Mining Precise Performance-Aware Behavioral Models from Existing Instrumentation

Tony Ohmann[a] Kevin Thai[b] Ivan Beschastnikh[c] Yuriy Brun[a]

[a]University of Massachusetts
Amherst, MA, USA
{ohmann, brun}@cs.umass.edu

[b]Facebook Inc.
Seattle, WA, USA
kevinthai@fb.com

[c]University of British Columbia
Vancouver, BC, Canada
bestchai@cs.ubc.ca

ABSTRACT

Software bugs often arise from differences between what developers envision their system does and what that system actually does. When faced with such conceptual inconsistencies, debugging can be very difficult. Inferring and presenting developers with accurate behavioral models of the system implementation can help developers reconcile their view of the system with reality and improve system quality.

We present Perfume, a model-inference algorithm that improves on the state of the art by using performance information to differentiate otherwise similar-appearing executions and to remove false positives from the inferred models. Perfume uses a system's runtime execution logs to infer a concise, precise, and predictive finite state machine model that describes both observed executions and executions that have not been observed but that the system can likely generate. Perfume guides the model inference process by mining temporal performance-constrained properties from the logs, ensuring precision of the model's predictions. We describe the model inference process and demonstrate how it improves precision over the state of the art.

Categories and Subject Descriptors: D.4.8 [Performance]: Modeling and prediction
General Terms: Algorithms, Design, Modeling
Keywords: Model inference, log analysis, performance, Perfume

1. INTRODUCTION

Software developers spend half their time looking for and fixing bugs [5,21] with the global annual cost of debugging topping $300 billion [5]. Mature software projects often ship with known defects [14], and even security-critical bugs remain unaddressed for long periods of time [11].

One significant cause of bugs is differences between what the developers envision their system does, what the developers understand the specification to mean, and what the system actually does [7]. To increase understanding, developers instrument key locations in the code and use runtime logging to peek into an implementation's behavior. Logging system behavior is one of the most ubiquitous, simple, and effective debugging tools. Logging is so important that production systems at companies like Google are instrumented to generate billions of log events each day. These events are stored for weeks to help diagnose bugs [22].

These logs are often incredibly rich, or trivial to make rich, with information about which events executed when and in what context, as well as with performance details such as resource use and timing. But the same richness that makes logs potentially useful also makes them complex, verbose, and difficult to understand. Further, logs contain linear series of events that represent individual executions, and it is difficult to use them to understand the aggregate system behavior. This paper focuses on improving the understanding of whole-system performance behavior at scale.

Dynamic behavioral specification mining, e.g., [3,4,16,18], tackles this problem by inferring behavioral models that summarize all observed executions in a concise form. Such models have been used to improve developers' understanding of system implementations and to find, diagnose, and remove bugs [3]. While state-of-the-art model inference algorithms rely on event names, message types, and sometimes data values stored in the logs, they ignore other rich information that makes logs so useful. We posit that by including more execution information available in system logs, the precision of model inference algorithms can improve, and the utility of the inferred models can increase.

We propose **Perfume**, a novel model inference algorithm that extends Synoptic [3] in a principled manner to account for performance information often available in system logs. Our focus is on execution timing information, although the algorithm we present extends trivially to resource use, such as memory utilization. Perfume mines temporal properties with performance constraints from the log and uses these properties to identify and remove imprecise generalizations in the Synoptic process.

Perfume's models can therefore more precisely describe the behavior of a system. In particular, system optimizations such as the use of caches, lazy evaluation, and loop perforation [19] both affect system performance and can cause bugs, which would be concealed in models that ignore performance information. Perfume models account for key differences in performance between observed executions both to improve model precision and to enable better prediction of unobserved executions. This is achieved by ensuring the models satisfy performance-constrained temporal properties mined from the observed executions. Further, it is achieved generally using existing runtime logs and requires access to neither the source code nor binaries of the modeled system.

Next, in Section 2, we illustrate why including performance information in a behavioral model is useful. We show that Perfume-generated models can be more insightful than those produced with

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the author/owner(s). Publication rights licensed to ACM.

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2768-8/14/05
http://dx.doi.org/10.1145/2591062.2591107

model-inference approaches that do not account for performance. Then, we detail the Perfume algorithm in Section 3 and discuss related research in Section 4. In Section 5, we summarize our contributions and describe the potential benefits of Perfume-inferred models in debugging and automated test generation.

2. WHY MODEL PERFORMANCE?

Consider a network diagnosis tool for identifying problematic client network paths. The tool first determines if the client is using narrowband or broadband and then runs a series of queries. Based on the speed and characteristics of the client's responses to the queries, the tool classifies the network path as `OK` or `problematic`.

The tool's developer wants to know what factors cause the tool to report client paths as problematic. Runtime logs of the tool, shown in Figure 1(a), can help answer this question, but the information is hard to infer manually. Instead, model-inference tools can summarize the log. Figures 1(b) and (c) depict models inferred using two well-known algorithms, kTails [4] and Synoptic [3], respectively. The kTails model differentiates execution paths of broadband and narrowband clients, but it contains no indication of the types of executions that suggest network problems because all paths pass through the common bottom node. Unlike the kTails model, the Synoptic model correctly conveys that no network problems are reported for narrowband clients. However, it does not help the developer further differentiate between those broadband clients who experienced a network problem and those who did not.

What the developer really wants to see is a model that reveals what types of executions imply a network problem. Our proposed approach, Perfume, infers the model shown in Figure 1(d) that exposes this information. This model is still predictive, and it still differentiates execution paths of broadband and narrowband users, but it also separates the sub-path `broadband` → `query` → `query` → `problem` from the sub-path `broadband` → `query` → `query` → `OK` based on the performance of the second query. The former sub-path reveals that the tool reports a network problem when a broadband client responds slowly to the second data query.

Note that simply adding performance information to the edges on the kTails and Synoptic models would not help identify what leads the tool to report a problem. The Synoptic model would predict that both slow and fast responses on broadband can lead to a problem. Meanwhile the kTails model would predict that all combinations of response speeds and bandwidths can lead to a problem. By contrast, the Perfume model not only displays the performance information, but is also more precise in its predictions, which leads to a more accurate differentiation between executions.

3. PERFORMANCE-AWARE MODEL INFERENCE

Perfume's goal is to produce a representative model of a system from an execution log containing examples of that system's behavior. In doing so, Perfume aims to improve on the state-of-the-art model-inference techniques by utilizing performance information in the log to guide the inference process and more accurately predict which unobserved executions are likely possible. Additionally, Perfume's goal is to be applicable to a wide range of systems and to produce concise and descriptive models. Figure 1(d) illustrates how Perfume can be used to help understand complex behavior.

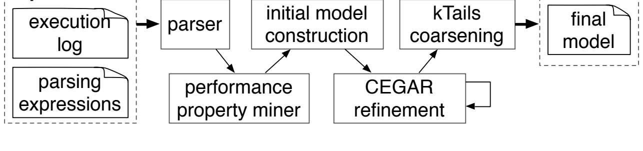


Figure 2: The Perfume model-inference process.

To infer descriptive but precise models, Perfume strives to produce concise models that are still consistent with constrained temporal properties observed in the logged executions. Figure 2 summarizes the high-level process Perfume uses to infer models, which extends the existing Synoptic algorithm [3]. Synoptic models do not enforce constrained temporal properties, and thus they are likely more concise but less precise than Perfume models. Still, Synoptic models have been shown to enhance developers' understanding of their systems and to help developers find bugs [3], which suggests that Perfume's more precise models may be similarly useful.

The rest of this section details the Perfume inference process.

3.1 Log Parsing

Perfume operates on system execution logs and requires access to neither system source code nor binaries, making Perfume broadly applicable to a wide range of systems. Perfume has two inputs: the system log and a set of regular expressions for parsing from the log the individual execution *traces*, *events* within those traces, and *timestamps* on the events. In the example log in Figure 1(a), a trace is a session for one IP address, and event instances are server actions, such as `query`. The parsing expressions for that log are `\k⟨ip⟩` for the traces and `(?⟨ip⟩) .+:(?⟨DTIME⟩.+)\] "GET HTTP/1.1 /(?⟨TYPE⟩.+)"` for the events and timestamps.

3.2 Property Mining

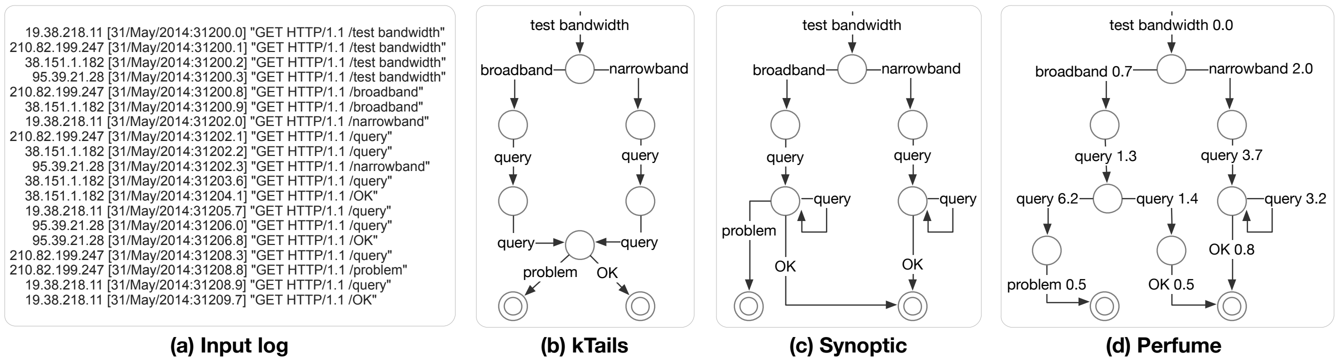
Perfume parses the log and mines five types of temporal, performance-constrained properties that hold for every observed trace in the log. Using variables `a` and `b` to describe possible event types and `t` to describe times, the five temporal property types are:

- If whenever `a` is present in a trace, `b` is also present later in the same trace in no more than `t` time, we say "`a` always followed by `b` upper-bound `t`".
- Similarly, if whenever `a` is present in a trace, `b` is also present later in the same trace in no less than `t` time, we say "`a` always followed by `b` lower-bound `t`".
- If whenever `b` is present in a trace, `a` is also present earlier in the same trace in no more than `t` time, we say "`a` always precedes `b` upper-bound `t`".
- Similarly, if whenever `b` is present in a trace, `a` is also present earlier in the same trace in no less than `t` time, we say "`a` always precedes `b` lower-bound `t`".
- Finally, if whenever `a` is present in a trace, `b` is never present later in the same trace, we say "`a` never followed by `b`".

More formally, we have defined each of these properties with timed propositional temporal logic (TPTL).[1]

These five properties capture the behavioral differences between the observed executions and executions the system likely cannot produce. When Perfume infers a model, it ensures that all executions predicted by the model adhere to the mined properties. Note that while there are only five property types, there can be many more instances of these types; the number of instances typically depends on the number of different event types the system can produce. The five property types are templates for constraints on the possible behavior of the system, with the first four property types also encoding the system's performance characteristics. For the

[1]For example, the "`a` always followed by `b` upper-bound `t`" property is represented with TPTL as $\Box x.(a \rightarrow (\Diamond y.(b \wedge y - x \leq t)))$



```
   19.38.218.11 [31/May/2014:31200.0] "GET HTTP/1.1 /test bandwidth"
210.82.199.247 [31/May/2014:31200.1] "GET HTTP/1.1 /test bandwidth"
   38.151.1.182 [31/May/2014:31200.2] "GET HTTP/1.1 /test bandwidth"
    95.39.21.28 [31/May/2014:31200.3] "GET HTTP/1.1 /test bandwidth"
210.82.199.247 [31/May/2014:31200.8] "GET HTTP/1.1 /broadband"
   38.151.1.182 [31/May/2014:31200.9] "GET HTTP/1.1 /broadband"
   19.38.218.11 [31/May/2014:31202.0] "GET HTTP/1.1 /narrowband"
210.82.199.247 [31/May/2014:31202.1] "GET HTTP/1.1 /query"
   38.151.1.182 [31/May/2014:31202.2] "GET HTTP/1.1 /query"
    95.39.21.28 [31/May/2014:31202.3] "GET HTTP/1.1 /narrowband"
   38.151.1.182 [31/May/2014:31203.6] "GET HTTP/1.1 /query"
   38.151.1.182 [31/May/2014:31204.1] "GET HTTP/1.1 /OK"
   19.38.218.11 [31/May/2014:31205.7] "GET HTTP/1.1 /query"
    95.39.21.28 [31/May/2014:31206.0] "GET HTTP/1.1 /query"
    95.39.21.28 [31/May/2014:31206.8] "GET HTTP/1.1 /OK"
210.82.199.247 [31/May/2014:31208.3] "GET HTTP/1.1 /query"
210.82.199.247 [31/May/2014:31208.8] "GET HTTP/1.1 /problem"
   19.38.218.11 [31/May/2014:31208.9] "GET HTTP/1.1 /query"
   19.38.218.11 [31/May/2014:31209.7] "GET HTTP/1.1 /OK"
```

Figure 1: (a) A sample network diagnosis tool's log of four execution traces, and models inferred by the (b) kTails algorithm with $k = 2$, (c) Synoptic, and (d) Perfume on that log.

upper-bound constraints, `t` is the maximum timestamp difference, for all traces, between the first event `a` and the last event `b` in each trace. For the lower-bound constraints, `t` is the minimum timestamp difference between any event `a` and any event `b` in each trace.

Perfume's properties extend Synoptic's properties [3] with performance data to capture behavior of the system more precisely. For example, from the log in Figure 1(a), one of the properties Perfume mines is "`broadband` always precedes `problem` lower-bound 8.7 seconds". This property helps explain the system's behavior because it differentiates the `query` events after `broadband` that lead to `problem` from those that lead to `OK`. This reveals that network problems are reported after a fast query followed by a slow query, whereas no problems are reported after two fast queries. Section 3.3.1 explains how Perfume uses these temporal properties to derive a model that separates paths with distinct behavior.

3.3 Model Construction

To construct a model, Perfume first builds the most concise model it can — all events of the same type lead to the same state. While concise, this *initial* model is imprecise because it predicts many executions that do not satisfy the mined temporal performance-constrained properties. Thus, Perfume iteratively *refines* the initial model to satisfy these properties. Section 3.3.1 explains the refinement process. Perfume's task is NP-hard [6], so it approximates a solution and may at times make suboptimal refinements. To partially correct these suboptimalities, once Perfume's refinement reaches a model that satisfies all mined properties, it *coarsens* the model where possible without introducing property violations. Section 3.3.2 explains the coarsening process. Finally, Section 3.3.3 summarizes important properties of the final Perfume model.

3.3.1 Refinement

The goal of this phase of the algorithm is to refine a model that violates some of the mined properties into a concise version that satisfies all of those properties. Creating such a model that is optimally concise is NP-hard [6]. Like prior work [3,4,16,18], Perfume finds an approximate solution.

Perfume iteratively performs counterexample guided abstraction refinement (CEGAR) [6] until the model satisfies all mined properties. In each iteration, Perfume uses model checking to identify a predicted path in the model that violates a mined property[2]. Using this identified *counterexample* path, Perfume splits states in the model to eliminate the path from the model. Then, Perfume iterates to find another violation of this or another property and further refines the model until all properties are satisfied.

[2]The model checking does not differentiate between observed and predicted paths, but observed paths cannot violate the properties mined from the observed paths themselves.

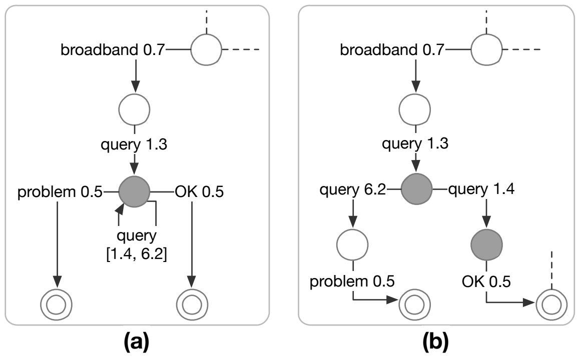


Figure 3: An example of a Perfume refinement step, part of the process for deriving the model in Figure 1(d). The partial model in (a) does not satisfy the mined property "`broadband` always precedes `problem` lower-bound 8.7 seconds." Perfume refines the shaded state in this model into the two shaded states in (b). The refined model in (b) satisfies the mined property.

Figure 3 shows one example refinement iteration. The partial model in Figure 3(a) does not satisfy the mined temporal property "`broadband` always precedes `problem` lower-bound 8.7 seconds." Perfume finds the counter-example path that contains the sub-path "`broadband` 0.7 $\rightarrow$ `query` 1.3 $\rightarrow$ `problem` 0.5" and refines the model to eliminate this path by splitting the shaded state into the two shaded states in Figure 3(b). The resulting model satisfies the mined property.

Perfume's refinement is guaranteed to produce a model that satisfies all of the mined properties because in the worst case, it will refine the model until it describes exactly the observed executions and makes no other predictions. In our experience, however, Perfume finds a more concise, predictive model.

3.3.2 Coarsening

Since model inference is NP-hard, refinement efficiently approximates inferring the most concise model by sometimes making suboptimal splits. Once refinement produces a model that satisfies all of the mined properties, a modified version of kTails [4] with $k = 1$ can make the model more concise through coarsening. This process checks each pair of states with the same incoming transition event types to see if these states can be merged without violating any properties. This guarantees that Perfume's model is locally minimal, although it cannot guarantee global optimality.

3.3.3 Model Construction Goals

The initial model construction, refinement, and coarsening address three goals of model inference: the final model is concise, predictive, and precise.

Concise models are human-readable and likely make it easier to understand the execution information contained in the log. Concise models also contribute to generalization, preventing or reducing overfitting to the observed executions. Perfume ensures conciseness in three ways: (1) the initial model is the smallest possible starting point that only separates different event types and limits the edges to only those observed during execution, (2) the refinement process only enlarges the model when it is necessary to eliminate a counter-example, and (3) the coarsening process corrects suboptimal refinements.

Predictive models generalize from the observed executions to also describe unobserved but likely possible executions. Predictiveness goes hand-in-hand with conciseness, as sharing model states among executions lends to combining observed traces into unobserved, predicted ones.

Precise models, particularly in their predictions, correctly identify which unobserved executions are possible. Perfume ensures precision by enforcing a rich set of temporal, performance-constrained properties that hold in the observed executions, and by allowing no new model edges other than those observed during execution, while allowing these edges to form unobserved paths.

4. RELATED WORK

Perfume builds on Synoptic [3], which infers behavioral models that obey temporal properties without time constraints. Walkinshaw et al. [20] also infer models constrained by temporal properties, but these properties are provided manually by the user. Other approaches infer different kinds of models, such as live sequence charts [15], or enrich models with other information, such as data invariants [16]. In contrast, Perfume infers precise models to help developers understand system performance characteristics.

Perfume mines temporal properties with constraints from the input log. Prior specification-mining work has focused on mining temporal [10, 23] and data [8] properties, as well as richer performance-related properties for distributed systems [12]. On their own, however, mined properties can easily overwhelm a developer, which is why Perfume uses mined properties to infer a more comprehensible, concise model of a system's performance.

Other approaches for tracing and capturing performance data in complex systems [1, 2, 9] are complementary to ours as they produce logs that Perfume can use to reveal system structure.

Performance debugging can be aided by lag hunting [13] and performance differencing [17]. The goal of Perfume is to help a developer understand the behavior of a system by deriving a performance-aware model. These models may be used for debugging, but also for comprehension, testing, and other tasks.

5. CONTRIBUTIONS AND VISION

The Perfume approach improves on the state of the art by augmenting the model inference process with performance information. The resulting models convey information that other approaches miss, which can improve a developer's system understanding. Perfume-generated models may also improve software processes and various forms of program analysis. For example, library developers can use Perfume to document the performance characteristics of the library API. These models can then be used to automatically optimize programs according to their library's usage patterns. In testing, Perfume models can be used to broaden a test suite's coverage by producing and testing Perfume-predicted executions. Further, testing can focus on likely executions that are especially slow and are more likely to contain performance bugs. Our early results show great promise for applying Perfume in these areas and in contexts where program behavior and performance modeling may produce insight into how the system operates.

6. REFERENCES

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP*, 2003.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[3] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *ESEC/FSE*, 2011.

[4] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972.

[5] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible Debugging Software. Technical report, University of Cambridge, Judge Business School, 2013.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[7] B. Dagenais, , and M. P. Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *FSE*, 2010.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, 2001.

[9] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[10] M. Gabel and Z. Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *FSE*, 2008.

[11] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE*, 2007.

[12] G. Jiang, H. Chen, and K. Yoshihira. Efficient and Scalable Algorithms for Inferring Likely Invariants in Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, 19(11):1508–1523, 2007.

[13] M. Jovic, A. Adamoli, and M. Hauswirth. Catch Me if You Can: Performance Bug Detection in the Wild. In *OOPSLA*, 2011.

[14] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[15] D. Lo and S. Maoz. Scenario-Based and Value-Based Specification Mining: Better Together. In *ASE*, 2010.

[16] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *ICSE*, 2008.

[17] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *NSDI*, 2011.

[18] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *ESEC/FSE*, 2013.

[19] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *ESEC/FSE*, 2011.

[20] N. Walkinshaw and K. Bogdanov. Inferring Finite-State Models with Temporal Constraints. In *ASE*, 2008.

[21] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Mining Software Repositories*, 2007.

[22] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Experience Mining Google's Production Console Logs. In *SLAML*, 2010.

[23] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *ICSE*, 2006.