Self-Adaptive Software Performance Monitoring

Jens Ehlers, Wilhelm Hasselbring

Software Engineering Group
Christian-Albrechts-University Kiel
24098 Kiel
{jeh,wha}@informatik.uni-kiel.de

Abstract: In addition to studying the construction and evolution of software services, the software engineering discipline needs to address the *operation* of continuously running software services. A requirement for its robust operation are means for effective monitoring of software runtime behavior. In contrast to profiling for construction activities, monitoring of operational services should only impose a small performance overhead. Furthermore, instrumentation should be non-intrusive to the business logic, as far as possible. Monitoring of continuously operating software services is essential for achieving high availability and high performance of these services. A main issue for dynamic analysis techniques is the amount of monitoring data that is collected and processed at runtime. On one hand, more data allows for accurate and precise analyses. On the other hand, probe instrumentation, data collection and analyses may cause significant overheads. Consequently, a trade-off between analysis quality and monitoring coverage has to be reached.

In this paper, we present a method for self-adaptive, rule-based performance monitoring. Our approach aims at a flexible instrumentation to monitor a software system's timing behavior. A performance engineer's task is to specify rules that define the monitoring goals for a specific software system. An inference engine decides at which granularity level a component will be observed. We employ the Object Constraint Language (OCL) to specify the monitoring rules. Our goal-oriented, self-adaptive method is based on the continuous evaluation of these rules. The implementation is based on the Eclipse Modeling Framework and the Kieker monitoring framework. In our evaluation, this implementation is applied to the iBATIS JPetStore and the SPECjEnterprise2010 benchmark.

1 Introduction

While for most software systems performance is a critical requirement, tools that monitor the operation of software systems at *application* level are rarely used in practice. This dichotomy is indicated by a survey carried out at a recent conference among Java practitioners and experts [Sna09]. Garbage collection, concurrency control, remote service calls, database access and legacy integration are identified as typical performance problem areas. Nevertheless, adequate monitoring tools that allow the analysis of these problems and their root causes are seldom known and employed in software engineering projects. The prevalent negligence of continuous operational monitoring is caused by the following symptoms: (1) a posteriori failure analysis, i.e. appropriate monitoring data

is seldom collected and evaluated systematically before a failure occurred, (2) inflexible instrumentation, i.e. probes are placed into a component's source code only at a limited number of fixed points such that recompilation and redeployment are required for future modifications, and (3) inability of request tracing in distributed systems, i.e. tracing of user requests beyond the borders of a single component or its execution container in a distributed system is not supported or not applied.

In this paper, we present a sophisticated method for self-adaptive performance monitoring and evaluate the design of an associated tool to handle the above shortcomings for existing component-based Java EE applications. Means for efficient and flexible observation, analysis, and reporting of a software system's runtime behavior are indispensable to ensure its proper operation. Therefore, two constraints have to be considered for continuous monitoring: (1) acceptable overhead and (2) non-intrusive instrumentation. In contrast to profiling at construction time, the monitoring overhead at operation time has to be kept deliberately small. Secondly, the instrumentation of probes should not pollute the application's business logic.

A main issue for dynamic analysis techniques addressing software behavior comprehension is the amount of information which is collected and processed at runtime. The more detailed monitoring data is the more precise subsequent analysis can be. On the other hand, instrumentation injection, data collection, logging, and online analysis itself cause measurable overhead. Consequently, a trade-off between analysis quality and monitoring coverage has to be reached.

It is not the injection of numerous dummy probes that causes overhead, but the complexity of real probe implementations. We have evaluated and quantified the decisive impact of how the observed data is collected, processed, and logged for subsequent analyses [vHRH+09]. A finding is that it is possible to inject diverse probes at a multitude of relevant join points, as long as not all of them are active at the same time during operation. Thus, our approach aims not at a fixed instrumentation to monitor a software system's timing behavior. Instead, a major objective is a self-adaptive activation of probes and their related join points. For that purpose, an inference engine decides at which granularity level a component will be observed. A performance engineer's task is to specify rules that define the monitoring goals for a specific software system.

Typical goals of software system monitoring include the required evidence of SLA compliance, the localization of faults causing QoS problems, capacity planning support, and mining of usage patterns for interface design or marketing purposes. In this paper, we focus on the goal to detect the origins of QoS problems or other system failures that effect a change in the system's normal behavior as perceived by its users. We employ the Object Constraint Language (OCL) [OMG10] to specify the monitoring rules. The rules refer to performance attributes such as responsiveness metrics and derived anomaly scores that change their values during runtime. Our goal-oriented self-adaptation is based on the continuous evaluation of these rules. Our presented implementation is based on EMF (Eclipse Modeling Framework) [SBPM08] meta-models which allow to evaluate OCL query expressions on object-oriented instance models at runtime. The contributions of this paper are

(1) the design of a generic monitoring process for component-based software systems which is based on our Kieker monitoring framework,[1]

(2) an approach for self-adaptive performance monitoring as an extension to (1),

(3) an implementation and evaluation of this approach.

The remainder of the paper is structured as follows: We briefly describe our Kieker monitoring framework in Section 2. In Section 3, we present our approach for self-adaptive performance monitoring. Its evaluation in lab experiments and industrial systems is summarized in Section 4. Related work is discussed in Section 5. Finally, a conclusion and outlook to future work is given in Section 6.

2 The Kieker Monitoring Framework

This section introduces the monitoring process underlying our Kieker monitoring framework for component-based software systems [vHRH+09]. The monitoring process consists of the following activities, as illustrated in Fig. 1: (1) probe injection, (2) probe activation, (3) data collection, (4) data provision, (5) data processing, (6) control and visualization, and (7) adaptation. The sequence of the activities and their interrelation with the major architectural components of the monitoring framework is annotated to the component diagram shown in Fig. 1. In this paper, we emphasize the (self-)adaptation activity which is presented in detail in Section 3. As the adaptation is closely interconnected with the preceding activities named above, our integrated monitoring process is subsequently explained.

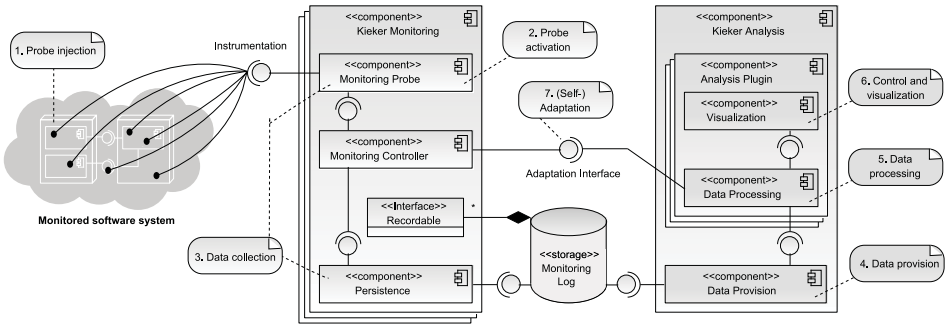

Figure 1: Self-Adaptive Monitoring Process and Architecture

Probe Injection: The software system to be monitored has to be instrumented with probes. A probe is positioned at each join point at which observation of control or data flow could be interesting. To measure service times, probes are placed at

[1] http://kieker.sourceforge.net/

operation entry and exit points. These include particularly the public services provided by a component's interface, but may also cover private methods or code blocks. Aspect-oriented programming (AOP) is an appropriate means to inject application-level probes into class methods in object-oriented software systems [FHRS07]. Thus, our default approach is to utilize the interception framework AspectJ for instrumentation. If method-level instrumentation is not fine-grained enough, it is proposed to use byte-code instrumentation approaches such as Javassist [Chi00]. To impurify the code by placing the probes manually, remains the least valued choice for instrumentation.

Probe Activation: As illustrated in the left part of Fig. 1, Monitoring Probes are triggered from various join points embedded in the monitored software system. The probes are part of a Kieker Monitoring component. In a distributed software system, an instance of this monitoring component is deployed on each execution container and interconnected with the hosted application components, e.g. by load-time aspect weaving. Each Kieker Monitoring component is controlled by a single Monitoring Controller. The controller manages which probes and probe join points are currently activated. Additionally, it provides access to a persistence unit for logging the monitored data and provides an adaptation interface for reconfiguration (see Section 3).

Data Collection: Our monitoring framework Kieker comes along with a set of different aspects that allow the detection of service call entries incoming via different interface technologies, including HTTP Servlets, JAX-WS, JAX-RS, EJB 3.0, Java Remote Method Invocation (RMI), or JMS. These aspects intercept specific framework methods, e.g. javax.servlet.http.HttpServlet.do*(..) for any HTTP operation processed by a Servlet or @javax.jws.WebMethod *.*(..) for JAX-WS annotated interface methods. Furthermore, Kieker provides an aspect to intercept application-specific operations. Typically, this includes all methods as part of a component realization that contribute to the system's application logic and which is in the scope of monitoring.

Each request is tagged with an id when it enters the system. By means of the framework aspects, it is possible to trace a request over any inter-component and inter-server communication protocol throughout a distributed system. To track the relation between a caller and its callee, the probe intercepting the calling operation can either store information about its join point in thread local memory or attach it to the meta-data of the service call. The latter is appropriate, if the call is asynchronous or crossing container borders. The probe linked to the called operation receives the information about its calling context and saves it in a monitoring record. Later, analyses will process the recorded calling context information of all operation calls related to a single request and reconstruct a monitored request trace. Besides calling context information, probes gather performance related metrics such as service times, call frequencies (to determine throughput), or resource utilization.

As monitoring data is collected at different nodes of a distributed system, it has to be collected for system-wide analysis. Fig. 1 illustrates that each monitoring component pushes its records into a central repository called Monitoring Log. Recording with Kieker can be applied to any persistent storage repository like a file system, a database, or a buffered message queue. As indicated in Fig. 1, probes can collect arbitrary data records and push them into the log, on condition that the record class implements the Kieker

interface for Recordable entities. Note that the persistence operations cause a major part of the measurement expense [vHRH+09].

Data Provision: While the Kieker Monitoring component is deployed several times, with one instance at each monitored execution container, it is sufficient to run only one instance of the Kieker Analysis component. The Kieker Analysis component frames a client application with a graphical control panel addressing the system's performance engineer. Our implementation of this component is realized as a tool based on Eclipse RCP and EMF. A screenshot is shown later in Section 3 in Fig. 4. The data stream in the Kieker Analysis component follows the pipes-and-filters pattern [TMD09]. The monitored records serve as input for the piped analysis data stream. The first and lowest-level filter is a Data Provision component. This data provider is configured to consume the records of multiple monitored execution container nodes, each one being equipped with a separate Monitoring Controller. Each controller provides information to the data provider about its Monitoring Log used for record persistence. Due to this information, a data provider can connect to the log(s) and continuously receive and forward the monitoring records to subsequent analysis filters. Thus, a data provider serves as a central consolidation filter that merges observations made in the single nodes of a distributed system.

Data Processing: The proceeding filters of the analysis data stream are joined together via a plugin mechanism. The Kieker Analysis component is constructed to be easily extensible with (third-party) Analysis Plugins containing Data Processing and/or Visualization components. Such data processing or visualization filters can be subscribed to a data provision filter or another supplying data processing filter, provided that the filters' input and output ports match. Visualizations that display the analysis results are usually a sink of the data stream. Each filter can implement any kind of processing based on the records received from its predecessors. The incoming records deliver pieces of information that can be assembled or visualized in different ways. For instance, records are related to tracing, responsiveness, or utilization.

Tracing allows to recap how one or more client requests have been executed by the software system under inspection. Tracing can be applied at different abstraction levels, e.g. studying interaction at the level of components, classes, or methods. In any case, information about the calling dependencies between arbitrary interacting entities is collected. This information can be represented in different ways, e.g. by a dynamic call tree [JSB97], by a call graph [GKM82], or by a calling context tree [ABL97]. These representations differ in their accuracy and efficiency, and hence are suitable for different tasks. A calling context tree (CCT) is an appropriate intermediate representation for continuous monitoring, as it is more accurate than a call graph and less resource consuming than storing every single call trace. The breadth of a CCT is limited by the number of observed entities and not by the number of requests over time. All requests with the same trace are aggregated in a CCT. Though the metrics of identical traces are merged, the stack context of each call is preserved [RvHG+08].

Our adaptive monitoring approach employs analyses that are based on the deviation of observed and expected service time. A derived measure is the anomaly score assigned to a sample of observed operation executions, see Section 3.

Control and Visualization: It is a major requirement of a monitoring tool to provide a control panel that conveniently visualizes the monitored data for analysts. The Kieker Analysis component serves as such a control panel framework and comes along with a set of plugins that implement the graph representations explicated above to picture tracing and compositional dependencies. Different libraries such as Graphviz[2] and Eclipse Zest[3] are used with Kieker for graph visualization. Various other plugins have been developed or are work in progress to be integrated into the framework, e.g. reverse engineered dependency and sequence diagrams, Markov chains, or 3D runtime behavior visualizations. As stated, the analysis component can easily be extended with further plugins if required. Concerning the runtime adaptability of monitoring probes, the related monitoring adaptation plugin described in the following section is of particular interest.

3 Adaptive Monitoring

Considering a software system in productive operation, it is not feasible to record each observed occurrence of any probe at any join point. In fact, this restriction is caused by the great number of expected requests per time slice, not by the number of injected probe join points. Thus, initial monitoring rules have to be configured prescribing which probes are where and when active. Regarding a component-based software system, an appropriate initial monitoring setting can be to activate only join points at operations which serve as part of a component's provided interface (see the activated monitoring points in Fig. 2). At runtime, the monitoring level can be increased to inspect the interior control flow of a component in case it does not behave as expected. The selective activation of a probe join point may depend on its call stack level, the responsiveness of the related operation, the current workload, a random probability, etc.

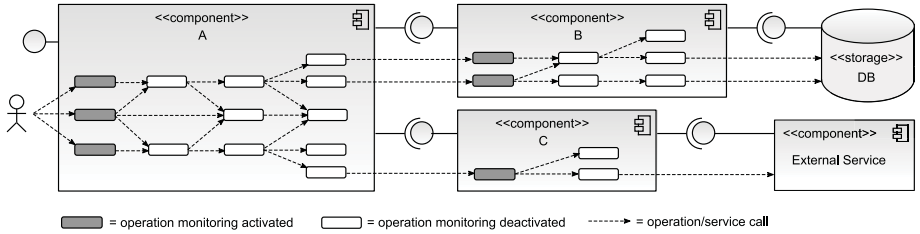

Figure 2: Software system with operation intercepting probes

In this section, we focus on the goal to detect the origins of QoS problems or other system failures that effect a change in the system's normal behavior as perceived by its users. A major part of the failure recovery time is required to locate a failure's root cause. In [KF05], the authors refer to an estimation that 75% of the recovery time is spent just for

[2] http://www.graphviz.org/
[3] http://www.eclipse.org/gef/zest/

fault detection. They report that it often takes days to search for a failure's cause, while it can be fixed quickly once the fault is found.

In case that harmful faults induce anomalous runtime behavior at application level, our proposed solution allows to adapt the monitoring coverage on demand. If the adaptation is conducted manually, the human decision to change the set of active probes or join points is usually based on a previous interpretation of performance visualizations provided by other Kieker plugins. A typical situation is that time series curves indicate a decline of a service's throughput though the load did not increase. A performance engineer who observes this incidence is interested in the cause of the phenomenon and therefore tries to activate more join points in the affected components. Afterwards, it takes a while until enough analysis-relevant records have been collected via the newly activated probe join points. Our self-adaptive monitoring process aims at reducing this potentially business-critical wait time that delays a failure or anomaly diagnosis.

Our proposed Adaptation Plugin is designed to be one of several plugins integrated into the Kieker Analysis component. It allows to (re)configure the set of active probes and probe join points that were previously injected into the system. The proposed adaptation process works like a simple rule-based expert system that employs deductive reasoning to reach a decision. The performance engineer acts as the expert who formulates the inference rules. To adjust the monitoring coverage, the desired conclusions are either to activate a set of currently disabled probe join points, to deactivate a set of currently enabled probe join points, or to leave the current setting unchanged. The set of join points to be changed is addressed in the premise of monitoring rules.

We employ the OCL to specify these rule premises. A premise consists of an OCL context element and an OCL expression. Indeed, the context element specifies the context in which the expression will be evaluated. Often, the adaptation filter itself is an appropriate context element, because it allows navigation to all records that have been supplied by preceding data processing filters.

The OCL expression specifies a set of join points in form of method signatures at which the referenced probes should be activated. The evaluation of the monitoring rules can either take place manually, i.e. released on a click in the control panel, or automatically, i.e. repetitive each time a specified time period has elapsed. As shown in Fig. 1, the Monitoring Controller of each monitored container provides an Adaptation Interface. The Adaptation Plugin contains a derived Data Processing filter that allows to start a concurrent thread for continuous evaluation of the monitoring rules. When a violation of the rule premises is detected, the filter uses the Adaptation Interface of each monitored container to execute the adaptation autonomously. Goal-oriented self-adaptation is based on the possibility to refer to (performance) attributes in the monitoring rules that change their values during runtime, e.g. responsiveness metrics and derived anomaly scores. The time interval between the two succeeding evaluations of the monitoring rules has to be set up. Enough time is required to measure reliable responsiveness and anomaly values in each cycle iteration, but also short enough to react flexibly and without distracting delays to performance degradations. An interval value in the scale of a couple of minutes is appropriate.

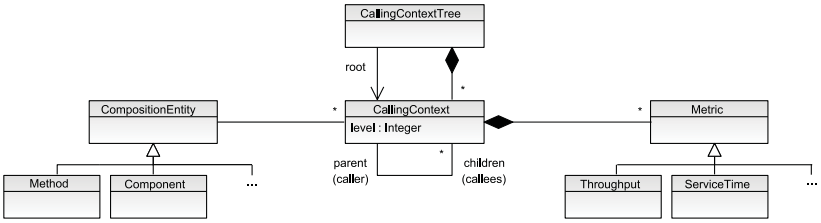

Figure 3: Meta-model of a calling context tree

In the following, example monitoring rules are discussed. The first one starts with a minimal set of active probe join points comprising only class methods (as concrete compositional entities) being placed at the topmost level of a CCT. It is assumed that the context element referenced by the OCL identifier self is set to a CCT instance which was previously delivered to the adaptation filter. A meta-model for a CCT is depicted in Fig. 3.

Example Rule R_1: "If a callee (such as a method) is at the top of an observed call stack (level 1 of the CCT), then enable the probe join points required to intercept and monitor calls to this callee." The corresponding OCL expression to monitor interface operations is:

context CallingContextTree: **self**.callingContexts→*select*(level = 1)→*collect*(method)

OCL provides several built-in collection operations to enable powerful ways of projecting new collections from existing ones. The expression above employs the operations *select* and *collect*. In case of the monitoring rules, the demand is to reduce the set of all join points to a selection of those ones to be activated at the moment. The operation *select* satisfies this demand, as it selects a subset of a collection based on a boolean expression. The OCL simple syntax form is collection→*select*(boolean-expression). The operation *collect* serves to come up to a derived collection that contains different object types than the collection it is originated from, i.e. the new collection is not a subset of the original one. The OCL simple syntax form is collection→*collect*(expression). OCL is mostly known for its purposes to specify invariants on classes, pre- and postconditions on operations, or guards annotated to UML diagrams. Despite that, the first purpose mentioned in the OCL specification suggests OCL to serve as a query language [OMG10]. Indeed, the premises of the monitoring rules can be seen as queries for a set of probe join points to be activated.

The second example rule is more sophisticated. It aims at intensifying the monitoring coverage within a component if it behaves anomalous.

Example Rule R_2: "If the premise of R_1 is fulfilled or if the corresponding caller is already monitored and behaves anomalous, i.e. its anomaly score exceeds a specified threshold t, then enable the probe join points required to intercept and monitor calls to the callee." The OCL expression to monitor interface and anomalous operations is:

context CallingContextTree: **self**.callingContexts→*select*(level = 1 **or** (level $>$ 1 **and** parent.monitoringActivated **and** parent.anomalyRating $>$ t))→*collect*(method)

Imagine that the monitoring rule R_2 is applied to the sample system depicted in Fig. 2. In the initial monitoring coverage only the system's interface operations are observed. If it is discovered that some of these operations do not behave as expected during runtime, the evaluation and appliance of the rule R_2 leads to the join point activation for all callees of the operations indicated as anomalous. Subsequently it can turn out that some of the newly observed callee operations behave anomalous as well. The repeated appliance of R_2 allows to zoom into the control flow of a component in order to seek for the root cause of higher-level anomalies. It is obvious that a manual exploration of such cause-and-effect chains is much more time-consuming and error-prone than an automated processing. A contribution of the self-adaptive monitoring approach is to save this time and effort. Due to space restrictions, the employed anomaly rating procedures are not discussed in this paper. An introduction to a variety of appropriate methods can be found in [CBK09, PP07].

4 Implementation and Evaluation

We have employed the Kieker monitoring component in the productive systems of a telecommunication company and a digital photo service provider, in order to show its practicability [RvHH+10]. Both industrial partners did not observe any perceivable runtime overhead due to the injection of our monitoring probes. In lab experiments, we evaluated the monitoring overhead in more detail and quantified the rates caused by instrumentation, data collection, and logging [vHRH+09].

We implemented the self-adaptive monitoring approach as a Kieker plugin. The implementation of the Kieker Analysis component employs EMF meta-models. For the Adaptation Plugin, we utilize the EMF Model Query[4] sub-project, which allows to construct and run queries on EMF models by means of OCL. Our realization of the Adaptation Interface is based on Java's remote method invocation (RMI) protocol. Probe (de)activation instructions are transmitted through this interface. We evaluated the Adaptation Plugin in lab experiments with the iBATIS JPetStore[5] and the SPECjEnterprise2010[6] benchmark. The sample applications have been stressed with intensity-varying workloads. As long as the system's capacity limits are not exceeded, no anomalies are reported and the monitoring coverage remains constant. In the next step, we manually induced faults into some component-internal operations. We tested successfully that the self-adaptive monitoring is able to locate these faulty operations via anomaly detection as the root cause for the resulting failures perceived by the system users.

Fig. 4 displays a screenshot of the Kieker Analysis component. The left frame (1) in the screenshot contains a project navigator that allows to organize the data processing and visualization filters of a monitoring project. In the frame on the top right (2), the configuration editor of the monitoring adaptation is depicted. It allows to specify a monitoring rule's OCL context element and OCL expression. The middle frame (3) shows a method-level CCT of the SPECjEnterprise2010 benchmark. Based on the failure

[4] http://www.eclipse.org/modeling/emf/?project=query
[5] http://sourceforge.net/projects/ibatisjpetstore/
[6] http://www.spec.org/jEnterprise2010/

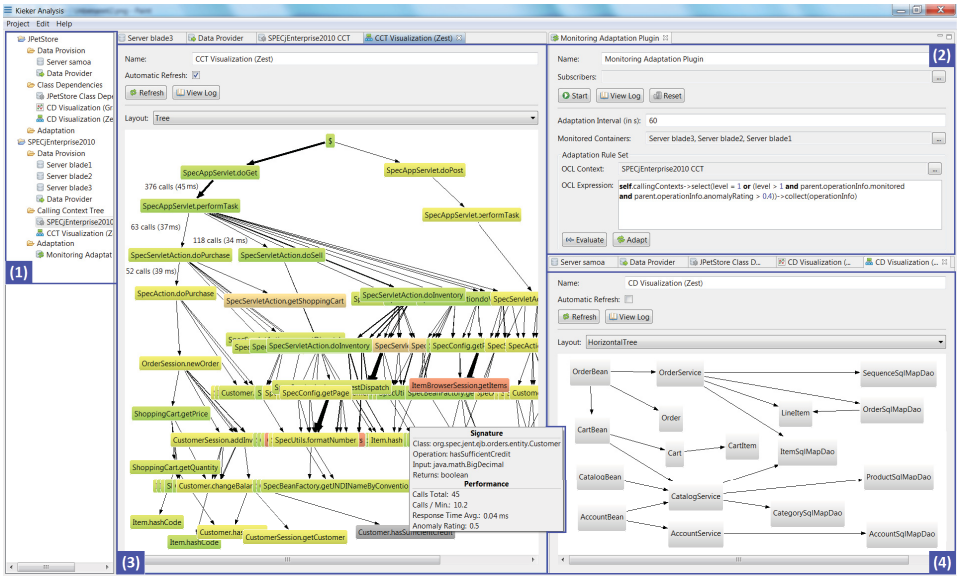

Figure 4: Screenshot of the Kieker analysis tool

diagnosis method of [MRvHH09], the colorings of the method nodes from green to red indicate their current anomaly rating. Call frequencies and average service times of calling contexts are displayed interactively due to clicks on the edges. The depicted tooltip highlights the collected performance metrics for one of the methods. In the frame on the bottom right (4), a class dependency graph of the JPetStore application is displayed. The edges are annotated with aggregated call frequencies among interdependent classes.

5 Related Work

Any integrated monitoring framework addresses two aspects: monitoring (i.e. instrumentation and data collection) and subsequent analysis. In [PMM06], the COMPAS JEEM tool is presented as such an integrated approach. It allows to inject probes as a component-level proxy layer into Java EE systems. Thus, interception is limited to the interface level of Java EE components such as EJBs or Servlets. Runtime adaptation of the monitoring coverage is studied in [MM04]. By observing component-internal operations, Kieker provides a finer-grained analysis of the runtime behavior than COMPAS.

Another approach is Magpie [BDIM04], which monitors resource utilization and component interactions in distributed systems. Magpie is implemented to monitor systems based on Microsoft technology, while Kieker concentrates on Java-based systems. The Pinpoint approach [KF05] utilizes monitoring data to determine anomalous runtime behavior. In contrast to Kieker, Pinpoint does not focus on performance, but applies data mining techniques to detect anomalies in the observed request traces. The Rainbow

project [GCH+04] employs monitoring for architecture-based adaptation of software systems. Magpie, Pinpoint, and Rainbow do not contribute means for self-adaptation concerning the monitoring coverage.

Like Kieker, the open-source projects Glassbox and InfraRED suggest AOP-based instrumentation to introduce monitoring probes. Both projects do not cover request tracing in distributed systems, as well as sophisticated data analyses. The popular open-source tool Nagios is rather intended for infrastructure monitoring than for application-level introspection. Related commercial products like CA Wily Introscope, DynaTrace, or JXInsight do not yet provide rule-based self-adaptation to control the monitoring coverage.

6 Conclusions and Future Work

Predictive support for failure prevention is desired in order to improve a system's fault-tolerance. If alarming system events indicating future performance drops are detected early enough, mechanisms that adapt the system's architectural configuration can be triggered. Therefore, responsiveness and scalability of the system components have to be continuously observed and evaluated. If, for example, a performance degradation is indicated, adaptation decisions and actions concerning the monitoring coverage are derived and the anomalous behavior may be reported or visualized. Filtering the set of active probes and join points on demand allows to zoom into a component if it behaves not as expected. In this case, zooming means to activate more (or less) join points in the control flow aiming at increasing (or decreasing) insight, e.g. into the operation call stack, effective loop iterations, or conditional branches taken. The activation control of probes and join points can either be applied manually or autonomously. For self-adaptive control a set of guiding monitoring rules is required.

In this paper, we presented the design of a generic monitoring process for component-based software systems. Tooling for the monitoring process is provided by our Kieker monitoring framework. Further, we proposed an approach for self-adaptive performance monitoring based on the continuous runtime evaluation of OCL monitoring rules. The feasibility of the self-adaptation has been shown by our implementation of a monitoring adaptation plugin as an extension for Kieker. In lab experiments, we applied our self-adaptive monitoring to benchmark applications such as SPECjEnterprise2010. In our future work, we will report on quantitative evaluation results for different anomaly rating procedures used for self-adaptive monitoring. Besides, we will concentrate on further evidence of the practicability of our self-adaptive monitoring approach by applying it to additional case studies and industrial systems.

References

[ABL97] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proc. of the ACM SIGPLAN 1997 Conf. on Programming Language Design and Implementation*, pages 85–96. ACM, 1997.

[BDIM04] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. of the 6th Conf. on Symposium on Operating Systems Design & Implementation*, pages 259–272. USENIX, 2004.

[CBK09] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys*, 41(3):1–58, 2009.

[Chi00] S. Chiba. Load-time Structural Reflection in Java. In *Proc. of the 14th European Conf. on OO Programming (ECOOP 2000)*, pages 313–336. Springer, 2000.

[FHRS07] T. Focke, W. Hasselbring, M. Rohr, and J.-G. Schute. Instrumentierung zum Monitoring mittels Aspekt-orientierter Programmierung. In *Tagungsband SE 2007*, volume 105 of *LNI*, pages 55–58. Köllen Druck+Verlag, 2007.

[GCH+04] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004.

[GKM82] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *ACM SIGPLAN Notices*, 17(6):120–126, 1982.

[JSB97] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *ICSE '97: Proc. of the 19th Intl. Conf. on Software Engineering*, pages 360–370. ACM, 1997.

[KF05] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Trans. on Neural Networks*, 16(5):1027 –1041, 2005.

[MM04] A. Mos and J. Murphy. COMPAS: Adaptive Performance Monitoring of Component-Based Systems. In *2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS 04), 26th Intl. Conf. on Software Engineering (ICSE 2004)*, pages 35–40, 2004.

[MRvHH09] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic Failure Diagnosis Support in Distributed Large-Scale Software Systems based on Timing Behavior Anomaly Correlation. In *Proc. of the 2009 European Conf. on Software Maintenance and Reengineering (CSMR'09)*, pages 47–57. IEEE, 2009.

[OMG10] OMG. Object Constraint Language, Version 2.2. `http://www.omg.org/spec/OCL/2.2/`, 2010.

[PMM06] T. Parsons, A. Mos, and J. Murphy. Non-intrusive end-to-end runtime path tracing for J2EE systems. *IEE Proc. – Software*, 153(4):149–161, 2006.

[PP07] A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.

[RvHG+08] M. Rohr, A. van Hoorn, S. Giesecke, J. Matevska, W. Hasselbring, and S. Alekseev. Trace-context sensitive performance profiling for enterprise software applications. In *Proc. of the SPEC Intl. Performance Evaluation Workshop (SIPEW '08)*, volume 5119 of *LNCS*, pages 283–302. Springer, 2008.

[RvHH+10] M. Rohr, A. van Hoorn, W. Hasselbring, M. Lübcke, and S. Alekseev. Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. In *WOSP/SIPEW '10: Proc. of the 1st joint WOSP/SIPEW Intl. Conf. on Performance Engineering*, pages 87–92. ACM, 2010.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2008.

[Sna09] R. G. Snatzke. Performance Survey 2008 – Survey by codecentric GmbH. `http://www.codecentric.de/de/m/kompetenzen/publikationen/studien/`, 2009.

[TMD09] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[vHRH+09] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Department of Computer Science, University of Kiel, 2009.