Suggesting Edits to Explain Failing Traces

Giles Reger

University of Manchester, UK

Abstract. Runtime verification involves checking whether an execution trace produced by a running system satisfies a specification. However, a simple 'yes' or 'no' answer may not be sufficient; often we need to understand why a violation occurs. This paper considers how computing the edit-distance between a trace and a specification can explain violations by suggesting correcting edits to the trace. By including information about the code location producing events in the trace, this method can highlight sources of bugs and suggest potential fixes.

1 Introduction

Runtime verification [4] is the process of checking whether an execution trace τ produced by a running system satisfies a specification φ. This means checking $\tau \in \mathcal{L}(\varphi)$ where $\mathcal{L}(\varphi)$ is the *language* of the specification. This paper considers traces as finite sequences of propositional symbols and specifications as finite state automata but the ideas could be transferable to both the parametric case, and cases where the specification can be translated to automata i.e. LTL. In the case of automata the check $\tau \in \mathcal{L}(\varphi)$ is well understood; however, if the answer is 'no' it does not reflect how close the trace is nor give any information about *why* the trace violations the specification.

The aim of this paper is to highlight and encourage the use of the *edit-distance* between a trace and the specification to *measure* the degree to which the trace satisfies the specification. If the trace is failing this approach can also suggest *edits* to the trace that could explain the violation and how to fix it. To motivate this approach consider the following trace of $open$ and $close$ events extracted from different parts of a system.

$$\underbrace{open.close.open.}_{\text{A.java}}\underbrace{open.close.}_{\text{B.java}}\underbrace{open.close.open.}_{\text{A.java}}\underbrace{open.close}_{\text{C.java}}$$

This violates the specification captured by the regular expression[1] $(open.close)^*$. There are different ways we can *edit* this trace to make it satisfy the specification. For example,

$$open.close.\cancel{open}.open.close.open.close.\cancel{open}.open.close$$
$$open.close.open.\cancel{open}.close.open.close.\cancel{open}.open.close$$
$$open.close.open. \overset{close}{\downarrow} .open.close.open.close.open. \overset{close}{\downarrow} .open.close$$
$$\ldots$$

Producing these edits can suggest ways in which the original system could be modified. Furthermore, if we can label the trace with the source of events we can use this information to suggest which sets of edits might be the most sensible. In this case we should prefer *consistently* editing the file A.java as it would involve the smallest change.

[1] For conciseness we will use regular expressions to represent the corresponding automaton

Some Related Work. Distance metrics have been considered before in the area of fault localization. For example, in [8] the distance is measured between a faulty run and a large set of correct runs to select a correct run similar to the faulty run, which is used to report suspicious parts of a program. Other similar approaches exist.

The ideas in this paper are similar to those presented in [3] in the context of using an FSA inferred from correct program runs to detect anomalous behaviours. A failing trace is used to *edit* the inferred FSA to give an interpretation of the failure.

In runtime verification it is common to simply report the event on which the first error occurred. Approaches for parametric runtime monitoring (i.e. JAVAMOP [5], MARQ [7]) can report on errors per set of parameters. Approaches that *restart* a monitor on failure effectively edit the trace but not necessarily in the most appropriate way.

Structure. Sec. 2 describes an existing method for computing the edit-distance between a trace and the language of an automaton. Sec. 3 introduces a method for suggesting sensible edits. Sec. 4 describes an experiment and Sec. 5 concludes.

2 Computing the Edit Distance

This section gives a brief overview of an existing method for computing the edit distance between a trace and an automaton's language. This was previously described in [6], which uses the approach in [2]. The general idea is to model the trace, automaton and edit operations as weighted transducers and compute their composition. Paths through the composed transducer to a final state represent different ways the trace can be edited to fit the automaton's language and the shortest path gives the edit distance.

Edit distance. The edit (or Levenshtein) distance between two traces is the minimal number of *edits* required to transform one trace into the other. This distance, using the standard edit operations of addition, removal and replacement, is defined as follows:

Definition 1 (Levenshtein distance). *The Levenshtein distance between traces τ_1 and τ_2 is* $\mathsf{distance}(\tau_1, \tau_2)$*, defined as*

$$\begin{array}{l}\mathsf{distance}(\tau_1, \epsilon) = |\tau_1| \\ \mathsf{distance}(\epsilon, \tau_2) = |\tau_2|\end{array} \qquad \mathsf{distance}(a\tau_1, b\tau_2) = \mathsf{min} \begin{cases} \mathsf{distance}(\tau_1, b\tau_2) + 1 & \\ \mathsf{distance}(a\tau_1, \tau_2) + 1 & \\ \mathsf{distance}(\tau_1, \tau_2) + 1 & \textit{if } a \neq b \\ \mathsf{distance}(\tau_1, \tau_2) & \textit{if } a = b \end{cases}$$

i.e. we repeatedly choose the edit (removing a, adding b, replacing a by b) or non-edit that leads to the fewest edits overall. The edit distance between a trace τ and an automaton φ is the smallest distance between τ and a trace in the language of φ:

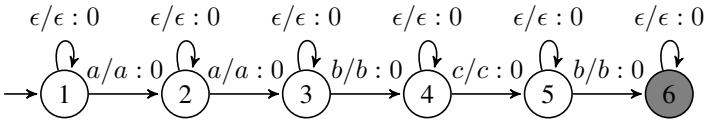
$$\mathtt{distance}(\tau, \varphi) = \mathsf{min}(\{\mathsf{distance}(\tau, \tau') \mid \tau' \in \mathcal{L}(\varphi)\})$$

Weighted Transducers. A weighted transducer (see [2]) has transitions labeled with input and output symbols and a weight (for this work this is 0 or 1). Input and output symbols can be ϵ i.e. can be taken without consuming or producing a symbol.

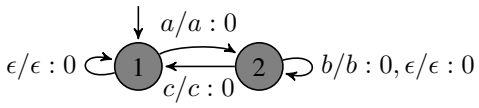
The composition $T \circ X$ of two transducers T and X considers all possible sequencing between strings of T and strings of X i.e. if $a/b.a/c$ is a string of T and $b/d.c/a$ is a string of X then $a/d.a/a$ is a string of $T \circ X$. Here we consider a three-way composition i.e. $T \circ X \circ P$. We compute this as a single operation for efficiency reasons - if we computed $T \circ X$ and then $(T \circ X) \circ P$ it is likely that $(T \circ X)$ would contain many superfluous transitions. An algorithm for doing this is presented in [1].

Computing the Edit Distance Using Weighted Transducers. This section briefly describes how to construct the three weighted transducers representing the trace, the edits and the automaton and how their composition is used to find the edit distance.

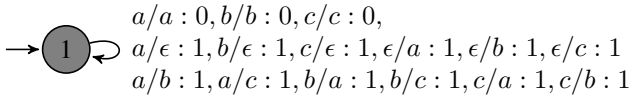
Traces are translated into weighted transducers by turning each event into a 0-weighted transition between states with ϵ self-looping transitions with the last state being final. The trace $a.a.b.c.b$ would become the following weighted transducer:



An automaton is translated into a weighted transducer by a) making each transition 0-weighted using the event as input and output, and b) adding a 0-weight self-looping ϵ transition to each state. The automaton for the regular expression $(ab^*c)^*$ would be:



The edit transducer has a single state with looping transitions for each of the edit operations it can perform - given below for alphabet $\{a, b, c\}$. Note how ϵ is used to model deletions and additions and all edit operations have a weight of 1.



The edit distance is computed using the composition $T \circ X \circ P$ where T, X and P are weighted transducers for the trace, edits and automaton respectively. Every path through this composition represents a way in which the trace T can be rewritten to a path of P using the edits in X. The edit-distance is given by the shortest path. Furthermore, a path describes *how* the trace can be edited. See [6] for an example.

Performance and Online Monitoring. In [1] it is shown that the three-way composition $T \circ X \circ P$ can be computed in time $O(|X||T|)$. This special method prevents a large intermediate result from dominating the computation. In [2] it is shown that the edit-distance between a string and automaton can be found in linear-space.

This approach assumes that the whole trace trace is available, which is reasonable for log file analysis but not for an online application of runtime monitoring. It may be possible to lazily compute the composition $T \circ X \circ P$ and perform the shortest path search in tandem online. This would involve keeping track of the states reachable in the composition and pruning this set (perhaps heuristically) to ensure it does not become unmanageable. Note that the composition represents a rewriting of the trace such that edits typically take the composition a finite number of steps away from the trace before returning to it. Further investigation is required.

It is clear that we cannot use any online approach that requires the trace to be stored as the space requirements would be impractical. As the composition itself is quadratic in the size of the trace this should also not be stored. Methods for compacting the trace online could be explored - this is briefly suggested in Sec. 4.

3 Suggesting Sensible Edits

The previous section described a technique that gives a set of minimal edit sequences that 'correct' a failing trace. This section explores how to produce *sensible* edits where:

1. Edits are consistently applied
2. Repeating the same edit is preferred to multiple separate edits

Here there is a notion of an *edit* as something more than altering the trace. To capture this we *label* the trace with the source of each event i.e. the source file and line number. In this work we only consider *consistent* labelled traces were each label is associated with a single event i.e. the source of events is deterministic Now a removal of event a at label l_1 is distinct from the removal of a at label l_2.

The composition construction in the previous section can be lifted to *labelled events* by ignoring the labels. The labels will be used to define a notion of a *sensible* edit path. An *edit path* is a finite sequence of edit records $((\langle a_1, l\rangle, a_2, w)$ starting (ending) in an initial (accepting) state of the composition where $\langle a_1, l\rangle$ is a labelled event from the trace and a_2 is an (edited) event from the automaton (possibly ϵ). The *path cost* of an edit path can be defined such that there is no cost to repeat an edit:

Definition 2 (Path cost). *The* cost *of an edit path τ is given as* $\mathsf{cost}(\tau, \{\})$ *defined as*

$$\begin{aligned}\mathsf{cost}(\epsilon, S) &= 0\\ \mathsf{cost}((((\langle a_1, l_1\rangle, a_2, w).\tau, S) &= \mathsf{cost}(\tau, S + (a_1/a_2, l_1)) + \begin{array}{ll} w & \textit{if } (a_1/a_2, l_1) \notin S\\ 0 & \textit{if } (a_1/a_2, l_1) \in S\end{array}\end{aligned}$$

A *sensible* edit path is one that preserves *consistency* and has a minimal path cost.

Finding Sensible Edit Paths. Firstly, it is possible to put an upper bound on minimal edit path cost using a modified version of Djkistra's algorithm; this is only an upper bound as the edit path cost is dependent on the whole path not just the state reached. To identify sensible edit paths we perform the following non-deterministic heuristic search of the composition structure. The search maintains a set S of previously performed

edits that is used to ensure consistency. The search is based around the idea that for a successful trace there will be a 0-weighted path through the composition and every failure can be seen as a blockage of this path. When a blockage is met we perform a local search to find the continuation of the trace. The steps taken by the search are:

1. Follow the 0-weighted path until there are no 0-weight transitions. If a transition matching an edit in S is found it must be taken to preserve consistency even if this diverges from the 0-weighted path.
2. Choose a (short) path p to the closest state with a 0-weight transition. The path p must be consistent with S and all edits in p should be added to S.
3. Repeat steps 1 and 2 until the final state is reached

Practically we perform the search in a breadth-first manner, keeping track of all possible choices. These can be pruned either by a max value given either by the approximated minimal edit path cost computed as described above or some other heuristic.

Paths found by this search are consistent by construction. Finding all paths of minimal path cost depends on how the path is chosen in step 2. An edit path of minimal path cost may be missed if a smaller set of edits are chosen too early, forcing a larger set to be used later. Note that if the search completes it will find a path of minimal path cost.

Performance. The complexity of this approach is exponential in the number of choices made. However, every choice restricts further choices as consistency must be preserved i.e. a path p cannot make a different edit at a label where an edit has already been made. Nevertheless, it is vital that paths are trimmed. We use the strategy of searching with $max = 0$ first (for the case where there are no failures) and increasing this by 1 until a path is found. Further heuristics for pruning paths should be investigated.

Online Monitoring. As discussed in the previous section, unless the composition can be created and searched lazily this approach will not be applicable to online monitoring. In [6] an approach that edited the trace as it was produced was explored. This was found to be very expensive as many different edit variations had to be tracked. Further work could apply the notion of *sensible path* to filter these edits.

4 A Scalability Experiment

The techniques presented here have been implemented in `Scala` and are available at `https://github.com/selig/RVsuggestEdit`. We use the *resource usage* property (ab^*c) to explore the scalability of the approach presented here. As a test trace we use the example trace given in Sec. 1 with `use` events inserted in-between the `open` and `close` events making the trace 100 events long. Let $t(m)$ be the test trace followed by m good events and let $t(m, n)$ be n repetitions of $t(m)$. We vary m and n to produce larger traces; the same edit paths are identified in each case. The results are as follows:

m	n	len	errs	Time (s)	m	n	len	errs	Time (s)	m	n	len	errs	Time (s)
0	1	100	4	0.005	900	1	1k	4	0.4	2.4k	1	2.5k	4	2.3
0	3	300	12	0.038	900	5	3k	12	3.3	2.4k	3	7.5k	12	20.4
0	6	600	24	0.144	900	10	6k	24	13.1	2.4k	6	15k	24	82.2

This demonstrates that this approach has the ability to scale but could certainly benefit from improvements. Two possible extensions would be a) collapsing repeated sections of the trace, represented by larger weights in T; and b) collapsing prefixes in the search to tame the exponential branching at each error. This experiment tentatively argues for the viability of the approach but further work with real software systems is needed to support the claim that this is a useful method for understanding failing traces.

5 Conclusion and Related Work

This paper has discussed a method for *measuring* the extent of failure in a trace and *suggesting* possible changes to the trace that could 'fix' it. The approach uses traces labelled with the points that generate events to suggest *sensible* edits. This was inspired by our earlier work using edit-distance in pattern-based specification mining [6].

The final point to be addressed is whether these edit paths are useful in explaining failing traces as the title of this short paper suggests. The motivation here is that by connecting the edit paths to program points via labels (and ensuring the edits are sensible) the edit paths correspond to edits to the program such as the deletion, insertion or replacement of a method call. Describing failures in this way is certainly not a new notion in the area of fault localisation.

Irrespective of their use for explaining failing traces, the author would suggest that using distance metrics is a reasonable direction for detecting *multiple errors* in a trace; an activity not often studied in the runtime verification community.

This is a preliminary investigation into this approach and much can be done to extend the ideas and make them more applicable to runtime verification problems.

References

1. C. Allauzen and M. Mohri. 3-way composition of weighted finite-state transducers. In *Proceedings of the 13th international conference on Implementation and Applications of Automata*, CIAA '08, pp. 262–273. Springer-Verlag, 2008.
2. C. Allauzen and M. Mohri. Linear-space computation of the edit-distance between a string and a finite automaton. *CoRR*, abs/0904.4686, 2009.
3. A. Babenko, L. Mariani, and F. Pastore. Ava: Automated interpretation of dynamically detected anomalies. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pp. 237–248. ACM, 2009.
4. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear*. IOS Press, 2013.
5. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the mop runtime verification framework. *J Software Tools for Technology Transfer*, pp. 1–41, 2011.
6. G. Reger, H. Barringer, and D. Rydeheard. Automata-based pattern mining from imperfect traces. In *The 2nd International Conference on Software Mining*, 2013.
7. G. Reger, H. C. Cruz, and D. Rydeheard. MARQ: monitoring at runtime with QEA. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, 2015.
8. M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, pp. 30–39, 2003.