Behavioral Resource-Aware Model Inference

Tony Ohmann[a] Michael Herzberg[a] Sebastian Fiss[a] Armand Halbert[a]
Marc Palyart[b] Ivan Beschastnikh[b] Yuriy Brun[a]

[a]University of Massachusetts
Amherst, MA, USA

[b]University of British Columbia
Vancouver, BC, Canada

{ohmann, mherzberg, sfiss, ahalbert, brun}@cs.umass.edu, {mpalyart, bestchai}@cs.ubc.ca

Abstract

Software bugs often arise because of differences between what developers think their system does and what the system actually does. These differences frustrate debugging and comprehension efforts. We describe Perfume, an automated approach for inferring behavioral, resource-aware models of software systems from logs of their executions. These finite state machine models ease understanding of system behavior and resource use.

Perfume improves on the state of the art in model inference by differentiating behaviorally similar executions that differ in resource consumption. For example, Perfume separates otherwise identical requests that hit a cache from those that miss it, which can aid understanding how the cache affects system behavior and removing cache-related bugs. A small user study demonstrates that using Perfume is more effective than using logs and another model inference tool for system comprehension. A case study on the TCP protocol demonstrates that Perfume models can help understand non-trivial protocol behavior. Perfume models capture key system properties and improve system comprehension, while being reasonably robust to noise likely to occur in real-world executions.

1. Introduction

Software developers spend half of their time looking for and fixing bugs [13,48] with the global annual cost of debugging topping $300 billion [13]. Mature software projects often ship with known defects [31], and even security-critical bugs remain unaddressed for long periods of time [25].

One significant cause of bugs is the inconsistency between what developers think their system does, and what the system actually does [15, 36]. To increase their understanding of a system, developers instrument key locations in the code and use logging to peek into an implementation's behavior at runtime. 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 [50].

The resulting logs are often incredibly rich with information such as legal behavior, conditions that result in errors, and resource use. Further, it is trivial to enhance systems to produce even richer logs by including more runtime information [52]. Unfortunately, the richness that makes logs useful simultaneously makes them complex, verbose, and difficult to understand. Additionally, logs contain linear series of events that represent individual system executions (e.g., the processing of a single client request), which makes it difficult to understand system behavior in aggregate. This paper focuses on helping developers understand how a system behaves and how it utilizes resources.

Dynamic behavioral specification mining, e.g., [10, 11, 19, 22, 28, 29, 35, 43] tackles the problem of inferring a behavioral model that summarizes a set of observed executions in a concise form. Such models have been used to improve developers' understanding of systems from logs [10], to generate test-cases [16] and test oracles [37], and to help make sense of complex distributed systems [9, 30]. While state-of-the-art model inference algorithms rely on event names, method names, message types, and sometimes data values stored in the logs, they ignore other rich information that makes logs so useful. One log feature that is not utilized by existing model inference tools is the prevalence of resource utilization information in logs. For example, a log may record how long each logged event took to execute (by including a timestamp with each event), how much memory each method used (by tracking memory allocation), or how much data was sent or received over the network. Our work is motivated by the observation that the precision of model inference algorithms and the utility of the inferred models can both be improved by using resource utilization information that is typically recorded in system logs. We describe Perfume, a novel model inference algorithm that extends Synoptic [10] in a principled manner to account for resource usage information in system logs, such as timing, CPU and memory utilization, and other resource measures.

Perfume infers a model directly from a text log and requires its user to specify a set of regular expressions to parse the log. Perfume works on existing runtime logs (with arbitrary log formats, as long as they can be parsed by regular expressions) and requires neither access to source code nor binaries of the modeled system.

Much of the prior model inference work has focused either on improving inference precision [33, 34], or on inferring richer kinds of models, such as extended finite state machines (FSMs) [35], message sequence graph [30], and communicating FSMs [9]. These more expressive models can describe more complex software (e.g., distributed systems) and also provide more information about the underlying behavior to the user or an automated analysis tool. However, this prior work often abstracts away the *context* in which the software executes. In this work, we augment the abstract execution of a system with data on its utilization of resources, such as time,

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'14 September 15–19 2014, Västerås, Sweden

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

ACM 978-1-4503-3013-8/14/09$15.00
http://dx.doi.org/10.1145/2642937.2642988

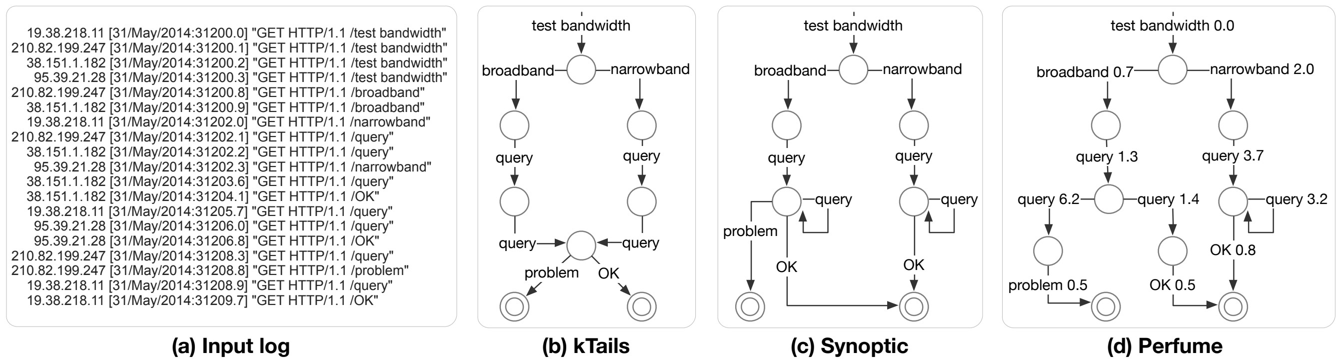

Figure 1: **(a)** A sample network diagnosis tool's log of four execution traces (one trace per IP address), and models inferred by the **(b)** kTails algorithm with $k = 2$, **(c)** Synoptic, and **(d)** Perfume on that log. In the Perfume model, each event is annotated with the time it took to complete.

power, and bandwidth. Our tool, Perfume, infers models that capture the context in which the execution took place. To ease human comprehension of models, Perfume limits each model's context to a single resource, described numerically. Still, for many analyses, a small amount of such contextual information provides valuable insight that is not present in the abstract sequences of events, likely data invariants, and other behavior descriptions. Perfume models are predictive: they generalize observed executions and predict unobserved combinations of behavior that likely represent possible system executions. This helps to keep the models concise, and to reason about the system behavior as a whole, as opposed to only about the observed executions.

Perfume infers models by (1) mining temporal properties with resource constraints from the log, (2) building an optimistically concise FSM model of log executions by overgeneralizing those executions, and (3) iteratively refining the model via counterexample guided abstraction refinement (CEGAR) [14] until the model satisfies all the mined, resource-constrained properties. Perfume builds on Synoptic [10] by extending its property mining, model checking, and refinement algorithms to account for and enforce the resource-constrained temporal properties, while also extending the underlying formalism to encode the executions' resource context. This enables Perfume models, unlike those of related work, to account for optimizations such as caching, lazy evaluation, and loop perforation [44], all of which impact system performance and can cause bugs.

Perfume models can be used to improve developers' comprehension of a system. Our small-scale user study showed that when given Perfume models, academic developers answer 12% more system comprehension questions correctly and in 12% less time than when shown only execution logs, and 4.4% more correctly in 4.2% less time than when shown Synoptic-inferred models.

This paper makes the following contributions:

- We introduce Perfume, a new approach to infer behavioral resource models of systems. The approach and the corresponding open-source tool are general-purpose: Perfume works on any resource that can be represented numerically, and any system log that contains a set of sequentially executed events.
- We implement and publicly deploy Perfume in the cloud: `http://bestchai.bitbucket.org/perfume` and release the source code: `http://synoptic.googlecode.com`.
- We demonstrate that Perfume models improve comprehension via a user study with 13 academic developers, showing that Perfume models make developers 12% more effective at answering system comprehension questions than execution logs, and 4.3% more effective than models inferred by Synoptic.
- We demonstrate that Perfume models can capture complex behavior via two case studies, one applying Perfume to the TCP protocol and another to a web log from a real estate website used in the BEAR framework [22] evaluation.

The rest of this paper is structured as follows. Section 2 illustrates how resource information can improve model inference. Sections 3, 4, and 5 detail the log parsing, property mining, and model construction stages of Perfume, respectively. Section 6 describes our prototype Perfume implementation, and Section 7 evaluates Perfume in a user study and two case studies. Section 8 places our work in the context of related research. Section 9 discusses our findings and future work. Finally, Section 10 summarizes our contributions.

2. Perfume Overview

Consider an example system of a network diagnosis tool that a server can use to identify problematic client network paths. The tool first determines if the client is using narrowband or broadband and then executes 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 1(c) depict models inferred using two well-known algorithms, kTails [11] and Synoptic [10], 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[1]. This model 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

[1]The Perfume implementation outputs event-based models, but for exposition, we convert them to the more standard, state-based FSMs with anonymous states.

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 resource 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 predicts that all combinations of response speeds and bandwidths can lead to a problem. By contrast, the Perfume model not only displays the resource information, but is also more precise in its predictions, which leads to a more accurate differentiation between executions.

2.1 Goals and Challenges

Perfume's high-level objective is to produce a representative model of a system from an execution log containing examples of that system's behavior. More specifically,

Goal 1. Generate models that are precise and predictive.

Predictive models generalize from the observed executions to also describe unobserved but likely possible executions.

Goal 2. Generate models that are concise.

Concise models are human-readable and are more likely to be understood by a developer than the logged executions. Concise models also contribute to generalization, preventing or reducing overfitting to the observed executions.

Achieving these goals requires solving three research challenges:

Challenge 1. Identify resource-based properties to precisely describe observed behavior without overfitting.

We use property mining to generalize observed behavior. These properties must be tractable to mine and to model check, and they must be descriptive. Section 4 addresses this challenge.

Challenge 2. Make Perfume general and applicable to a wide range of systems, logs, and resource measures.

To apply to a broad range of systems Perfume must neither require access to the system source code nor binaries, and should handle a large variety of logs. Section 3 addresses this aspect of this challenge. Further, Perfume must make few assumptions about the resource measures. For example, while timestamps increase monotonically in many log formats (and this monotonicity eases property mining and model checking), many resources, including memory usage, power availability, and network congestion, are not typically monotonic, and Perfume should mine and model check such properties. Section 4.2 addresses this aspect of this challenge.

Challenge 3. Efficient model checking of resource-based temporal properties.

Model checking efficiency is necessary for model refinement: inferring the minimal model is NP-complete [4, 23] and Perfume approximates an optimal solution. Sections 5.2 and 5.4 address this challenge.

2.2 The Perfume Approach

Figure 2 summarizes the Perfume approach. Perfume infers FSM models in which each event is annotated with a *distribution* of resource metric values. Visually, we represent this distribution as a range, e.g., $[1,10]$, and represent singleton ranges with a single number.

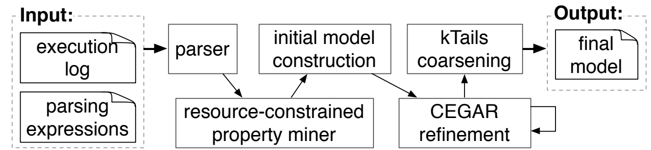

Figure 2: The Perfume model-inference process.

Perfume infers models of unmodified systems, as its input is a runtime log of system executions. Perfume does not require a specific format, as long as the log can be parsed with regular expressions, as described in Section 3.

Perfume produces precise and concise models by generalizing observed executions. Perfume ensures precision by enforcing a rich set of temporal, resource-constrained properties that hold in the observed executions. For example, if Perfume finds that the log of observed executions captures timeout behavior, then the model it infers will disallow predicted, unobserved executions from reaching a timeout sooner than the fastest observed timeout. Section 4 will formalize the property templates using timed propositional temporal logic (TPTL) [3]. Perfume mines instances of these properties from the observed executions and refines the model to ensure that predicted but unobserved executions behave accordingly.

Perfume ensures conciseness in three ways: (1) Perfume starts the inference process with the smallest possible (but imprecise) initial model that separates only different event types and limits the edges to only those observed during execution, (2) Perfume refines the initial model to eliminate counter-example generalizations that violate the mined properties, and (3) Perfume coarsens the final model using the kTails algorithm [11] to clean up after suboptimal refinement. Section 5 further details this three-step process.

3. Log Parsing

Perfume operates on system execution logs and requires access to neither system source code nor binaries, making Perfume general and broadly applicable to a wide range of systems, satisfying part of Challenge 2 from Section 2.1. Perfume has two inputs: the system's runtime log and a set of regular expressions for parsing the log. The regular expressions must extract from the log the individual execution *traces* of the system. 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), a trace is a session for one IP address, and event instances are specific server actions that appear on each log line, such as the strings `test bandwidth` and `query`. The resource measurements are the timestamps associated with each log line. To parse the log in Figure 1(a), the developer would need to specify the following two regular expressions:

- `(?⟨ip⟩).+:(?⟨DTIME⟩.+)\] "GET HTTP/1.1 /(?⟨TYPE⟩.+)"`
- `\k⟨ip⟩`

The first expression matches the log lines and extracts the IP address, time, and event type from each line. The second expression maps log lines with the same IP address value to the same execution.

There are two common points at which systems log events, either before starting the event, or after completing it. Perfume works for both cases, but models inferred from two such logs should be interpreted differently. For example, consider two adjacent event instances `a` and `b`, and time as the logged resource metric. If logging occurs before starting an event, the difference between the metrics

Description	Timed Propositional Temporal Logic formula	Notation in this paper
a always followed by b upper-bound t: whenever a is present in a trace, b is also present later in the same trace with metric difference at most t.	$\Box$x.(a $\rightarrow$ ($\Diamond$y.(b $\wedge$ y - x $\leq$ t)))	$a \xrightarrow{\leq t} b$
a always followed by b lower-bound t: whenever a is present in a trace, b is also present later in the same trace with metric difference at least t.	$\Box$x.(a $\rightarrow$ ($\Diamond$y.(b $\wedge$ y - x $\geq$ t)))	$a \xrightarrow{\geq t} b$
a always precedes b upper-bound t: whenever b is present in a trace, a is also present earlier in the same trace with metric difference at most t.	$\Box$x.(a $\cup$ ($\Diamond$y.(b $\wedge$ y - x $\leq$ t)))	$a \xleftarrow{\leq t} b$
a always precedes b lower-bound t: whenever b is present in a trace, a is also present earlier in the same trace with metric difference at least t.	$\Box$x.(a $\cup$ ($\Diamond$y.(b $\wedge$ y - x $\geq$ t)))	$a \xleftarrow{\geq t} b$
a interrupted by b upper-bound t: between any two consecutive a events there must be a b event, and the metric difference between the two a events must be at most t.	$\Box$x.(a $\rightarrow$ $\Diamond$(b $\wedge$ $\Diamond$y.(a $\wedge$ y - x $\leq$ t)))	$a \xrightarrow{b, \leq t} a$
a interrupted by b lower-bound t: between any two consecutive a events there must be a b event, and the metric difference between the two a events must be at least t.	$\Box$x.(a $\rightarrow$ $\Diamond$(b $\wedge$ $\Diamond$y.(a $\wedge$ y - x $\geq$ t)))	$a \xrightarrow{b, \geq t} a$
a never followed by b: whenever a is present in a trace, b is never present later in the same trace.	LTL formula: $\Box$(a $\rightarrow$ $\Box\neg$ b)	$a \not\rightarrow b$

Figure 3: Perfume property types, including a description, the corresponding TPTL formula, and the short-hand notation used in this paper.

of a and b is the time to complete a. (It follows that the time to complete the last event in each trace is undefined.) However, if logging occurs after completing an event, the metric difference between a and b is the time to complete b. (It follows that the time to complete the first event in each trace is undefined.)

4. Property Mining

Perfume models predict possible, unobserved executions that likely could be produced by the underlying software system. To ensure this prediction is accurate and satisfies Goal 1 from Section 2.1, Perfume (1) approximates the unknown properties of the underlying system using properties mined from the observed traces, and (2) enforces those properties on all paths in its inferred model. A path that violates one or more of these properties represents an execution that exhibits behavior dissimilar to all observed behavior, and therefore, Perfume assumes that the system is unlikely to produce such an execution. Section 4.1 will describe the types of properties Perfume uses, and Section 4.2 will discuss Perfume's solution to the challenge of mining such properties from logs with monotonic and non-monotonic resources.

4.1 Property Types

Perfume parses the log of observed executions and mines seven types of temporal, resource-constrained properties that hold for every trace in the log. The seven property types (Figure 3) Perfume mines and enforces are based on the most common and representative specification patterns presented by Dwyer et al. [17]. We formalize these properties using timed propositional temporal logic (TPTL) [3]. These properties help Perfume address Challenge 1 from Section 2.1 by ensuring that Perfume's predictive models are precise. For example, if every observed execution that had a send event also had a receive event, and the most time that ever passed between these events were 9.9 seconds, Perfume would mine the property "send always followed by receive upper-bound 9.9", represented as send $\xrightarrow{\leq 9.9}$ receive. When inferring the model, Perfume would ensure that no predicted execution (1) had a send event without a later receive event, and (2) had a receive event generated more than 9.9 seconds after a send event.

Three of these property types, "always followed by," "always precedes," and "never followed by," were presented in Synoptic [10], and we extend the former two with resource constraints. While these ostensibly simple property types were shown in [10] to capture complex behavior, Perfume can be easily extended to support more complicated property types. We introduced resource-constrained "interrupted by" properties, which are binary properties with some characteristics of ternary properties, after empirically finding that they can express additional system behavior.

Note that while there are only seven property types, there can be many more instances of these types mined from a log; the number of instances typically depends on the number of different event types the system can produce. The seven property types are templates for constraints on the possible behavior of the system, with the first six property types also encoding the system's performance characteristics. For the upper-bound constraints on properties "always followed by" and "always precedes," t is the maximum resource metric difference, for all traces, between the first event a and the last event b in each trace. For the lower-bound constraints on the same properties, t is the minimum resource metric difference between any event a and any event b in each trace. Bounds on the "interrupted by" property behave identically, except relevant metric differences are between the first and last event of the same type a. Recall that these resource metric differences include the metric measurement during the completion of a but not b if events are logged before they start. The reverse is true if events are logged after completion.

Perfume's properties extend Synoptic's properties [10] with performance data, and capture key behavioral properties of the system more precisely. For example, from the log in Figure 1(a), one of the properties Perfume mines is broadband $\xleftarrow{\geq 8.7s.}$ problem. This property is crucial to understanding 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,

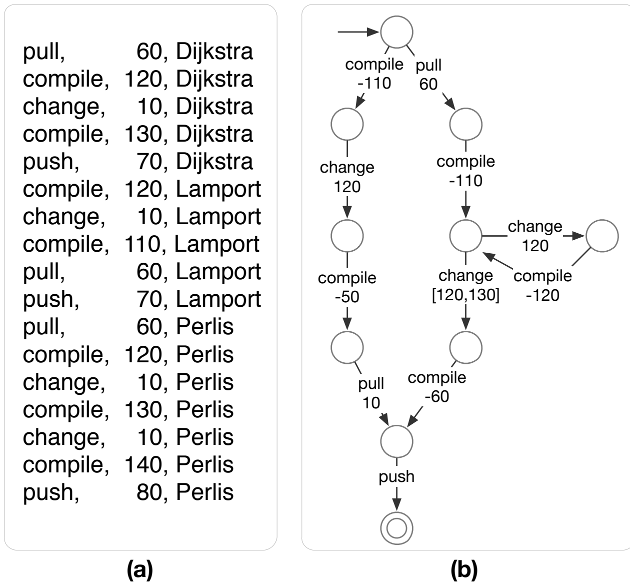

Figure 4: Memory is a non-monotonic resource. The log **(a)** of three executions of a version control system is annotated with the user's local machine memory use. Perfume handles non-monotonic resources and infers the model **(b)** of that log.

whereas no problems are reported after two fast queries. Section 5.2 will explain how Perfume uses these temporal properties to refine the model, separating paths that exhibit distinct behavior, and enforcing that all predicted paths satisfy the mined properties.

4.2 Handling Varied Resource Types

The log in Figure 4 represents the typical workflow of three developers changing and compiling code, and interacting with a distributed version control system via pulls and pushes. The log keeps track of the memory consumption of each developer's system at the time each event executes. Unlike time, system memory consumption is non-monotonic.

Mining properties constrained by resources that either increase or decrease monotonically (e.g., time) can be done by comparing relatively few pairs of event type instances. For example, to mine instances of the property $\texttt{a} \xrightarrow{\geq t} \texttt{b}$, one has to consider all adjacent `a` and `b` instances in every trace, and identify the pair with the least `t` difference in the resource. Similarly, for $\texttt{a} \xrightarrow{\leq t} \texttt{b}$, one only has to consider the first instance of `a` in every trace, and the last instance of `b`. However, for logs with non-monotonic resources, as in Figure 4(a), such algorithms cannot work because the two instances of `a` and `b` with the least and most resource differences may neither be adjacent nor most distant in the trace. For example, in Figure 4(a), the first-last pair algorithm would report $\texttt{compile} \xrightarrow{\leq -50} \texttt{push}$, which is a false property because the difference between Lamport's second `compile` event instance and only `push` instance is -40.

Figure 5 lists the unoptimized resource-constrained temporal property mining algorithm that handles both monotonic and non-monotonic resources. This algorithm inspects all *aType* and *bType* event instance pairs to compute the lower and upper bounds for each mined property. Perfume uses Synoptic's miner for unconstrained property types (described in Section 3.2 in [10]). Perfume's implementation of the algorithm in Figure 5 also minimizing the number passes over the log. This algorithm addresses part of Challenge 2 from Section 2.1, ensuring that Perfume is applicable to a broad range of resource measures and log types.

```
1  function ComputeBounds(aType, bType, log):
2    let lBound ← POSITIVE_INFINITY
3    let uBound ← NEGATIVE_INFINITY
4    foreach trace in log:
5      foreach e1 in trace where e1.eType = aType:
6        foreach e2 in trace after e1 where e2.eType = bType:
7          // computeDelta returns the difference in resource
8          // utilization between two event instances.
9          lBound ← min(lBound, computeDelta(e1, e2))
10         uBound ← max(uBound, computeDelta(e1, e2))
11 return (lBound, uBound)
```

Figure 5: Perfume's algorithm for computing the lower-bound and upper-found resource constraints for two event types *aType* and *bType* on a given *log* input. This algorithm is run on each pair of event types that appear in a mined unconstrained property. The constraints for the "interrupted by" property type are computed differently: we omit this algorithm for brevity.

5. Model Construction

To construct a concise model, satisfying Goal 2 from Section 2.1, Perfume first builds the most concise model it can, as Section 5.1 describes. While concise, this *initial* model is imprecise because it predicts many executions that do not satisfy the mined temporal, resource-constrained properties. Thus, to satisfy Goal 1 from Section 2.1, Perfume iteratively *refines* the initial model to satisfy these properties. Section 5.2 explains the refinement process. Perfume's task is NP-complete [4, 23], 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 5.3 explains the coarsening process. Both refinement and coarsening require model checking — detecting property violations in the model — and Section 5.4 describes this process.

5.1 Initial Model Construction

Initially, Perfume constructs the most concise possible FSM model. In this model, all events of the same type lead to the same state. For example, the initial model for the log in Figure 1(a) would have all `query` events lead to a single state, and no other events lead to that state.

The initial model is very concise but only somewhat precise. All the edges in this model represent an observed state transition. However, because all events of a given type lead to the same state, this model is overgeneralized — it predicts many unobserved executions without regard for their satisfying the temporal, resource-constrained properties mined from the observed executions (Section 4), making the model somewhat imprecise.

Next, Perfume refines this initial model to eliminate mined property violations, and increase the model's precision.

5.2 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-complete [4, 23]. Like prior work [10, 11, 35, 43], Perfume finds an approximate solution. To satisfy Challenge 3 from Section 2.1, this refinement must be efficient.

Perfume iteratively performs counterexample guided abstraction refinement (CEGAR) [14] until the model satisfies all the mined properties. In each iteration, Perfume uses model checking to iden-

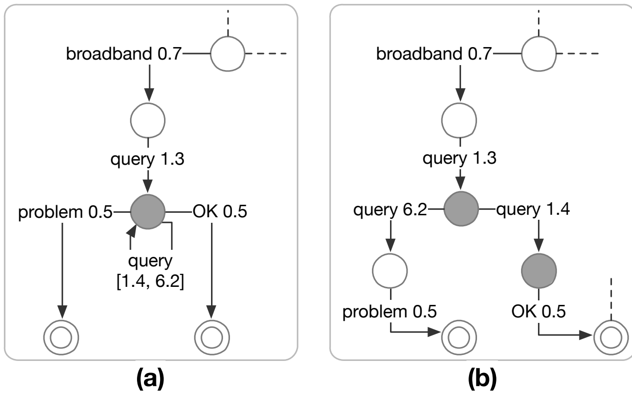

Figure 6: An example of a Perfume refinement step, part of the construction process of the model from Figure 1(d). The partial model in (a) does not satisfy the mined temporal property `broadband` $\xleftarrow{\geq 8.7}$ `problem`. Perfume refines this model by splitting the shaded state into the two shaded states in (b). The resulting model satisfies the mined property.

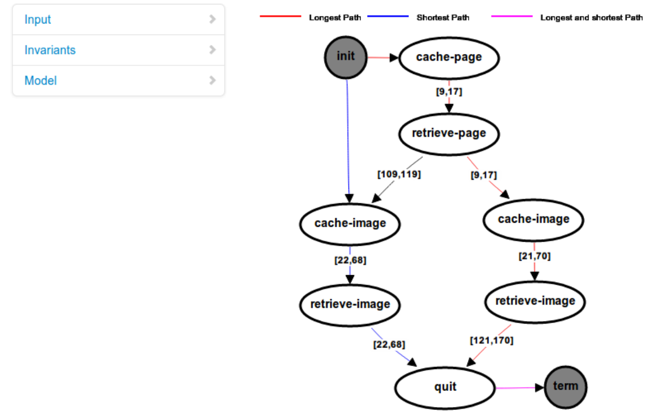

Figure 7: A screenshot of the Perfume front-end. The three-tab interface allows viewing the Perfume inputs, mined invariants, and the model. Shown here is one of the models used in the user study (Section 7.1). Another view of this model is shown in Figure 8.

tify a predicted path in the model that violates a mined property[2]. Model checking (Section 5.4) recognizes violations by finding paths that falsify a mined property's TPTL definition. Using the first identified *counterexample* path, Perfume localizes the violation and splits states in the model to eliminate the path from the model. Localization helps approximate the optimal states to split in order to keep the model as concise as possible. Then, Perfume iterates to find another violation of this or another property and further refines the model until all properties are satisfied.

Figure 6 shows one example refinement iteration. The partial model in Figure 6(a) does not satisfy the mined temporal property `broadband` $\xleftarrow{\geq 8.7}$ `problem`. 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 6(b). The resulting model satisfies the mined property.

Perfume's refinement is guaranteed to produce a model that satisfies all the mined properties because in the degenerate 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.

5.3 Coarsening

Since model inference is NP-complete, the refinement procedure efficiently approximates the most concise model by sometimes making suboptimal refinements. Once refinement produces a model that satisfies all the mined properties, a modified version of kTails [11] with $k = 1$ can make the model more concise through coarsening. This process checks each pair of events of the same type to see if their source and destination states can be merged without introducing a property violation. This guarantees that Perfume's model is locally minimal, although it cannot guarantee global optimality. This coarsening helps keep the inferred model concise, satisfying Goal 2 from Section 2.1.

5.4 Model Checking

Perfume uses model checking to identify property-violating counterexamples to guide refinement (Section 5.2) and, during coarsening, to identify a locally concise model that preserves the mined properties (Section 5.3). This model checking procedure must be efficient to address Challenge 3 from Section 2.1.

During model checking, for each mined property instance, Perfume tests if there exists a path through the model that falsifies the property instance's TPTL definition. To do this, Perfume encodes each property as a property FSM that accepts only the executions that satisfy the property. Model checking simulates all paths in the model in this property FSM. If model checking discovers a path the FSM rejects, that path serves as the counterexample. When used for refinement, this counterexample is simulated in the model to locate the violation. When used for coarsening, the violation signifies that the proposed merge of two states should be rejected.

6. Implementation

Perfume is implemented in Java and is released as open source: `http://synoptic.googlecode.com`. The implementation builds on the Synoptic code-base — we added a total of 2,200 new lines of code to Synoptic (about 10% of the codebase) to implement Perfume. The Java prototype outputs models in the GraphViz dot format[3], which can then be converted into standard image formats. As well, we deployed a web-based front-end to Perfume: `http://bestchai.bitbucket.org/perfume/`. Figure 7 shows a screenshot of the front-end. Using this front-end, users can upload a log, enter the regular expressions to parse that log, and then explore the Perfume-generated model. The front-end model visualization highlights the paths that consumed the most and least resources, and supports several other queries to allow users to better understand the modeled system behavior.

By default, Perfume generates models in event-based form, as shown in Figure 7. These can be trivially converted to state-based models akin to other figures in this paper. The event name within each node is simply moved to all its outgoing edges if events are logged before they start (as is the case in Figure 7), or all its incoming edges if events are logged after they complete (recall Section 3).

7. Evaluation

We evaluated Perfume in three ways. First, we carried out a user study to see if users can interpret Perfume models and use them to answer questions about system behavior (Section 7.1). We found

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

[3] `http://www.graphviz.org/`

that, on average, subjects answered questions more quickly and more correctly with Perfume than with only logs and Synoptic [10]. Second, we applied Perfume to a log of TCP packets captured from a browsing session (Section 7.2). We used the inferred TCP model to identify three standard timing-based protocol behavior. Finally, we applied Perfume to a subset of a web access log for a real estate website that was the subject of evaluation in [22] (Section 7.3). Using the inferred model we replicated some of the findings from [22] and also identified a new analysis specific to the Perfume model.

7.1 User Study

To understand if Perfume-derived models support developer system understanding, we designed and conducted a user study. All of the user study materials are available online [39]. Our goal was to compare Perfume behavioral resource models against the standard behavioral models derived with Synoptic [10] as well as to compare them against the raw log information that a developer typically inspects.

The study was structured as a take-home assignment in a graduate software engineering course. A total of 10 students completed the survey, and 3 additional participants completed it voluntarily. The respondents had 6.1 years of programming experience on average, and 11 of the 13 reported that they use logging to debug their code "frequently" or "very frequently." To our knowledge, none of the participants had previous experience with Perfume.

Participants were asked to read a short description of a system and then answer questions about the system's behavior using one of three treatments, (1) a runtime log, (2) the log and an inferred Synoptic model, or (3) the log and an inferred Perfume model. Each respondent answered questions about three systems, exactly one with each treatment. The order of the systems and treatments were random. Every question had one correct answer, and the responses were timed. Participants using the Perfume treatment answered 81.4% of the questions correctly, meanwhile, the Synoptic treatment resulted in 78.0% correct responses (4.4% fewer correct responses), and the log treatment 72.4% (12.4% fewer correct responses).[4]

The mean time taken to answer questions in the Perfume, Synoptic, and log treatments (averaged by system) was 682 seconds, 712 seconds, and 778 seconds, respectively. Using Perfume models reduced question response times by $\frac{712-682}{712} = 4.2\%$ versus Synoptic models and $\frac{778-682}{778} = 12.3\%$ versus logs.

For the study, we selected three systems: the RADIUS authentication protocol [40], a web browser, and a hypothetical Connection Tester tool modeled in Figure 1:

1. The RADIUS protocol authenticates clients on a network. Clients send a request message with a username and password, and the server can grant access, reject access, or ask the client to send another request message containing additional credentials. Logs included these messages with their sizes in bytes.

Participants using the Perfume model correctly answered the most questions (81.7%) about the RADIUS protocol. They also spent the least amount of time (543 seconds).

2. Caching Web Browser. Web browsers often cache previously-viewed pages and resources to disk to speed up subsequent requests. We manually wrote a log that simulated a web server cache, recording when an object was cached and retrieved, along with the total number of kilobytes read from and written to disk. Figure 8 shows the perfume-inferred Perfume model for this log.

For this system, participants performed best using a Synoptic model. They spent an average of 527 seconds to correctly answer

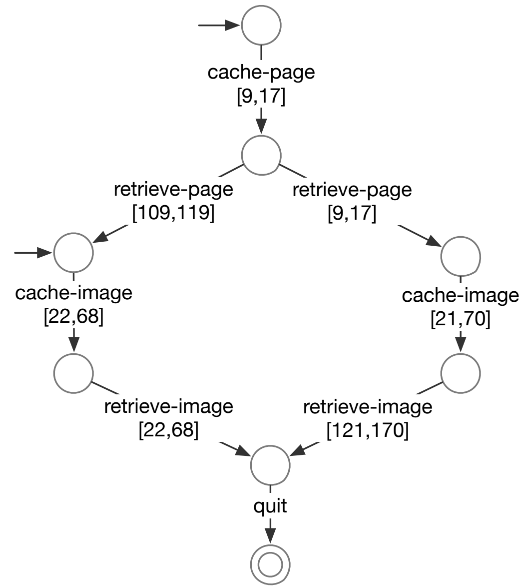

Figure 8: Perfume model of web-browser caching events used in the user study (Section 7.1).

80.6% of the questions. Using a Perfume model, they averaged 632 seconds were correct 60.0% of the time.

3. Connection Tester is a fictional tool that diagnoses network issues. It tests a client's bandwidth, executes a series of queries, and then classifies the network path as "normal" or "problematic" based on the results. Our manually written logs included server events with timestamps.

Participants using Perfume models averaged 995 seconds to answer questions about Connection Tester, the most time of the three tools. However, Perfume model users correctly answered almost every question, averaging 97.2% correct. Users answered correctly less than 83.4% of the time using the other two tools.

After completing the survey, respondents were asked to describe their experiences with the three tools. Of the 13 respondents, 9 found Perfume models to be the most useful for some types of questions. Further, 8 said they would use Perfume models if available, 3 said they might, 1 said he would not, and 1 did not respond. Multiple participants found Perfume models to be "easier to understand," and one envisioned applying them to regression testing. One respondent commented that comparing these models *"with [his] mental model of a system would be very useful in making sure the contracts [he] expect[s] to be enforced are actually holding."* A respondent also suggested that Perfume models *"would make abnormal behaviour [...] much more clear"* and *"would be very useful for debugging."*

No one tool dominated the others across all three systems, but participants using Perfume did best on two of the three systems. And, on average, participants using Perfume models spent the least time answering and answered the most questions correctly. While this study was small, these preliminary results suggest that Perfume models can help developers understand unfamiliar systems quickly.

7.2 Modeling TCP

TCP is a network transport protocol for in-order and reliable exchange of packets (sequences of bytes) between a source and a destination host. A TCP connection starts with a three-way handshake between a client (active end-point) and a server (passive end-point). In this exchange, the client first sends a `syn` packet, to which the server replies with a `syn,ack`, which the client in turn

[4]Because the number of questions for each system varied, we first computed the average of the correct answers for each system, and then averaged these across the three systems for each treatment.

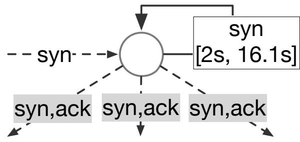

Figure 9: A part of the Perfume model derived for TCP. The server-generated events are shaded. The model illustrates that Perfume correctly inferred the syn re-transmission timeout.

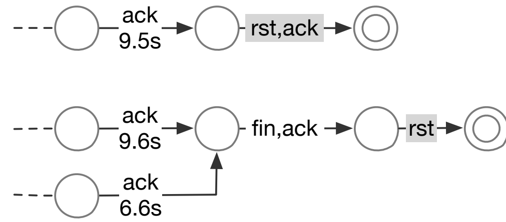

Figure 10: A part of the Perfume model derived for TCP. Server-generated events are shaded. The model illustrates that Perfume correctly inferred that if the server transmits a rst packet when the client does not communicate for a long time.

answers with an ack. Once established, a TCP connection transfers data bidirectionally between the two end-points, using ack packets to acknowledge data reception. When the transfer is completed, the server typically closes the connection with a fin, and the client responds to this with a fin,ack.

TCP packets include flags to encode packet types, such as syn, ack, and fin, and special events, such as rst and psh. An end-point sends a packet containing the rst flag if it decides to terminate the connection unilaterally (e.g., due to a technical issue), which relieves it from executing the closing handshake involving fin. Since TCP implementations typically buffer data in the host operating system, interactive applications need a way to flush these internal buffers. The psh flag causes data in the buffers to be transmitted immediately.

7.2.1 Methodology

We used Perfume to infer a model of TCP with the flag types serving as model events. We then examined this model manually. We used Wireshark[5] to capture all incoming and outgoing TCP packets on server port 80 into a log. We then opened a Wikipedia page[6] in the Google Chrome (version 34) web browser and opened the article for each of the five UMass campuses in a separate window, always letting one page fully load before opening the next. These pages included text and images. We exported all captured TCP packets into a file.

We treated the flags of a single captured packet as an event and all packets sent/received via a specific client port (the server port was always 80) as a trace. We filtered out traces longer than 16 events, as these traces contained long psh,ack and ack sequences that occur when transferring large amounts of data, such as images. Since we were specifically inspecting behavior while establishing and closing connections, these traces would have complicated the model without adding useful information. Note that such filtering is a likely part of a process developers may follow in using Perfume to investigate a system's behavior.

We collected and processed 63 traces containing a total of 602 events. The largest trace had 15 events and the shortest trace had 6 (the minimum for the handshake information exchange). Perfume mined 125 properties such as syn $\xleftarrow{\geq 36ms}$ psh,ack meaning that it takes at least 36 milliseconds from the connection initialization (the client sending a packet with the syn flag set) to the first packet containing application data (the server sending packet with the psh,ack flags set).

7.2.2 Results and Observations

The model Perfume inferred has a total of 168 nodes and 240 edges and is available online [39]. It displays the initial three-way handshake that utilizes syn. Additionally, the model shows an alternating pattern of ack packets between the client and the server as well as graceful termination via fin packets. The Perfume model also captured several TCP corner-case behaviors:

- If the syn packet during the initial handshake is not acknowledged by the server within a short timespan, the client assumes it was lost and resends the syn packet. This is indicated by the syn-loop in the model and explains how TCP reacts to lost syn packets. Figure 9 illustrates this behavior, with the empirically derived timeout range of 2–16 seconds.
- A connection may not terminate gracefully with fin, and may be aborted by the server with a rst. The model inferred by Perfume indicates that this behavior occurs after a long break (6–9 seconds) since the client's last ack without any further packets. This corner-case demonstrates how the server terminates the TCP connection when the connection has remained quiescent for too long. Figure 10 illustrates this behavior.
- The inferred model also captures the behavior in which no data is sent after the initial handshake, leading to a slow transition between the client's ack and a fin,ack. In the other cases, data is sent with a psh flag, and Perfume shows a faster transition in these cases. This illustrates how the client terminates the connection if it requires no new data after the initial handshake. The slow transition is most likely caused by a timeout, as the client waits for new data.

We used Perfume to infer a model of flags within TCP packets in connections between a web browser and a web server. One of the authors, who had minimal prior knowledge of TCP, identified three corner-case behaviors. This case study demonstrates that Perfume-generated models can capture performance and resource-based behavior and may be useful for improving comprehension of a protocol or for confirming specific protocol behavior.

7.3 Modeling User Behavior

Several existing tools use web logs to infer models of how users interact with websites [22, 43]. In this section, we compare Perfume with one of these tools — BEAR [22] — which mines behavioral models using probabilistic Markov models.

7.3.1 Methodology

The BEAR authors [22] shared with us an anonymized subset of the log they used in their experiments. This log contains web request data from a real estate website on which users browse or search for houses and apartments to rent or buy. Each logged web request is associated with a timestamp, which we use to compute the time between a user's web requests.

The input log consisted of about 150,000 lines. We reused BEAR's pre-processing and taxonomy of events. The pre-processing cleans the log (e.g., by removing events generated by web crawlers) and provides each event with semantics by categorizing the events using manual rules (see Section 4.1 in [22]). Reusing these BEAR components allows us to directly compare the BEAR's and Perfume's model inference processes.

[5] http://www.wireshark.org/
[6] http://en.wikipedia.org/wiki/UMass

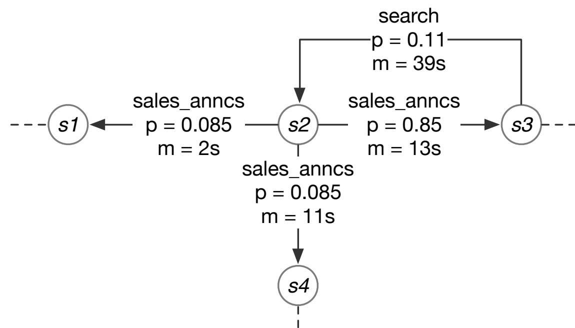

Figure 11: A part of the Perfume-generated real-estate website user-behavior model from the log used in the BEAR evaluation [22]. This part describes the behavior relevant to the *search* page (state *s*2), indicating that users leave this page quickly, presumably toward their intended target pages.

The pre-processed log contained about 12,000 lines, categorized into 13 categories. Perfume executions took, on average, 3 hours to infer a model. This runtime is an average of 7 executions on an Intel Core i7-3630QM laptop with 16 GB RAM, running 64bit Java 1.7.0. Compared to BEAR, which completes in a few minutes, Perfume is slow. We believe this is due to the complexity of the constrained-invariants (by comparison, Synoptic [10] also takes a few minutes to complete). We believe there is ample room for optimization in our implementation.

7.3.2 Results and Observations

Perfume extracted 1,093 traces with a total of 7,706 events from the log. It mined 67 property instances and inferred a model with 153 states and 467 transitions. The complete model is available online [39].

To study the model, we used Gephi [7], a graph exploration tool. Additionally, we developed a script to highlight certain configurations in the model. Our analysis identified states with high probability of a short stay to focus on webpages on which users spent a short time. Such an analysis is relevant to web developers and maintainers who strive to keep the users' attention. We used a prior study [32] to define a *short stay* threshold value of 20 seconds — we use this as a cut-off for the median of the range of times on an edge in the Perfume model. To find pages with high probability of a short stay, we rely on the trace-based transition probabilities present (but not shown by default) in the Perfume models. The probability of a transition between two states on some event is the number of traces from the log that transition along the event divided by the total number of traces that pass through the source state. Given a state, we add the probability of all the state's outgoing transitions (except self transitions) that represent a short stay. We identified several states with a high probability of a short stay; these states correspond to the *search*, *sales*, and *renting* pages of the real estate website. Here, we discuss the *search* page.

Figure 11 presents a part of the model relevant to the *search* page (state *s*2). This page has a high probability of a short stay, since transitions out of *s*2 have a median time below 20s with probability 1. Other states in the model representing the *search* page had similar high probabilities of a short stay. This result suggests that the *search* page is well designed, since users are routed off the search page quickly.

The BEAR framework can detect navigational anomalies, such as the difference between actual and expected user navigation flow, by comparing the BEAR model to the site map [22]. The BEAR study [22] used such anomalies to identify problems with the *terms of use* page: This page, which is only accessible from the *contact* page, was missing a back button. This resulted in unexpected transitions from the state corresponding to the *terms of use* page. The Perfume model can also support this task by showing reachability in the inferred model from the *terms of use* page state. However, Perfume cannot easily support some other analyses supported by BEAR. For example, BEAR can report the probability of a user browsing the sales page and not the renting ads page.

Overall, the Perfume model improved our understanding of the real estate website. While Perfume allowed us to answer certain questions that BEAR cannot answered (e.g., finding pages with high probability of a short stay), and Perfume allowed us to replicate some of the analysis that BEAR supports, other BEAR-supported analyses could not be easily performed on Perfume models. Thus Perfume is complementary to BEAR, assisting some of the same, and some different comprehension tasks.

8. Related Work

We have previously proposed Perfume at a high-level in [38], focusing on log event timestamps. In this paper, we generalize Perfume to other resources, present a complete and detailed description of the involved algorithms, evaluate Perfume, and present, deploy, and release the source code of a Perfume prototype.

Perfume builds on Synoptic [10, 42], which infers behavioral models that obey temporal properties without resource constraints. As Figure 1 illustrates, Synoptic models are likely more concise but less precise than Perfume models. Walkinshaw et al. [47] 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 [34], or enrich models with other information, such as data invariants [35]. By contrast, Perfume infers models to help developers understand system resource utilization characteristics. The timed automata [2] formalism extends finite state machines with real-valued clocks to describe systems for which time plays an important role. Perfume can be seen as a first step toward inferring timed automata system models.

InvariMint [8] proposes declaratively specifying FSM-based model inference algorithms. InvariMint does not support resource-based properties. We plan to express Perfume with InvariMint using an extended set of intersection and union operations over automata that encode metric values.

Perfume mines temporal properties with constraints from the input log. Prior specification-mining work has focused on mining temporal [21, 51] and data [18] properties, as well as richer performance-related properties of distributed systems [26]. 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 resource utilization.

Other approaches for tracing and capturing performance data in complex systems [1, 6, 20, 24] are complementary to ours as they produce logs that can reveal system structure [9]. Perfume targets sequential systems and requires a totally-ordered log of system behavior; Perfume models cannot describe multi-threaded or distributed systems.

Recent work on statistical debugging indicates that most user-reported performance issues are diagnosed through comparison analysis [45]. Building on [41], Perfume can support performance comparison by differencing two Perfume-derived models. More generally, Perfume models can be used for performance fingerprinting [12], testing, and other tasks. Depending on the specificity of the logged information, Perfume models may also be used to support lag hunting [27].

Software performance engineering research can be classified into

two distinct approaches [49]: predictive model-based approaches used in early phases of a software development, and measurement-based approaches like Perfume that improve the comprehension of existing systems. Predictive model-based approaches have been extensively surveyed [5].

Perfume is not intended to replace specialized and fine-grained performance analysis tools, such as runtime profilers like YourKit[7], or memory tools like Valgrind[8]. These specialized tools provide thorough performance analysis of a specific resource and require instrumentation. Perfume relies on existing log messages (though more can be added to improve its analysis), does not require instrumentation, and can model any numerically-valued resource that is interesting to a developer. Further, Perfume can take advantage of prior work on fine-grained logging of performance information [46], and recent tools that make it easy to collect and manage event logs[9].

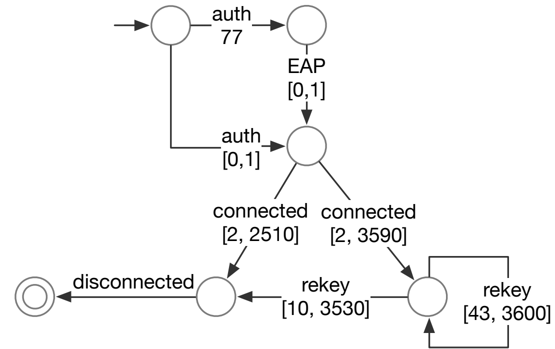

Figure 12: A ground truth model of *wpa_supplicant*'s high-level behavior, including time bounds in milliseconds.

9. Discussion and Future Work

Our evaluation results indicate that Perfume models can help developers understand the performance characteristics of their systems. We are working to expand the capabilities of the Perfume front-end website to allow developers to interact with Perfume models in more meaningful ways. For example, developers will be able to visualize answers to queries like "which path in the model has the least sum total resource use?" (front-end answer: highlight path) or "why are these two states split?" (front-end answer: highlight property type that is violated when the two nodes are merged).

The focus of our evaluation was primarily on the inference of models from logs with timing information like timestamps, though the user study described in Section 7.1 included two non-temporal models. We believe that Perfume models generalize easily to many other kinds of resources that can be expressed numerically, such as energy, throughput, and memory use. We will explore the modeling of these various resource types in our future work.

Perfume-inferred models describe not only the system but also its environment. Thus a model inferred from a set of executions in an environment with environment-specific resource use may not generalize to other contexts, but does accurately describe that and similar deployments. Our view is that the model is a compromise between the design-time specification model and the low-level log of an execution, helping developers reason about the system and its deployment.

Perfume models may have more nodes and edges than Synoptic models (as Perfume satisfies a superset of the Synoptic properties). This may hurt human comprehension. More generally, there is a trade-off between including more information in the models and making the modes useful for comprehension. However, there are tools (e.g., Gephi) that are designed to make large graph exploration more tractable. In fact, we used Gephi when dealing with the more complex model in Section 7.3.

We also envision other non-comprehension uses for Perfume models. For example, developers can use these models to identify erroneous executions based on performance characteristics during debugging. Also, Perfume models can be thought of as specifications of a system's actual runtime behavior. This can be useful to library developers who can use the inferred models to document good and bad library usage practices.

In ideal circumstances, executions in the input log would exercise all possible system behavior. In reality, however, most logs will be incomplete, so Perfume predicts system behavior that is not present in the log. One way to assess the effectiveness of Perfume's predictions is to evaluate its ability to recover a known ground truth model of a system's behavior using logs covering varying proportions of all possible system behavior. We have begun a preliminary evaluation in this direction on the *wpa_supplicant* wireless authentication daemon[10] model pictured in Figure 12. We used this model as the ground truth (although Perfume originally inferred this model from daemon logs). We generated a log from this model that provides full coverage of model behavior by traversing each path in the model two times: one execution exercised all minimum metric values, and another exercised all maximum metric values. For each loop in the model, at least one execution traversed that loop, and no executions traversed any loops more than once. Using the complete set of such paths, Perfume recovered the ground truth model in Figure 12. We are working to extend this early result to circumstances in which the set of observations generated from a ground truth model is incomplete or noisy.

10. Contributions

We have presented Perfume, a resource-aware approach for inferring behavioral software models. A preliminary evaluation shows promise that Perfume-inferred models encode important behavior missed by prior approaches and improve developers' comprehension. For example, a small-scale user study showed that using Perfume improves comprehension task correctness and speed by 12% over using logs, and by 4% over using a prior model-inference tool.

By using resource-use information only available at runtime, Perfume can improve the inferred model's precision, and convey information that other approaches ignore. Perfume-inferred 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 optimize programs automatically 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 Perfume's model quality, and suggest applying Perfume in these contexts.

11. Acknowledgments

We thank Kevin Thai for his initial work on prototyping Perfume. This work was partially supported by NSERC and by Microsoft Research via the Software Engineering Innovation Foundation Award.

[7] `http://www.yourkit.com`
[8] `http://valgrind.org`
[9] `http://logstash.net`
[10] `http://hostap.epitest.fi/wpa_supplicant/`

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 *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[3] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–203, January 1994.

[4] D. Angluin. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, 21(1):46–62, 1980.

[5] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering (TSE)*, 30(5):295–310, May 2004.

[6] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[7] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In *International AAAI Conference on Weblogs and Social Media*, 2009.

[8] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 252–261, San Francisco, CA, USA, May 2013. DOI: 10.1109/ICSE.2013.6606571.

[9] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 468–479, Hyderabad, India, June 2014. DOI: 10.1145/2568225.2568246.

[10] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 267–277, Szeged, Hungary, September 2011. DOI: 10.1145/2025113.2025151.

[11] 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.

[12] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *European Conference on Computer Systems (EuroSys)*, pages 111–124, Paris, France, 2010.

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

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

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

[16] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 85–96, Trento, Italy, 2010.

[17] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 1999.

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

[19] D. Fahland, D. Lo, and S. Maoz. Mining branching-time scenarios. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013.

[20] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[21] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Atlanta, GA, USA, 2008.

[22] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, Hyderabad, India, 2014.

[23] E. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[24] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2012.

[25] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 34–43, 2007.

[26] 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 (TKDE)*, 19(11):1508–1523, 2007.

[27] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: Performance bug detection in the wild. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 155–170, Portland, OR, USA, 2011.

[28] I. Krka, Y. Brun, and N. Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Hong Kong, China, November 2014.

[29] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *New Ideas and Emerging Results Track at the ACM/IEEE International Conference on Software Engineering (ICSE NIER)*, pages 179–182, Cape Town, South Africa, May 2010. DOI: 10.1145/1810295.1810324.

[30] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo. Mining message sequence graphs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 91–100, Honolulu, HI, USA, 2011.

[31] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 141–154, San Diego, CA, USA, 2003.

[32] C. Liu, R. W. White, and S. Dumais. Understanding web browsing behaviors through weibull analysis of dwell time. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 379–386, New York, NY, USA,

October 2010.

[33] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 265–275, Portland, OR, USA, 2006.

[34] D. Lo and S. Maoz. Scenario-based and value-based specification mining: Better together. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 387–396, Antwerp, Belgium, 2010.

[35] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2008.

[36] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 18–28, Washington, DC, USA, 1995.

[37] C. D. Nguyen, A. Marchetto, and P. Tonella. Automated oracles: An empirical study on cost and effectiveness. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Saint Petersburg, Russia, 2013.

[38] T. Ohmann, K. Thai, I. Beschastnikh, and Y. Brun. Mining precise performance-aware behavioral models from existing instrumentation. In *New Ideas and Emerging Results Track at the ACM/IEEE International Conference on Software Engineering (ICSE NIER)*, pages 484–487, Hyderabad, India, June 2014. DOI: 10.1145/2591062.2591107.

[39] Perfume (ASE 2014 supplementary content), `http://people.cs.umass.edu/~ohmann/perfume/ase2014/`.

[40] C. Rigney, S. Willens, Livingston, A. C. Rubens, Merit, W. A. Simpson, and Daydreamer. Remote authentication dial in user service (RADIUS). Technical Report 2865, June 2000.

[41] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[42] S. Schneider, I. Beschastnikh, S. Chernyak, M. D. Ernst, and Y. Brun. Synoptic: Summarizing system logs with refinement. In *Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML)*, Vancouver, Canada, October 2010. DOI: 10.1145/1928991.1928995.

[43] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 422–432, Saint Petersburg, Russia, 2013.

[44] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 124–134, Szeged, Hungary, 2011.

[45] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Portland, OR, USA, October 2014.

[46] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The NetLogger methodology for high performance distributed systems performance analysis. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 260–267, Chicago, IL, USA, 1998.

[47] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 248–257, L'Aquila, Italy, September 2008.

[48] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *International Workshop on Mining Software Repositories*, Minneapolis, MN, USA, 2007.

[49] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *Future of Software Engineering (FOSE)*, pages 171–187, Minneapolis, MN, USA, 2007.

[50] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Experience mining Google's production console logs. In *Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML)*, 2010.

[51] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 282–291, Shanghai, China, 2006.

[52] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):4:1–4:28, February 2012.