A Self-adaptive Monitoring Framework for Component-Based Software Systems

Jens Ehlers and Wilhelm Hasselbring

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

Abstract. To allow architectural self-adaptation at runtime, software systems require continuous monitoring capabilities to observe and to reflect on their innate runtime behavior. For software systems in productive operation, the monitoring overhead has to be kept deliberately small. By consequence, a trade-off between the monitoring coverage and the resulting effort for data collection and analysis is necessary. In this paper, we present a framework that allows for autonomic on-demand adaptation of the monitoring coverage at runtime. We employ our self-adaptive monitoring approach to investigate performance anomalies in component-based software systems. The approach is based on goal-oriented monitoring rules specified with the OCL. The continuous evaluation of the monitoring rules enables to zoom into the internal realization of a component, if it behaves anomalous. Our tool support is based on the Eclipse Modeling Project and the Kieker monitoring framework. We provide evaluations of the monitoring overhead and the anomaly rating procedure using the JPetStore reference application as a Java EE-based test system.

Keywords: Adaptive monitoring, failure diagnosis, anomaly detection.

1 Introduction

Performance is a critical characteristic for software systems. Even though monitoring the operation of systems is often neglected in practice. A recent survey among Java practitioners and experts [10] indicates this antagonism: Adequate application-level monitoring tools that allow to analyze the causes of performance problems are seldom known and employed in software engineering projects.

It is difficult to decide in advance where to place the monitoring probes and which data should be collected. Thus, probes are typically instrumented only in reaction to prior performance degradations or system failures. In contrast to construction-time profiling, continuous monitoring at operation time has to regard a deliberately small monitoring overhead. Consequently, a main issue is the limited amount of information that can be collected and processed. More detailed monitoring data allows for more detailed analyses of the underlying software system's behavior. We evaluated and quantified the impact of how monitoring data is collected, processed, and persisted for subsequent analyses. A finding is that it is feasible to instrument probes at a variety of

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 278–286, 2011.

possibly relevant measuring points, as long as not all of them are active at the same time during operation.

In this paper, we present a self-adaptive, rule-based monitoring approach that allows on-demand changes of the monitoring coverage at runtime. The monitoring rules follow the goals for which monitoring data is required, e.g. the evidence of SLA compliance, dynamic adaptation of resource capacities, or usage pattern recognition for interface design. We will concentrate on the monitoring goal to localize performance anomalies that change the valid behavior of software system as perceived by its users. For the specification of the monitoring rules, we employ the Object Constraint Language (OCL) [8]. The rules refer to performance attributes of a previously extracted system runtime model. As the model values (particularly anomaly scores rating the timing behavior of system-inherent operations) change during operation, a continuous evaluation of the monitoring rules is required. Our implementation is based on EMF (Eclipse Modeling Framework)[1] meta-models which allow for evaluation of OCL query expressions on object-oriented instance models at runtime.

The remainder of this paper is structured as follows: In Section 2, we describe our approach for self-adaptive performance monitoring and the underlying anomaly rating procedure. Its evaluation in lab experiments and industrial systems is summarized in Section 3. Related work is discussed in Section 4. Finally, a conclusion and an outlook to future work are given in Section 5.

2 Self-adaptivity for Continuous Software System Monitoring

In this section, we present our approach for self-adaptive software system monitoring, which is embedded into our Kieker monitoring framework[2] [5]. Kieker facilitates to monitor and to analyze the runtime behavior of component-based software systems. The underlying monitoring and analysis process is structured into the following activities: probe injection, probe activation, data collection, data provision, data processing, visualization, and (self-)adaptation. Different plugins can be integrated into this analysis process via the pipes-and-filters pattern. In the following, we present a Monitoring Adaptation Plugin addressing rule-based adaptation of the current monitoring coverage.

To enable fine-grained monitoring of component-internal behavior, probes have to be instrumented at various measuring points in the components' control flow. At operation time with extensive system workload, it is not possible to process each probe actuation. Only a selection of the measuring points can be activated. An adequate initial coverage is to activate the measuring points that intercept the execution of system interface operations. Our proposed Monitoring Adaptation Plugin allows the specification of monitoring rules which are evaluated continuously and may effect changes of the current monitoring coverage. We will specify a rule that the coverage of a component's interior control flow should be increased if it does not behave as expected. In this way, our approach affords automatic on-demand adaptation of the effective software system monitoring.

[1] http://www.eclipse.org/modeling/emf/
[2] http://kieker.sourceforge.net/

If the monitoring adaptation is conducted manually, the human decision to change the set of active measuring points is usually caused by (critical) incidents that imply anomalous runtime behavior. A performance engineer who observes such an incident is interested in the root cause and tries to activate more measuring points in the affected components. Subsequently, it takes a while until enough relevant records have been collected via the newly activated measuring points. It is well-known that a major part of the failure recovery time is required to locate the root cause of a failure. An estimation of 75% of the recovery time being spent just for fault localization is referred to in [6]. Our self-adaptive monitoring approach will reduce this potentially business-critical wait time that delays a failure or anomaly diagnosis.

2.1 Runtime Evaluation of OCL-Based Monitoring Rules

We employ the OCL to specify the monitoring rule premises. OCL is well known for its purposes to specify invariants on classes, pre- and postconditions on operations, or guards in UML diagrams. Nevertheless, the first objective listed in the OCL specification suggests OCL to be a query language [8]. In our case, the monitoring rule premises can actually be regarded as queries that select a set of measuring points to be activated or deactivated. The Monitoring Adaptation Plugin provides an editor with syntax highlighting and code completion for performance analysts to specify the required OCL expressions. The context in which an OCL expression will be evaluated is determined by a selectable context element. In the expression, the context element can be referenced by the OCL identifier self. Appropriate context elements are the analysis models of other Kieker plugins which provide the input of the Monitoring Adaptation Plugin. As all plugins are based on EMF meta-models, we are able to utilize the EMF Model Query sub-project, which allows constructing and running queries on EMF models by means of OCL. Goal-oriented self-adaptation is based on the possibility to refer to attributes in the OCL expressions that change their values during runtime, e.g. responsiveness metrics and derived anomaly scores.

P_1: **context** CallingContextTree:
self.callingContexts→*select*(level = 1 **or**
(parent.op.monitored **and** parent.anomalyScore > *t*))
→*collect*(op)

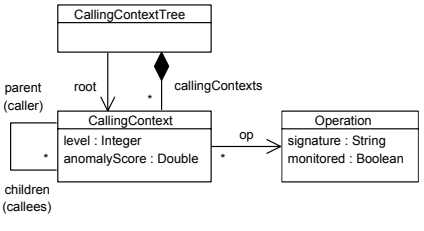

Fig. 1. Monitoring premise and corresponding simplified CCT meta-model

The Monitoring Adaptation Plugin runs a thread that evaluates the specified monitoring rules. The time interval between succeeding evaluations of the monitoring rules has to be configured manually. Enough time is required to collect reliable new monitoring data in each iteration. On the other side, the delay time must be short enough to react promptly to observed anomalies. An interval value in the scale of a couple of minutes is appropriate. Probe (de)activation instructions are delegated to the monitoring agent instances via their remote adaptation interface.

In the following, we discuss an example monitoring rule: "If an operation is selected by the rule premise P_1 as described in Figure 1, then activate the probe measuring point required to intercept and monitor calls to this operation." The context element of P_1 is a calling context tree (CCT) model. The simplified meta-model of a CCT is depicted in Figure 1. P_1 selects all operations that are called from a caller operation that is already monitored (parent.op.monitored) and behaves anomalous in a particular calling context (with an unique call stack), i.e. the context's anomaly score exceeds a specified threshold t (parent.anomalyScore $> t$). Additionally, all operations are added to the result set that are at the topmost level of the CCT (level = 1), i.e. system-level interface operations for incoming client requests.

2.2 Software Performance Anomaly Rating

The above monitoring rule references an operation-level anomaly score metric. As it strongly depends on the context if an observation has to be considered as anomalous or not [2], we capture and separate different contexts as far as possible (e.g. separation by calling context), but we assume that it is not possible to separate all context-determinant impact factors (e.g. operation input parametrization, component state, system workload). Even from a fine-grained contextual viewpoint, response times can be arbitrarily distributed and do not necessarily converge to a parametric distribution model. Thus, we suggest an anomaly rating procedure based on time series analysis that disregards any technical or economical influences. The characteristic features of the underlying stochastic process are recovered from the present time series of response times. Our anomaly rating procedure consists of four steps, which are summarized in the following and explicated in detail in [3]:

(1) Forecast expected response times for each software service in dependence of the stack context based on historical observations. We provide different forecast models such as single exponential smoothing (SES), Holt-Winters smoothing, and ARIMA models.

(2) Test if a sample of newly measured service response times is to be rated as normal or anomalous related to the expected forecast value from (1). A Student's t-test is conducted based on the measured sample variance.

(3) Based on the sequent rating of response times samples from (2), calculate an anomaly score expressing the recent degree of a software service to exhibit anomalous timing behavior. Here, we construct an anomaly scoring function that reflects the frequency and the trend of anomalous samples over time.

(4) Aggregate and correlate anomaly scores from (3) to higher levels of abstraction, e.g. component-level anomaly scores.

3 Evaluation

We employed the Kieker Monitoring component in the productive systems of a telecommunication company and a digital photo service provider [5]. These previous case studies confirmed the practicability and the robustness of our approach. Regarding

the monitoring cost, our industrial partners were not able to perceive any monitoring overhead due to the instrumentation of our probes. Thus, we set up lab experiments to quantify the monitoring overhead and to evaluate the self-adaptive anomaly detection. The results of these evaluations are recapped in the following.

Monitoring Overhead: The goal of our monitoring cost evaluation is to quantify and to decompose the monitoring overhead. In Figure 2, we apportion monitoring costs for instrumentation (Δ_I), data collection (Δ_C), and data logging (Δ_L). In the experiment, we monitored an operation that takes 500 μs to be processed on a specific test system (Sun Blade X6270 with 2x Intel Xeon E5540, total 8 cores at 2.53 GHz, 24 GB RAM, ZFS RAID, SunOS 5.1, Java HotSpot x86 Server VM 1.6). The response time deviation is minimal as we carried out an extensive warm-up phase to saturate the JVM behavior, particularly the just-in-time compilation. In the experiment, the monitored operation was continuously executed by 15 concurrent threads. The boxplots show that (1) instrumentation, i.e. processing previously woven, but inactive dummy probes, causes negligible overhead (Δ_I is less than 1 μs) compared to (2) data collection and (3) logging, i.e. creating and persisting the monitoring records to a Monitoring Log (Δ_C and Δ_L are each about 4 μs). In case (3) of the experiment where logging is enabled, the records were written asynchronously into the local file system by a dedicated writer thread. This avoids a direct delay of the response time perceived by the system users. The remaining logging overhead is effected by the thread concurrency. Our evaluation results suggest the conclusion that injecting probes at a multitude of measuring points is not critical as long as data collection and logging can be (de)activated systemically. This finding underpins the adaptive activation of measuring points proposed in Section 2.1.

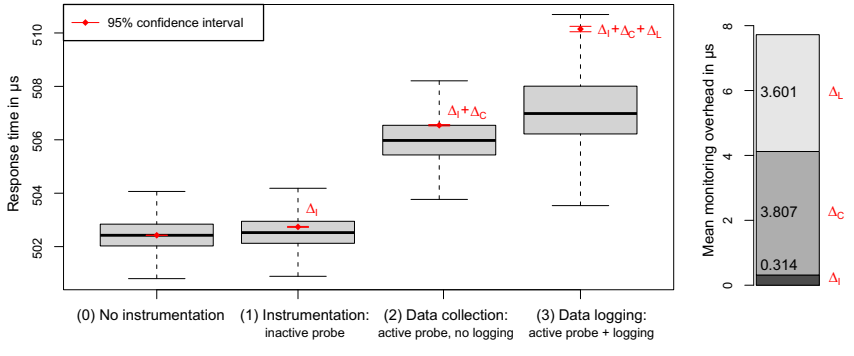

Fig. 2. Evaluation of the monitoring cost

Anomaly Detection: In [3], we presented our evaluation results for different forecast models. Here, we show that consecutive divergences of measurements and forecasts indicate anomalous timing behavior. For our evaluation, we use the JPetStore[3] reference application as a test system. Initially, we monitor only the system's interface operations.

[3] `http://sourceforge.net/projects/ibatisjpetstore/`

Fig. 3. Kieker screenshots with calling context tree (left) and responsiveness time series (right)

In our experiment, we stress the system under test with workload that causes a desirable level of resource utilization, i.e. the CPUs are continuously utilized in a range between 30% and 50%. Given this load, the system was run and observed for 2 days, using SES for forecasting with a fixed smoothing factor of 0.15. The anomaly scores of the monitored services did not exceed a threshold value of 0.5. That is, there have never been 7 of 10 subsequent samples that were rated anomalous. For example, the mean response time of the viewProduct service was 48.7 ms with a standard deviation of 10.3 ms. A response time curve of this operation is depicted in the top right view part of Figure 3. The green line indicates the observed mean response time surrounded by a light green confidence interval whose range depends on the observed variance and the specified significance level. The blue line indicates the expected response time determined by the used forecast model. During failure-free operation, the forecast value is mostly within the confidence interval. By consequence, the anomaly score indicated by the red line does not rise considerably. In the very right part of the depicted time series, a fault is injected that disrupts the failure-free operation. Suddenly, response times nearly double. Without this pattern being expected in advance, the forecast model

adapts itself only moderately to the new timing behavior. In this period, measurement and forecast diverge so that the anomaly score increases rapidly and exceeds the defined alarm threshold.
This situation is captured in the top left view part of Figure 3, where an extract of the system's calling context tree is shown. The color of the operation nodes from green to red indicate their current anomaly score. Operations that are hitherto not monitored are colored in light gray. As a consequence of the anomalous behavior of the viewProduct and viewItem operations, the evaluation and appliance of the monitoring rule P_1 explicated above leads to the monitoring activation for their callees. As shown in the bottom left view part of Figure 3, the monitoring coverage is adapted to localize the root cause of the anomalous behavior. In the scenario depicted in the screenshots, both anomalous interface operations depend on a common operation called getExchangeRate effecting the anomaly. The time series in the bottom right view part of Figure 3 demonstrates that the getExchangeRate operation has not been monitored continuously until the anomaly occurred. Only sparsely distributed sample observations have been made to check over the learned expected behavior. In the discussed evaluation scenario, we systematically changed the responsiveness of the external web service which is invoked from inside of the getExchangeRate operation. Though we injected this fault on purpose, a similar incident can easily occur in a productive system that depends on third-party services.

Furthermore, we studied two more fault injection scenarios, which are not described in detail due to space restrictions: In the first scenario, we dropped and recreated a database index causing significant changes in the response time of several operations called from lower levels of the system's CCT. We kept track of how our self-adaptive monitoring approach successively adapts the monitoring coverage to zoom in and out the CCT. In a second scenario, we increased the overall system load abruptly to simulate a situation where a load-balanced system replica drops out and the remaining replicas have to absorb the capacity reduction. As in our experiment setup particularly the CPU resources have not been heavily underutilized, almost all expensive operations react anomalous. It is obvious that a manual exploration of such cause-and-effect chains as constructed in our experiments is much more time-consuming and error-prone than an automated processing. A major contribution of the self-adaptive monitoring approach is to save this time and effort.

4 Related Work

An integrated software system monitoring framework such as Kieker is concerned with two aspects: (1) monitoring, i.e. instrumentation and data acquisition, and (2) subsequent analysis. Related work comprises the COMPAS JEEM project [9] which facilitates the injection of probes as a component-level proxy layer in Java EE systems. In the context of COMPAS, adaptation of the monitoring coverage at runtime is studied in [7]. However, monitoring is restrained to the interface level of Java EE components such as EJBs or Servlets. By the observation of component-internal operation responsiveness, Kieker allows a finer-grained insight.

Concerning application-level fault determination without addressing self-adaptation, related work is provided by the Pinpoint approach [6]. In contrast to Kieker, Pinpoint does not focus on performance time series, but applies pattern-oriented data mining techniques to detect anomalies in the request traces. A further related approach addressing monitoring of resource utilization and component interactions in distributed systems is Magpie [1]. While the implementation of Kieker concentrates on Java-based systems, Magpie is realized to monitor systems based on Microsoft technology. The Rainbow project [4] employs monitoring for architecture-based adaptation of software systems. To our knowledge, Magpie, Pinpoint, and Rainbow so far do not contribute means for rule-based self-adaptation to control the monitoring coverage. The same applies to related commercial products like CA Wily Introscope, DynaTrace, or JXInsight. The popular open-source tool Nagios is intended for infrastructure monitoring, not for application-level monitoring.

5 Conclusions and Future Work

Responsiveness and scalability of a software system components have to be monitored and analyzed continuously. In case a system component responds anomalous, our self-adaptive monitoring approach enables zooming into the component-internal behavior on demand. Zooming means to activate more (or less) measuring points in the application-level control flow aiming at increasing (or decreasing) insight, e.g. into the operation call stack, effective loop iterations, or conditional branches taken. A set of OCL-based monitoring rules is proposed to control the monitoring coverage automatically. In Section 2, we presented an extension to our Kieker monitoring framework supporting self-adaptive software system monitoring based on the continuous evaluation of OCL-based monitoring rules at runtime. Further, we briefly described our underlying anomaly rating procedure for the timing behavior of software systems. In Section 3, we quantified the monitoring overhead and evaluated the anomaly detection procedure in lab experiments.

In our future work, we plan to study the adaptive monitoring approach in case studies with industrial partners. Further, we intend to implement unsettled practical issues concerning our Kieker monitoring framework such as model-driven instrumentation, IDE integration, and support for other programming languages in addition to Java.

References

1. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for request extraction and workload modelling. In: Proc. of the 6th Conf. on Symposium on Operating Systems Design & Implementation, pp. 259–272. USENIX (2004)
2. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: A survey. ACM Computing Surveys 41(3), 1–58 (2009)
3. Ehlers, J., van Hoorn, A., Waller, J., Hasselbring, W.: Self-adaptive software system monitoring for performance anomaly localization. In: Proc. of the 8th IEEE/ACM Intl. Conf. on Autonomic Computing (ICAC 2011), pp. 197–200. ACM, New York (2011)

4. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
5. van Hoorn, A., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., Kieselhorst, D.: Continuous monitoring of software services: Design and application of the Kieker framework. Tech. Rep. TR-0921, Dept. of Computer Science, University of Kiel (2009)
6. Kiciman, E., Fox, A.: Detecting application-level failures in component-based internet services. IEEE Trans. on Neural Networks 16(5), 1027–1041 (2005)
7. Mos, A., Murphy, J.: COMPAS: Adaptive performance monitoring of component-based systems. In: 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems, 26th Intl. Conf. on Software Engineering (ICSE 2004), pp. 35–40 (2004)
8. OMG: Object Constraint Language, Version 2.2. http://www.omg.org/spec/OCL/2.2/ (2010)
9. Parsons, T., Mos, A., Murphy, J.: Non-intrusive end-to-end runtime path tracing for J2EE systems. IEE Proc. – Software 153(4), 149–161 (2006)
10. Snatzke, R.G.: Performance survey 2008 – survey by codecentric GmbH (2009), http://www.codecentric.de/de/m/kompetenzen/publikationen/studien/