Computer and Information Science; Vol. 9, No. 1; 2016
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Improving Analysis and Visualizing of JVM Profiling Logs Using Process Mining

M. M. MohieEl Din[1], Neveen I. Ghali[1], Mohamed S. Farag[1] & O. M. Hassan[1]

[1] Department of Mathematics, Facility of Science Al-Azhar University Cairo, Egypt

Correspondence: Mohamed S. Farag, Department of Mathematics, Facility of Science Al-Azhar University, Nasr city, 11884, Cairo, Egypt. Tel: 20-1006-574-243. E-mail: mohamed.s.farag@azhar.edu.eg

Received: November 23, 2015 Accepted: December 28, 2015 Online Published: January 6, 2016

doi:10.5539/cis.v9n1p54 URL: http://dx.doi.org/10.5539/cis.v9n1p54

Abstract

Growing size and complexity of modern software applications increase the demand to make the information systems self-configuring, self-optimizing and with flexible architecture. Although managed languages have eliminated or minimized many low-level software errors there are many other sources of errors that persist. Java Virtual Machine (JVM), as managed language has many adaptive optimization techniques, which needs tools to analysis program behavior determines where the application spends most of its time. In this paper, new approached has been introduced to use process-mining techniques to represent the analysis and visualize phases of JVM profilers. They are flexible enough to cover so many perspectives in several ways. That can form a unified layer for analysis and visualize across profiling.

Keywords: JVM profiling, process mining, heuristics miner

1. Introduction

The modern software applications are complicated enough that leads to increasing the demand for automating the process of managing the software environments that allows developers to identify performance bottlenecks with minimum effort.

Likewise for the Java Virtual Machine (JVM), as managed language based on interpreter it requires more processing for execution, and there are several approaches to enhance JVM performance like Just-In-Time Compilation (JIT), interpretation directly in hardware by specialized architecture and improving JVM performance by understanding of the behavior of Java-based applications (Bowers & Kaeli, 1998). Optimizing the compilers and software applications by understanding the dynamic behavior of it; it is an effective approach (Driesen et al., 2003).

The process of automatic collection and presentation of data that is representing the dynamic behavior of the program is called profiling (Dmitriev, 2004). After profilers collect and analyze the data, it can be either automatically feedback to the compiler or present it for the developers. Each case has different requirements in designing the profiler (Liang & Viswanathan, 1999). For example the feedback profilers should avoid the "observer effect" program that may affect the program's behavior (Snyder et al., 2011), while this problem not critical if profile will just present the result for developers.

From Another Perspective, Process Mining techniques aim to extract non-trivial information from event logs recorded by information systems. According to their abilities to assist in understanding and (re)design the complex process by extracting the workflow model that represent the information system behavior, the process mining techniques have received notable attention and promising vision (Van Der Aalst & Weijters, 2004). ProM framework is a pluggable environment for process mining. This framework is flexible with respect to the input and output format, and is also open enough to allow for the easy reuse of code during the implementation of new process mining ideas (De Medeiros et al., 2005).

This paper is mainly concerned with profilers that provide information about java program or JVM (HotSpotTM); these profilers mainly have three phases: collecting data, analyzing data and visualizing results. In this paper, the process mining techniques and ProM tool implemented to represent the analysis and visualize phases. They are flexible enough to cover so many perspectives in several ways. That can form a unified layer for analysis and visualize across profiling perspectives. Process mining applied on two different profiling data for java

programs/VM, and the result compared with the original profiling tools. The profiling data mapped to process-based, and with each different mapping new analysis perspective obtained. The DaCapo (Garner et al., 2006) benchmark suite has been used to apply profiling perspectives on some of its component. In this context, the java-event-logs tool has been provided to mapping Java profiling data to process mining event logs.

This paper organized as follows. Section 2 provides background information about the profiling data and process mining. Section 3 describes related work. Section 4 describes the architecture of java-event-logs tool. Section 5 provides a detailed description on how to use process mining during profiling and presenting the experimental results. Finally, Section 6 concludes and suggests directions for future work.

2. Literature Review

In this section, the profiling data will be dissected and study the existence tool for analysis and visualizing, then an overview about process mining perspectives and event log format will be discussed.

2.1 JVM Profilers

There are two different profiling data with different perspectives selected to study. The following is a breakdown of them:

2.1.1 Dependence Graph

The Java HotSpotTM server compiler uses a program dependence graph as the intermediate data structure when compiling Java bytecodes to machine code. When using the compiler in debug mode, it is providing a textual output of the graph (Ottenstein et al., 1987; Vick et al., 2001; Wimmer et al., 2008). The Ideal Graph Visualizer (IGV) tool used to analyze the compiler by providing a graphical representation of the program dependence graph. During the compilation process, the IGV tool captures snapshots of the graph then use it to create visual presentation to reconstruct the transformations applied to the graph by compiler optimizations. Figure 1 shows the interaction between the visualization tool and the server compiler (Würthinger, 2007; Wimmer et al., 2008). In IGV the data transferred from the server compiler to the visualization tool is represented in XML. Figure 2 shows the XML elements and their relations (Würthinger, 2007).

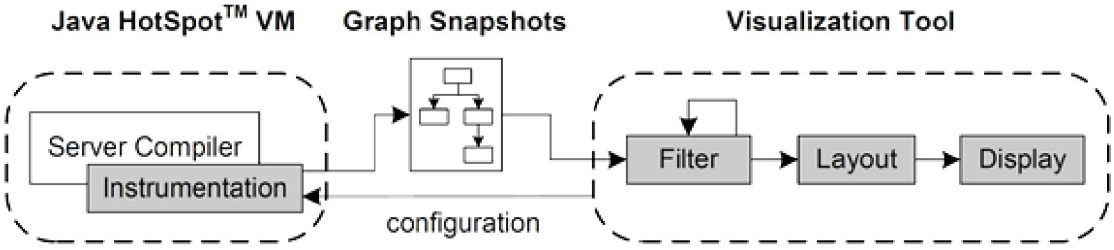

Figure 1. Interaction between the compiler and the visualization tool

Table 1. Description of the main elements in IGV XML

Element	Level	Description
graphDocument	1st	top-level element and can contain group child elements
group	2nd	server compiler creates a group element for every traced method
method	3rd	describes the bytecodes and inlining of the method
graph	3rd	describe the traced states of the graph during compilation of the method, it include the state title and starting time.
nodes	4th	contain definitions of nodes as node elements or removeNode elements, which state that a certain node of the previous graph is no longer present
edges	4th	contain definitions of edges as edge elements or removeEdge elements, which state that a certain edge of the previous graph is no longer present
controlFlow	4th	contains the information necessary to cluster the nodes into blocks

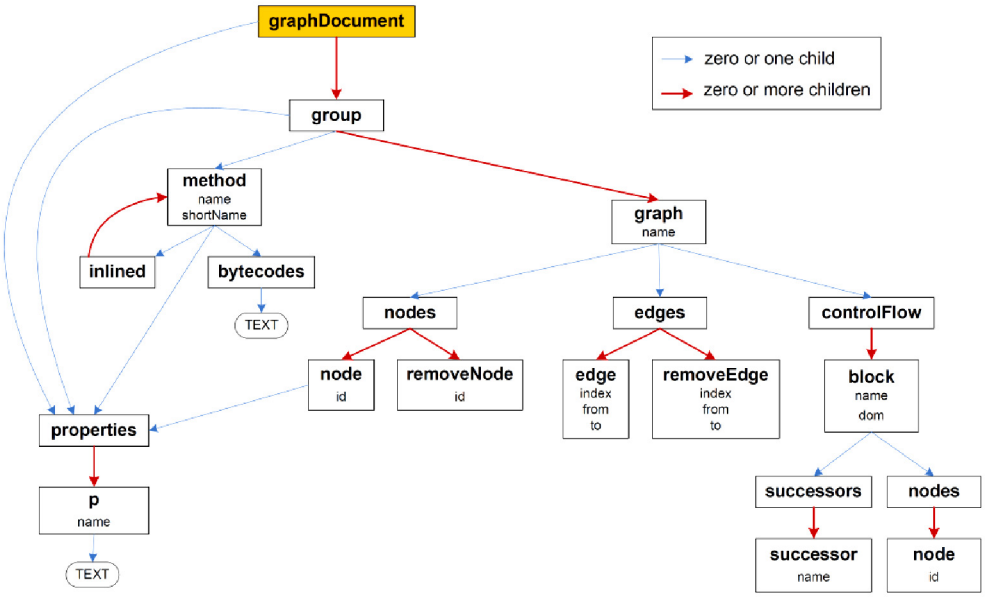

Figure 2. IGV XML elements and their relations

Figure 3 shows the output of the IGV tool, which represents the dependency graph; also, IGV tool supports some options like filtering the graph component manually or using JavaScript function and display the difference between snapshots graphically.

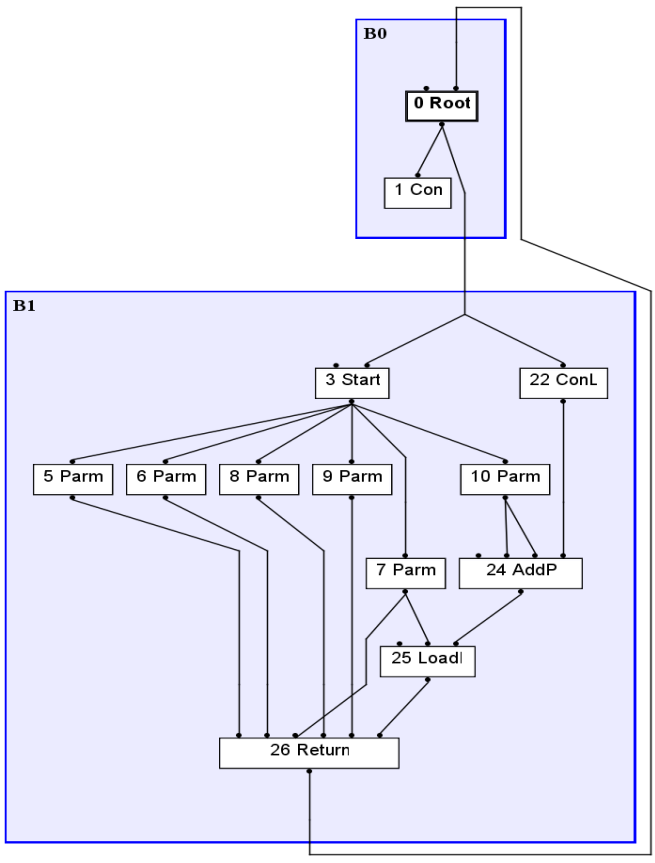

Figure 3. Dependence Graph as represented in IGV tool

2.1.2 Compilation Logs

The JVM HotSpotTM developers create internal diagnostic option in the JVM itself. The diagnostic options "-XX: +LogCompilation" emits a structured XML log of compilation related activity during a run of the virtual machine. By default it ends up in the standard "hotspot.log" file, though this can be changed using the -XX: LogFile= option. Note that both of these are considered diagnostic options and have to be enabled using -XX: +UnlockDiagnosticVMOptions (Snyder et al., 2011).

Figure 4 shows very rough overview of the LogCompilation output XML, and Table 2 describes the main elements in this XML.

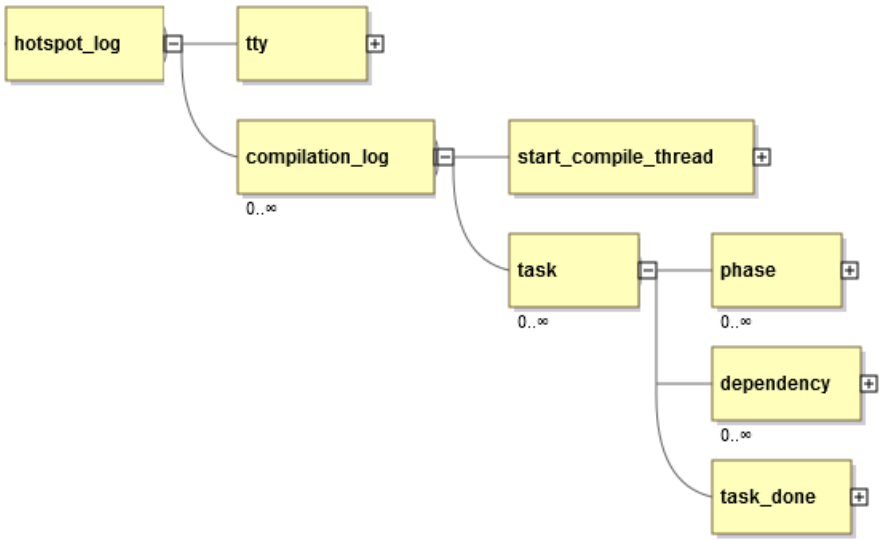

Figure 4. Very rough overview of the LogCompilation XML

Table 2. Description of the main elements in LogCompilation XML

Element	Level	Description
hotspot_log	1st	The main element in the XML.
tty	2nd	Output from normal Java threads and contains events for methods being enqueued for compilation, uncommon traps that invalidate a method and other events.
compilation_log	2nd	Comes from the compiler threads themselves. The output from the compiler is basically a log of the stages of the compile along with the high level decisions made during the compile such as inlining.
start_compile_thread	3rd	Mainly gives a timestamp for the start time of the compiler thread.
task	3rd	Methods to be compiled are placed into the compile queue and the compiler threads dequeue them and compile them as long as there are elements in the queue. Each individual compile is shown as a 'task' element.
phase	4th	Each phase of the compilation is wrapped by a 'phase' element which records the name of the phase, the maximum number of nodes in the IR at that point and the timestamp. Phases may nest and may be repeated.
dependency	4th	A "dependency" element indicates that class hierarchy analysis has indicated some interesting property of the classes that allows the compiler to optimistically assume things like there are no subclasses of a particular class or there is only one implementor of a particular method.
task_done	4th	indicates completion of the compile and includes a 'success' element to indicate whether the compile succeeded.

The LogCompilation tool created to parse the related XML file. This tool is part of the JVM HotSpotTM and provides textual output. This output provides information about compilation statistics (Number of compilation tasks, Number of bytes of the compiled method, Compilation type, …) and list the compilation events ordered by start time or elapsed time. Figure 5 shows sample for the output of this tool.

```
@ 12 java.util.HashMap::get (29 bytes)
  @ 11 java.util.HashMap::getEntry (77 bytes)
    @ 10 java.util.HashMap::hash (59 bytes)
      @ 31 java.lang.Integer::hashCode (5 bytes)
    @ 24 java.util.HashMap::indexFor (6 bytes)
    @ 59 java.lang.Integer::equals (29 bytes)
      @ 15 java.lang.Integer::intValue (5 bytes)
  @ 25 java.util.HashMap$Entry::getValue (5 bytes)
```

Figure 5. Sample for the output of the LogCompilation tool

2.1.3 Other Profilers

In (Sewe et al., 2012) the JP2 tool designed to extract the valuable calling context tree without exposure to analysis or visualize. In each of (Krinke, 2004; Balmas, 2001; Lee & Sim, 2015) specially programmed tools have been provided to display the program dependence graph. In (Driesen et al., 2003; Hendren et al., 2003) a new tool has been developed to use the internal JVM profiling APIs for gathering the information about the program then computing and presenting the results from the standpoint of the dynamic metrics. The NetBeans/JFluid Profiler (Dmitriev, 2004; Schulz et al., 2015) depends on dynamic bytecode instrumentation and code hotswapping to turn profiling on and off dynamically. However, this tool needs a customized JVM and is therefore only available for a limited set of environments. The Spy framework (Banados et al., 2012) builds profilers and visualizes profiling information for the Pharo-Smalltalk programming language. However, the limitations of the language reflect on the profiler. There is a wide range of related work in the area of profiling perspectives and tools; but the common thing across all that there is no unified data model and each tool designs its analysis and visualize technique which make it hard to integrate.

2.2 Process Mining

The main goal of process mining is to extract the information from the logs of the systems and representing it in workflow model to reconstruct the order of activities in the form of a graphical model. The basic idea of process mining is to learn from observed executions of a process (Van der Aalst & Weijters, 2004; Van Dongen et al., 2007); this used to: Discover new models (e.g., constructing a Petri Net that is able to reproduce the observed behavior), Check the conformance of a model by checking whether the modeled behavior matches the observed behavior and Extend an existing model by projecting.

The basic perspective of process mining is the so-called control-flow (process) perspective, which focuses on the control-flow, i.e., the ordering of activities. However, in addition to that could also consider: the organization perspective which focuses on which performers are involved and how they are related, and the case perspective that focuses on properties of cases (Van der Aalst & Weijters, 2004).

Event logs can be very different in nature, i.e. an event log could show the events that occur in a specific machine that produces computer chips, or it could show the different departments visited by a patient in a hospital. However, all event logs have one thing in common: they show occurrences of events at specific moments in time, where each event refers to a specific process and an instance thereof, i.e. a case (Van Der Aalst & De Medeiros, 2005).

ProM framework (De Medeiros et al., 2005) is a pluggable environment for process mining. Since each system has its own format for output log files, ProM framework works with a generic XML formats like MXML and XES (Van Der Aalst & Van Der Aalst, 2011). Regardless the elements name in file formats; there are main elements for each process that should be represented in any format, these elements listed in Table 3.

Plug-ins in ProM framework can be divided to mining plug-in which implements algorithms that mine models from event logs, analysis plug-in which typically implement some property analysis on some mining result and others plug-ins related to file formats input/output. Moreover, ProM has enormous potential in filtration and

general statistics about the input event logs.

Table 3. The main elements in the event log

Element	Required	Description
Case	Mandatory	Each case has unique ID and includes related actions.
Activity	Mandatory	The name of the action.
Timestamp	Optional	The time of the action.
Originator	Optional	The name of the action performer.

3. Proposed Tool Architecture

In this section, the architecture of java-event-logs tool and the usage of it described in details. XML format is the common thing between the types of input files and the output files too. So, the XMLBeans library for accessing XML by binding it to Java types, XMLBeans provides a way to get at the XML through XML schema that has been compiled to generate Java types that represent schema types, the XML schemas that descript the three types of input data has been have been included in the tool.

Figure 6 shows the class diagram for the java-event-logs tool. The "MainMiner" is the main class which receives the user options and delegates it to the right miner. The "LogMiner" is the abstract parent class for the three miners which applies the factory method pattern, the "IGVLogMiner","LogCompilationLogMiner" are the miners that responsible for extract the event logs patterns from dependence graph and compilation logs and finally the "MXMLLogBuilder" class which is responsible about the event logs output format.

The java-event-logs tool has two execution options "-igv" and "-logc" for dependence graph and compilation logs respectively. As shown in Figure 7 and Figure 8, the tool apply certain algorithm based on each log input.

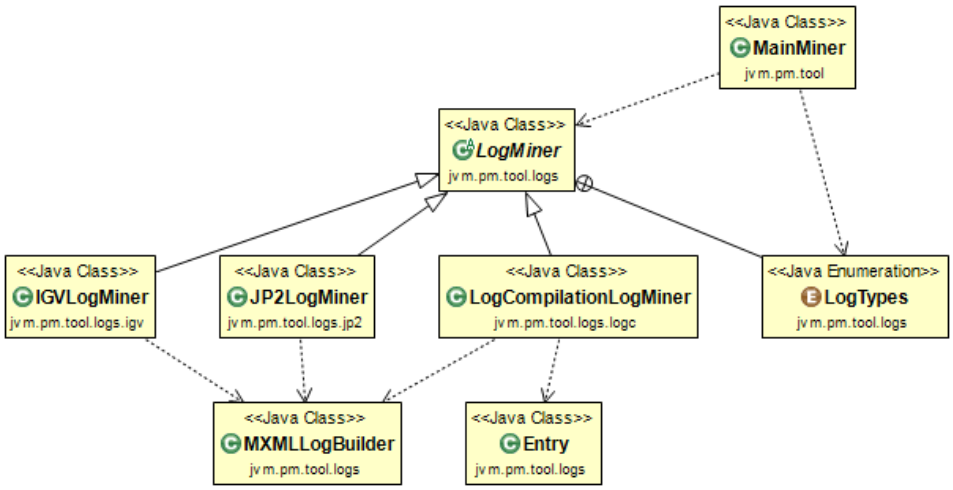

Figure 6. Class diagram for java-event-logs tool

4. Process-Based Profiling in Action

In this section, the process mining techniques applied on the profiling data that previously mentioned by mapping the data using java-event-logs tool and present the faces of various analysis perspectives supported by ProM. Data extracted by profiling the fop application in the DaCapo benchmark suite. As in Table 3, time of the action is one of elements that process mining uses during analysis, although it is optional, some important analysis techniques rely on it. So, the time attribute added to profiling data by modifying the profiler agent.

The Heuristics Miner (HM) algorithm (Weijters & Van Der Aalst, 2003; Van Der Aalst et al., 2006; Burattin, 2015) focuses on the control flow perspective and generates a process model in form of a Heuristics Net for the underlying event log. Also HM is a practical applicable mining algorithm that can deal with noise, and can be used to express the main behavior of the event log. So, for the control flow perspective of the next cases, the HM algorithm selected. The HM provides list of parameters to control the level of details in the extracted model by it.

The following parameter values have been used: (1, 1, 0.01, 0.01, 0.01, 0.01, 1, 0.1, false, true, false) for ("Relative-to-best threshold", "Positive observations", "Dependency threshold", "Length-one-loops threshold", "Length-two-loops threshold", "Long distance threshold", "Dependency divisor", "AND threshold", "Extra information", "Use-all-events-connected heuristic", "Long distance threshold dependency heuristics") respectively.

```
Input: graph Document GD.
Output: even log L
Method: Perform the following steps:

1.      Initialize L as empty log // this log will contain all processes extracted from graph document
2.      Initialize G as list of all graphs in GD // each graph g in G is representing code execution state
3.      For each graph g in G do
3.1.            Let N := list of nodes in g;
3.2.            Let E := list of edges in g;
3.3.            Filter list of nodes N to discard nodes marked as removed in graph g
3.4.            Filter list of edges E to discard edges marked as removed in graph g
3.5.            For each edge e in E do
3.5.1.                  Add new Process Instant P to event log L
3.5.2.                  Let ns := search list of nodes N and get the one is linked to start of edge e
3.5.3.                  Let ne := search list of nodes N and get the one is linked to end of edge e
3.5.4.                  Add new Entity ws to process instance P where Activity and Time is node ns title and time.
3.5.5.                  Add new Entity we to process instance P where Activity and Time is node ne title and time.
3.6.            End for
4.      End for
```

Figure 7. The algorithm steps to convert dependency graph to event log

```
Input: Compilation logs CL.
Output: even log L
Method: Perform the following steps:

1.      Initialize L as empty log // this log will contain all processes extracted from graph document
2.      Initialize E as empty list of task entities
3.      For each log cl in CL do // flat line all entities in all tasks in compilation logs to be single list
3.1.            Let T := list of tasks in cl;
3.2.            For each task t in T do
3.2.1.                  Let e := new task entity
3.2.2.                  Set title of e := method name in task t
3.2.3.                  Set time of e := time of task t
3.2.4.                  Set origin of e := class name in task t
3.2.5.                  Add e to E
3.3.            End for
4.      End for
5.      Sort the list of entities E // sort the list after adding all entities from all tasks in compilation log
6.      For each entity e in E do
6.1.            Let P := empty process instance of event log L
6.2.            Let fe := new process entry
6.3.            Let te := new process entry
6.4.            Add fe to process instance P
6.5.            Add te to process instance P
7.      End for
```

Figure 8. The algorithm steps to convert Compilation logs to event log

The timestamp of an activity used to calculate these ordering. Therefore, HM introduce the following notations and defines an event log as follows:

Let T be a set of activities, $\sigma \in T^*$ is an event trace and $W \subseteq T^*$ is an event log. And let a, $b \in T$, $a >_W b$ iff there is a trace $\sigma = t1,t2,t3 \ldots tn$ and $i \in \{1, \ldots, n-1\}$ such that $\sigma \in W$ and $ti = a$ and $ti+1=b$, $a \rightarrow_W b$ iff $a >_W b$ and

b ≯W a, a#W b iff a ≯W b and b ≯W a, a‖W b iff a >W b and b >W a, a >>W b iff there is a trace σ =t1t2t3 …tn and i∈{1, …, n-2} such that σ∈ W and ti = a and ti+1=b and ti+2 = a, a >>>W b iff there is a trace σ =t1t2t3 …tn and i<j and i,j ∈{1, …, n} such that σ∈ W and ti = a and tj=b.

HM Algorithm Steps: The HM algorithm is a three-step algorithm: Construct a dependency graph on the basis of the event log. For each task in the event log establish the input-output expressions in form of type of dependencies between activities. Discover the long distance dependency relations.

Mining of the dependency graph: The starting point of the Heuristics Miner is the construction of a so-called dependency graph. A frequency based metric is used to indicate how certain that there is truly a dependency relation between two events a and b (notation a ⇒W b).

Let W be an event log over T, and a, b ∈ T. Then |a >W b| is the number of times a >W b occurs in W, and

$$\mathbf{a} \Rightarrow_{\mathbf{W}} \mathbf{b} = \left(\frac{|\mathbf{a} >_{\mathbf{W}} \mathbf{b}| - |\mathbf{b} >_{\mathbf{W}} \mathbf{a}|}{|\mathbf{a} >_{\mathbf{W}} \mathbf{b}| + |\mathbf{b} >_{\mathbf{W}} \mathbf{a}| + \mathbf{1}} \right)$$

Equation 1. Dependency measure between a and b

4.1 JVM Dependence Graph Implementation

For each different data mapping from dependence graph to event log, different analysis perspective obtained. The timestamp attribute added for each graph element. Two different mapping listed below:

4.1.1 Method Snapshots

This pattern provides a graphical representation of each method snapshot of the program dependence graph. This represent an equivalent for what provided by the IGV tool. Each process will represent states of single method; each case will represent two nodes attached with one edge, any case constructed in two actions, first one is the source node and second action is the destination node. The filtration functionality used to select the specific state to work on. The profiling data mapped to event log as in Table 4.

Table 4. Data mapping to extract the method snapshot

Event Log	Profiling Data	Description
Case	Graph: state title	Each case represents graph stat of the method
Activity	node & edge	Each activity represents connection between nodes

Figure 9 shows the control flow graph represent method state extracted using HM algorithm after filtering it using instance name filter with regular expression value "^(?!After|Parsing).*$" to model "After Parsing" state only. Also there are different analysis techniques are available for direct applying like LTL and SCIFF checkers (Lamma et al., 2009) which uses a logic-based approach to mining declarative models and DWS clustering algorithm (Guzzo et al., 2008) which provides solution for over-fitting problem that appear with complex methods and which is not handled in IGV tool.

4.1.2 Compilation Workflow

This pattern provides very detailed information about compiler behavior during the process of compilation of the monitored code, the compilation process changes based on method structure and complexity. The event log will have only one process; ach case will represent one method compilation steps and each action will include the step title, event time is the time of starting this step and the originator will be the full name of the method itself. The profiling data mapped to event log as in Table 5.

Table 5. Data mapping to extract compilation workflow from dependency graph

Event Log	Profiling Data	Description
Case	method	Each case represents single method compilation states.
Activity	Graph: state title	Each activity represents one state.
Timestamp	Graph: state time	The starting time of this state.
Originator	method	Putting originator as method full name to use in analysis.

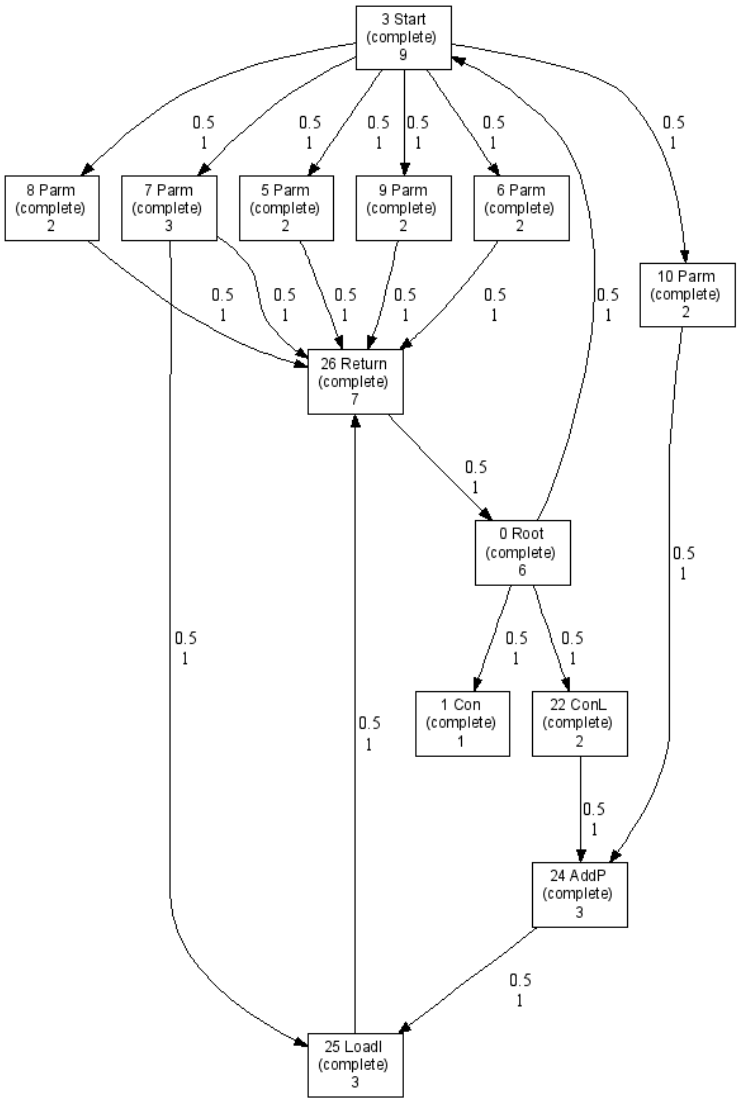

Figure 9. Dependence Graph state as represented in ProM tool

Starting with the control flow model for compilation process, Figure 10 shows part of the HM model that explains which compiler state has triggered and in which sequence and frequency, that allows understanding the code complexity and the corresponding compiler behavior. For example, how many "Phase Ideal Loop" states compiler has to call, did the compiler call the "eliminating allocations and locks" state or not and for how many times and so on. The applying of LTL and SCIFF checkers allows defining which compilation pattern to check, also clustering over-fit patterns and simplify then using DWS.

Some different analysis perspectives can be extracted directly; like basic statistics about the occurrences of the compilation states as in Figure 11, using the basic performance analysis we can easily identify the time that each state takes in average as in Figure 12 to identify the costly states or the time that each method takes in general while compilation, by using the "Originator by Task Matrix" we can identify which method trigger specific state in high frequency as in Figure 13.

To study the compilation patterns we can use "Sequence Diagram Analysis" to list the paths that the control flow constructed from them with identification for the most frequent path that was happened during compilation as in Figure 14, in this case the total unique compilation paths is 9 paths represent 38 cases and the most frequent path happened 21 times. For studying the changes in compiler behavior from method to another, we can use the "Trace Diff Analysis" to compare compilation steps for two methods, Figure 15 shows the common steps in order between two method and when changes start and end.

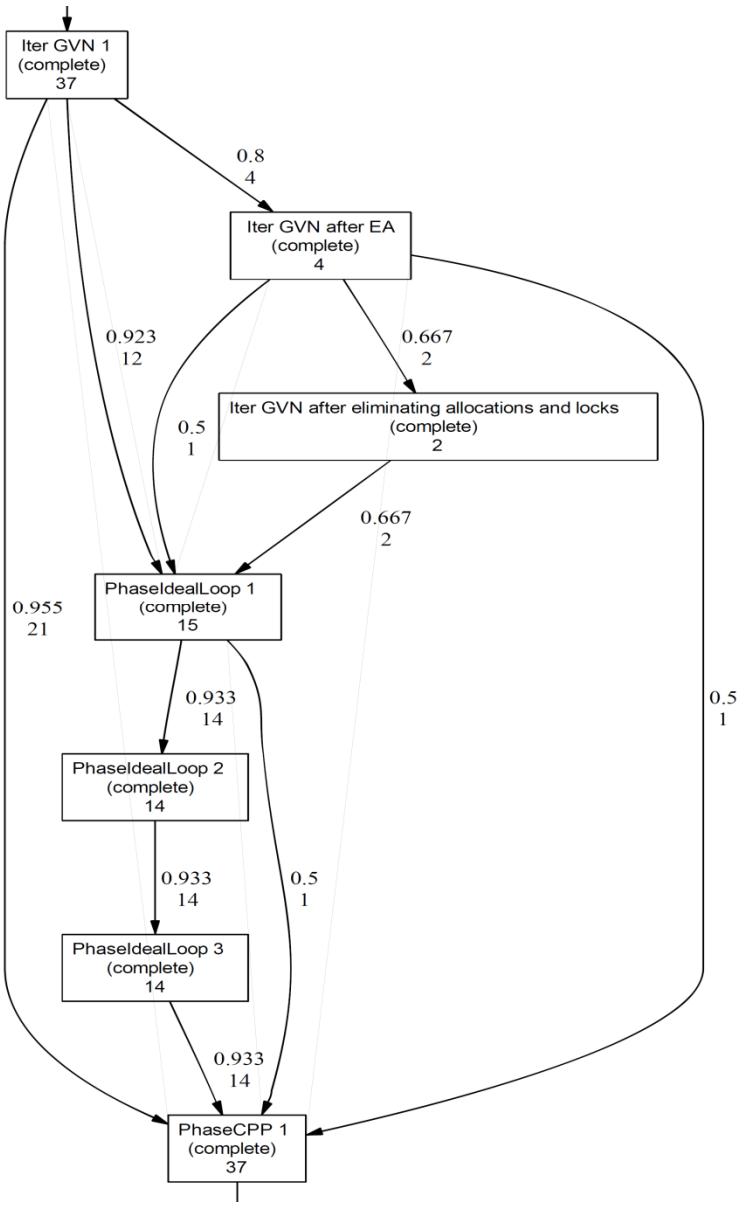

Figure 10. Workflow diagram that represent some methods compilation processes in JVM according to dependency graph

Model element	Occurrences (absolute)	Occurrences (relative)
After Parsing	38	10.243%
Iter GVN 1	37	9.973%
PhaseCPP 1	37	9.973%
Iter GVN 2	37	9.973%
Optimize finished	37	9.973%
Before Matching	37	9.973%

Figure 11. Compilation states basic statistics

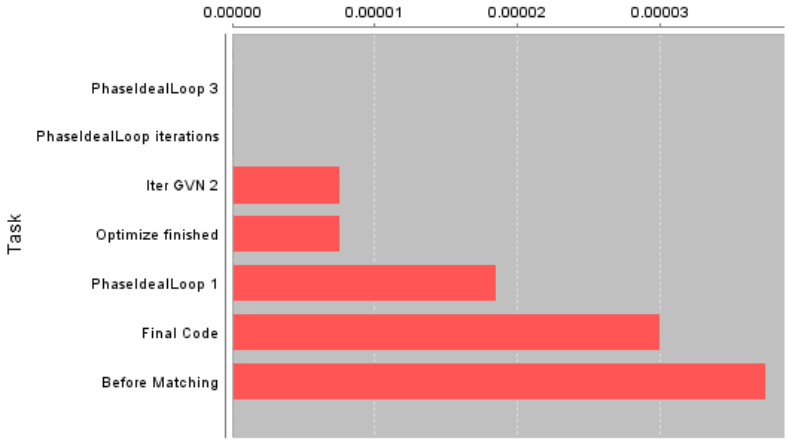

Figure 12. Bar chart for compilation states time

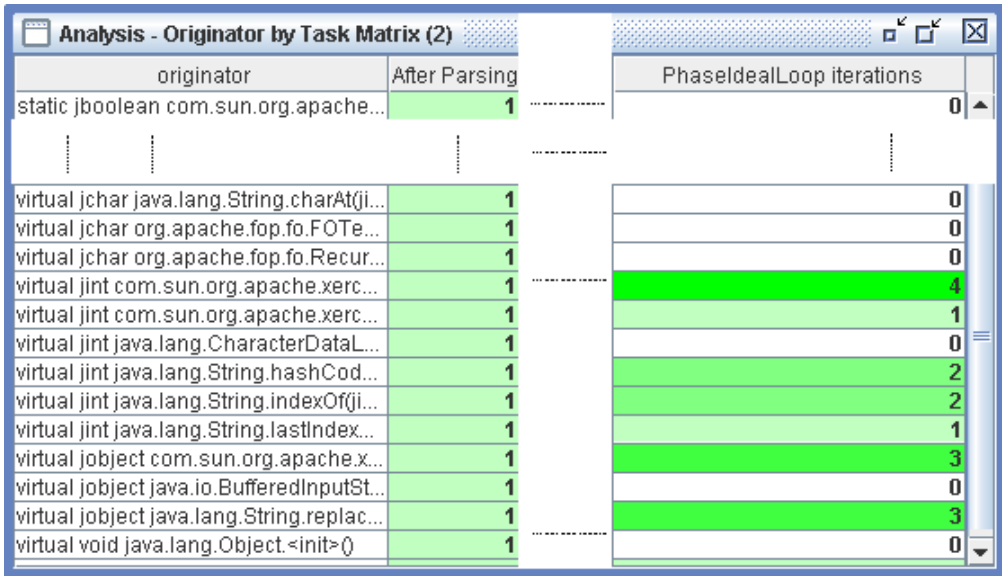

Figure 13. States' frequency for each individual method

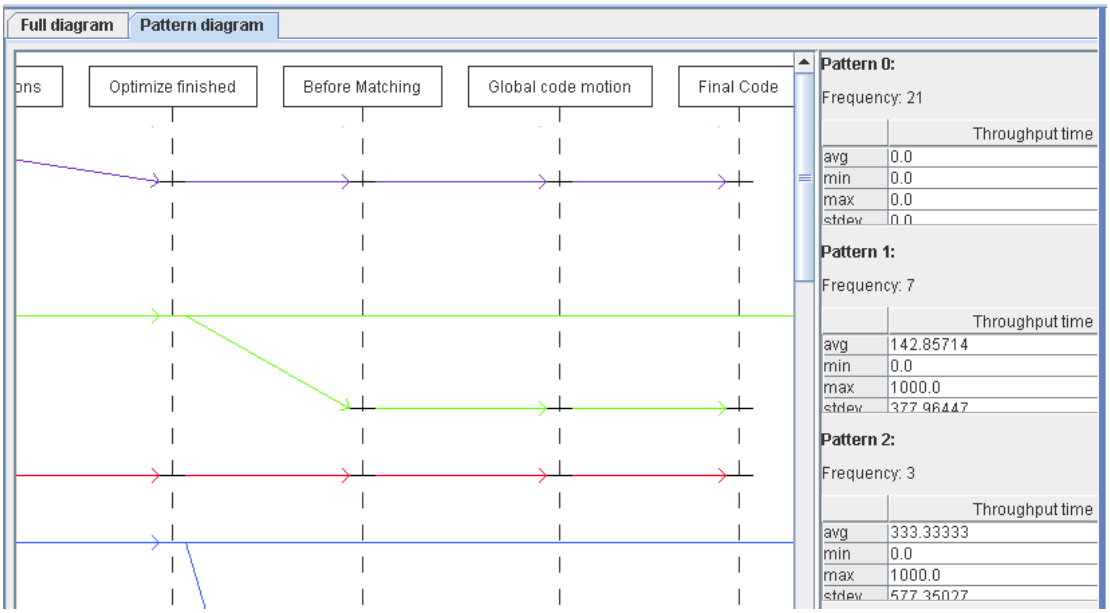

Figure 14. Sequence Diagram for the compilation paths

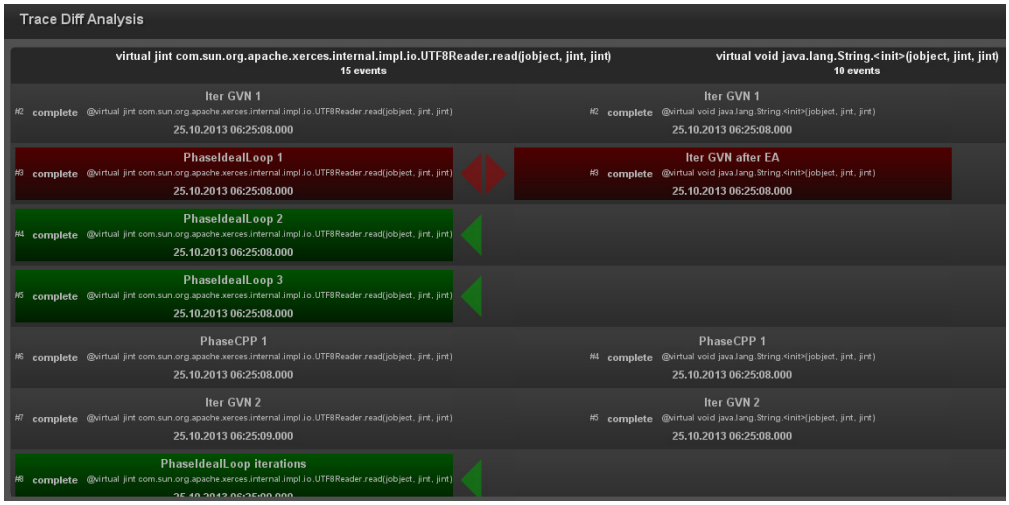

Figure 15. Compare two compilation paths by comparing the process activities

4.2 Compilation Logs Implementation

Compilation logs provided by JVM HotSpot developers inspects the compilation process. Compilation logs focuses on the compilation of method without describing the method architecture. Two patterns listed below, first is about classes relationship according to compilation, and the other pattern is compilation workflow to describe compiler behavior during the process of compilation.

4.2.1 Classes Relationship Based on Compilation

In JVM HotSpot method compilation happens under some optimization conditions, for this pattern, and to extract a valid classes relationship based on compilation, the compilation should happens for all method. So, the "-Xcomp" option has been used in this pattern to force compilation for all methods. Each process will represent single classes sequence; each case will represent two classes relationship according to the order of compilation methods in both of them as in Table 6.

Figure 16 shows part of the HM workflow model that describes the classes' relationship based on compilation process. The dependency between classes is so clear in such a model, by filtering this model to cover specific classes with predefine relation, we can make sure that what we designed actually applied. Obviously there are

some basic statistic can be extracted from this pattern directly. Like the time has been consumed with each method, frequencies of method compilations in each class and which class has more compilation frequency. Listing the methods based on their compilation order as in the LogCompilation tool extracted directly from the log inspector as in Figure 17.

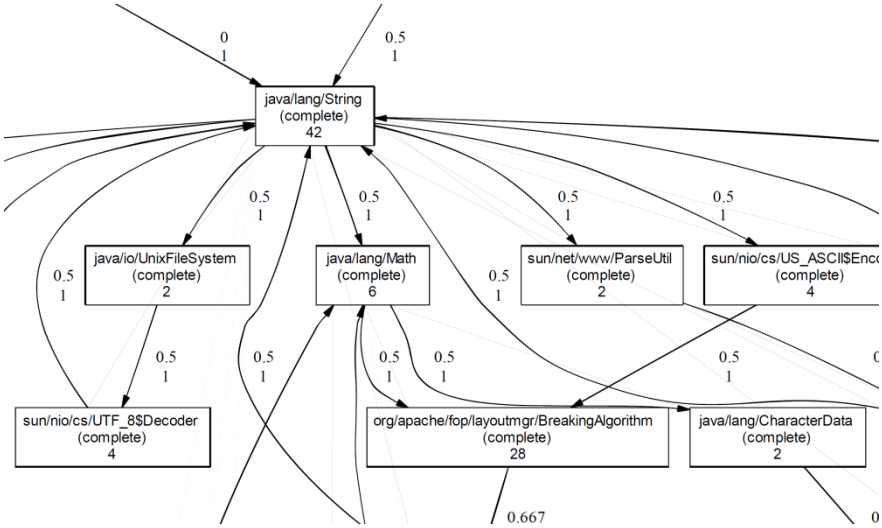

Figure 16. HM workflow model that represent classes relationship based on compilation

Table 6. Data mapping to extract classes' relationship based on compilation

Event Log	Profiling Data	Description
Case	Two Tasks	Each case represents one sequence of compilation
Activity	Task: Class Name	Each activity represents connection between two classes

Figure 17. Methods compilation order as shows in log inspector

4.2.2 Compilation Workflow

This pattern describes compiler behavior during the process of compilation as organized in JVM compilation logs. The event log will have only one process; ach case will represent one method compilation process and each action will include the compilation phase, event time is the time of starting this phase and the originator will be the full name of the class that contains this method. The profiling data mapped to event log as in Table 7. Figure 18 shows the HM model, and all the analysis patterns that mentioned with dependency graph can be extract as well from JVM compilation logs.

Table 7. Data mapping to extract compilation workflow from JVM compilation logs

Event Log	Profiling Data	Description
Case	Task	Each case represents one method compilation process.
Activity	Phase	Each activity represents one phase of compilation process.
Timestamp	Phase time	The starting time of this phase.
Originator	Task: Class Name	Putting originator as method full name to analysis.

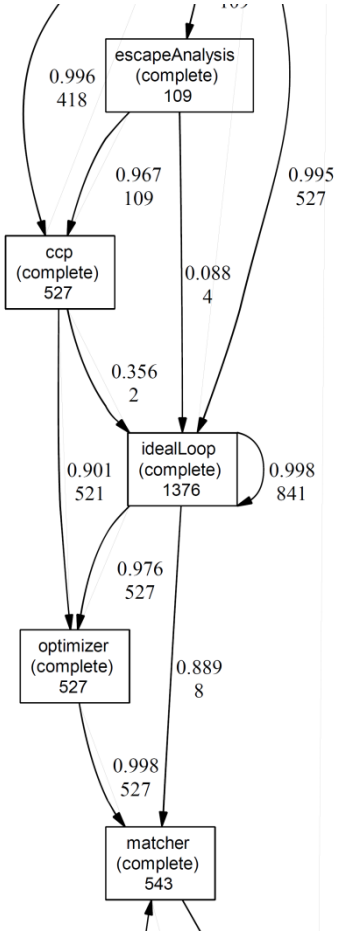

Figure 18. Workflow diagram that represent some methods compilation processes in JVM according to compilation logs

5. Conclusions

In this paper, new approach has introduced to use process-mining techniques to represent the analysis and visualize phases of JVM profilers. They are flexible enough to cover so many perspectives in several ways. That can form a unified layer for analysis and visualize across profiling perspectives.

To do so, new tool java-event-logs has introduced to implement this approach. The java-event-logs tool has two

execution options "-igv" and "-logc" for dependence graph and compilation logs respectively, and the outputs provide new perspectives for profiling data analysis.

Applying this new approach on the JVM profilers provides information about java program or JVM (HotSpotTM) and helps JVM developers with new aspect of analysis with each process mining perspective. On the other hand, we will work on how to use process mining to provide interactive approached that can help to analysis the Java Byte code program and provide feedback to JVM to enhance execution time and memory management.

References

Ahmad, S. et al. (2014). Dependence flow graph for analysis of aspect-oriented programs. *International Journal of Software Engineering & Applications, 5*(6), 125.

Balmas, F. (2001). *Displaying dependence graphs: A hierarchical approach*. Reverse Engineering, 2001. Proceedings. Eighth Working Conference on, IEEE.

Bergel, A. et al. (2012). Spy: A flexible code profiling framework. *Computer Languages, Systems & Structures 38*(1), 16-28.

Blackburn, S. M. et al. (2006). *The DaCapo benchmarks: Java benchmarking development and analysis*. ACM SIGPLAN Notices, ACM.

Bowers, K. R., & Kaeli, D. (1998). *Characterizing the SPEC JVM98 benchmarks on the Java virtual machine*. Te hnical Report ECE-CEG-98-026, Northeastern University, Department of Electrical and Computer Engineering.

Burattin, A. (2015). *Heuristics Miner for Time Interval*. Process Mining Techniques in Business Environments, Springer: 85-95.

Chesani, F. et al. (2009). *Exploiting inductive logic programming techniques for declarative process mining*. Transactions on Petri Nets and Other Models of Concurrency II, Springer: 278-295.

De Medeiros, A. K. A. et al. (2008). *Process mining based on clustering: A quest for precision*. Business Process Management Workshops, Springer.

Dmitriev, M. (2004). *Profiling Java applications using code hotswapping and dynamic call graph revelation*. ACM SIGSOFT Software Engineering Notes, ACM.

Dufour, B. et al. (2003). *Dynamic metrics for Java*. ACM SIGPLAN Notices, ACM.

Dufour, B. et al. (2003). *J: a tool for dynamic analysis of Java programs*. Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM.

Ferrante, J. et al. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS), 9*(3), 319-349.

Krinke, J. (2004). *Visualization of program dependence and slices*. Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, IEEE.

Lee, S. H., & Sim, S. (2015). Aspect Refactoring Techniques for System Optimization. *Indian Journal of Science and Technology, 8*, 412.

Liang, S., & Viswanathan, D. (1999). *Comprehensive Profiling Support in the Java Virtual Machine*. COOTS.

Paleczny, M. et al. (2001). *The java hotspot TM server compiler*. Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1, USENIX Association.

Sarimbekov, A. et al. (2012). *JP2: Call-site aware calling context profiling for the Java Virtual Machine*. Science of Computer Programming.

Sjoblom, I. et al. (2011). *Can You Trust Your JVM Diagnostic Tools?* MICS.

Van Der Aalst, W. M. et al. (2007). *ProM 4.0: comprehensive support for real process analysis*. Petri Nets and Other Models of Concurrency–ICATPN 2007, Springer, 484-494.

Van Der Aalst, W. M., & De Medeiros, A. K. A. (2005). Process mining and security: Detecting anomalous process executions and checking process conformance. *Electronic Notes in Theoretical Computer Science 121*, 3-21.

Van Der Aalst, W. M., & Van Der Aalst, W. (2011). *Process mining: discovery, conformance and enhancement of business processes*. Springer.

Van Der Aalst, W. M., & Weijters, A. (2004). Process mining: a research agenda. *Computers in Industry, 53*(3), 231-244.

Van Dongen, B. F. et al. (2005). *The ProM framework: A new era in process mining tool support*. Applications and Theory of Petri Nets 2005, Springer, 444-454.

Weijters, A. et al. (2006). *Process mining with the heuristics miner-algorithm*. Technische Universiteit Eindhoven, Tech. Rep. WP 166.

Weijters, A. J., & Van Der Aalst, W. M. (2003). Rediscovering workflow models from event-based data using little thumb. *Integrated Computer-Aided Engineering, 10*(2), 151-162.

Wert, A. et al. (2015). *AIM: Adaptable Instrumentation and Monitoring for automated software performance analysis*. Automation of Software Test (AST), 2015 IEEE/ACM 10th International Workshop on, IEEE.

Würthinger, T. (2007). *Visualization of program dependence graphs*. Johannes Kepler University Linz. Master.

Würthinger, T. et al. (2008). *Visualization of program dependence graphs*. Compiler Construction, Springer.