Visually Reasoning about System and Resource Behavior

Tony Ohmann[a] Ryan Stanley[a] Ivan Beschastnikh[b] Yuriy Brun[a]
[a]College of Information and Computer Science
University of Massachusetts
Amherst, MA, USA
[b]Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
ohmann@cs.umass.edu, rwstanle@umass.edu, bestchai@cs.ubc.ca, brun@cs.umass.edu

ABSTRACT

Understanding how software utilizes resources is an important software engineering task. Existing software comprehension approaches rarely consider how resource utilization affects system behavior. We present Perfume, a general-purpose tool to help developers understand how resource utilization impacts their systems' control flow. Perfume is broadly applicable, as it is configurable to parse a wide variety of execution log formats and applies to all resource types that can be represented numerically. Perfume mines temporal properties that hold over the logged executions and represents system behavior in a resource finite state automaton that satisfies the mined properties. Perfume's interactive interface allows the developers to understand system behavior and to formulate and test hypotheses about system executions. A controlled experiment with 40 students shows that Perfume effectively supports understanding and debugging tasks. Students using Perfume answered 8.3% more questions correctly than those using execution logs alone and did so 15.5% more quickly. Perfume is open source and deployed at http://perfume.cs.umass.edu/.

Perfume demo video: http://perfume.cs.umass.edu/demo

CCS Concepts

•Software and its engineering → Software system models; Abstraction, modeling and modularity; Dynamic analysis; *Software maintenance tools;* Model checking;

Keywords

Model inference, Specification mining, System understanding, Software comprehension, Resource modeling

1. INTRODUCTION

Debugging software systems requires understanding implementation behavior. One of the factors complicating such understanding is the system behavior's dependency on resources. For example, a system that uses a cache may exhibit different behavior in seemingly identical situations because the cache is in different states.

To help developers understand and debug system behavior and how resource use affects that behavior, we developed Perfume. Perfume uses logs of system executions with resource-use data to infer resource-aware execution properties and concise, precise models of system behavior. Perfume then visualizes these properties and behavioral models, enabling the developer to visually explore system behavior and to interact with individual executions and abstracted behavioral representations. These processes support understanding system behavior and testing hypotheses, an important step in the debugging process.

Unlike other tools, Perfume is applicable to all resources that can be represented numerically. For example, Perfume can be used to model the LED luminosity on a Rasberry Pi device, a workstation's memory management, a mobile phone's network usage, and a web server's response time.

This paper presents the design of an interactive, graphical, web-based Perfume interface for behavioral understanding and debugging. We have previously described Perfume's property mining and model inference algorithms [16]. These algorithms use the temporal relationships and resource-use information encoded in the logs to generalize the logged executions into likely system behavior, abstract common execution behavior, and infer a resource finite state automaton (RFSA) that expresses the generalized, abstracted behavior. Each edge in the RFSA is associated with a system behavior (e.g., a method call, an event execution, or a sent message) and a distribution of resource utilization.

In a controlled study with 40 students performing system understanding and debugging tasks, we found that students using Perfume answered 8.3% more questions correctly than those using execution logs alone and did so 15.5% more quickly. Perfume source code is available at https://people.cs.umass.edu/~ohmann/perfume/, and the tool is deployed at http://perfume.cs.umass.edu/.

Perfume's interactive interface overcomes four research challenges. Perfume's objective is to effectively help developers understand and debug systems by inferring a behavioral RFSA model using the information recorded in execution logs. This objective requires that the model captures and summarizes key system behavior in a human-readable way (Challenges 1 and 2) and that the interactive visualization effectively connects model behavior to recorded system executions (Challenges 3 and 4).

Challenge 1. Knowing invariant properties of system behavior can reveal both desired and unexpected behavior. *Precisely* mining and effectively presenting these properties can thus aid understanding and debugging.

Challenge 2. Inferred behavioral models must be understandable by humans. Inferred models generalize the system behavior from the observed executions. To be used effectively by developers, the models must be *concise*, abstracting and grouping similar behavior together.

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 the author(s) 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.

ICSE'16 Companion, May 14–22, 2016, Austin, TX, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4205-6/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2889160.2889166

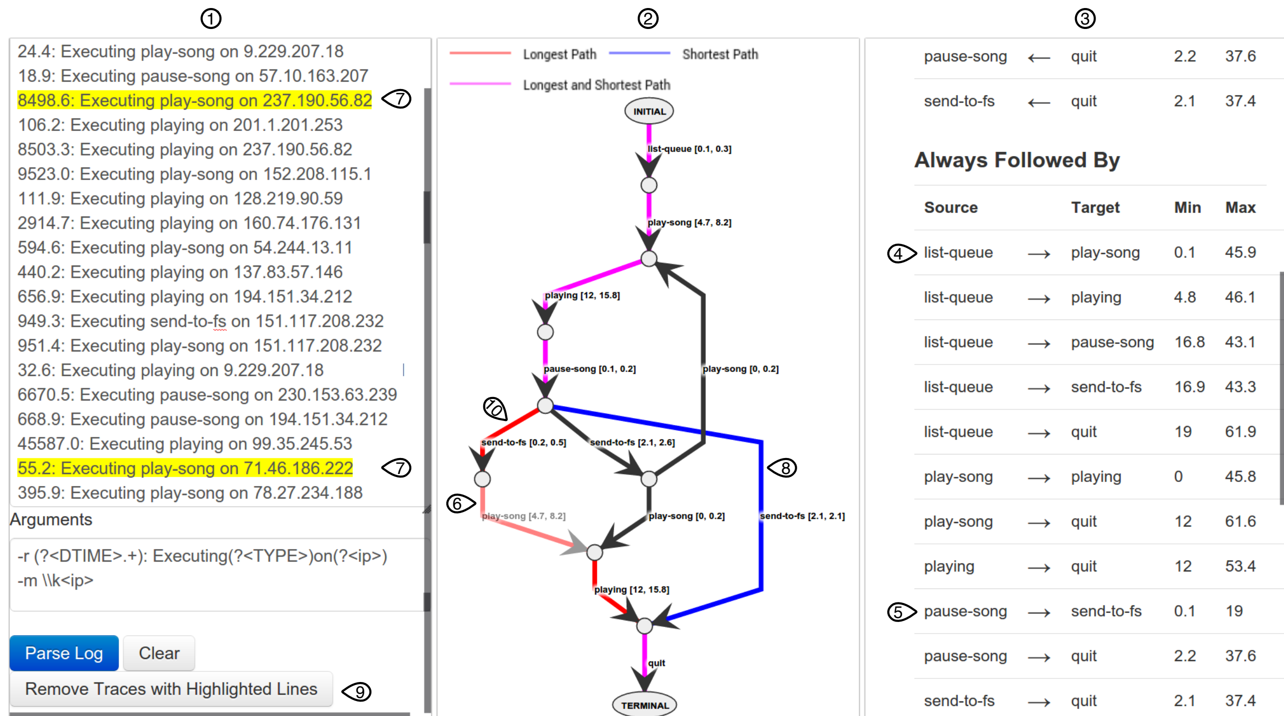

Figure 1: The online Perfume interface. ① The log and parsing expressions. ② The inferred RFSA model. ③ The inferred temporal properties. ④ & ⑤ Two inferred properties. ⑥ A slow `play-song` edge, highlighted by a click. ⑦ Two log lines corresponding to the highlighted `play-song` edge. ⑧ A `send-to-fs` edge indicating that the execution never played after pausing. ⑨ A button to remove executions containing the highlighted log lines. ⑩ A fast `send-to-fs` edge indicates the cause of the bug.

Challenge 3. Abstract behavior represented in the model must be *connected* to the execution traces to enable developers to simultaneously reason about system behavior and executions.

Challenge 4. Understanding system behavior as a whole must be viewed in the context of the individual executions. Developers need to visualize individual executions within the behavioral models and see the effects of removing select executions.

Section 2 describes how the Perfume interface solves these four challenges, and Section 3 summarizes how the model inference algorithm supports these solutions. Section 4 evaluates Perfume, and Section 5 places our work in the context of related research. Finally, Section 6 summarizes our contributions.

2. INTERACTIVE PERFUME INTERFACE

We explain Perfume using a simple cloud-based music player called djQ. djQ loads a static playlist and allows the user to perform only two actions: play and pause. When the user presses play for the first time, or when a song finishes playing, djQ downloads the next song and plays it. When the user presses pause, djQ stores the currently playing song on the local file system so it can be un-paused later without re-downloading it. Unfortunately, several users have reported that hitting play when the song is paused sometimes causes an unexpectedly long wait before the song starts playing.

A djQ developer wants to understand this buggy behavior and find its cause using runtime logs from djQ users. Analyzing the logs manually is hard. Even if the developer uses tools to separate the intertwined executions and parse the logs, she is still forced to consider each execution individually rather than system behavior as a whole. Using Perfume is a better idea. Figure 1 shows a screenshot of Perfume working with the djQ logs. The left side (① in Figure 1) shows the execution logs (only a small part of the log is visible). The middle part (② in Figure 1) shows the RFSA model Perfume infers (see Section 3). Each edge in the RFSA describes a system action (e.g., `play-song`) and a range of time that this action took (e.g., `[4.7, 8.2]` seconds). A path from the `INITIAL` to the `TERMINAL` state in the RFSA represents a djQ execution. Finally, the right side (③ in Figure 1) shows temporal, resource-bounded properties that hold true for all djQ executions; for example, every instance of `list-queue` was eventually followed by `play-song` in no less than 0.1 and no more than 45.9 seconds (④ in Figure 1).

Understanding behavior with runtime properties. The developer wants to discover why djQ sometimes takes a long time to `play-song` after `pause-song`; other than the initial `play-song`, this should happen quickly because the song should be stored locally. The developer hypothesizes that djQ has a bug that sometimes prevents the song from being stored to the local file system after being paused. Challenge 1 (recall Section 1) posits that such system properties should be easy to access and reason about. Perfume addresses this challenge by mining and displaying (③ in Figure 1) temporal properties that are true of all logged executions and that are enforced in all abstracted executions displayed in the model. The developer can check that the property `pause-song` Always-Followed-by `send-to-fs` is present (⑤ in Figure 1), which disproves this hypothesis.

Understanding behavior with the behavioral model. The RFSA (② in Figure 1) describes system-wide behavior of djQ. It groups similar behavior in the observed and abstracted traces, addressing Challenge 2 and helping developers understand system behavior as a whole. Inspecting this model, the developer notes that the first `play-song` of a user's execution always takes between 4.7 and 8.2 seconds. This is expected, since the song needs to be downloaded

from the cloud. Subsequent play-song events should be much faster, and they are most of the time, but one model edge (⑥ in Figure 1) is unexpectedly slow. This edge manifests the buggy behavior, and the developer can focus her attention there. Clicking the edge highlights the relevant execution log lines (⑦ in Figure 1).

Reasoning about executions and abstract model behavior. To focus on the buggy behavior, the developer may remove unrelated executions. Perfume supports using the model to find executions in the log that exhibit specific behavior, addressing Challenge 3 of connecting the model to the original logs. Here, executions that do not play-song after pause-song are irrelevant to the buggy behavior. Clicking on the send-to-fs edge that goes straight to quit instead of to play-song (⑧ in Figure 1) highlights the relevant log lines (highlighting not shown). Perfume allows the developer to hide a behavior from the logs and the model, addressing Challenge 4; clicking on "Remove Traces with Highlighted Lines" (⑨ in Figure 1) removes the executions exhibiting this behavior from consideration.

Armed with a more concise model of djQ executions, the developer refocuses on the bottom-left play-song [4.7, 8.2] edge (⑥ in Figure 1). Following the path from INITIAL to that edge, the developer notices that this edge is preceded by a much shorter send-to-fs edge ([0.2, 0.5], ⑩ in Figure 1) than the faster play-song edges. This means executions with slow play-song events have less time to store the song on the local file system. The developer now understands the bug: a pause-song followed very quickly by a play-song causes djQ to re-download the song from the cloud. The developer may reproduce the bug to generate new logs and verify that they follow the expected path in the model. Clicking on the slow play-song edge (⑥ in Figure 1) highlights the buggy executions.

Once the developer repairs the bug, she can analyze new execution logs with Perfume to help confirm that the slow play-song edge is gone and the bug is fixed.

3. PERFUME BEHAVIORAL INFERENCE

Perfume has two inputs: the system's runtime log and a set of regular expressions for parsing the log. The regular expressions extract the individual execution *traces* from the log. Perfume supports any log that regular expressions can parse where each trace consists of a sequence of *event instances* and each event instance is associated with a *resource measurement*. For example, in the log in Figure 1, a trace is a session for one IP address, an event instance is a specific user action that appears on each log line, such as play-song, and the resource measurement is the elapsed time associated with each event. The regular expressions below the log in Figure 1 parse this log. The first expression matches the log lines and extracts the elapsed time, the event type, and the IP address from each line. The second expression maps log lines with the same IP address to the same execution.

Perfume first *mines temporal properties* from the logs and then uses the logs and temporal properties to *construct and iteratively refine an abstract model* of behavior. The model generalizes the observed behavior and groups similar behavior. We now describe these two steps.

Property mining. To generalize observed behavior, Perfume mines resource-bounded temporal properties true of every logged execution. One such property for the log in Figure 1 is "pause-song Always-Followed-by send-to-fs in at least 0.1 and at most 19 seconds." These properties later guide which predicted, generalized behavior is allowed in the model. By default, Perfume mines four types of properties [16] based on the most common and representative specification patterns [8] but can be extended to mine others.

Model construction. To construct a behavioral model, Perfume first builds a minimal model. This *initial* model is imprecise because it allows many executions that do not satisfy the mined properties. Next, Perfume iteratively *refines* the initial model to satisfy the mined properties. For example, if the model allows an execution that falsifies a mined property (e.g., an execution in which a pause-song is not followed by send-to-fs, or is followed by send-to-fs but either less than 0.1 or more than 19 seconds later), Perfume refines the model by splitting states to disallow such executions. Perfume uses counterexample guided abstraction refinement (CEGAR) [7]: Iteratively, Perfume (1) model checks the model to find a predicted path that violates a mined property, and (2) removes this *counterexample* path by using the path to localize the violation and split into two a state that allows the violation. Perfume iterates model refinement until all the mined properties are satisfied, which is guaranteed to happen [16]. Perfume's model inference task is NP-complete [1, 11], so it approximates a solution and may at times make suboptimal refinements; a post-processing step corrects some such refinements. In our experience, this refinement process finds concise and precise models. We have previously described the property mining and model inference procedures in more detail and proven inference termination and model precision properties [16].

4. CONTROLLED USER STUDY

To understand if Perfume models support system understanding and debugging, we conducted a controlled, within-participants mixed design experiment across 40 students at the University of Massachusetts, Amherst in a joint undergraduate-graduate software engineering course. The user study materials are available online [18]. The study compared Perfume behavioral resource models to the raw execution log information that developers typically inspect.

The study participants had 5 years of programming experience on average, and 25 of the 39 (64%) who answered the question reported that they use logging to debug their code "frequently" or "very frequently." None of the participants had previous experience with Perfume. The participants were asked to perform a debugging task on one system, djQ described above, and another on a second system, a video game. For djQ, we manually constructed execution logs exhibiting the bug described in Section 2. For the video game, we manually constructed execution logs that simulate user play and exhibit a memory-leak bug; in these logs, the resource recorded in the logs was the memory usage. The study employed two treatments: one had the participant use only the logs, and the other, the logs and the Perfume deployment at http://perfume.cs.umass.edu/. Each participant performed one task with djQ and one with the video game, and each treatment once. The order of the tasks and the treatment were chosen at random. Each task entailed reading a short description of a system, studying system behavior, and answering eight system behavior understanding questions, leading to the underlying system bug. The participants' answers were recorded and timed.

The study answered two research questions:

- **RQ1:** How did Perfume affect the correctness of the participants' answers?
- **RQ2:** How did Perfume affect the participants' efficiency in answering questions?

Perfume positively affected both the answer correctness and the efficiency. Participants who used Perfume answered, on average, 8.3% more questions correctly (284 out of 312 questions vs. 269 out of 320 questions, going from 91.0% correctness when using Perfume to 84.1% correctness when using logs; $\frac{284/312-269/320}{269/320} = 8.3\%$). Student's t-test finds that the two answer distributions are the same with only $p = 0.0489$. Perfume also reduced the time to answer the

eight questions by 15.5% (1,325 sec. vs. 1,569 sec., on average). Student's t-test finds that the two timing distributions are the same with only $p = 0.0596$.

After completing the two tasks, we asked the respondents to reflect on their experience, and 38 did. Of those, 28 (74%) found Perfume to be more useful than logs. Further, 27 said they would use Perfume in the future, 3 more said they would use it if they had to analyze logs, 5 were uncertain, 2 said they would not use Perfume, and 1 did not respond to this question. Participants reported that Perfume was "very useful for visualizing the information a log gives you" and that "it gives a nice view of what's going on in the program." One respondent suggested that "Perfume would be very useful when performing testing to see which paths lead to inefficiencies" and envisioned using it "to track performance analytics for a large set of users." Others liked that Perfume "simplifies and compacts" the log and that it "makes system design easy to understand and follow." One participant found Perfume especially intuitive, reporting that when using runtime logs, he "establish[es] a similar flow in [his] mind to debug."

Overall, participants using Perfume answered questions about debugging more correctly and more quickly. While the study was small, these preliminary results suggest that Perfume can help developers understand unfamiliar systems and debug effectively.

5. RELATED WORK

Perfume's interactive interface builds on the Perfume algorithm and prototype [16, 17]. Other model-inference algorithms, e.g., [3, 4, 5, 6, 14, 15], can benefit from similar interfaces.

Perfume mines temporal, resource-bounded properties. Prior work has focused on mining pure temporal [10, 21] and data [9] properties, and richer performance-related properties of distributed systems [13]. Mined properties alone can easily overwhelm a developer, so Perfume uses them to infer a more comprehensible, concise behavioral model.

Statistical debugging work suggests that most user-reported performance issues are diagnosed through comparison analysis [19], which Perfume can be extended to support. Perfume is not intended to replace specialized and fine-grained performance analysis tools, such as visualization techniques designed for performance debugging and optimization [12], runtime profilers like YourKit (http://www.yourkit.com), or memory tools like Valgrind (http://valgrind.org). These specialized tools provide thorough performance analysis of a specific resource and require instrumentation. Finally, Perfume-like measurement-based approaches to software performance engineering focus on improving existing system comprehension [20], and are complementary to predictive model-based approaches used in early development [2].

6. CONTRIBUTIONS

Perfume's interactive interface enables users to visualize, understand, and debug system behavior. Perfume uses execution logs to infer precise, concise models of system behavior constrained by the system's resource use and tightly connects visual representations of the behavior to the logs. A 40-student controlled experiment showed that users answer 8.3% more questions correctly with Perfume than when using execution logs alone and do so 15.5% more quickly.

7. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under grants CCF-1453474 and CNS-1513055, by Google via a faculty research award, by NSERC via a discovery award, and by the Office of the Privacy Commissioner of Canada (OPC). The views expressed herein are those of the authors and do not necessarily reflect those of the OPC and of the other funding organizations.

8. REFERENCES

[1] D. Angluin. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, 21(1), 1980.

[2] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE TSE*, 30(5), 2004.

[3] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying fsm-inference algorithms through declarative specification. In *ICSE*, 2013.

[4] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE TSE*, 41(4), 2015.

[5] 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.

[6] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE TC*, 21(6), 1972.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counter-example-guided abstraction refinement. In *Computer Aided Verification*, 2000.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999.

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

[10] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *FSE*, 2008.

[11] E. Gold. Language identification in the limit. *Information and Control*, 10(5), 1967.

[12] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the art of performance visualization. In *EuroVis STAR*, 2014.

[13] G. Jiang, H. Chen, and K. Yoshihira. Efficient and scalable algorithms for inferring likely invariants in distributed systems. *IEEE TKDE*, 19(11), 2007.

[14] I. Krka, Y. Brun, and N. Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *FSE*, 2014.

[15] D. Lo and S. Maoz. Scenario-based and value-based specification mining: Better together. In *ASE*, 2010.

[16] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun. Behavioral resource-aware model inference. In *ASE*, 2014.

[17] T. Ohmann, K. Thai, I. Beschastnikh, and Y. Brun. Mining precise performance-aware behavioral models from existing instrumentation. In *ICSE NIER*, 2014.

[18] Perfume survey, https://people.cs.umass.edu/~ohmann/perfume/icse2016/survey.html.

[19] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, 2014.

[20] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *FOSE*, 2007.

[21] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.