internal events, and tend to neglect external and heterogeneous event sources. A major contribution of ADBMSs is their very well defined and proven semantics for definition and execution of Event-Condition-Action (ECA) rules. This leads to general classifications for parameters and options in ADBMS core functionality [13]. We may capture options that are relevant to event monitoring within parts of our general event model. QoS aspects are handled within ADBMS, for example, within the context of database transactions. However, since ADBMSs mostly do not concentrate on heterogeneity (and distribution), let alone SOAs, our research work extends into such directions.

The closest relationship to our research is the work, which directly combines the aspects QoS and SOA. As many as 2002 several articles fall into this category. However, in almost all known articles the SOA part focuses on WS-* technologies. This is in contrast to our work, which takes the operational environment of our insurance industry partners into account.

Examples of Webservice (WS-*) related QoS work include QoS-based dynamic service bind [14][15], related WS-* standards such as WS-Policy [16], and general research questions for QoS in SOA environments [17]. Design aspects and models for QoS and SOA are, for example, addressed in [14][18][19][20][21]. As for WS-* Web services, we also take XML as foundational modelling language for our work. SOA performance including QoS is discussed in articles [22], and monitoring for SOA in articles such as [23][24][25][26].

Uniqueness of our research is, that it takes all the above mentioned aspects into account. We provide a detailed XML based measurement model, as well as a generator-supported, generic SOA monitoring framework. All of it takes especially the operational environment of our insurance industry partners into account, which is a large scale SOA, but only partially WS-* technology based. This makes our work highly relevant in practice. Even more, since we base our modelling on standards, which are highly relevant for German insurance businesses (cf. VAA [27], ISO/IEC 9126 [28][29]), our work is of a quite general nature and thus can be transferable (at least within the insurance domain).

III. Plan the Measurement Process

The Core Measurement Process can be divided into two parts. First of all, the planning of the measurements takes place, which determines how the Information Need can be answered. In the second part, the planned methods of measurement will be implemented.

A. Core Concepts of the Abstract Information Model

To measure the response time behavior of a dynamic system, the definition of static response time limits is often not sufficient. When a system component (service) is deployed in a different hardware environment or in a different cloud environment, this will affect the response time of this system component. Static limits would have to be adapted manually to the new execution environment of the services. Furthermore, individual services share hardware resources with many other services. This can lead to an unpredictable system behavior, especially in complex business processes. Therefore Static limits are not sufficient, but a more flexible solution is required. The approach of this work is the investigation of a measurement concept, which is more flexible and based on the standard deviation of system load of a specifiable measuring period.

The insurance industry in particular is characterized by strong seasonal fluctuations. Towards the end of the year, many customers switch their insurance contracts and are provoking high system loads. In times of such high system loads, the mentioned static limits would be continuously exceeding. The information would be lost at the time when high loads are peaking in such a strongly demanded period. It is important to know when the current system is heavily loaded. Knowledge about this information represents the so-called Information Need (Fig. 1) of our partner from the insurance industry.

To answer this Information Need, the average response time behavior μ of a system component is firstly computed for a freely definable time period. For example, on the basis of the last $n = 500$ measured response times of the services. On the basis of this, the standard deviation is calculated within this period, shown in (1):

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n-1}} \qquad (1)$$

After this calculation, the current response time r of a service is set in relation to this standard deviation s. If the response time r of a currently requested service exceeds this standard deviation by the factor of 2x, this is considered as an overload situation:

$$\text{Spike detected: } r > \mu + 2 * s \qquad (2)$$

This calculation takes place continuously. As soon as a service is requested again, its response time is recorded and set in relation to the last one (e.g., the last 500 measured values). It is therefore a continuous, rolling measurement. This measuring system can be applied both for very slow system components on a daily base and also to very fine-granular services that interact in the range of milliseconds.

The important fact is that the standard deviation is calculated continuously over a defined time period, and the current response time of a service is set in relation to this. Therefore the measuring system adapts to seasonal fluctuations, and it is possible to identify, which user requests (service calls) are currently very critical with respect to the general response time behavior, independently of the prevailing current load situation. This allows fast and more precise analysis of systems and less misinterpretation due to incorrectly set static time limits. This dynamic measurement concept can give a more reliable answer to the Information Need of our project partners.

B. Mapping of the Concepts of the Information Model

In this subsection, a QoS Information Model (QoS IM) is presented in a more detailed manner. The QoS IM is a XML document that includes values of the concepts for a given application scenario. The concepts and their relationshipts with each other are introduced in [2]). Here we focus on the implementation of the concepts.

The QoS IM is created during the planning stage when executing the subprocess 'Plan the Measurement Process', cf. Fig. 1). The XML document is used to automatically generate the QoS Platform's artefacts. The measurements results (i.e. the output of 'Perform the Measurement Process') are produced by the QoS Platform. They are persistently stored for subsequent analysis, typically in a database system.

We opted for XML as universally accepted standard that is highly flexible, platform and vendor independent and supported by a wide variety of tools. Furthermore, XML comes with a standardized schema definition language, namely XML Schema. This is a big advantage against other languages such as JSON for example.

In the QoS IM, we specify the measurement concepts for the check24.com scenario, or the Proposal Service respectively. Due to space limitation, the discussion is restricted to the following concepts (cf. [2]):

- Measurable Concept – outlines in an abstract way, how the Quality Attributes are determined to satisfy the Information Need,
- Base Measure – specifies by its Measurement Method *how* the value of Quality Attribute is to be determined,
- Derived Measure – uses one or more Base Measures or other Derived Measures, whilst the Measurement Function specifies the calculation method and thus the combination of the Measures used,
- Indicator – is a qualitative evaluation of Quality Attributes, which directly addresses the issue raised in the Information Needs.

```
<MeasurableConcept Name="Processing_Time">
 <SubCharacteristic Name="Performance"/>
 <BaseMeasure Name="t_inst"/>
 <BaseMeasure Name="t_term"/>
 <DerivedMeasure Name="t_proc"/>
 <DerivedMeasure Name="Count_StdDev_Calls"
     />
 <DerivedMeasure Name="Count_Calls"/>
 <DerivedMeasure Name="
     StdDev_Calls_Percentage"/>
 <DerivedMeasure Name="Failed_Calls"/>
</MeasurableConcept>
```

Listing 1. Calculation of the Proposal Service's Processing Time

The Measurable Concept `Processing_Time` references by name all necessary Base and Derived Measures (cf. listing 1).

The definition of the Base Measure `t_inst` is shown in listing 2. Its task is to capture the start time of a Proposal Service call (by a user request). The element `Attribute` specifies the attribute of the Proposal Service to be observed. The element hierarchy of `Implementation` defines all platform specific information to automatically generate all artefacts needed for the measurement, i.e., the agent class with attributes and the measurement method (cf. subsection IV).

```
1  <BaseMeasure Name="t_inst">
2   <Scale TypeOfScale="Rational" Type="R"/>
3   <Attribute ServiceID="BAS_001" AttributeName="ServiceCallID"/>
4   <MeasurementMethod Name="recordTimeOfServiceCall">
5   <Implementation>
6    <Agent Class="ServiceAgent">
7    <Method>
8    <Attribute Name="ServiceCallID" Type=.../>
9    <Attribute Name="Time" Type=... Computed="time"/>
10   <Event Name="ServiceStartEvent"/>
11   </Method>
12   </Agent>
13  </Implementation>
14  </MeasurementMethod>
15 </BaseMeasure>
```

Listing 2. Start Time of a Proposal Service Call

The Derived Measure `Count_StdDev_Calls` presented in listing 3 calculates the number of Proposal Service calls that exceeds twice the standard deviation (cf. subsection IV, (2)).

```
1  <DerivedMeasure Name="Count_StdDev_Calls">
2   <Uses><DerivedMeasure Name="t_proc"/></Uses>
3   <MeasurementFunction Name="calculateNumberOfCallsAboveSTDDEV">
4    <Implementation>
5     <Analyzer>
6      <Query Class="ServiceDuration" Type=...>
7       <Plain>
8       SELECT COUNT(*) FROM serviceduration
9       WHERE
10      TINTS > TIME_SECS(DATEADD('DAY',
11                  -30,NOW()))
12      AND TPROC > (SELECT AVG(TPROC)
13                        + (2 * STDDEV(TPROC))
14      FROM serviceduration
15        WHERE TINTS > TIME_SECS(
16            DATEADD('DAY',-30, NOW())))
17      </Plain>
18     </Query>
19    </Analyzer>
20   </Implementation>
21  </MeasurementFunction>
22  <UnitOfMeasurement>ms</UnitOfMeasurement>
23  <TargetValue>1</TargetValue>
24 </DerivedMeasure>
```

Listing 3. Compute the Number of Proposal Service Calls that Exceed Twice the Standard Deviation

`Count_StdDev_Calls` is based on a different Derived Measure, namely `t_proc`, which computes the processing time of a Proposal Service call (cf. `Uses` element, line 2). The element `Implementation` comprises of all information that is used to generate the analyzer class (cf. subsection IV). The analyzer executes the SQL Select statement (cf. lines 8 to 16), which represents the content of the element `Plain`. This is done by the measurement function `calculateNumberOfCallsAboveSTDDEV`, shown in line 3, whenever an event `ServiceDurationEvent` has been fired (cf. line 6).

Finally, the Indicator `SLoT_proc`, shown in listing 4, evaluates the adequacy of the processing time of all Proposal Service calls.

`SLoT_proc` is based on two different Derived Measures, namely `StdDev_Calls_Percentage`, and `Failed_Calls` respectively (cf. `Uses` element, lines 4 to 7). The first measure, `StdDev_Calls_Percentage`, takes `Count_StdDev_Calls` and `Count_Calls` and does some basic arithmetic computation.

The element `DecisionCriteria` specifies a decision table, so that a value, computed by the Derived Measures, can be mapped to the entry of the given nominal scale (i.e., high, medium, low). The element `Implementation` comprises all information to generate the analyzer class (cf. subsection IV), which implements the decision table and the mapping.

```
1  <Indicator Name="SLoT_proc">
2   <AnalysisModel Name="computeAdequacyOfProcessingTime">
3   <Scale TypeOfScale="Nominal" Type=.../>
4   <Uses>
5   <DerivedMeasure Name="StdDev_Calls_Percentage"/>
6   <DerivedMeasure Name="Failed_Calls"/>
7   </Uses>
8   <DecisionCriteria>
9   <Implementation>
10   <Analyzer>
11   <IndicatorTable Class="IndicatorController" Type="HMN">
12   <IndicatorEntry>
13   <Input>devPercentageCount < 5 && badCount == 0</Input>
14   <Result>low</Result>
15   </IndicatorEntry>
16   ...
17   </IndicatorTable>
18   </Analyzer>
19  </Implementation>
20  </DecisionCriteria>
21  </AnalysisModel>
22 </Indicator>
```

Listing 4. Compute the Adequacy of the Processing Time of all Proposal Service Calls

IV. Concepts Implementation based on Generators

The initial phases of applying an IM (cf. Fig. 2) were shown in Section III-B. This section discusses subsequent phases (especially about generators, artefacts, etc.) in detail. Please note, although its concepts are transferable, our QoS Generator aims not to be of generic nature, but is tailored specifically towards our XML based IM and needs of our partner companies. Furthermore, the generated artefacts are specific to our current QoS Platform. Both offer the flexibility to tailor each part to the specific needs of each of our partner companies.

A. Design of the QoS Generator

Several different artefacts have to be generated to apply a specific IM. The basic design of the QoS Generator is

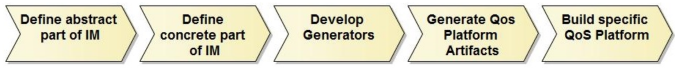
Define abstract part of IM
Define concrete part of IM
Develop Generators
Generate Qos Platform Artifacts
Build specific QoS Platform

Figure 2. Phase from IM to QoS System.

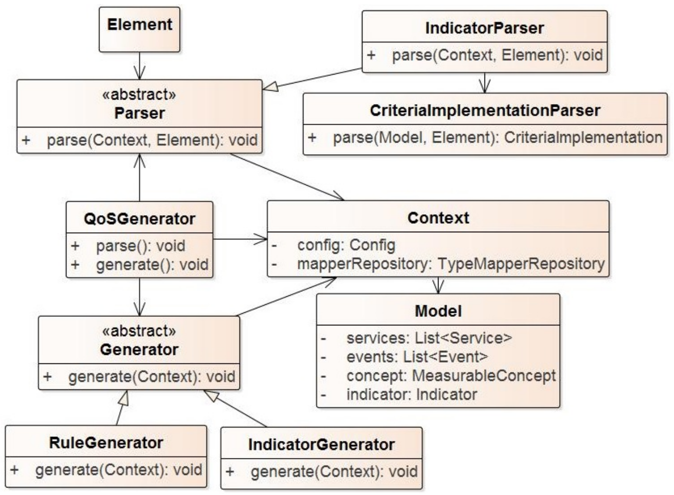
Element
IndicatorParser
+ parse(Context, Element): void
«abstract» Parser
+ parse(Context, Element): void
CriterialmplementationParser
+ parse(Model, Element): CriterialImplementation
QoSGenerator
+ parse(): void
+ generate(): void
Context
- config: Config
- mapperRepository: TypeMapperRepository
Model
- services: List<Service>
- events: List<Event>
- concept: MeasurableConcept
- indicator: Indicator
«abstract» Generator
+ generate(Context): void
RuleGenerator
+ generate(Context): void
IndicatorGenerator
+ generate(Context): void

Figure 3. Design of the QoS Generator.

given in Fig. 3. In general, it consists of a parser step and a generator step. Purpose of the first step is to build an optimized in-memory model of an IM. A parser gets the XML root element and parses the abstract part (as shown through the IndicatorParser) and then the concrete part (as shown by the CriteriaImplementationParser). This distinction is necessary for the desired flexibility of the QoS Platform itself.

The optimized model is part of the Context class and given to each generator as part of the second step. Also, the Context contains general configuration information and a TypeMapperRepository. This latter contains mappers to translate XML types into implementation specific types (e.g., SQL, Java, etc.).

While the parsers are tailored towards the IM model, the generators are tailored towards implementation artefacts or concepts. Hence, there are generators for the QoS Agent, Indicator implementation or complex event processing (CEP) rules. Each generator has a specific task concluding in the generation of certain artefacts. This further supports the flexibility of the QoS Platform itself.

B. Implemented concepts and their artefacts

In the following paragraphs, different concrete parts and their corresponding artefacts are presented. Note, only excerpts are shown and currently not all elements of the abstract part are used.

The concrete part of the Base Measure `t_inst` is given in Listing 2. It defines `Attribute` elements and references the computed QoS Event. The ServiceCallID is parsed from Service Call data. The Time attribute will be computed through the Agent itself. Furthermore, a class attribute is given in the Agent element. It is used to structure the generated code and the corresponding artefacts. The specific method name is derived from the `MeasurementMethod` element.

The concrete part of the Derived Measure `t_proc` is given in Listing 5. It contains the definition of the CEP rule, which computes the complex event ServiceDurationEvent. Plain element indicates that this code fragment will be placed ”as is” into a rule file. Only import definitions (e.g., for event classes) will be added. The rule file is loaded on start up by the CEP engine (JBoss Drools) of the QoS Measurement module.

```
<Rule>
 <Event Name="ServiceDurationEvent"
  Handle="output"/>
 <Plain>
  rule "Service Duration Rule"
  when
   $start : ServiceStartEvent()
   $end : ServiceEndEvent(
    this after[ 0s , 2s ] $start &&
    this.id == $start.id
   )
  then
   channels["analyzer"].send(
    new ServiceDurationEvent(...)
   );
  end
 </Plain>
</Rule>
```

Listing 5. Concrete Part of a Rule.

The concrete part of the Derived Measure `COUNT_STDEV_CALLS` is given in Listing 3. It contains the SQL query to get the count of all events with a runtime above the doubled standard deviation. The generated class is shown in Listing 6. The query attribute contains the SQL query of the Plain element. Again, the class attribute is used to structure the code and artefacts, but the Type attribute is specific for a query and specifies the return type (in this case Long) of the query. The name of method is given in the `MeasurementFunction` element. Furthermore, needed imports and Spring code to integrate the jdbcOperations object are generated.

```
public class ServiceDurationQuery {
 ...
 public Long ⤸
     calculateNumberOfCallsAboveSTDDEV() {
  return jdbcOperations.queryForObject(
    query, Long.class );
 }
}
```

Listing 6. Generated query class.

The concrete part of the Indicator `SLoT_proc` is given in Listing 4. Each `IndicatorEntry` element consists of an `Input` where the Indicator condition is defined and a `Result` element, which contains the actual Indicator response. Each of these results have to be a valid HMN type. The generated IndicatorController class is given in Listing 7. The dependencies to other Measure results are given through the `Uses` element. This information is also used for generation and manual modifications.

While the QoS Agent is only designed as part of the QoS System, it is actually placed directly into the SOA as part of the ESB.war. The used ESB is a partner specific implementation. The generated class and rule files for the DerivedMeasures are part of the QoS Platform (and part of the QoS Platform.war). For example, the generated classes of

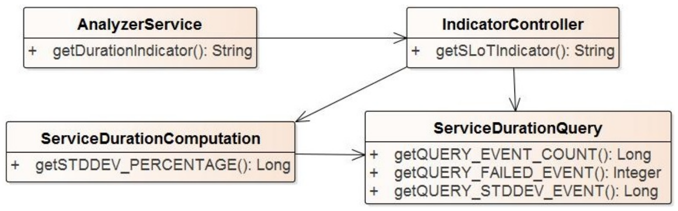
AnalyzerService
+ getDurationIndicator(): String
IndicatorController
+ getSLoTIndicator(): String
ServiceDurationComputation
+ getSTDDEV_PERCENTAGE(): Long
ServiceDurationQuery
+ getQUERY_EVENT_COUNT(): Long
+ getQUERY_FAILED_EVENT(): Integer
+ getQUERY_STDDEV_EVENT(): Long

Figure 4. Detailed view into the Analyzer Module.

the Analyzer module are shown in Fig. 4. Indicator and *-Duration classes are integrated, if needed manually, onto the QoS Platform. The AnalyzerService class is considered part of the QoS Platform core and implements the REST interface for downstream systems (e.g., alerting).

```
public class IndicatorController {
 public String ⤸
     computeAdequacyOfProcessingTime() {
  ...
  if(devPercentageCount < 5 &&
      badCount == 0) {
   return "niedrig";
  }
  if(devPercentageCount >= 5 &&
      badCount == 0) {
   return "mittel";
  } else {
   return "hoch";
  }
}}
```

Listing 7. Generated indicator class.

V. MEASUREMENTS

For the evaluation of the described measurement concept, it is stressed with an initial load test. The general 'Information Need' (Fig. 1) is the information about how volatile a software system is currently being stressed. Static thresholds cannot fulfill the desired 'Information Need' of the partner companies in the insurance industry. The dynamic approach of measuring the spikes, which exceed the standard-deviation of a measuring period, can provide better answers here. For the evaluation, such spikes are directly provoked. When generating the spikes, two parameters are randomly influenced:

- Intensity: The intensity of the spikes.
- Frequency: The frequency at which the spikes occur.

In the stress test, the two parameters 'Intensity' and 'Frequency' are set. A high intensity means that a spike is generated with a high level of volatility. The intensity describes, how long the response time of a service request is and how 'intensive' the standard deviation is exceeded according to (1). The frequency determines, how often such a spike should occur in the stress test. The stress test therefore generates very volatile measurement events, which must be recorded dynamically by the measuring system. So, a random variation of these two parameters will provoke volatile stress situations with unpredictable intensity and frequency. This allows the measuring system to be tested as strongly and dynamically as possible. Some of the preliminary results measured with the QoS System are shown in Fig. 5. The yellow line shows the

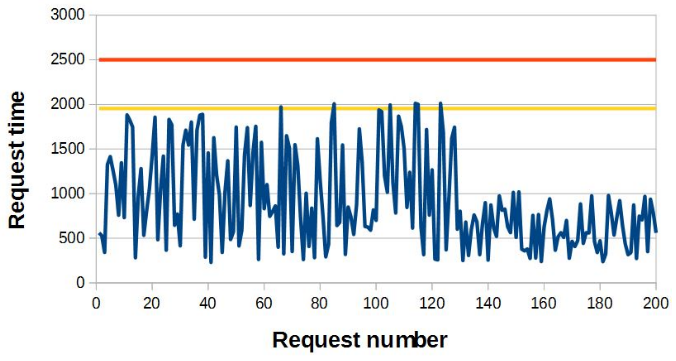

Figure 5. Preliminary Measurement Results.

standard deviation barrier. In this case, 3 % of all measured requests (6 service calls) are violating the barrier, thus the computed indicator would be low. Above 5 % the indicator would be middle. The red line shows the SLA barrier introduced by service consumers like check24.de. If one request exceeds this barrier the indicator switches to high. A more thorough test and evaluation based on these loads will be given in our future work. But based on these results, the measurement concept can be used to even measure very volatile stress situations

VI. CONCLUSION AND FUTURE WORK

In this article we presented an approach for monitoring a distributed SOA environment, which we see as a promising path to take. Our SOA Quality Model is aimed to follow the ISO/IEC-Standard 15939 (cf. [30]), which enables a wide range of use cases. Our Measurement Concept outlines an execution platform for the specific QoS Information Model, which should cause minimal impact on the SOA environment.

The separation of Measurement Agents and QoS-Analyzer on one hand allows lightweight agents and on the other hand a very capable analyzer component. Furthermore certain parts of our QoS Platform can be replaced or complemented with common tools, e.g., from the microservices eco system. For example, Netflix's Hystrix could be used to implement a BaseMeasure or Prometheus to implement DerivedMeasures. This flexibility in our architecture with the general concept given through our SOA Quality Model offers new opportunities for our partner companies.

Already in previous work [1] [2] we presented our general measurement concept, an initial business process (the 'check 24' Proposal Service insurance use case, a basic business relevant scenario), and our information model and concept. The core contributions of the present article are implementation details of our approach. Therefore in Section III we dive deeply into our information model and in Section IV our model-driven, generator based implementation is described in depth.

Our ongoing work of applying the QoS System to an application scenario relevant to our partner in the insurance industry (the so called 'Check 24 process'), will provide evidence of the practical usability of the created framework. It is expected, that our monitoring system will help to discover potential bottlenecks in the current system design of our partner's distributed services. Therefore, it will create value in the process of solving these issues.

In future work, we have planned to apply our existing work to the more complex insurance process 'Angebot erstellen' ('create individual proposal') of the VAA [27]. Thus, we will implement a more complex insurance scenario. Moreover, the actual measurement and analysis of the results are an ongoing process, which is yet to be finished.

We also have plans to apply these results onto cloud based environments. Furthermore, a deeper subdivision or extraction from the current coarse granular SOA services into more fine grained microservices will be investigated by us in future work 'where it makes sense', for example, to allow for a better scalability of individual microservices.

References

[1] A. Hausotter, A. Koschel, J. Busch, M. Petzsch, and Malte Zuch, ”Implementing a Framework for QoS Measurement in SOA”, submitted for publication, 2017.

[2] A. Hausotter, A. Koschel, J. Busch, M. Petzsch, and Malte Zuch, ”Agent based Framework for QoS Measurement Applied in SOA,” in: The 9th International Conferences on Advanced Service Computing (Service Computation), IARIA, Athens, Greece, 2017, pp. 16-23.

[3] T. Bergemann, A. Hausotter, and A. Koschel, ”Keeping Workflow-Enabled Enterprises Flexible: WfMS Abstraction and Advanced Task Management,” in: 4th Int. Conference on Grid and Pervasive Computing Conference (GPC), 2009, pp. 19-26.

[4] C. Gäth, et al., ”Always Stay Agile! – Towards Service-oriented Integration of Business Process and Business Rules Management,” in: The Sixth International Conferences on Advanced Service Computing (Service Computation), IARIA, Venice, Italy, 2014, pp. 40-43.

[5] A. Hausotter, C. Kleiner, A. Koschel, D. Zhang, and H. Gehrken, ”Always Stay Flexible! WfMS-independent Business Process Controlling in SOA,” in: IEEE EDOCW 2011: Workshops Proc. of the 15th IEEE Intl. Enterprise Distributed Object Computing Conference, IEEE: Helsinki, Finnland, 2011, pp. 184-193.

[6] A. Koschel and R. Kramer, ”Configurable Event Triggered Services for CORBA-based Systems,” Proc. 2nd Intl. Enterprise Distributed Object Computing Workshop (EDOC'98), San Diego, U.S.A, 1998, pp. 1-13.

[7] M. Schaaf, I. Astrova, A. Koschel, and S. Gatziu, ”The OM4SPACE Activity Service - A semantically well-defined cloud-based event notification middleware,” in: IARIA Intl. Journal On Advances in Software, 7(3,4), 2014, pp. 697-709.

[8] B. Schroeder, ”On-Line Monitoring: A Tutorial,” IEEE Computer, 28(6), pp. 72-80, 1995.

[9] S. Schwiderski, ”Monitoring the Behavior of Distributed Systems,” PhD thesis, Selwyn College, University of Cambridge, University of Cambridge, Computer Lab, Cambridge, United Kingdom, 1996.

[10] Dynatrace LLC, ”Dynatrace Application Monitoring,” [Online]. URL: https://www.dynatrace.com/de/products/application-monitoring.html [accessed: 2017-12-06].

[11] Nagios.ORG, ”Nagios Core Editions,” [Online]. URL: https://www.nagios.org/ [accessed: 2016-12-26].

[12] N. W. Paton (ed.), ”Active Rules for Databases,” Springer, New York, 1999.

[13] ACT-NET Consortium, ”The Active DBMS Manifesto,” ACM SIGMOD Record, 25(3), 1996.

[14] M. Garcia-Valls, P. Basanta-Val, M. Marcos, and E. Estévez, ”A bi-dimensional QoS model for SOA and real-time middleware,” in: Intl. Journal of Computer Systems Science and Engineering, CLR Publishing, 2013, pp. 315-326.

[15] V. Krishnamurthy and C. Babu, ”Pattern Based Adaptation for Service Oriented Applications,” in: ACM SIGSOFT Softw. Eng. Notes 37, 2012(1), 2012, pp. 1-6.

[16] T. Frotscher, G. Starke (ed.), and S. Tilkov (ed.), ”Der Webservices-Architekturstack,” in: SOA-Expertenwissen, Heidelberg, dpunkt.verlag, 2007, pp. 489-506.

[17] F. Curbera, R. Khalaf, and N. Mukhi, ”Quality of Service in SOA Environments. An Overview and Research Agenda,” in: it - Information Technology 50, 2008(2), 2008, pp. 99-107.

[18] S.W. Choi, J.S. Her, and S.D. Kim, ”QoS Metrics for Evaluating Services from the Perspective of Service Providers,” in: Proc. of the IEEE International Conference on e-Business Engineering, Washington DC, USA : IEEE Computer Society (ICEBE'07), 2007, pp. 622-625.

[19] Z. Balfagih and M.F. Hassan, ”Quality Model for Web Services from Multi-stakeholders' Perspective,” in: Proceedings of the 2009 International Conference on Information Management and Engineering, Washington DC, USA : IEEE Computer Society (ICIME'09), 2009, pp. 287-291.

[20] G. Wang, A. Chen, C. Wang, C. Fung, and S. Uczekaj, ”Integrated Quality of Service (QoS) Management in Service-Oriented Enterprise Architectures,” in: Proceedings of the 8th IEEE Intl. Enterprise Distributed Object Computing Conference (EDOC'04), Washington DC, USA, IEEE, 2004, pp. 21-32.

[21] M. Varela, L. Skorin-Kapov, F. Guyard, and M. Fiedler, ”Meta-Modeling QoE”, PIK-Praxis der Informationsverarbeitung und Kommunikation, 2014, Vol. 37(4), pp. 265-274.

[22] R.W. Maule and W.C. Lewis, ”Performance and QoS in Service-Based Systems”, Proc. of the 2011 IEEE World Congress on Services, IEEE Computer Society, 2011, pp. 556-563.

[23] B. Wetzstein, et al., ”Monitoring and Analyzing Influential Factors of Business Process Performance,” in: Proc. IEEE Intl. Enterprise Distributed Object Computing Conf. (EDOC'09), 2009, pp. 141-150.

[24] F. Rosenberg, C. Platzer, and S. Dustdar, ”Bootstrapping Performance and Dependability Attributes of Web Services,” in: Proc. International Conference on Web Services (ICWS'06), 2006, pp. 205-212.

[25] M. Schmid, J. Schaefer, and R. Kroeger, ”Ein MDSD-Ansatz zum QoS-Monitoring von Diensten in Serviceorientierten Architekturen,” in: PIK Praxis der Informationsverarbeitung und Kommunikation, 31 (2008) 4, 2008, pp. 232-238.

[26] S.M.S. da Cruz, R.M. Costa, M. Manhaes, and J. Zavaleta, ”Monitoring SOA-based Applications with Business Provenance”, Proc. of the 28th Annual ACM Symposium on Applied Computing (ACM SAC), ACM, 2013, pp. 1927-1932.

[27] GDV (Gesamtverband der Deutschen Versicherungswirtschaft e.V. – General Association o.t. German Insurance Industry), ”Die Anwendungsarchitektur der Versicherungswirtschaft: Das Objektorientierte Fachliche Referenzmodell (The application architecture of the German insurance business – The functional object-oriented reference model”, VAA Final Edt. Vers. 2.0, 2001, [Online]. URL: http://www.gdv-online.de/vaa/vaafe_html/dokument/ofrm.pdf [accessed: 2017-01-11].

[28] ISO - International Organization for Standardization (ed.), ”ISO/IEC 25010:2011 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models”, 2011.

[29] M. Azuma, ”SQuaRE: the next generation of the ISO/IEC 9126 and 14598 international standards series on software product quality, ” in: Proc. of European Software Control and Metrics (ESCOM), 2001, pp. 337-346.

[30] ISO - International Organization for Standardization (ed.), ”ISO/IEC 15939:2007 - Systems and software engineering - Measurement process,” 2007.