An Aspect Oriented Programming Framework to Support Transparent Runtime Monitoring of Applications

Abdullah O. AL-Zaghameem
Department of Computer and Information Technology
Faculty of Science, Tafila Technical University, Tafila, Jordan

***Abstract*—Monitoring the runtime state and behavior of applications is very important to evaluate the performance of these applications and to inspect their behavior. In case of legacy applications that have been developed without monitoring capabilities, there is a real challenge to accomplish runtime state monitoring. This research redefines runtime monitoring concept, and then presents an Aspect Oriented Programming (AOP) framework to equip applications with the capabilities to monitor their runtime state transparently. The framework, called RM Framework, supports three monitoring modes; Invasive-mode, Controlled-mode/(Functionality and Attribute), and Controlled-mode/Selective. The framework is applied on a Java application as a case study. The results show smooth integration between application and runtime monitoring capabilities without affecting the target application consistency.**

Keywords*—Runtime state monitoring; application behavior; aspect oriented programming technique; statistical analysis; bytecode transformation

I. Introduction

Runtime monitoring has been defined in [1] as "the act of observing an executing system in order to learn something about its dynamic behavior". In this research, runtime monitoring, as a term, will be used to point out the state of executing application at a specific moment or during a period of time. The runtime state of an application includes information about components, amount of data processed during execution, and resource consumption; like CPU time and memory. Monitoring the runtime state of applications has several benefits like understanding and analyzing of software behavior [2, 3], detection of performance problems and bottlenecks [4-6], and building applications' execution history and datasets [7].

Monitoring the runtime state of legacy applications is a very challenging mission. From one side, the source code of these applications most probably is unavailable, which makes it hard to perform white-box monitoring. From the other, a modular and systematic mechanism is required to perform smooth monitoring without violating the functionality and structure consistency of application. In this context, the Aspect Oriented Programming (AOP) [8] is vital and efficient technique. Because its capability to intersect the execution of application at several points, the behavior and runtime state of that application could be inspected. In this research, the runtime state monitoring is presented as an AOP aspect, and a framework is developed to serve monitoring the runtime state of applications developed using Java™ programming language. The research is partially guided by the fundamentals of runtime monitoring presented in [3].

The paper is organized as follows: Section 2 discusses the runtime state of object-oriented applications and redefines the term "runtime state monitoring". In Section 3, the term "Runtime Monitoring Aspect" is introduced, and a framework called Runtime Monitoring Framework is presented in Section 4. The section presents the details of framework. Section 5 applies RM framework on application as a case study. Section 6 discusses some related works. Finally, Section 7 concludes research results and lists some limitations.

II. Runtime State of Object-Oriented Applications

The "runtime state" points out the statistical and behavioral measurements of software execution at a given point of time, or during a period of time. As for statistical measurements, countable amounts of data (e.g. counters, data sizes, etc) during application execution need to be recorded and stored for further analysis. Collecting data will help answering questions like "How many times ..?" and "How much data ..?" For behavioral measurements, collecting specific data and observing links between components could facilitate the description of application behavior.

Inspecting runtime state of applications assists monitoring their performance, detecting any possible structural deformations or functional bottlenecks, and establishing execution history and test cases. In order to enable applications to record runtime state measurements, either they had been programmed to do so, or they should be *modified* and provided with the capabilities to record runtime state measurements. A real challenge arises for the second case; especially for legacy applications which their source code is missing. Providing these applications with the capability to record runtime state requires *injecting* the necessary monitoring code; taking into account not to violate application structure or functionality consistency.

III. Runtime Monitoring Aspect

To emphasize the runtime state concept and highlight its importance to software applications, the term "Runtime Monitoring" will be introduced as an AOP *Aspect*. The aspect represents the process of recording runtime state measurements. Fig. 1 illustrates a pseudo code for the Runtime Monitoring Aspect (RMA). As shown in the figure, the aspect defines four pointcuts that determine the locations (join points (JPs)) at application code where runtime state measurements need to be recorded. The first location is just before any method call. In RMA definition, JP1 represents this case. JP1 aims to collect information about the called method and the arguments passed to it to calculate the amount of data flow.

The second location is when a call to the method is completed. Information about the returned value(s) could be recorded. JP2 represents this case, where **ret** identifier points to the value returned after executing method **m**. In addition, recording successful execution of methods, as well as failed executions, is important to observe application behavior. The third important part in code that affects the runtime state of application is when new objects are instantiated. Collecting information about objects' creation is not only important for statistical analysis, but also for monitoring application behavior and resources consumption. For this purpose, JP3 represents this concern.

In object oriented software, it is important to monitor the access of object's fields (or attributes). In order to monitor fields' access operations, type of access (either read or write) and the points in code at which a specific field is accessed need to be determined. For this purpose, JP4 in RMA inspects the occurrence of access operations and their types.

The definition of RMA listed in Fig. 1 is a generic case of interception. In other words, it defines how to apply runtime state monitoring overall the application functionality and behavior. For more applicability options of RMA, we suggest applying the following monitoring modes:

1) The invasive mode: In this mode (the default) all application components are put to monitoring. The mode represents the comprehensive monitoring if all functionalities are to be monitored.

2) The controlled mode: In this mode, we can monitor partial parts of runtime state. In this case, more concentration is oriented toward specific functionalities. In practice, this mode could be divided further into the following sub-modes:

a) The Functionality Mode: In which only objects' methods are monitored for the sake of recording statistical data about application behavior and data flow.

b) The Attribute Mode: This mode targets the monitoring of object's fields. It records measurements about fields' access operations. This mode can help tracking the access of objects' fields and how their values changed over runtime.

c) The Selective Mode: This mode could be considered as a custom mode. It supports the monitoring of the access of specific fields, and the execution of specific methods. This specialized mode provides the flexibility to monitor complex and large-scale applications as individual parts.

```
Aspect Runtime_Monitoring_Aspect
{
JP1: before *.call(m, args) →updateRuntimeState(m, args);
JP2: after  *.call(m, ret)  →updateRuntimeState(m, ret);
JP3: after  newobj                    →updateRuntimeState(obj);
JP4: on field access f          →updateRuntimeState(acc, f);
 Advice1: updateRuntimeState(Method m, Args[])    { .. }
 Advice2: updateRuntimeState(Method m, Object ret) { .. }
 Advice3: updateRuntimeState(Object obj)          { .. }
 Advice4: updateRuntimeState(AccessType acc, Field f)  { .. }

}
```

Fig. 1. Runtime Monitoring Aspect (RMA)–A Pseudo Code.

IV. RM Framework (RMF)

In this research, the static bytecode transformation approach is used to realize the RMA concepts. The static bytecode transformation performs the necessary bytecode modifications on the target software to produce a new version of that software by injecting the RMA mechanism. In this section, a detailed description of the realization of RMA as a framework will be presented.

A. Framework General Overview

Fig. 2 illustrates the general structure of RM Framework. The framework works at two main phases where a set of operations are performed. At the first one, called static phase, the target application (i.e. the software to be monitored) is given as input to the RMA Injection Unit (see Fig. 2), which performs three main operations:

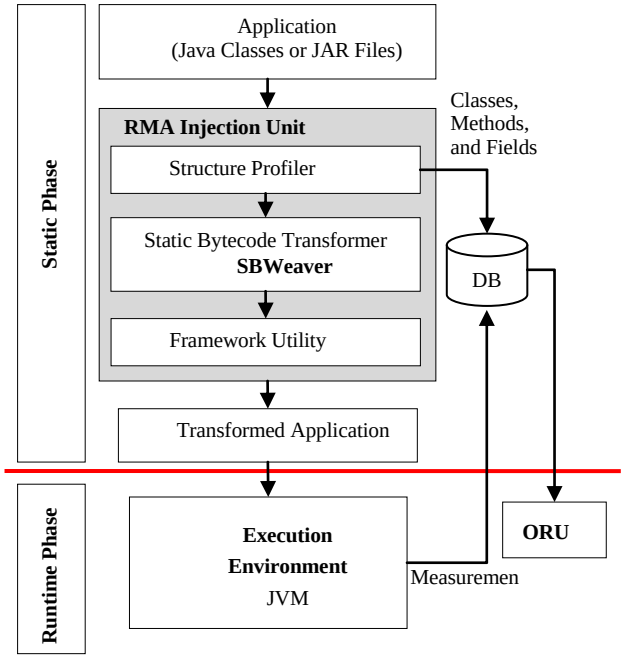

Fig. 2. The Structure of Runtime Monitoring (RM) Framework.

1) Analyzing the input software: At this point, the bytecode of the target software is inspected and analyzed. In other words, the structural units of the software (i.e. Java classes) are enclosed, and for each recognized class the member methods, member attributes (fields), and class constructors are inspected. The "Structure Profiler" is responsible of building a structure database for the entire software. This step is very important as it gives a road map that assists describing software classes, and helps manipulating classes easily. A full application profile is generated and stored in the database as a result.

2) Once the structural database of the software is built: The bytecode of software classes is then transformed. The "SBWeaver" transforms software classes and, for each class, it injects the necessary code required to realize the RMA at the specific join-points.

3) To facilitate gathering the measurements of runtime state: The framework utility code is added to the transformed application. The utility code serves mainly connection to database.

The output of static phase is a transformed version of the original application. In this context, it is important to assure the consistency of application structure and functionality transformation. This is to say, application should perform what it has been developed for without knowing it has been transformed.

The second phase is the runtime phase. The transformed application is executed, as normal, on top of its executing environment. During execution, the environment will gather and store the measurements of application runtime state in database.

B. SBWeaver

The SBWeaver is the centric component in RM framework. It performs code transformation. It follows the algorithm depicted in the pseudo code of Fig. 3.

As this research targets Java applications, the Byte Code Engineering Library (BCEL) [9] is used. BCEL is an open source framework for Java bytecode transformation. The RM framework uses BCEL version 6.2.

```
1  SBWEaver(ApplicationPath appPath)
2  {
3    For each class claz in the appPath do
4     If not already_transformed(claz) then
5    For each method m implemented by clazdo
6       | Inject advices 1,2, and 3in method m according to
        | joint-points declarationsJP-1, JP-2, and JP-3
        | respectively
7      For each field f declared in class claz do
8       | Determine access operations (get or set) on fin
        | method m, and injectadvice4 properly
9     End For
10   End For
11     | Mark claz as transformed
12   End For
13 }
```

Fig. 3. A Pseudo Code of SB Weaver at the Invasive-Mode.

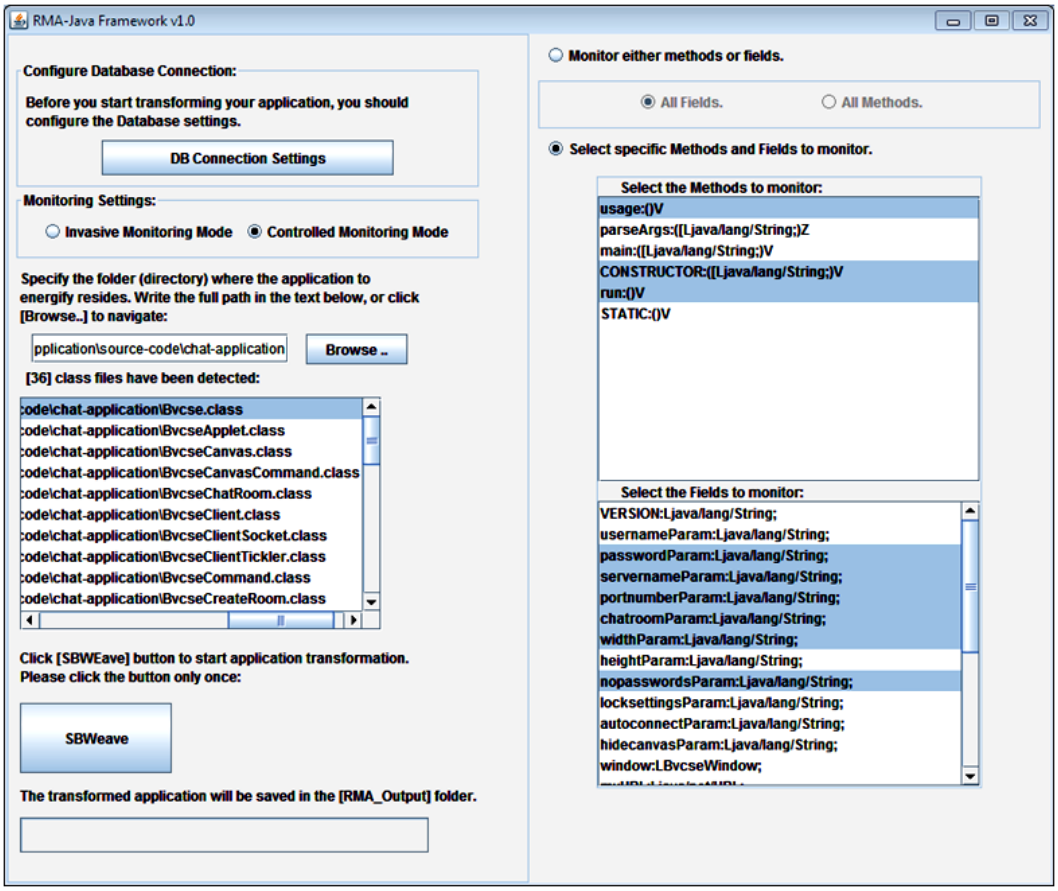

Fig. 4. The RMA Framework in Action.

The SBWeaver needs to know only where application classes reside. The BCEL is used to facilitate inspection of Java bytecode instructions in easy and powerful way. Therefore, the SBWeaver can traverse, for example, all access operations of specific fields. In Fig. 3, the SBWeaver enumerates the access operations on a field, and then injects the necessary code to collect the measurements of field access. The pseudo code shown in Fig. 3 depicts the SBWeaver at the Invasive-Mode. As for the step at line 11, transformed classes are marked *transformed* so they will not be transformed twice. For this purpose, the SBWeaver adds an implementation of the empty interface *ITTU_SBWeaverMark*, which is used as a marking interface.

The output of SBWeaver is a transformed version of the application that is ready to be monitored at runtime. Fig. 4 illustrates the RMA framework GUI in action. The framework has been implemented totally using Java. Note the options of monitoring. In Controlled-Mode, users can select to perform monitoring (per class) either on methods or fields only. For fine-grain selection, users can select subset methods and subset fields to be involved in runtime monitoring. This flexibility in selection can increase target application suitability for monitoring.

C. Output Representation Unit

During the execution of transformed application, runtime measurements and behavioral measurements are recorded. The Output Representation Unit (ORU) is part of RM framework and is responsible of representing these measurements in various ways. The ORU makes it easy to browse the results in proper and compatible way. However, users of RM framework can issue their customized reports as they have access to the database.

V. Case Study: Monitoring a Chatting Application

An experiment has been conducted to test the RMA concepts on an object-oriented application. The application, called *Bvcse*, is a chatting application in Java. It is open source and free to download from the Internet.

A. The "Bvcse" Chatting Application

The application consists of two parts; the server program, which coordinates and organize connections between clients. The second part is the client program, which allows users to chat in public global rooms or privately in user rooms. It allows chatters to share a drawing panel that provides a simple drawing toolbox to draw colorful sketches. Both programs are having GUI to facilitate chatting features. In total, "Bvcse" application includes (36) Java classes.

The application is an interactive application that broadcasts global chat messages and syncs the drawing board to all clients. Therefore, it has been selected to check monitoring applications with intensive interaction with different monitoring modes. Next section discusses how we apply the RMA monitoring modes on "Bvcse" application.

B. Applying RMA on "Bvcse"

The RM Framework is used to augment "Bvcse" application with runtime state monitoring capability several times; each time with a specific monitoring mode. For each case, the application is then deployed on five machines; four for clients (chatters) who do not know that the application is being monitored, and one for server.

1) Invasive monitoring mode: In this round, all the 36 classes of "Bvcse" application have been transformed. The SBWeaver excludes automatically all Interface and Enumeration classes. The direct impact of applying this mode is the inflation of application classes. Depending on the number of fields defined and methods implemented, the SBWeaver injects new code chunks. For example, the size of class "Bvcse.class" before transformation was (4,351 Bytes) and after transformation becomes (8,881 Bytes). All in all, the more fields defined by application class the more increasing in transformed class size. This is expected because SBWeaver injects for each field two methods; one to get field's value and one to set a new value. In addition, all field access operations (gets and sets) in the original code will be replaced by calls to getter and setter methods.

2) Controlled mode [functionality and attribute]: The experiment is performed twice in this round; the first one to monitor method executions and the other to monitor field access operations. As for monitoring methods executions, the application works fine and all statistics have been recorded and stored in the database during the experiment duration (one hour). For field access monitoring, the application performance suffers from periodic congestion at clients and server sides; especially when syncing the drawing panel. Once again, a large number of fields need to be accessed upon message sending or drawing panel synchronization; which causes these congestions.

3) Controlled mode [selective]: In this round, three fields and three methods from each class have been randomly selected for monitoring (a customized set of fields and methods could be selected per class). The transformed application is then deployed for execution. As expected, the application works fine without any "stutters" as clients experience no delays. The statistical information gathered for the monitored fields and methods are recorded and stored in the database during experiment. Fig. 5 shows a short snippet for statistical field access as appeared in the database.

C. Results and Discussion

According to experiment results, it is obvious that monitoring in the Invasive-Mode is improper for interactive and dense applications. As expected, the connection traffic was very dense and the performance of all five copies suffers strong bottlenecks. Technically, the experiment of round one comes out with the worst results; because large set of fields need to be accessed at the server and client programs each time a new message or new drawing panel update arrive. For RM framework, this means a connection to the database is required to execute update query for the corresponding accessed field; which means extra runtime.

fieldID	classID	appID	fieldName	fieldSignature	fieldACC	Put	Get
18	2	5	windowWidth	I	V	39	6
19	2	5	windowHeight	I	V	34	6
23	2	5	showCanvas	Z	V	7	120
17	2	5	room	Ljava/lang/String;	V	7	79
22	2	5	autoConnect	Z	V	5	5
21	2	5	lockSettings	Z	V	5	5
20	2	5	requirePasswords	Z	V	5	5

Fig. 5. Statistical Measurements of Some Fields as Appeared in the Database.

However, in all rounds, application consistency (structure and hierarchy) has been preserved. In addition, users give no comments on differences in application functionality; as they use the original application before applying RMA. Therefore, RM framework supports the monitoring transparency claimed in this research.

VI. Related Works

Several researches have been conducted to deal with application runtime states. In the context of applications behavior, the authors in [10] have presented *blended program analysis* as a new paradigm to analyze large framework-based Java applications. The researchers have designed a blended escape analysis for approximating the effective lifetimes of objects. The approach was supposed to help explain how temporary structures are built and used. We found a relevancy between our work and their work from the purpose. This research collects information about the creation of objects statistically, while blended analysis inspects objects' lifetime lines.

A new metric to measure the degree of cohesion in relation to objects of a class at runtime has been presented in [11]. A runtime cohesion metric called LCOM-Desouky has been defined and experimented on Rhino 1.7R4 – an open source JavaScript framework written in Java. The study results in a large negative correlation between the metric and tested data. The author did not mention how they collect statistics for metric calculations at runtime. Our framework explains this step in details, and provides a modular and cohesion method for collecting runtime statistics. We do believe that more metrics could be constructed according to runtime statistics collected by RM framework.

A runtime state fetching method has been proposed in [5]. The authors designed a language called State Fetching Description Language (SFDL) to express monitoring requirements, and implemented a framework to compile SFDL rules into monitor probes which gather information and store them. The approach has been applied on distributed software and some performance bottlenecks have been detected. The approach is similar to RM framework from the point that both of them present runtime monitoring as a separate concern. Our approach, however, is transparent; there is no need to a description language such as SFDL to describe what to monitor. Instead, RM framework provides three monitoring modes.

The work proposed in [12] presents a similar approach to the one in [5]. The proposed technique uses Online Evolution Module (OEM) which receives monitoring information and compares state event to pre-defined evolution rules. It performs evolution actions and real-time corrections if these rules are triggered. In [13], the author presented a performance analysis of large-scale object-oriented software by finding repeated patterns of dynamic behavior in calling context tree (CCT) extracted from software profile data. In the tested application with over 64 thousand unique calling contexts, 10 patterns account for over 50% of application execution.

The authors in [3] presented a survey of software runtime monitoring. The research introduced the fundamentals of runtime monitoring; which include the architecture of a monitoring system, and the monitoring levels. The study introduced, classified, analyzed, and compared the typical software runtime monitoring methods. The research presented some problems related to runtime monitoring methods, and gave some future directions.

VII. Conclusions and Future Work

Monitoring the runtime state of applications is important to study the behavior of these applications. Collecting runtime measurements is one of the vital methods to perform this task. In this research, Runtime Monitoring Aspect (RMA) has been presented, which describes how and when to collect statistical data about runtime state of object-oriented applications transparently. The research introduced the RM Framework as an implementation to RMA. The framework suggests three monitoring modes; the Invasive-Mode to monitor the overall application runtime state. The Controlled-Mode/(Functionality and Attribute) monitors method executions, which helps inspecting application behavior and tracks field access operations. Finally, the Controlled-Mode/Selective allows monitoring customized set of fields and methods.

An experiment has been conducted to apply RMA on a chatting application written in Java. In the experiment, all monitoring modes have been applied. Because the chatting application is highly interactive, bottlenecks and congestion problems appear when applying Invasive-Mode and Controlled-Mode/Attribute.

The RM framework has some limitations. First, it injects extra code into target applications, which may inflate their sizes. If verification on class sizes is an issue, then RM framework causes a violation. The framework is limited in scope because it targets applications developed in Java. For future work, more applications need to be monitored by RM framework.

REFERENCES

[1] Nusayr and J. Cook, "AOP for the Domain of Runtime Monitoring: Breaking Out of the Code-Based Model", DSAL'09, March 3, 2009, Charlottesville, Virginia, USA,ACM. (2009)

[2] J. Sundararaman and G. Back, "HDPV: interactive, faithful, in-vivo runtime state visualization for C/C++ and Java", SOFTVIS 2008, Herrsching am Ammersee, Germany, September 16–17, 2008. (2008)

[3] L. Gao, *et. al.*, "A Survey of Software Runtime Monitoring". The 8th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, 2017, pp. 308-313. IEEE. (2017)

[4] G. Franks, *et. al*., "Layered Bottlenecks and Their Mitigation", Third International Conference on the Quantitative Evaluation of Systems (QEST'06), IEEE. (2006)

[5] G. Changguo and W. Tao, "A Method and Framework for Fetching Software Runtime State", International Conference on Computer, Mechatronics, Control and Electronic Engineering (CMCE), IEEE. (2010)

[6] J. Xu, "Rule-based Automatic Software Performance Diagnosis and Improvement", WOSP'08, June 24–26, 2008, Princeton, New Jersey, USA, ACM. (2008)

[7] H. Rajan and K. Sullivan, "Aspect Language Features for Concern Coverage Profiling", In Proceedings of the 4th international conference on Aspect-oriented software development (AOSD '05). ACM, New York, NY, USA, 181-191. (2005)

[8] G. Kiczales, *et. al*., "Aspect-Oriented Programming", In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997. (1997)

[9] M. Dahm, "Byte Code Engineering with the BCEL API", Freie Universitaat Berlin, Institut fur Informatik, (Technical Report B-17-98), April 3, 2001. (2001)

[10] B. Dufour, B. Ryder, and G. Sevitsky, "Blended Analysis for Performance Understanding of Framework-based Applications", ISSTA'07, July 9–12, 2007, London, England, United Kingdom. ACM. (2007)

[11] A. Desouky and L. Etzkorn, "Object Oriented Cohesion Metrics: A Qualitative Empirical Analysis of Runtime Behavior", ACM SE '14, March 28 - 29 2014, Kennesaw, GA, USA. (2014)

[12] L. Zhen-Dong, et. al., "Bytecode-based Software Monitoring and Trusted Evolution Programming Framework", 2012 IEEE Symposium on Robotics and Applications (ISRA), Kuala Lumpur, 2012, pp. 494-497. IEEE. (2012)

[13] D. Maplesden, "Performance Analysis of Object-Oriented Software", ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India. ACM 978-1-4503-2768-8/14/05. ACM. (2014)