Automated Detection of Discontinuities in Models Inferred from Execution Traces

Alessandro Marchetto, Cu Duy Nguyen, Paolo Tonella
Software Engineering Research Unit
Fondazione Bruno Kessler
Trento, Italy
marchetto, cunduy, tonella@fbk.eu

Abstract—Modern applications (e.g., the so called Future Internet applications) exhibit properties that make them hard to model once for all. In fact, they dynamically adapt to the user’s habits, to the context, to the environment; they dynamically discover new services and components to integrate; they modify themselves through reflection, automatically. Model inference techniques are based on the observation of the application behavior (trace collection) and on its generalization into a model. Model inference supports testing, understanding and evolution of the software. However, inferred models may become obsolete at run time, due to the evolution or the self-modifications of the software. We investigate an approach for the automated detection of model discontinuities, based on a trade off between delay of the detection and accuracy, measured in terms of false negatives.

Keywords—Model Inference; High Variability.

I. INTRODUCTION

Current trend in software development is toward deferring several design decisions to the run time. Applications are deployed with a number of alternatives embedded into them and run time conditions determine which of them should be activated. This paradigm has been envisioned by the autonomic, self-adaptive systems vision [1] and takes nowadays multiple forms, among which that of the so-called Future Internet applications. Adaptive software has the capability to monitor the execution environment and make decisions on how to adapt itself to changes. Examples of such adaptations are load distribution among computational nodes, GUI reconfiguration to the user behavior, dynamic service discovery and integration, code mobility. Modeling such applications is a hard task and the software engineering research community is extremely active in this field (see, e.g., the SEAMS workshop series). The problem becomes even more difficult when models are inferred from observations instead of being provided upfront. In fact, model inference should also automatically adapt to the discontinuities of behavior that the application under analysis may exhibit.

Model inference has been successfully employed in many areas, including program comprehension, software testing and evolution [2], [3], [4]. Models can be inferred by means of state abstraction or event sequence abstraction. ADABU is a tool for model inference based on state abstraction. Abstraction functions are defined to map concrete states into abstract states, so as to control the size of the inferred model and to allow for generalization from the actually observed states [2]. Event sequence abstraction takes advantage of regular language inference algorithms, such as $k$-tail [5], or its variants [6], [7]. A finite state machines is obtained which recognizes the language of the event sequences observed in execution logs. Such finite state machine is actually a generalization of the observed sequences (not just their union), hence it may give raise to both false positives and false negatives when applied to new event sequences (i.e., it is an unsound generalization). Reducing the amount of both types of errors is the goal of most research in the area of even sequence based model inference.

Models inferred from execution traces have been used to test the application. Test cases have been generated for Ajax Web applications [3] by abstracting the concrete DOM of the Ajax Web pages into abstract model states (using state based abstraction). Search based generation of test cases from the inferred model has been also investigated for Ajax Web applications, with the aim of increasing the test case diversity [4]. Inferred models have been also used for anomaly detection [8]. When an event sequence is not accepted by the finite state model inferred for the application, an anomaly is automatically detected. If properly classified as a fault, it may trigger self-protection actions [9] or it can be simply reported to the developers. Models are also inferred to help understanding of the application under analysis. This is particularly beneficial for applications used to implement complex, organizational business processes (e.g., e-government, e-health, etc.). Model inference has been proved to be very effective in extracting business models from execution logs [10], [11].

Despite its enormous potential to help software developers, model inference is threatened by nowadays dynamic, adaptive software. The inferred model may in fact become obsolete when run time changes occur. We propose a technique to automatically detect the occurrence of discontinuities in the application behavior that are regarded as demanding for a model update. While the cause for such a discontinuity may be diverse, ranging from changed user behavior, to newly discovered services or components, the effect is that a new model must be inferred for the application. If the model is used for testing purposes, the
Figure 1. Using a sliding window to detect model discontinuities

detection of a discontinuity might indicate the need for a new testing session, aimed at validating the newly exhibited behavior. If the model is used for anomaly detection, it should be updated to avoid false positives. If the model is used for code understanding, it should be updated to reflect the new underlying application behaviors and processes.

Our technique for automated discontinuity detection is based on a similarity metric that is used to compare a model inferred in the past with a more recently inferred model. When the similarity goes below a given threshold, a discontinuity is detected. The two key parameters of this approach are the window size, i.e., the number of log events used for model inference, and the similarity threshold. We argue that different values of these two parameters bring different, contrasting benefits. Low window size allows for fast detection of discontinuities, but typically requires a low threshold to avoid false positives, which in turn gives raise to false negatives. On the other hand, high threshold is typically possible only at high window size, which makes the recognition of discontinuities slower. We resort to multi-objective Pareto optimality to find good trade-offs between these two contrasting requirements.

After presenting the approach in Section II, we provide some preliminary experimental results in Section III. The presented discontinuity detection approach can be potentially applied in conjunction with several existing model inference techniques. In the case study, however, we detailed the results we obtained applying our technique to an Ajax application for which state abstraction was used as model inference technique [3]. Results are reported in Section III, followed by conclusion and future work, in Section IV.

II. AUTOMATED MODEL CHANGE DETECTION

Let us consider a context in which execution traces are continuously collected while the application to be modeled is running and a model inference algorithm is used to infer a model from the traces. In such a context, we can automatically detect a model discontinuity when the similarity between a recently inferred model and a previously inferred one goes below a given threshold. In fact, this indicates that there is substantial difference between the recent behavior of the application, as represented in the inferred model, and the old one.

Figure 1 provides a graphical representation of the idea behind the proposed approach. A sliding window is moved over time as new execution events are collected by monitoring the application being modeled. Such a window has a size of \(2w\). Its first half, of size \(w\), is used to infer a model from past executions. This is indicated as model \(M_1\). The second half of the window, still of size \(w\), is used to infer a model \(M_2\) from the most recently traced events. The idea is that when the similarity between the old model \(M_1\) and the recent model \(M_2\) goes below the threshold \(T\), a discontinuity in the application behavior is detected. Hence, a new model for the application needs to be inferred, to properly model the new behavior of the application. If the model is used for testing purposes, this might indicate the need for additional test case generation and execution. It should be noticed that the detected discontinuity might be associated with a (possibly autonomic) evolution of the application, but also with a changed usage profile. We think both are relevant and worth automated detection for successive model update (and maybe application re-test).

The two critical parameters involved in the automatic change detection shown in Figure 1 are the window size \(w\) and the similarity threshold \(T\). In fact, a too large window size \(w\) introduces a big delay in the point in time when the change is detected. In fact, the second half of the window must be filled in with events associated with the new behavior before the change can be detected and when \(w\) is big this requires more time. Hence, we want to keep \(w\) small for fast reaction to changes. However, a too small \(w\) results in models that do not account for a sufficient sample of application behaviors, such that models changes are continuously detected just because of model instability, due to the small window size, rather than actual changes of behavior. The other parameter, the similarity threshold \(T\), suffers similar problems. A too high value results in lots of false positives, i.e., similarity goes below threshold often, even when only minor changes happen. on the other hand, a too low threshold \(T\) gives raise to false negatives, since actual behavior changes may go undetected because the two inferred models \(M_1\) and \(M_2\) do not have enough differences.

It is thus critical to choose optimal values for parameters \(w\) and \(T\). We use the calibration procedure described in the next subsection to determine the set of candidate pairs \((w, T)\) and we resort to Pareto optimality, as described in subsection II-B to choose among the available alternatives.

A. Calibration procedure

The calibration procedure for the parameters \(w\) and \(T\) is run on an execution log that is known to contain no model discontinuity. At increasing window size, it reports the maximum threshold that produces no false positive. This value of the threshold \(T\) for a given window size \(w\) is given by the following equation:
The similarity metric used in equation (1) depends on the model inference technique being used. When state abstraction is used, it is possible to match the abstract states in two different finite state machines $M_1$ and $M_2$, hence similarity can be simply defined as a Jaccard similarity coefficient, for example computed on the model transitions:

$$\text{sim}(M_1, M_2) = \frac{|\text{Trans}(M_1) \cap \text{Trans}(M_2)|}{|\text{Trans}(M_1) \cup \text{Trans}(M_2)|}$$

When state matching is not possible, e.g., because event sequence abstraction is used, we resort to metrics derived from precision and recall. For instance, we can measure precision and recall of $M_1$ and $M_2$ using the procedure described by Pradel et al. [12] and take the harmonic mean of the two (i.e., the so called F-measure):

$$\text{sim}(M_1, M_2) = \frac{2 \cdot \text{prec}(M_1, M_2) \cdot \text{recall}(M_1, M_2)}{\text{prec}(M_1, M_2) + \text{recall}(M_1, M_2)}$$

Algorithm 1 Calibration procedure

**Input** $L = \{Tr_1, ..., Tr_k\}$: execution log, consisting of $k$ execution traces, known to contain no model discontinuity

**Output** $C = \{(w, T)|1 \leq w \leq S/2\}$: set of candidate pairs $(w, T)$, with $T$ the maximum threshold ensuring no false positive at window size $w$. $S = |Tr_1 + \ldots + Tr_k|$.

1. $C := \emptyset$
2. for $w := 1$ to $S/2$
3. $T := 1$
4. for each log $l \in \text{randomLogPermutations}(L)$
5. for $t := 1$ to $S - 2w$
6. $M_1 := \text{infM}(t, t + w)$
7. $M_2 := \text{infM}(t + w, t + 2w)$
8. if $\text{sim}(M_1, M_2) < T$ then
9. $T := \text{sim}(M_1, M_2)$
10. end if
11. end for
12. end for
13. add $(w, T)$ to $C$
14. end for
15. return $C$

The result obtained from the calibration procedure is a set of candidate pairs $(w, T)$ that do not give raise to false positives when used with the calibration traces $L$ and that is expected to be very sensitive to discontinuities, when used with newly collected traces, because the maximum threshold producing no false positives has been chosen. A lower value of $T$ would still give raise to no false positives, but would be less sensitive to changes. As an extreme case, $T = 0$ would produce no false positives, but would not allow the detection of any discontinuity at all.

The problem is how to select one pair $(w, T)$ from the set $C$ reported by the calibration procedure. Lower values of $w$ are associated with a lower delay in discontinuity detection, but typically this comes also with lower values for $T$, which means more false negatives become possible. Hence there are contrasting goals associated with the selection of $(w, T)$. We can deal with them by resorting to Pareto optimality, as described in the next subsection.

### B. Multi objective optimization

In the selection of the pair $(w, T)$ among those reported by the calibration procedure we have two contrasting goals:

- $G_1$: Reducing the delay necessary to detect a model discontinuity.
- $G_2$: Minimizing the false negatives (missed model discontinuities).

$G_1$ is determined by the window size $w$. Larger $w$ is associated with longer time before the window includes a portion of changed events associated with a model that differs from the previous one enough to allow change detection. So, we can assume that delay $d$ of discontinuity detection is roughly approximated by $w$, in that there is a monotonic relation between the two $(d \approx w)$.

$G_2$ is focused on false negatives (the pairs $(w, T)$ exclude false positives by construction). For small values of $w$, $T$ tends to be also small, because the window size is not sufficient to allow stabilization of the model. Instabilities produce similarities that are less than 1, which in turn reduce the value of $T$ that ensures the absence of false positives. However, small $T$ means also that the probability of false negatives is high, since only changes that result in a very
low similarity will be detected. We can thus approximate the probability of missing a true model change by \(1 - T\). If \(T\) is one, there is no false negative, but any dissimilarity will be reported as a potential model change. With \(T\) equals to zero no change is detected (similarity cannot be less than zero). Hence, to minimize the false negative rates, we want a high threshold \(T\) (low probability of missing model changes \(1 - T\)). However, a low probability of false negatives is expected to be obtained by increasing the the window size \(w\), in this case an high \(T\) is expected. This means that small delay and small probability of missing model changes are contrasting requirements.

From \(C\) we extract all non-dominated pairs \((w, 1 - T)\), i.e., all pairs such that no other pair has both higher \(w\) and \(1 - T\). The extracted, non-dominated pairs represent the Pareto front for the multi-objective optimization (minimization) of \(w\) and \(1 - T\). The Pareto plot \(P(w, 1 - T)\) can be interpreted as the optimal trade-off between model change detection delay and probability of missing a model change. The shape of the plot can be used to select the best trade-off for the application under analysis.

Given the Pareto plot \(P(w, 1 - T)\), we can choose trade-offs associated with:

1) Lower delay and higher probability of missing changes (left-up portion of the Pareto plot).
2) Intermediate delay and intermediate probability of missing changes (middle part of Pareto plot);
3) Higher delay and lower probability of missing changes (right-bottom portion of the Pareto plot).

While the shape of the Pareto front is useful in guiding the selection of an appropriate trade-off between detection delay and missed changes, empirical validation of the various alternatives is of fundamental importance to gain increased confidence on such a choice. The next section is devoted to an empirical investigation of the trade-offs available in the Pareto plot \(P(w, 1 - T)\).

III. EXPERIMENTS

We conducted some preliminary experiments to evaluate the proposed technique for its capability to avoid false positives when no discontinuity happens, i.e., when used within a single version of the application, and we assessed its accuracy in detecting actual discontinuities. In the latter case, we have run experiments in which we know in advance that two different versions of the application are used and that the execution traces belong to one version up to a given, known point in time, and then to the second. We assess the capability of our technique to accurately detect the discontinuity and we measure the detection delay.

Overall, our experiments address the following research questions:

**RQ1:** What is the false positive rate obtained when the parameters \((w, T)\) are taken from the Pareto front and no discontinuity occurs?

**RQ2:** In the presence of discontinuities, what is the false negative rate?

**RQ3:** In the presence of discontinuities, what is the delay of discontinuity detection?

To answer RQ1 we conduct a **within version experiment**, i.e., an experiment on execution traces collected from an application that is known not to change for the entire duration of event collection. Hence, no discontinuity detection (false positive) is expected to happen in such setting. To answer RQ2 and RQ3 we conduct a **between version experiment**. We resort to a known discontinuity, associated with a known change of behavior of the application, occurring when a new version is deployed. This new version is selected so as to differ substantially from the old version, which means that any model inferred from the old version is regarded as inadequate to represent the behavior of the new version of the application. We then run model inference on a sliding window moved along a concatenation of the old and of the new execution trace, to see if and when similarity goes below threshold, hence indicating the occurrence of a discontinuity.

The metrics collected for RQ1 is \(FP\) (false positives), measuring the number of times similarity goes below threshold \(T\) even though no discontinuity is in the trace. We repeat the measurement for different selections of the pair \((w, T)\) from the Pareto front. Moreover, for each given pair \((w, T)\) we repeat the experiment multiple times, with different concatenations of execution traces collected within a single version of the application (i.e., with no discontinuity). We present the results by boxplotting \(FP\).

For RQ2 we measure \(FN\) (false negatives), i.e., the number of cases when we concatenate traces from different versions of the application, but similarity never goes below threshold, such that no discontinuity is detected. The outcome of a single between version experiment is boolean: either the discontinuity is detected, or it is not. Hence, we repeat the experiment multiple times, with different execution traces collected for both versions and then concatenated together. We also repeat for alternative selections of \((w, T)\) from the Pareto front.

For RQ3 we consider the first event of the trace collected when the new version of the application was running and we measure the distance from this event of the first point in time when the similarity goes below threshold. This is the delay between the appearance of the new version of the application and the detection of a discontinuity. It is actually an overestimate of the actual detection delay, since it might happen that for some time, when the new version is deployed, no state/event sequence deviating from the old behavior occurs. A more accurate measure of the detection delay would be between the first event that deviates from the old behavior and the time when the discontinuity is detected. However, it is hard to determine such first event. In fact, this is quite subjective and dependent on the “ground truth” model that one has in mind when making
such an assessment. To avoid such problems, we make
the simplifying assumption that a discontinuity should be
detected from the first event produced by the new version of
the application. In interpreting the results, we know that the
delay measured in this way is an over estimate of the “true”
delay. If the over estimate is regarded as small enough,
the actual delay will be even more acceptable. Since the
experiment is repeated multiple times for different traces
and trace concatenations, we boxplot the delay values. The
experiment was also repeated for different selections of
\((W, T)\).

A. Case study

The object of the experiment is the Web application
Tudu\(^1\), supporting Web-based management of personal todo
lists. The user can create, remove or change her/his todo lists
and can share them with other, registered users. Moreover
she/he can manage the events associated with each todo list.
The Tudu application consists of around 12k lines of codes,
written in Java/JSP and it uses several frameworks (e.g.,
Struts, Spring, Oro, Aspectj, Log4j, Velocity, Xalan, etc.),
among which DWR, which supports Ajax programming. The
application can store its data both into an external database
(e.g., MySql) or into the embedded database Hsqldb.

In the experiment, we consider two versions of Tudu
that differ for some (minor) issues: Tudu2.0alpha and
Tudu2.0beta. The beta version, in fact, has been improved
with respect to the alpha one mainly by adding a couple of
Ajax events to the application user interface with the aim of
increasing the capability of the users of managing their todo
lists. In particular, in the experiment we focus on the event
(called menuToMe) that allows the user of Tudu2.0beta to
visualize only those items of a todo list assigned to the user
itself. This helps the users to easily identify their todo-tasks
instead of having access only to the whole set of items of a
todo list, as happens using Tudu2.0alpha.

In our experiment, we overall traced 248 executions of
these two Tudu versions (respectively 144 for Tudu2.0alpha
and 104 for Tudu2.0beta) using our tool [3], able to infer a
model from the execution of Ajax applications by abstracting
the concrete DOM of the Ajax Web pages into abstract
model states. Each trace consists of a sequence of executed
events (user-interface events, such as buttons and clicks, as
well as Ajax-events) and abstract instances of the DOM
captured before and after the event execution. The inferred
model, hence, is a Finite State Machine in which each state
represents an abstract instance of the application DOM while
each transition represents an Ajax-event of the application
pages impacting the DOM. See Marchetto et al. [3], [4] for
details about execution traces and model inference.

In the rest of this section, we describe the three main
phases of our experiment: (1) Calibration, (2) Within version
experiment, and (3) Between version experiment.

\(^1\)http://tudu.sourceforge.net

\[\text{Figure 2. The Pareto front}\]

B. Calibration

In the calibration procedure we used 62 executions of
Tudu2.0alpha randomly selected among the traced ones. Overall,
in the considered set of traces, 609 user-interface and Ajax events have been exercised (9.8 on average for each traced execution).

Figure 2 shows the Pareto front generated in the cal-
ibration considering 10 randomly generated permutations
of the 62 traces. In such a figure, we can observe that a lower threshold \(T\) corresponds to lower window size \(w\) (i.e., left-part of the Pareto plot) while a higher threshold \(T\)
corresponds to higher window size \(w\) (i.e., right-part of the
Pareto plot).

Figure 2 shows also the three points \((w, T)\) spread across
the Pareto front that we have chosen for the next phases
of our experiments: \(P_1 = (60.0, 0.21)\), \(P_2 = (138, 0.35)\),
\(P_3 = (276, 0.56)\).

C. Within version experiment

In the within version experiment, we analyzed a set
of 62 executions traces (different from those used in the
calibration) in which 596 events (9.6 on average per exec-
ution) of the Tudu2.0alpha Web pages have been traced.
Moreover, we considered 50 permutations of such traces.
For each permutation we first computed the \(FP\) (i.e., false
positive) for the three selected pairs \((w, T)\), i.e., \(P_1\), \(P_2\),
\(P_3\). Then, to answer \(RQ_1\), we computed the \(FPrate\) (i.e.,
percentage of false positive in the whole set of considered
trace permutations) and we boxploted such values.

We obtained that \(FPrate(P_1) = 2.3\%\), \(FPrate(P_2) =
13.7\%\) and \(FPrate(P_3) = 100\%.\) Figure 3 shows the
boxplots for the three pairs \((w, T)\). These boxplots show the
frequency of \(FP\) and \(TN\) (true negative, i.e., those points in
which the technique correctly does not detect the presence of any discontinuities) in the sliding windows analyzed in the traces for each considered point \((w, T)\). Observing both the boxplots and the \(FP\) rate measured for the tree points \(P_1\), \(P_2\) and \(P_3\), we can see that the number of false positive tends to be smaller when considering points with small window size and small threshold, e.g., \(P_1\). The number of false positive tends to increase when considering points with higher window size and threshold, e.g., \(P_3\). Therefore, we can conclude that lower window size and threshold are better to reduce the occurrence of false positives.

**D. Between version experiment**

In the between version experiment, we considered a set of other 124 executions traces (i.e., 20 of Tudu2.0alpha and 104 of Tudu2.0beta) and we separated them into two sets \(S_1\) and \(S_2\) composed of 62 traces each. Both \(S_1\) and \(S_2\) are composed of 10 traces of Tudu2.0alpha and 52 traces of Tudu2.0beta. In detail, in the 52 traces of \(S_1\) from Tudu2.0beta, 477 events have been traced and 31 of them are related to the new event \(menuToMe\) \((density_{menuToMe}(S_1) = 6.5\%)\). In \(S_2\), 488 events have been traced and 10 of them are related to the new event \(menuToMe\) \((density_{menuToMe}(S_2) = 2.4\%)\). For both sets \(S_1\) and \(S_2\) we considered 50 permutations of their 62 traces and, for each permutation, we computed the \(FN\) (i.e., false negative) and, in the presence of detected discontinuities, we also computed the \(DD\) (i.e., delay of discontinuity detection).

For answering \(RQ2\), we analyzed the considered permutations of \(S_1\) and \(S_2\) by firstly computing the \(FNrate\) (i.e., percentage of false negative in the whole set of considered trace permutations) for \(P_1\), \(P_2\), \(P_3\), and then boxplotted the \(FN\) values. Table I shows the \(FNrate\) measured for the three points \(P_1\), \(P_2\), \(P_3\), both for \(S_1\) and \(S_2\). Figure 4 and
Figure 5 show the boxplots for S1 and S2 respectively. Such boxplots show the frequency of FN (false negatives) and TP (true positives, occurring when actual discontinuities have been correctly detected) obtained in the analysis of the traces for each point.

Observing both the FN rate obtained for S1 and S2 and the boxplots, we can see that the number of false negatives is higher when considering small window size and threshold, e.g., P1, but it decreases when increasing the values of window size and threshold, e.g., for P3. Considering this result and the finding of the Within version experiment we can conclude that increasing the value of the considered point \((w, T)\) we increase also the possibility of detecting actual needs for model update (i.e., discontinuity points), but we increase also the rate of the false positives.

One relevant difference in the results obtained for the two sets of traces S1 and S2 concerns their different density of the new event added to Tudu2.0beta. This can be observed by considering the FN rate of, e.g., P2 in the two sets: FN rate S1 (P2) = 5% and FN rate S2 (P2) = 95%. The new event density strongly impacts the results of the technique and also the selection of the point \((w, T)\) to be used to analyze new traces. In fact, with a lower density of new events in the traces, the points on the left-part of the Pareto front are not quite good at detecting real discontinuities. This is due to the low value of the threshold, which works effectively only if major discontinuities happen (i.e., at high density of new events). This result is partially confirmed by the FN rate measured for P3: FN rate S1 (P3) = 0% and FN rate S2 (P3) = 30%. Similarly to the Within version experiment, small variations of the traces are not detected as discontinuities when using points \((w, T)\) from the left-part of the Pareto front.

For the considered permutations of S1 and S2, we also computed the avgDD (i.e., average of the delay of discontinuity detection on the whole set of trace permutations) for each pair \((w, T)\) and then we plotted it to answer RQ3. Table I shows the avgDD obtained, while Figure 6 and Figure 7 show the boxplots of DD for S1 and S2 respectively (the latter does not include any boxplot for P1 since on S2 the associated false negative rate is 100%, i.e., the discontinuity is always missed).

Observing both the avgDD values and the boxplots, we can conclude that with high \((w, T)\) the delay in detecting discontinuity is on average very low. With small \((w, T)\) the delay is reasonable low. When we consider an intermediate value of \((w, T)\) the delay can be high. We can argue that the ability of detecting a discontinuity early is strongly dependent on the threshold \(T\). In fact, high \((w, T)\) can detect any small variation in the traces, while smaller \((w, T)\) can detect only large changes. The larger window size associated with a higher threshold is not detrimental to the ability of reacting quickly to the discontinuities, while a lower threshold has a major negative impact, despite the lower windows size, which makes model inference more sensitive to changes.

E. Overall Considerations

The results obtained in the experiments, in summary, confirm that the capability of detecting needs for model update depends on the position of the considered pair \((w, T)\) in the Pareto front. In particular, (RQ1) the rate of false positives of a point \((w, T)\) tends to increase at
increasing \((w, T)\), while \((RQ2)\) the rate of false negatives of a point \((w, T)\) tends to decrease at increasing \((w, T)\). The experiment, however, shows that the rate of false negatives depends also on additional factors such as the amount of the trace variability (i.e., density of changes in the new traces). Another finding of the experiment concerns the delay of discontinuity detection \((RQ3)\). A high delay occurs when the point \((w, T)\) is selected in the middle part of the Pareto front. The reason is that with high values of \((w, T)\) any small variation in the traces is reported as a discontinuity, hence, actual discontinuity can be detected quickly, if they exist. At low values of \((w, T)\) the false negative rate can be as high as 100\% (corresponding to an infinite delay). Hence, we conclude that intermediate points in the Pareto front are better, because they offer a reasonable trade-off between false positives and false negatives at reasonable delay.

Several threats affect the validity of the obtained results. For instance, it is difficult to generalize the results since we analyzed only a couple of versions of the same application and the used set of traces is quite small. However, this preliminary experimentation shows the potentiality of our technique in detecting actual needs for model update. Further experimentation will be devoted to refine the technique starting from these preliminary findings.

IV. CONCLUSION AND FUTURE WORK

Automated detection of model discontinuities is fundamental when model inference techniques, used for testing, anomaly detection or program comprehension, are applied to dynamically changing, self-adaptive applications. We proposed a novel approach to detect such discontinuities based on the computation of the similarity between recent and past models inferred from the execution log. We resort to multi-objective optimization to select the best trade off between the two key parameters of our approach, window size, related to the delay of discontinuity detection, and similarity threshold, affecting the rate of false negatives (i.e., missed model changes). Our preliminary experimentation with the proposed approach indicates that it is possible to achieve high accuracy, in terms of reduced false positives and negatives, at reasonably low delay, i.e., with a quite fast reaction to changes.

Future work will be devoted to further experimentation on larger applications that produce execution logs, associated with larger and more complex inferred models. We will also evaluate the effectiveness of the approach with respect to the final usage of the inferred model (e.g., testing), by measuring the limited usefulness of the old model compared to the benefits achieved by inferring a new model when a discontinuity is detected. Further experimentation with model inference techniques that are based on event sequence abstraction is also needed for a more complete assessment of the proposed approach.

ACKNOWLEDGMENT

This work has been partially funded by the European Union FP7 project FITTEST (grant agreement n. 257574).

REFERENCES