Runtime Monitoring in Continuous Deployment by Differencing Execution Behavior Model

Monika Gupta[1(✉)], Atri Mandal[3], Gargi Dasgupta[3], and Alexander Serebrenik[2]

[1] IBM Research, New Delhi, India
gupmonik@in.ibm.com
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
a.serebrenik@tue.nl
[3] IBM Research, Bengaluru, India
{atri.mandal,gaargidasgupta}@in.ibm.com

Abstract. Continuous deployment techniques support rapid deployment of new software versions. Usually a new version is deployed on a limited scale, its behavior is monitored and compared against the previously deployed version and either the deployment of the new version is broadened, or one reverts to the previous version. The existing monitoring approaches, however, do not capture the differences in the execution behavior between the new and the previously deployed versions.

We propose an approach to automatically discover execution behavior models for the deployed and the new version using the execution logs. Differences between the two models are identified and enriched such that spurious differences, e.g., due to logging statement modifications, are mitigated. The remaining differences are visualized as cohesive diff regions within the discovered behavior model, allowing one to effectively analyze them for, e.g., anomaly detection and release decision making.

To evaluate the proposed approach, we conducted case study on Nutch, an open source application, and an industrial application. We discovered the execution behavior models for the two versions of applications and identified the diff regions between them. By analyzing the regions, we detected bugs introduced in the new versions of these applications. The bugs have been reported and later fixed by the developers, thus, confirming the effectiveness of our approach.

Keywords: Continuous deployment · DevOps · Execution logs
Runtime flow graph · Release decision · Visualization

1 Introduction

Increasing speed of the changing priorities of customers causes many companies to adopt continuous deployment [1,8,15,23]. A continuous deployment model is crucial for service delivery business as it ensures that software services are always

© Springer Nature Switzerland AG 2018
C. Pahl et al. (Eds.): ICSOC 2018, LNCS 11236, pp. 812–827, 2018.
https://doi.org/10.1007/978-3-030-03596-9_58

in a releasable state, and changes are incremental. To ensure high quality release in continuous deployment, the upcoming release is staged in production environment using such strategies as blue-green deployment [13], dark launches [9], canary release [13,26] and shadow testing [23], and its performance is monitored [3,13] to quickly identify whether it is misbehaving [23,26].

Vast amount of data is logged during the execution of the new and previously deployed software versions. Existing monitoring systems keep track of suspicious events in logs (e.g., errors, warning messages, stack traces) and raise alerts. However, such systems do not leverage the unstructured data captured in the execution logs to efficiently derive and compare the dynamic behavior of the new and the previously deployed versions in a holistic manner.

In this work, we present a novel approach to automatically detect discrepancies in the fast evolving applications adopting continuous deployment. This is achieved by identifying the differences in the behavior model of the previously deployed and new version, derived by mining the execution logs.

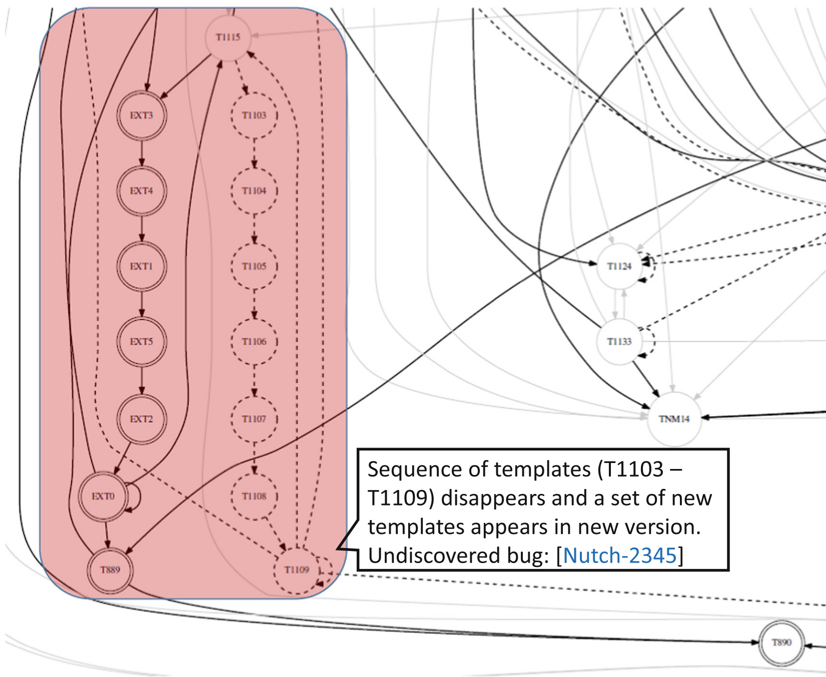

Fig. 1. Differences between the execution behavior models of two versions of Nutch. Vertices added in the new version are encircled twice, added edges–bold, deleted vertices and edges–dashed. The analysis of differences allowed us to discover a bug that we reported as NUTCH-2345. The bug was fixed by the Nutch developers.

2 Motivating Example: A Bug in Nutch

As part of an issue *[NUTCH-1934]*[1] the class *Fetcher* counting ca. 1600 lines of code is refactored to improve modularity. We took the version before and after refactoring to identify differences between the two versions. We used Nutch to crawl a set of URLs thus generating the execution logs for both the versions. We map the generated execution logs to templates derived from the Nutch source code using string matching. A subset of log lines was not mapped to any source code template (that is, from third party library) and clustered using a combination of approximate and weighted edit distance clustering. Execution behavior model is discovered automatically for each of the versions using the respective templatized execution logs. Each vertex in the model corresponds to a unique template. Using our automated approach, many diff regions are detected between the two discovered models.

Figure 1 presents one of the diff regions, i.e., deletion of a set of vertices $T1103$–$T1109$ (represented as dashed) from class *Fetcher.java* and addition of new vertices $EXT0$–$EXT5$ (double circled) from apparently third party library (prefixed with EXT). We manually investigated this diff region and found that the code fragment corresponding to templates $T1103$–$T1109$ has been moved from *Fetcher.java* to *FetchItemQueue.java*[2]. Inspecting *FetchItemQueue.java* we found that *FetchItemQueue**s*** is used as logger instead of *FetchItemQueue*. Consequently, the log messages from *FetchItemQueue* had a wrong class name, and thus were not mapped to the corresponding source code logging statement and treated as log statements from third party library ($EXT0$–$EXT5$).

This issue was introduced in Nutch 1.11 and fixed after we reported it[3] in Nutch 1.13. Using our approach, the issue would have been detected in the version 1.11 itself. This highlights the potential of our approach for discovering anomalies by analyzing automatically identified diff regions.

3 Proposed Approach

The proposed approach takes executions logs and source code for the deployed and new version as starting points. Since our approach is targeted towards continuous deployment, access to both these artifacts can be assumed. The approach leverages execution logs without instrumenting the code because instrumentation overhead is not possible in a fast evolving production software [31]. Nevertheless, execution paths are successfully captured from the existing logs because in practice, sufficient logging is done to facilitate runtime monitoring [6,16].

Our approach consists of three broad phases: *template mining* that maps each line in execution logs to a unique template (Sect. 3.1), *execution behavior model mining* that derives execution behavior models from the templatized logs and refines the model using multimodal approach (Sect. 3.2), and *analysis of*

---
1 https://issues.apache.org/jira/browse/NUTCH-1934.
2 http://svn.apache.org/viewvc?view=revision&revision=1678281.
3 https://issues.apache.org/jira/browse/NUTCH-2345.

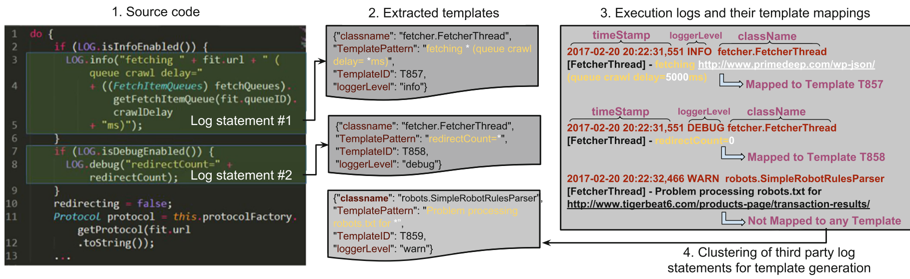

Fig. 2. Given the source code (1) templates are extracted (2), and log lines are mapped to them (3). Log lines from external libraries are clustered to create new templates (4).

the model differences to identify the differences between the execution behavior models and classify them into cohesive diff regions (Sect. 3.3).

3.1 Template Mining

A template is an abstraction of a logging statement in the source code consisting of a fixed part and variable part (denoting parameters) [2,18]. Due to presence of parameters templates often manifest themselves as different log messages. Thus, identifying the templates from the execution log messages has inherent challenges [20]. If no source code is available, templates can be inferred by clustering log messages [20,27]. However, often log messages from different logging statements are clustered together, resulting in inaccurate templates. Since we have access to source code, we extract templates using regular expressions (cf. Fig. 2).

Derive Templates from the Source Code: In this step the print statements are identified from the source code along with the class name and severity level (e.g., INFO, WARN and DEBUG) [6]. We search for the logging statements in the source code using regular expressions with some enhancements to identify ternary print statements and ignore commented logging statements in the source code. As shown in Fig. 2, logging statement is parsed and represented as a regular expression which is then assigned a unique template id. Class name and severity level are also stored as additional information to disambiguate templates which have identical invariant pattern but appear in different classes of the code.

While the complete source code is used to extract templates for the deployed source code version, to extract the templates for the new version we only analyze the *diff* between the two source code revisions as indeed, continuous deployment encourages incremental changes. Not only is the extraction more efficient, this also ensures that the unchanged templates between the two versions are represented by the same template ID. The main shortcoming of *diff* is that if a logging statement is modified, it is represented in the *diff* as a combination of addition and deletion, that will be interpreted as addition of a new template and deletion of the old template. Thus, two execution behavior models will appear

different for the templates which are actually the same. Since modification of logging statement is frequent [6,16], we address this shortcoming using a novel multimodal approach for template merging and model refinement (Sect. 3.2).

Templatize Log Messages: In this step, template id is assigned to each log line appearing in the execution logs, by matching with templates obtained from the previous step. To reduce the search space for the match, class name and severity level (if included as part of the log messages) are used as additional matching parameters (cf. Fig. 2). While regular expression matching can find the matching template, log lines matching multiple templates, templates with no fixed part and log lines generated by the third party libraries require special treatment. If a *log line matches more than one template* it is mapped to the most specific template, i.e., the template with the largest fixed part. If there is *one logging statement in a class without constant part* then all the unmapped log lines from that class with same logging level are mapped to it[4]. Finally, *log lines from external sources* such as third party libraries for which we do not have access to source code cannot be templatized as explained above. These log lines are clustered using a combination of approximate clustering [20] and weighted edit distance similarity [10]. Each cluster generated after the refinement is represented as a template and is assigned a unique template ID. Thereafter, non-templatized log lines are matched with the templates derived from clustering step such that all the log lines are assigned a unique template id.

3.2 Mining Execution Behavior Model Using Multimodal Approach

Execution Behavior Model (EBM) is a graphical representation of the templatized execution logs capturing the relationship between the templates. Each vertex in the model corresponds to a unique template and the edges represent the flow relationship between the templates. Since template represents a logging statement from the code, EBM captures a subset of possible code flows.

Accuracy of identified diff regions directly depends on the accuracy of the EBM mining which in turn depends on the accuracy of the template mining. As discussed in Sect. 3.1, the execution logs are templatized with high precision using source code. However, for log lines being generated from third party libraries we had to resort to the clustering based technique which has inherent limitations. This limits the template mining accuracy and consequently the accuracy of EBM mining. This is even more apparent in the new version because only a limited amount of logs is available, which is a hindrance to accurate mining [20]. Further, inconsistency in the templates because of the modified log statements in source code being recorded as new templates leads to many spurious differences between the compared models thus, making the *diff* analysis practically less effective. To overcome this problem, we propose an iterative EBM refinement

[4] The case with multiple such statements is very rare and hence does not affect our approach.

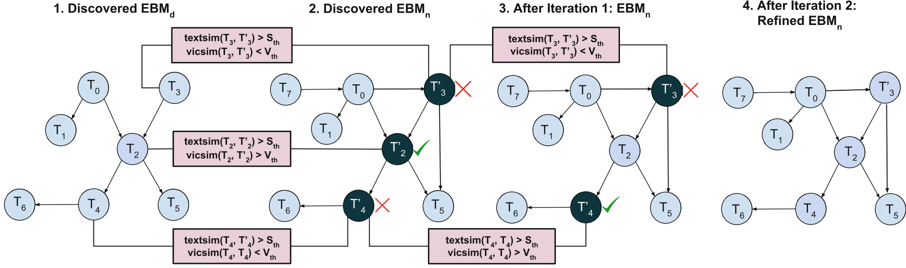

Fig. 3. Iterative multimodal execution behavior model refinement

strategy using multimodal signals that is, *text* and *vicinity* (i.e. predecessors and successors in EBM) of the template.

Iterative Execution Behavior Model Refinement: We derive execution behavior model for the deployed, EBM_d and the new version, EBM_n using corresponding templatized execution logs. We compare EBM_d and EBM_n to identify the vertices which are present in EBM_n but not in EBM_d (that is, $\triangle T_{add}$) and vice-versa, i.e. $\triangle T_{del}$. It is possible that the vertex from $\triangle T_{add}$ set is actually same as the vertex from set $\triangle T_{del}$ but captured as different template as discussed above. We identify and resolve such cases using proposed multimodal approach thus, reducing the spurious *diff* and making the comparison more effective.

One of the multimodal signals that we use is *textual similarity* between the templates from $\triangle T_{del}$ and $\triangle T_{add}$. If there are m templates in $\triangle T_{del}$ and n templates in $\triangle T_{add}$ then similarity is calculated between $m \times n$ pairs. The pairs with textual similarity above a threshold are captured as *potential merge candidates*. We do not merge the templates simply based on text similarity because there can be two textually similar templates corresponding to different logging statements in the code. Hence, to improve the precision, we evaluate the similarity for one more modality, i.e., *vicinity similarity*, where vicinity is the set of predecessors and successors. If the vicinity similarity is above a threshold, the templates are marked as identical. Thresholds for textual similarity and vicinity similarity can be selected based on grid search and fine tuned to project requirements [20].

We continue the process iteratively with each step leading to a more refined EBM_n. With every iteration some of the vertices are marked as identical which in turn can change the value of vicinity similarity for other candidate pairs. We stop the iterations when no more candidate pairs can be merged and the EBM_n output of subsequent steps no longer changes.

Example 1. Consider the EBMs shown in Fig. 3. By comparing EBM_d and EBM_n, we observe that $\triangle T_{del} = \{T_2, T_3, T_4\}$ and $\triangle T_{add} = \{T_2', T_3', T_4'\}$. We calculate *text similarity* for all the nine pairs and find the *potentially similar candidate set*, $C = \{(T_2, T_2'), (T_3, T_3'), (T_4, T_4')\}$. *Vicinity similarity* is checked for all the candidates and in first iteration vicinity similarity is above the threshold only

for one pair, (T_2, T_2') which is marked as identical and removed from C. In the next iteration, the remaining pairs from C are analyzed for the vicinity similarity which is found to be greater than the threshold for (T_4, T_4'), which is again marked as identical and removed from C. Only one pair, (T_3, T_3') is not marked as same because its vicinity similarity is below threshold even though the textual similarity is high. Consequently, *diff* set after EBM_n refinement is reduced to $\triangle T_{add} = \{T_3'\}$ and $\triangle T_{del} = \{T_3\}$.

3.3 Analyzing Differences Between Execution Behavior Models

Since EBMs are graphs identifying the differences between them can be seen as the graph isomorphism problem [14], known to be in NP. However, since we ensure the consistency in the template ID across the two models, the comparison of two models is simplified. The refined models are compared to identify the following differences: sets of vertices, $\triangle \mathit{diff}_v$ and edges, $\triangle \mathit{diff}_e$ which are added/deleted, as well as the set of vertices for which the relative frequency of outgoing transitions has changed $\triangle \mathit{diff}_{dist}$ in EBM_n when compared to EBM_d. For efficient follow-up analysis, we group the identified differences into cohesive regions such that the related differences are investigated as single unit.

Example 2. Deletion of T1103–T1109 and the corresponding edges, and addition of EXT0–EXT5 and the corresponding edges in Fig. 1 are grouped together.

Vertex Anchored Region: Intuitively, we would like to find the maximum point from which the difference in execution behaviors is observed and the minimum point up to which there are differences in the execution behavior. For the differences with same maximum point, it is highly likely that they are caused due to modification in same code, and, thus, should be investigated as a single unit.

A vertex, v_i is randomly selected from $\triangle \mathit{diff}_v$ as a seed to detect the region. We back traverse the graph till an unchanged ancestor (that is, vertex common between the two models) is detected along all the paths to v_i. All the vertices and edges along the path (including unchanged ancestor) are marked as part of the region. For all marked vertices, all the outgoing branches are traversed and marked till an unchanged child vertex (that is, vertex common between the two models) is detected. Unchanged child vertex is not included in the region because the boundary of region is defined till the last difference in the included path. Effectively, a region covering a set of vertices and edges is identified. The process repeats as long as there remain unmarked vertices in $\triangle \mathit{diff}_v$. At the end of this step, all vertices from $\triangle \mathit{diff}_v$ and *some* edges from $\triangle \mathit{diff}_e$ are marked as part of some region. We call these regions as *vertex anchored regions.*

Example 3. Consider Fig. 4 where $\triangle \mathit{diff}_v = \{T_0, T_2, T_3, T_4, T_6, T_7, T_8, T_{10}\}$ and $\triangle \mathit{diff}_e = \{(T_0, T_{11}), (T_1, T_0), (T_1, T_4), (T_1, T_3), (T_1, T_2), (T_1, T_6), (T_4, T_5), (T_3, T_5), (T_2, T_5), (T_{11}, T_6), (T_6, T_7), (T_7, T_{11}), (T_{10}, T_{11}), (T_{10}, T_9), (T_9, T_{10}), (T_9, T_8), (T_8, T_9),$

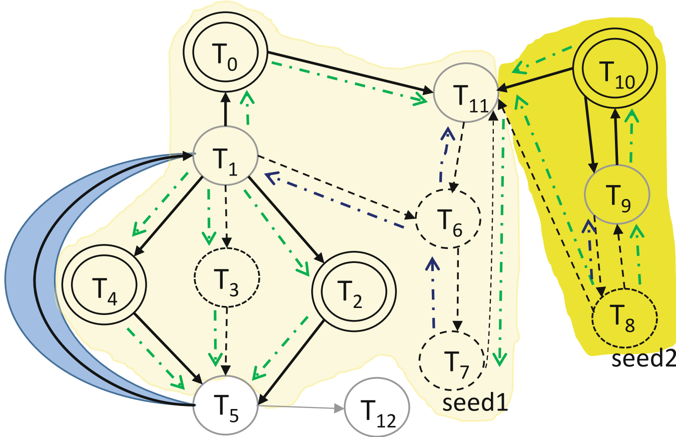

Fig. 4. There are two vertex anchored region (different shades of yellow) and one edge between the unchanged vertices (blue). Blue pointers correspond to backtracking, and green pointers depict forward tracking. (Color figure online)

$(T_8, T_{11}), (T_5, T_1)\}$. We choose T_7 as the first seed and back traverse its incoming path (blue pointers) up to maximum unchanged vertices, i.e., $\{T_1, T_{11}\}$, marking vertices $\{T_7, T_6, T_1, T_{11}\}$ and edges $\{(T_7, T_6), (T_6, T_1), (T_6, T_{11})\}$. Next, the outgoing branches are traversed (green pointers) till unchanged vertex is detected and the corresponding vertices are marked. As a result, the light yellow region is created consisting of $V_{r1} = \{T_0, T_2, T_3, T_4, T_6, T_7\}$ and $E_{r1} = \{(T_0, T_{11}), (T_1, T_0), (T_1, T_4), (T_1, T_3), (T_1, T_2), (T_1, T_6), (T_4, T_5), (T_3, T_5), (T_2, T_5), (T_{11}, T_6), (T_6, T_7), (T_7, T_{11}), (T_5, T_1)\}$ from $diff_v$ and $diff_e$ respectively. For the next iteration we choose T_8 as a seed from the set of uncovered vertices in $diff_v$ and repeat the process to identify another region. The second region becomes $V_{r2} = \{T_8, T_9, T_{10}\}$ and $E_{r2} = \{(T_{10}, T_{11}), (T_{10}, T_9), (T_9, T_{10}), (T_9, T_8), (T_8, T_9), (T_8, T_{11})\}$. Hence, all the vertices from $diff_v$ and a subset of $diff_e$ are grouped in one of the cohesive regions.

Edge Anchored Region: Not all edges from $\triangle diff_e$ belong to one of the vertex anchored regions. These are mainly the edges added/deleted between unchanged vertices and should be analyzed separately. We refer to each of these edges along with its vertices as an *edge anchored region*.

Example 4. After detecting the vertex anchored regions in Fig. 4 only one edge in $\triangle diff_e$ is unmarked. The only edge anchored region is hence $V_{r3} = \{T_1, T_5\}$ and $E_{r3} = \{(T_5, T_1)\}$.

Distribution Anchored Region: Apart from the above two cases of structural changes (addition or deletion of vertex or edge) in execution behavior model, we investigate the vertices common in both the versions of the model to detect

the deviations in changes in the relative frequency of outgoing transitions. To capture the distribution change, for a given vertex v and its outgoing transitions common between the two models we compute $\frac{|f_d(i)-f_n(i)|}{f_d(i)}$, where $f_d(i)$ ($f_n(i)$) is a relative frequency of transition i in EBM_d (EBM_n) among the outgoing transitions of v common between the two models. If the metric value is above threshold for at least one transition from the vertex v, it is marked as *distribution anchored region*. Threshold needs to be decided manually based on the project requirements such that minor changes are discounted (that is, not considered as part of the differences) and major changes are marked in the differences.

4 Evaluation: Open Source and Proprietary Applications

We evaluated our approach on two different applications: (i) Nutch[5], an open source web crawler, and (ii) an industrial log analytics application. We have already shown some initial results on the Nutch project in Sect. 2 and discuss the other findings here. Also all the artifacts such as execution logs, templatized logs, execution behavior model and diff files are made publicly available for reproducibility of the results[6]. Details of the industrial application cannot be divulged for confidentiality reasons. We selected these applications primarily because of the availability of the source code and historical data on bugs and the corresponding fixes, as well as frequently occurring incremental changes in these applications. Execution logs for these projects were not available so we use a custom load-generator to generate logs for different source code versions.

Table 1. Properties for the two versions of Nutch application

Attribute	Nutch	
	Ver 1	Ver 2
Classes	415	420
Total LOC	67658	67891
Logging statements in src	1098	1097
Total lines in execution log (approx)	94137	125695
Total [Info, Debug]	19K,73K	26K,98K
Total [Error, Warn]	408,178	354,604
Vertices in model	106	104
Edges in model	328	310

[5] http://nutch.apache.org/.
[6] https://github.com/Mining-multiple-repos-data/Nutch-results.

4.1 Experimental Results for Nutch

Two Nutch versions were used: (i) before the commit for [NUTCH-1934], henceforth called version 1 (deployed/prod version) and (ii) after the commit for [NUTCH-1934], henceforth called version 2 (new version). This commit is considered a major change as a big class, *Fetcher.java* (ca. 1600 lines of code) is refactored into *six* classes. Table 1 presents details for two Nutch versions. We derive the templates from the source code for version 1, henceforth called $templates_{v1}$. To derive the templates for version 2, $templates_{v2}$, 46 templates are deleted, and 47 templates are added to $templates_{v1}$ in accordance with the code *diff* (here, *git-diff*) between the two code versions. We generate the execution logs for both the versions by crawling same URLs (that is, mimic prod) and observe that number of loglines generated for version 1 are less than that for version 2 (cf. Table 1). The execution logs for version 1 and version 2 are templatized using $templates_{v1}$ and $templates_{v2}$, respectively. Around 12% log lines are not templatized, and hence are clustered. 80 clusters are obtained. The clusters are further refined and grouped using weighted edit distance reducing their number to 26. Non-templatized log lines are matched with the templates generated after clustering and every line in the execution log is templatized for both the versions. We discover the execution behavior model (EBM) for both the versions, EBM_1 and EBM_2 and refine them using multimodal approach.

The behavior model shows that there are 53 added and 47 deleted vertices in EBM_2 as compared to EBM_1. For every pair of the added and deleted vertices, text similarity is calculated from the source code. The *text similarity* is found to be above a threshold (here, 0.8) for 36 out of 2491 pairs and the corresponding vicinity is compared in EBM_1 and EBM_2. *Vicinity similarity* is also found to be above a threshold (that is, 0.5) for all the 36 candidate pairs. Thus, these vertices are marked to be the same templates across the two EBMs. For better understanding, diff refinements file is made publicly available at the *link* (See footnote 6). As a result, all the templates which are captured as new template because of refactoring got mapped to the corresponding old templates reducing the number of differences significantly. Refined EBM_2 is compared with EBM_1 to identify and analyze the differences. The final refined model with diffs is made publicly available (See footnote 5).

We observe several differences which are grouped as cohesive regions using approach discussed in Sect. 3.3.

We identified one region which is explained in Sect. 2. In the additional region we observe (i) deletion of vertex corresponding to template "Using queue mode : byHost" (though present in source code of both the versions), and (ii) significant change in distribution of a vertex $T1135$ such that the edge $T1135 \rightarrow T1131$ traversed only twice in EBM_1 has been traversed 601 times in EBM_2. Both observations are related to *FetcherThread.java* which is investigated manually and a bug is identified in the way URLs are redirected. Instead of following the correct redirect link, the code was following the same link over and over again. After the maximum number of retries is exceeded further processing of the URL stopped with the message $T1131$ ("- redirect count exceeded *"), thus, increasing frequency of this edge traversal. This bug has already been reported as

NUTCH-2124[7] and attributed to patch commit we are analyzing. This validates the findings of our approach and highlights its usefulness. Therefore, using our approach we not only detect differences but also provide the context to derive actionable insights.

4.2 Experimental Results for the Industrial Application

The EBM generated automatically by our code is shown in Fig. 5 with annotations. Grey denotes the part which is common in EBM_d and EBM_n, dashed is the part which is present only in EBM_d but not in EBM_n, and bold edges/double encircled correspond to the part which is present in EBM_n but not in EBM_d. We selected two code revisions (referred to as $v1$ and $v2$) of the project such that it captures different kinds of code changes possible in software development cycle.

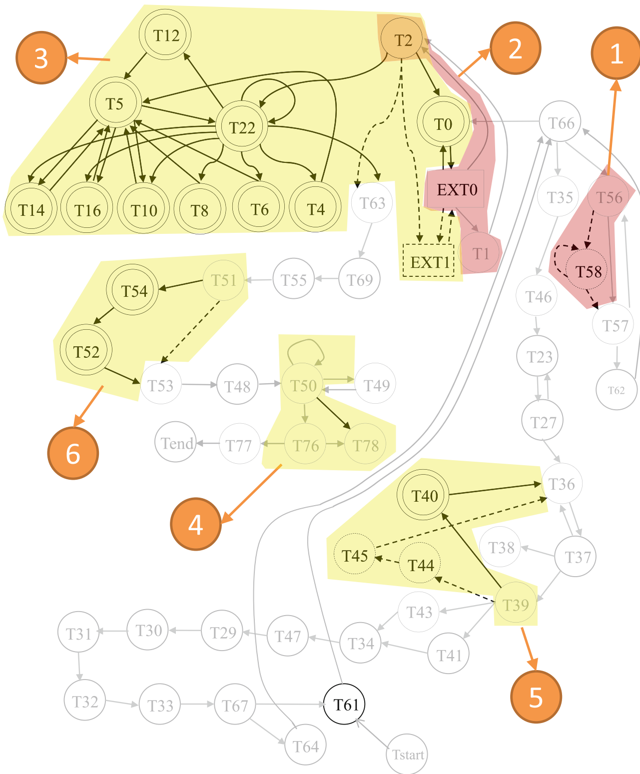

Fig. 5. Annotated execution behavior model highlighting the regions of *diff* for internal analytics application. Grey part is common in EBM_d and EBM_n, dashed is the part which is present only in EBM_d but not in EBM_n, and bold edges/double encircled are present in EBM_n but not in EBM_d

7 https://issues.apache.org/jira/browse/NUTCH-2124.

As shown in the Fig. 5, our approach detected six different regions of change between the two revisions, which we explain below.

Region 1: $T58$ template is present in both the source code versions but is not observed in EBM_n. Manual inspection of the code and commit history reveals that this is actually a bug caused due to faulty regular expression match and hence one conditional statement is skipped completely.

Region 2: A shift in distribution between edges $(EXT0, T1)$ and $(EXT0, T2)$ that is, increase in transition to $T1$ by factor of 8. Manual inspection reveals that the cause of this anomaly is a wrong Boolean condition check, which caused the distribution to be flipped between two conditional statements.

Region 3: Many new nodes appeared in EBM_n because a new Java class is added (identified in manual investigation) which gets invoked in the new version, i.e., this is an evolutionary design change. Addition of $T0$ however is not exactly related to this change. It is from the class that invokes the new feature but was added in Region 3 alongside the new class because of its close proximity.

Region 4: It has only one change viz. the addition of edge $T50 \rightarrow T78$ and an accompanying decrease in the frequency of $T76 \rightarrow T78$. Manual investigation highlights that $T78$ corresponds to a new exception check added in the class containing $T50$ thus whatever is not caught at $T50$ level is caught at $T76$.

Region 5: Main change is addition of node $T40$ and disappearance of nodes $T44$ and $T45$. Both $T44$ and $T45$ are exception nodes which exist in both code revisions while $T40$ is a new node. On manual inspection, it is revealed that this change is actually a result of bug fix that is, for `ArrayOutOfBounds` exception. This validates that the bug fix is working as intended.

Region 6: Two new nodes appeared in EBM_n and investigation of revision history reveals that a new function was added with two prints which is invoked just after $T51$ in the code thus, an evolutionary change.

To summarize, our approach has successfully detected all seven regions of code change between the two code revisions. It coalesced two of the regions (in Region 3) but this does not affect the usability of our approach as these regions are in close proximity. Manual investigation of *diff* regions in EBM highlights regression bugs as well as validates the evolutionary changes.

5 Discussion

Based on the project requirements, the information from the proposed approach can be leveraged in different ways to help improve the continuous deployment process. It provides additional insights (not to replace the existing practices) for some of the use cases as discussed below:

– *Go/No-go during Release Decision Making:* Most software companies have the concept of a go/no-go meeting before a production release where product, development and operations managers get together to decide whether to go ahead with the release of newer version or not. Our approach provides a way to visualize differences between the code versions in a graphical way thus being easily consumable for decision-making.
– *Update Test Suite to Cover Modified Execution Flows:* The proposed approach identifies region of differences between two execution behavior models thus, merit for comprehensive review. Regression testing can be performed for the diff regions instead of testing the whole application thus making regression testing leaner and at the same time more reliable and effective.
– *Optimal Test Suite Coverage:* By looking at discovered execution flow graphs, it is possible to identify the code paths which are frequently taken during runtime, and tests can be designed intelligently. Hence ensure that more frequent paths are tested rigorously using sophisticated techniques.

The approach assumes the presence of an identifier (i.e. thread ID) to capture the trace for an execution. Since thread ID is often present in the execution logs [32], it is fair to make this assumption. When the identifier for an execution is not present, execution behavior model can be mined using other techniques [20].

To keep our approach language independent and light weight, we do not use static analysis techniques [19]. Also static analysis will not capture the complete reality of execution behavior which gets influenced by prod configuration.

We mine the differences between the execution behavior models of two versions. However, we do not associate the differences with the change type such as bug fixing or feature addition. This kind of classification will not only help in quick resolution of bugs but will also act as an additional check to see if all the release items have been properly taken care before signing off on deployment.

6 Threats to Validity

Threats to Construct Validity: It focuses on the relation between the theory behind the experiment and the observation [29]. Performance of the approach depends on the pervasiveness of logging hence, if less logging statements then it may not be possible to derive useful inferences. However, given that logs are primary source for problem diagnosis, sufficient logging statements are written in the software [6].

Threats to External Validity: External validity is concerned with the generalizability of the results to other settings [29]. We conducted experiments on one open source and one proprietary project to illustrate the effectiveness of the approach. However, both are Java based projects using Log4j library for logging thus, very similar in terms of logging practice. While the approach does not make any project specific assumptions, it is possible that the performance can vary for different project characteristics. Accuracy of multimodal approach depends on the thresholds and thus can vary across projects.

7 Related Work

Execution logs have been extensively studied in such contexts as anomaly detection [4,20], identification of software components [24], component behavior discovery [17], process mining [28], behavioral differencing [12], failure diagnosis [25], fault localization [30], invariant inference [5], and performance diagnosis [10,26]. In this section, we focus on automatic analysis of execution logs.

Goldstein *et al.* [12] analyze system logs, automatically infer Finite State Automata, and compare the inferred behavior to the expected behavior. However, they work on system logs with predefined states while we identify these states (templates) first. Moreover, they present the differences as independent units whereas we group them together rendering the representation more usable.

Cheng *et al.* [7] propose to extract the most discriminative subgraphs which contrast the program flow of correct and faulty execution. Fu *et al.* [10] derive a Finite State Automata to model the execution path of the system and learn it to detect anomalies in new log sequences. However, these are supervised approaches assuming the presence of ground truth for correct and faulty executions to learn a model. Nandi *et al.* [20] detect anomalies by mining the execution logs in distributed environment however, anomalies are detected within the same version, no differencing between the flow graphs of two versions.

Tarvo *et al.* [26] automatically compare the quality of the canary version with the deployed version using a set of performance metrics such as CPU utilization and logged errors however, do not detect the differences in execution flow which is crucial for finding discrepancies. A set of techniques compare multiple versions of an application. Ramanathan *et al.* [22] consider program execution in terms of memory reads and writes and detect the tests whose execution behavior is influenced by these changes. Ghanavati *et al.* [11] compare the behavior of two software versions under the same unit and integration tests. If a test fails in the new version, a set of suspicious code change is reported. This approach works best when comprehensive test suites are available which is not the case in considered Agile environment.

Beyond the specifics of the execution log analysis, our work can be positioned in the context of continuous deployment. Continuous deployment can be seen as an extension of continuous integration [33] and the following step on "the stairway to heaven", the typical evolution path for companies [21].

8 Conclusion and Future Work

We have presented an approach to efficiently highlight the differences in the execution behavior caused due to incremental changes in fast evolving applications. We automatically discover execution behavior model using multimodal approach for the deployed and the new version by mining the execution logs. The models are compared to automatically identify the differences which are presented as cohesive diff regions. Since we have used a graphical representation, we not only identify diff regions but also the context to facilitate in-depth analysis.

Our preliminary evaluation on the open source project Nutch and internal log analytics application illustrates the effectiveness of the approach. Using our approach, we were able to detect multiple bugs introduced in new version for both the applications. Following the analysis, we found that some of the detected bugs were already reported in their issue tracking system therefore, we reported the remaining ones which were later fixed by the developers.

As part of future work, we plan to evaluate the approach on several other applications. Also we plan to automatically classify identified *diff* regions as anomaly and drill down to the root cause commit(s) using revision history.

References

1. Adams, B., McIntosh, S.: Modern release engineering in a nutshell-why researchers should care. In: SANER, pp. 78–90 (2016)
2. Arnoldus, J., van den Brand, M.G.J., Serebrenik, A., Brunekreef, J.: Code Generation with Templates. Atlantis Press, Amsterdam (2012)
3. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional, Boston (2015)
4. Bertero, C., Roy, M., Sauvanaud, C., Trédan, G.: Experience report: log mining using natural language processing and application to anomaly detection. In: ISSRE, pp. 351–360 (2017)
5. Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., Ernst, M.D.: Leveraging existing instrumentation to automatically infer invariant-constrained models. In: ESEC/FSE, pp. 267–277. ACM (2011)
6. Chen, B., Jiang, Z.M.J.: Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. EMSE **22**(1), 330–374 (2016)
7. Cheng, H., Lo, D., Zhou, Y., Wang, X., Yan, X.: Identifying bug signatures using discriminative graph mining. In: ISSTA, pp. 141–152. ACM (2009)
8. Claps, G.G., Svensson, R.B., Aurum, A.: On the journey to continuous deployment: technical and social challenges along the way. IST **57**, 21–31 (2015)
9. Feitelson, D.G., Frachtenberg, E., Beck, K.L.: Development and deployment at facebook. IEEE Internet Comput. **17**(4), 8–17 (2013)
10. Fu, Q., Lou, J.G., Wang, Y., Li, J.: Execution anomaly detection in distributed systems through unstructured log analysis. In: ICDM, pp. 149–158 (2009)
11. Ghanavati, M., Andrzejak, A., Dong, Z.: Scalable isolation of failure-inducing changes via version comparison. In: ISSRE Workshops, pp. 150–156 (2013)
12. Goldstein, M., Raz, D., Segall, I.: Experience report: log-based behavioral differencing. In: ISSRE, pp. 282–293 (2017)
13. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education, London (2010)
14. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: an analysis of approaches to support model differencing. In: ICSE Workshop on Comparison and Versioning of Software Models, pp. 1–6 (2009)
15. Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.P., Itkonen, J., Mäntylä, M.V., Männistö, T.: The highways and country roads to continuous deployment. IEEE Softw. **32**(2), 64–72 (2015)
16. Li, S., Niu, X., Jia, Z., Wang, J., He, H., Wang, T.: Logtracker: learning log revision behaviors proactively from software evolution history. In: ICPC (2018)

17. Liu, C., van Dongen, B.F., Assy, N., van der Aalst, W.M.P.: Component behavior discovery from software execution data. In: ICPC (2018)
18. Messaoudi, S., Panichella, A., Bianculli, D., Briand, L., Sasnauskas, R.: A search-based approach for accurate identification of log message formats. In: ICPC (2018)
19. Muske, T., Serebrenik, A.: Survey of approaches for handling static analysis alarms. In: SCAM, pp. 157–166 (2016)
20. Nandi, A., Mandal, A., Atreja, S., Dasgupta, G.B., Bhattacharya, S.: Anomaly detection using program control flow graph mining from execution logs. In: KDD, pp. 215–224 (2016)
21. Olsson, H.H., Alahyari, H., Bosch, J.: Climbing the "stairway to heaven" - a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: Software Engineering and Advanced Applications, pp. 392–399 (2012)
22. Ramanathan, M.K., Grama, A., Jagannathan, S.: Sieve: a tool for automatically detecting variations across program versions. In: ASE, pp. 241–252 (2006)
23. Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., Stumm, M.: Continuous deployment at Facebook and OANDA. In: ICSE Companion, pp. 21–30 (2016)
24. Shatnawi, A., Shatnawi, H., Aymen Saied, M., Al Shara, Z., Sahraoui, H., Seriai, A.: Identifying software components from object-oriented APIs based on dynamic analysis. In: ICPC (2018)
25. Tan, J., Pan, X., Kavulya, S., Gandhi, R., Narasimhan, P.: SALSA: analyzing logs as StAte machines. In: USENIX Workshop on Analysis of System Logs, pp. 1–8 (2008)
26. Tarvo, A., Sweeney, P.F., Mitchell, N., Rajan, V., Arnold, M., Baldini, I.: CanaryAdvisor: a statistical-based tool for canary testing. In: ISSTA, pp. 418–422. ACM (2015)
27. Vaarandi, R., Pihelgas, M.: Logcluster–a data clustering and pattern mining algorithm for event logs. In: Network and Service Management, pp. 1–7. IEEE (2015)
28. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process discovery using integer linear programming. In: Petri Nets, pp. 368–387 (2008)
29. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29044-2
30. Wong, W.E., Debroy, V., Golden, R., Xu, X., Thuraisingham, B.: Effective software fault localization using an RBF neural network. IEEE Trans. Reliab. **61**(1), 149–169 (2012)
31. Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., Pasupathy, S.: SherLog: error diagnosis by connecting clues from run-time logs. In: ACM SIGARCH Computer Architecture News, vol. 38, pp. 143–154 (2010)
32. Yuan, D., Zheng, J., Park, S., Zhou, Y., Savage, S.: Improving software diagnosability via log enhancement. ACM Trans. Comput. Syst. (TOCS) **30**(1), 4:1–4:28 (2012)
33. Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., Vasilescu, B.: The impact of continuous integration on other software development practices: a large-scale empirical study. In: ASE, pp. 60–71 (2017)