ABSTRACT

Virtual execution environments and middleware are required to be extremely reliable because applications running on top of them are developed assuming their correctness, and platform-level failures can result in serious and unexpected application-level problems. Since software platforms and middleware are often executed for long time without any interruption, large part of the testing process is devoted to investigate their behavior when long and stressful executions occur (these test cases are called workloads). When a problem is identified, software engineers examine log files to find its root cause. Unfortunately, since of the workloads length, log files can contain a huge amount of information and manual analysis is often prohibitive. Thus, de-facto, the identification of the root cause is mostly left to the intuition of the software engineer.

In this paper, we propose a technique to automatically analyze logs obtained from workloads to retrieve important information that can relate the failure to its cause. The technique works in three steps: (1) during workload executions, the system under test is monitored; (2) logs extracted from workloads that have been successfully completed are used to derive compact and general models of the expected behavior of the target system; (3) logs corresponding to workloads terminated unsuccessfully are compared with the inferred models to identify anomalous event sequences. Anomalies help software engineers to identify failure causes. The technique can also be used during operational phase, to discover possible causes of unexpected failures by comparing logs corresponding to failing executions with models derived at testing time. Preliminary experimental results conducted on the Java Virtual Machine indicate that several bugs can be rapidly identified thanks to the feedbacks provided by our technique.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—Monitors, Tracing

General Terms
Reliability

Keywords
log file analysis, automated analysis, model inference, workloads execution, JVM monitoring

1. INTRODUCTION

Nowadays we are witnessing to an increasing use of virtual execution environments and middleware platforms for the development of large and complex applications. Examples are Java Virtual Machines (JVMs), enterprise systems and Corba-based platforms. During application testing, large part of the testing process is devoted to investigate the behavior of underlying platforms when long and stressful executions occur (these test cases are called workloads). When a problem is detected, software engineers examine log files to gain insights about failure manifestations and to discover potential root causes. Although several research efforts have been performed in the design and the implementation of (semi)automatic tools for log file analysis, such as [9] and [10], the problem of automatically analyzing huge log files has been only partially solved [1, 13, 4]. De-facto, it is extremely hard to understand failure manifestations, errors propagation and isolation, and to discover potential root causes.

In this paper, we propose a technique to collect and analyze log files to obtain important information about failure manifestations. The technique works in three steps:

1. during workload executions, the system under test is monitored;
2. logs corresponding to workloads terminated correctly are used to derive a model of the behavior expected from the target system;
3. logs corresponding to workloads terminated unsuccessfully are compared with the models derived at testing time to identify anomalous event sequences.
3. logs corresponding to either workloads terminated un-
   successfully or failures observed in the field are com-
   pared with the reference model to identify anomalous
   event sequences.

The empirical experience presented in this paper focuses
on the Sun Java Virtual Machine (JVM), which is widely
used in the development of large and complex applications.
In order to apply such technique to JVM, we used JVM-
Mon [14], which is a tool for on-line monitoring and analysis
of JVM, and kBehavior [12], which is an inference engine
that generates general and compact Finite State Automata
(FSA) from execution traces. Our driving idea is to trace
the events raised by internal components of the JVM and, in
turn, to provide the logged data to the kBehavior inference
engine. Exploiting the information about the events internal
to the JVM and deriving a general behavioral model,
we can point out anomalous interactions within the JVM
by comparing the reference models with the data logged in
faulty executions. Information about anomalous sequences
support testers in the identification of failure causes.
In this paper, we considered a 2 hours workload to derive the
reference models and we investigated causes of 8 publicly
documented bugs of the JVM.

The paper is organized as follows. Section 2 presents
the overall approach. Section 3 provides insights about the
JVMMon monitoring infrastructure that has been used in
our experiments. Sections 4 indicates how the collected data
is pre-processed before being provided to the inference en-
gine. Section 5 presents the kBehavior inference engine. Sec-

2. INTEGRATION OF WORKLOAD EXE-
   CUTION AND DYNAMIC ANALYSIS

The execution of ad-hoc workloads is often used to assess
reliability of execution environments and software platforms
in particular operative conditions. For instance, if testers
need to investigate the behavior of a JVM when running
applications that consume large amount of memory, a work-
load that continuously creates, and sometime destroys, ob-
jects can be executed. Monitoring the underlying platform
helps to reveal problems (if any) and their causes, but also
produces huge logs that are difficult to be manually ana-
yzed. The proposed technique automatically analyzes the
information extracted during workload execution by com-
bining pre-processing, to reduce noise and skim irrelevant
information, and model inference, to automatically gener-
ate general and compact models of the expected behavior.
When the system under test fails, the inferred models can
identify anomalous sequences that are inspected by testers
in order to identify the fault that has been responsible for the
experienced failure.

Figure 1 shows our approach. In the first step, we record
logs while the platform under test is executed with several
workloads. Workloads usually focus on specific issues, e.g.,
they stress garbage collection or concurrency. Logs obtained
in this step represent the behavior observed when the sys-
tem under test correctly manages the aspects stressed by
workloads.

In the second step, we produce models that summarize
and generalize the behavior traced into log files. To this end,
several intermediate phases must be completed. Firstly, we
produce multiple log files, smaller than the original one, that
include only events related to specific issues. For instance,
the initial log file can split into two smaller log files, one with
events related to memory management and one with events
related to concurrency. The generation of issue-specific log
files focuses model inference and analysis on specific prob-
lems. Typical monitoring frameworks annotate events with
additional information. For example, JVMMon annotates
the thread start event with the id of the thread that is
started. Including explicit values, such as numbers and
strings, in the model can over-restrict the inference, thus
resulting in several false positives at the analysis phase. For
instance, invoking a wait on a monitor with a hashcode that
has never been observed before can be erroneously recog-
nized as an anomalous event. To avoid these problems, but
still incorporating data-flow information in the model, we
refine the log files by replacing explicit values with symbolic
values that represent data-flow information. The obtained
log files are finally used to generate compact and general
models that represent the set of observed event sequences
with FSAs.

In the third step, if a target software platform fails, testers
can investigate failure causes supported by models generated
during testing. In particular, the events observed in the
failed execution are compared with inferred models. Anom-
alous sequences are reported to testers, who inspect both
the application and the traces to identify the exact cause of
the problem. The technique has the benefit to immediately
focus the attention of testers to few problematic sequences
of events. Early experiments show that false positives are
limited and anomalous sequences are extremely useful to
quickly identify causes of problems.

3. MONITORING

In this Section we present the architecture of the JVM,
which is the software platform that has been considered in
this paper, and the JVMMon tool, which is the monitoring
framework that has been used to extract events from the
JVM.
Java Virtual Machine

The JVM is a virtual machine belonging to the High Level Language VMs (HLL-VM) category [17]. An HLL-VM is a VM which i) adds support for cross-platform programming, ii) provides a virtual Instruction Set Architecture (ISA), and iii) abstracts the Application Binary Interface (ABI) and the ISA exposed by the underlying Operating System and Hardware, thus making applications written for the virtual machine platform-independent.

The virtual ISA of the JVM is a set of instructions called bytecode; programs written in Java are compiled into bytecode. The JVM is composed of four main components, depicted in Figure 2:

- **Execution Unit** - It dispatches and executes operations, emulating a CPU. An operation could be i) a translated bytecode instruction, ii) a compiled bytecode instruction, or iii) a native instruction. The Interpreter translates single bytecode instructions into native machine code whereas the Just-In-Time (JIT) compiler translates entire methods into native code doing some optimizations. Instead, native instructions need no translation since they are native machine instructions. They are dynamically loaded, linked and executed by the Java Native Interface (JNI). Moreover, the Exception Handler handles exceptions thrown by both Java Applications and the Virtual Machine. Exceptions thrown by applications are defined checked, while exceptions thrown by the VM are defined unchecked and are related to errors originated into the virtual machine.

- **OS Virtualization Layer Unit** - It provides a platform-independent abstraction of the host system’s ABI. This abstraction layer provides a common gateway for all JVM components to access host system resources.

- **Memory Management Unit** - It handles both the JVM heap area and the stack area, managing object allocation, reference handling, object finalization and garbage collection. Moreover, Fast Allocation Mechanism are provided to allocate temporary memory areas for internal VM operations.

- **System Services Unit** - Components included in this unit offer services to Java Applications. The Thread Management component handles thread creation and termination and it implements mechanisms for thread synchronization as specified by the Java Virtual Machine Specification [11] and the Java Language Specification [8]. The Class Loader is in charge of dynamically loading and verifying Java classfiles. Timers component provides functionalities to access system timers through the JVM. Finally, the Management and Debugging component includes functionalities for debugging Java applications and for the management of the JVM.

**JVMMon**

JVMMon is JVM monitoring infrastructures which has been presented in [14]. Unlike other systems conceived to collect failure data, JVMMon has been designed to intercept events related to state changes of the JVM, thus collecting the evolution of JVM state along with errors and failures. JVMMon allows on-line analysis of JVM state evolution through a three-step process: i) a monitoring agent, developed using the JVM Tool Interface (JVMTI) (which stems from the Java Platform Profiling Architecture [7]) and ByteCode Instrumentation (BCI), intercepts events generated inside the JVM and collects data about its state; ii) a monitoring daemon processes these information and updates the state of the virtual machine; iii) a data collector stores collected data in a database, allowing on-line and off-line analysis.

Since JVMMon is built upon JVMTI, it is applicable to all JSR-163 compliant Virtual Machines. Interested readers can find further details in [14].

4. PRE PROCESSING

In general, pre-processing is based on two phases: feature selection and data transformation [5]. The former removes noisy and irrelevant data from the trace files that are used to infer behavioral models. The latter suitably transforms the data values that are specific to a given execution and cannot be immediately compared with values traced in different executions, e.g., timestamps and hashcodes.

**Feature selection**

Feature selection filters events and attributes that are useful, and thus should be considered in model generation, from the ones that are of little use, and thus should not be considered in model generation. Since the complete set of events recorded by JVMMon and the type of investigated problems are known a-priori, we associate to each class of problems its sets of relevant events. When a model is inferred to identify a particular problem type, e.g., concurrency issues, all irrelevant events are removed from log files before starting the inference process, e.g., only information related to threads are preserved in the input file.

In this paper, we focus on two classes of problems: concurrency and memory handling problems. Events selected as relevant for concurrency problems are thread start and end, wait for a lock and lock acquisition, thread disabled and
enabled, and exceptions related to concurrency. Events selected as relevant for memory handling problems are garbage collector start and stop and object creation. More events can be considered for both classes of problems, however JVM-Mon does not yet support complete traceability.

**Data transformation**

Data transformation consists of filtering, normalizing and aggregating attribute values to increase quality of the outputs provided by inference algorithms. The choice of the transformation criterion depends from the nature of the attributes. Table 1 shows the JVM events and the parameters selected for our experiments. Most of parameter values are specific to the single executions where they are observed and are not suitable to be directly included into (general) behavioral models. For instance, the hashcode of a monitor is not usually preserved between different executions and it makes little sense to include it into an inferred model.

**Table 1: Events recorded by JVM-Mon.**

<table>
<thead>
<tr>
<th>Kind of issue</th>
<th>Event Name</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>Thread start</td>
<td>tid</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Thread stop</td>
<td>tid</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Thread waiting</td>
<td>tid,mhash</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Thread notified</td>
<td>tid,mhash</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Thread waiting for a lock</td>
<td>tid,mhash</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Thread acquiring a lock</td>
<td>tid,mhash</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Concurrency Exception</td>
<td>class name</td>
</tr>
<tr>
<td>Memory</td>
<td>Garbage collector start</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>Garbage collector stop</td>
<td></td>
</tr>
</tbody>
</table>

**Legend:** tid indicates the thread id, mhash indicates the hashcode of the monitor.

To cope with attribute values, we defined three feature extraction functions that abstract from concrete values and use symbols (in our case numbers) to specify data-flow information, i.e., information about the values that are generated and used in different events. In particular, the feature extraction functions rewrite each sequence of parameter values according to specific strategies. Table 2 shows how the three rewriting strategies (GO, RI and RA) transform a same input (original data).

**Global ordering** (GO) assigns a different number (it starts with 0 and proceeds incrementally) to each different concrete value, and always uses the same number when a same concrete value is used. This function is useful to identify patterns of reused values. Figure 3 column GO shows an application of this rewriting strategy to an example trace. Figure 3, E3, E6 column GO shows the same pattern of thread waited and notified independently from their concrete values. For instance, thread names can change in different executions, but global ordering can reveal recurrent pattern of thread waited and notified independently from their name.

**Relative to instantiation** (RI) replaces each attribute value with a number that indicates how many attribute values never observed before have been detected from the first appearance of value to be rewritten. This rewriting strategy is extremely useful to capture pattern that consists of attribute values that are generated, used and then discarded. For instance, thread names can change in different executions, but global ordering can reveal recurrent pattern of thread waited and notified independently from their name.

**Relative to access** (RA) replaces each attribute value with a number that indicates the number of values that have been observed from its last occurrence. This strategy is useful to identify recurrent definitions and uses of attribute values, independently from their concrete values. Figure 3 column RA shows an application of this rewriting strategy to an example trace. In this example RA captures that pairs of

**Table 2: Results obtained with data transformation techniques.**

<table>
<thead>
<tr>
<th>Orig. data</th>
<th>GO</th>
<th>RI</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1 A T1 21</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>E1 B T2 23</td>
<td>1 1 1</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>E1 A T1 23</td>
<td>0 0 1</td>
<td>2 2 1</td>
<td>2 2 1</td>
</tr>
<tr>
<td>E4 A 21</td>
<td>2 2 1</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>E5 T1 21</td>
<td>0 0 0</td>
<td>2 2 1</td>
<td>1 3 1</td>
</tr>
<tr>
<td>E3 C T2 21</td>
<td>2 1 0</td>
<td>0 1 2</td>
<td>0 3 1</td>
</tr>
<tr>
<td>E3 C T2 27</td>
<td>2 1 2</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>E2 Y T0 39</td>
<td>3 2 3</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>E2 Y T0 42</td>
<td>3 2 4</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>E1 A T3 39</td>
<td>0 3 3</td>
<td>4 0 2</td>
<td>5 0 2</td>
</tr>
<tr>
<td>E1 B T4 42</td>
<td>1 4 4</td>
<td>5 0 2</td>
<td>8 0 2</td>
</tr>
<tr>
<td>E1 A T3 42</td>
<td>0 3 4</td>
<td>4 2 1</td>
<td>2 2 1</td>
</tr>
<tr>
<td>E4 A 0</td>
<td>4 1 1</td>
<td>9 1 1</td>
<td></td>
</tr>
</tbody>
</table>

**Legend**

Orig. data is the data recorded into log files, GO indicates the global ordering data transformation strategy, RI indicates the relative to instantiation data transformation strategy and RA indicates the relative to access data transformation strategy.
Figure 3: Excerpt of a trace taken from our experiments.

variables, e.g., 39, 42 and 21, 23, are used according to a recurrent pattern.

These strategies have been applied to traces extracted from JVM executions. An excerpt of a log file containing events related to concurrency is shown in Figure 3. The first column indicates the original set of events and related attributes, while the other columns show the string that is used to replace attribute values (values of single attributes, while the other columns show the string that is used to replace attribute values (values of single attributes are separated by _). Go provided the worst results. Go is useful when recurrent event sequences are highly deterministic, otherwise observation of additional events modifies the values used to rewrite attributes, causing many potential matching labels to not match anymore. Unfortunately, deterministic behaviors seldom occurs in practice. Ri captured several interesting patterns, notwithstanding size and complexity of the log file. For instance, in the early portion of the log, it discovered a repeated sequence of threads that start and then wait. Finally, Ra captured both the initial pattern and several recurring patterns that indicate threads that are enabled, disabled, re-enabled, disabled again (on a different monitor), and finally disabled.

In this empirical experience, Ra provided the best results. However, both Ri and Ra appear to be promising rewriting strategies. The former because of its robustness to noisy events and the latter because of its high effectiveness in discovering recurring patterns. Finally, Go appears to be useful with simple examples, but extremely limited with realistic log files.

5. MODEL GENERATION

In this phase, the kBehavior inference engine [12] incrementally analyzes the pre-processed log files and generates a FSA that both summarizes and generalizes the observed event sequences. At each step, kBehavior reads a trace and updates the current FSA according to the content of the trace. The updated FSA guarantees to generate all traces that have been analyzed. Since kBehavior is incremental, as well as the pre-processing techniques, models can be completely produced at run-time without recording traces, resulting in an enormous reduction of occupied disk space.

Figure 4: An example FSA extended with a new trace. The state with the triangle is the initial state. The state with the double border is the final state. The dotted arrows and state 8 are added as a consequence of the extension step.

The algorithm used by kBehavior to extend a current FSA given a new trace is based on the identification of sub-machines in the current FSA that generates sub-sequences in the input trace. Once these relations are identified, the portions of the input trace that do not correspond to any sub-machine are used to create new branches in the current FSA so that the updated FSA generates both all event sequences generated by the previous FSA and the event sequences represented in the input trace. For example, Figure 4. shows how a FSA can be extended providing a new input string to kBehavior. In this simple example, the portion of the input string that does not correspond to any sub-machine lead to addition of a new branch to the current FSA. However, in the general case, a FSA is extended by gluing FSAs obtained from the recursive execution of kBehavior on the portions of the input string that cannot be associated with sub-machines. Technical details can be found in [12].

6. FAILURE ANALYSIS

In the failure analysis phase, we execute the target application either in the new field or with new workflows and, when a failure is observed, we compare the behavior described by inferred models with traces recorded in this phase. The goal of the comparison is to automatically identify anomalous patterns, i.e., event sequences that differ from the ones observed in previous executions. Anomalous patterns are likely to indicate the source and the cause of the failure, and can be efficiently inspected by testers independently from the original size of the log file.

A straightforward way to compare a trace and a model...
is to check if the model generates the trace. If the trace is generated by the model, there are no anomalies. If the trace is generated only up to a given point, namely $p$, there is an anomaly. The tester can thus inspect the set of events surrounding $p$ and the sub-machine around the state that has been reached by generating all symbols of the trace up to position $p$. Unfortunately, this strategy has little effectiveness when multiple anomalies or noisy data are present in a single trace. If a trace includes multiple anomalous patterns, the first anomaly in the trace hides all successive anomalies. Similarly, if the early portion of a trace includes unexpected but legal event sequences, the anomaly associated with the noisy data hides any successive anomaly. This happen because when an anomaly is detected, the remaining portion of the trace cannot be matched anymore and the checking is interrupted. This strategy is extremely ineffective when applications are executed for long time, thus traces are extremely long and several interesting information can be located in several points of a same trace, such as the case of workflow executions.

To avoid loss of important information, we implemented the matching process between traces and FSAs on top of the kBehavior extension mechanism, which is extremely useful in pairing event sequences and sub-machines independently from their positions. For instance, a sub-sequence that is located at the beginning of the trace can be associated with any sub-machine of the FSA. Thus, we use the trace to be matched to extend the current model, and we consider all the extensions points as the set of anomalous events that must be inspected by testers. In this way, presence of noisy data or multiple anomalies do not hinder effectiveness of the matching process because all anomalous sequences are identified. Figure 5 shows an example of a matching between a trace and a FSA.

7. PRELIMINARY EXPERIMENTS

We validated the technique proposed in this paper by evaluating its capability to detect common problems in JVM, which is a widely adopted software platform. In our validation, we considered the two classes of faults that have been identified as the major causes of failures in the JVM [3]: problems related to memory handling and problems related to concurrency. To reproduce these problems, we extracted eight bug descriptions and the corresponding sample programs that exhibit these faulty behaviors from Sun [18] and Jikes [6] online repositories. Three bugs are related to memory handling issues and five bugs are related to concurrency issues. The analyzed bugs are summarized in Table 3.

In this empirical experience, we executed the JVM with a workload and we collected data (monitoring), then we derived models of the observed behavior (model inference) and finally we used these models to identify problem causes on the sample programs (failure analysis).

To stress both memory handling and concurrency, we executed for 2 hours the James mail server [2], which is known to extensively use both aspects. The underlying JVM has been monitored with JVMMon. The collected data has been pre-processed and analyzed to produce general models for both memory handling and concurrency aspects. Finally, the sample applications have been executed with the same JVM and the collected data has been matched with the inferred models. In all cases, our technique detected anomalous event sequences. In three of the experiments, the anomalous event sequences directly pointed to the cause of the failure (bugs 4916841, 5073365 and 6450205), while in five of the experiments the technique indicated unexpected event sequences, but they were not clearly related to the failure (bugs 4916841, 4485042, 6450200, 6281487 and 5047214).

In most cases, the lack of a clear correlation between the anomalous sequences and the failure of the application can be related to the lack of useful events that have not been logged since the actual prototype implementation of JVMMon does not capture all event types yet. However, for all five bugs, the feedback provided by our technique has been useful to track part of the relevant events that incrementally lead the JVM to failure. In one of the completely successful experiences, the pattern of anomalous events identified by our technique interestingly represents a sequence that can be generalized as representative of an entire class of problems.

To provide an example of the real problems that our technique can automatically identify, we describe two of the three bugs that have been completely captured by our technique.

**Bug 4697804**

Bug 4697804 reproduces a fault leading the JVM to crash because of an out of memory error. Such error is related to a faulty behavior of the garbage collector, that, after several cycles, is not able to free memory anymore. In particular, the garbage collector needs to allocate data structures of fixed size and data structures requiring an amount of memory proportional to the amount of live data in the heap. The faulty behavior can be observed when resources available to
JVM diminish, and the garbage collector attempts to expand the heap but cannot successfully obtain space for data structures it needs. If this happens, the JVM fails to throw an OutOfMemory exception because no further Java code can be run, thus it prints an out of memory error and exits.

Matching the log file with our inferred models lead to the identification of six anomalous sequences. Three of them were related to new classes that have been loaded by the JVM. However, these anomalies were clearly dependent from the new application that has been considered and have been immediately discarded. Two anomalies indicated a new class loading schema for the class Cleaner, which is an important information related to the failure. The most important information is the last anomaly that indicated an unexpected sequence of Garbage Collector Start-Garbage Collector Finish that indicates the continuous attempts of freeing memory and expanding the heap. It is worth to point out that this kind of violation can be considered as representative of a wider class of faults able of leading the garbage collector to such anomalous behavior. Bug 6450205 reproduces a thread management problem of the ThreadPoolExecutor class in J2SE 1.5.0, which manages the execution of submitted tasks using pooled threads. When a new task is submitted, it is normally assigned to one of the idling thread in the thread pool. If there are fewer running threads than corePoolSize parameter or the queue is full, a new thread is created to handle the request. If a thread is idling for more than KeepAliveTime parameter, it is terminated. The correct behavior of this class prescribes that the number of threads must never be less than corePoolSize parameter value. Code reproducing bug 6450205 violates this condition by enabling the unexpected termination of all threads.

Our tool analyzed the log file corresponding extracted from the faulty application and identified an anomalous event sequence that indicates that all previously created threads terminated in an unexpected way. This fault causes loss of performance because when all threads have been terminated new tasks have to wait for a creation of a new thread before they can be executed. This problem is hard to diagnose, because it causes performance degradation, and a failure can be observed only when a significantly amount of tasks are assigned to thread pool. Information provided by our tool pointed out to event sequences representing this misbehavior.

### 8. RELATED WORK

Log file analysis is commonly used to investigate causes of failures that have been experienced either in-the-field or during testing. These techniques may differ for the kind of data that is analyzed, the purpose of the analysis and the results that can be produced. We can classify log file analysis techniques in two main groups: techniques that infer models of faulty behaviors to predict and identify failures [10, 15], and techniques that use user-specified models of the expected behavior to analyze logs and detect anomalous executions [1].

Techniques in the former group analyze traces to infer models (e.g. decision trees or data clusters) that represent observed faulty behaviors. These models can be used to predict further occurrences of a same problem. Traces can be either extracted by monitoring faulty applications [10] or injecting faults into correct systems [15]. The latter group of techniques verify if traces satisfy user-supplied models of the expected behavior, e.g., FSA, to identify possible problems [1].

The technique presented in this paper avoids the shortcoming of requiring the existence of user-specified models and complements techniques that infer faulty behaviors [10, 15]. Inferring faulty behaviors may be useful when large amount of data about faulty executions is available. In this case, inference techniques can accurately model faulty behaviors, thus being able to precisely predict future occurrences of a same problem. Moreover, observing new legal behaviors do not generate false positive on techniques based on models of faulty behaviors, while can induce false positives in techniques that model correct behaviors. However, the technique presented in this paper is useful in the many cases in which little information about faulty executions is available.

Finally, clustering and machine learning algorithms have been extensively used to reduce the size of log files and to improve their quality [16]. These approaches are useful in many contexts, but are not able to capture the recurrent patterns that we identify thanks to data transformation strategies. In particular, the experience presented in this paper shows that data transformation strategies revealed recurrent patterns - which would not be visible otherwise - that can be suitably processed by kBehavior.
automatically analyzed [1]. Unfortunately, formal specifications are expensive to be produced and are seldom available.

If repositories of faulty executions are available, it is possible to infer models of these problems and then match them with the logged data [10, 15]. However, repositories of faulty executions are not commonly available for all applications.

In this paper, we presented a technique based on the inference of models of expected behavior, which can be easily derived during testing (when workloads are executed). These models are used to identify anomalous event sequences in log files extracted during faulty executions. Models are generated from sets of events specific for a given class of problems, i.e., log files are partitioned into multiple issue-specific log files. Moreover, our technique suitably supports the use of attribute values that can be associated with recorded events.

We applied the technique to the JVM, which is one of the most used virtual execution environment. Preliminary experiments were conducted with 8 known bugs of the JVM. Results show that failure causes can be automatically identified for 3 of the 8 bugs. Interesting unexpected event sequences have been detected in the 5 remaining classes of bugs, but they were not completely correlated to the failure causes. This limitation is mainly due to the partial monitoring that is currently performed by JVMMon.

Future work is about increasing the accuracy of the monitoring provided by JVMMon, developing automated analysis techniques that correlate the unexpected event sequences to the root cause of the problem, defining techniques for automatically removing false positives and extending the empirical investigation to a larger set of platforms.

10. REFERENCES


