Compatibility Testing via Patterns-Based Trace Comparison

Venkatesh-Prasad Ranganath Pradip Vallathol Pankaj Gupta

Created: 03/15/2012
Compiled: March 16, 2013

Technical Report
MSR-TR-2012-87

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

In its most simple form, *compatibility testing* checks if software component A can use or serve another software component B via published interfaces.[1]

The most common form of compatibility (testing) is *backward compatibility (testing)*, e.g. can Microsoft Excel 2010 be used open and manipulate worksheets created using Microsoft Excel 2007? As software gets more modular and extensible, it is necessary to ensure that the set of components constituting a software program are mutually compatible. To this end, most software take proactive measures to allow only the use of compatible components.

A good example scenario is where users upgrade their favourite web browser. Upon upgrading their browsers from version V_1 to V_2, users are notified of any installed browser plug-ins that need to be disabled due to incompatibilities with version V_2 of the browser or that need to be upgraded to newer versions. These incompatibilities can stem from syntactic or semantic changes to the published interfaces of the browser.

Of these changes, incompatible syntactic changes to a published interface (e.g. change in function signatures) can be easily detected by compiling plug-ins against version V_2 of the browser.

As for semantic changes to a published interface, very few of them (e.g. change in values of an enumeration type) can be easily detected by compiling plug-ins against version V_2 of the browser. Most semantic changes will require plug-ins to be executed with version V_2 of the browser and the change to be exercised during the execution.

For example, suppose version V_2 of the browser removed the value *Monday* from an enumeration type for weekdays. If none of the decisions in plug-in A depend on the value *Monday*, then the behavior of plug-in A will be unaffected with version V_2 of the browser. On the other hand, if a decision in plug-in B depends on *Monday*, then the behavior of plug-in B with version V_2 of the browser will most likely be altered; possibly, leading to malfunction of both plug-in B and the browser.

[1]The notion of published interfaces was introduced by Martin Fowler [13].

Beyond such semantic changes, *programs can depend on any behaviors observable at published interfaces of other programs.*

As an example, consider a C language library L that exposes two functions f and g that consume a pointer to a structure with a field q as their first argument and require field q to contain value v upon invocation. In version V_1 of L, suppose the effect of f on field q is unspecified and q is unmodified upon returning from f. This observation can be exploited to optimize clients of version V_1 of L — when invoking f and g in sequence, assign v to q and then invoke f and g without any intervening assignment of v to q. However, in a subsequent version V_2 of L, if the implementation of f modifies q, then clients optimized against version V_1 of L will fail when operating with version V_2 of L.

We encountered a real-world instance of this scenario during the development of Windows 8. To support USB 3.0 protocol in Windows 8, USB team built a new USB 3.0 driver stack from scratch. Since USB 3.0 protocol is backward compatible with USB 2.0 protocol, USB 3.0 driver stack needed to support USB 2.0 devices along with their device drivers that were built against the existing USB 2.0 driver stack. Hence, when servicing USB 2.0 devices, USB 3.0 driver stack was required to mimic the observable behavior of the USB 2.0 driver stack. In this context, an incompatibility issue (deviation) could be the USB 3.0 driver stack completes isochronous transfer requests at `PASSIVE_LEVEL` interrupt request level while USB 2.0 driver stack always completes such requests at `DISPATCH_LEVEL` interrupt request level. Further, such deviations may not affect the observable behaviors of USB 2.0 devices used to test USB 3.0 driver stack. However, they could possibly affect untested USB 2.0 devices which could be numerous given the number of unique USB devices in the world. So, we needed a way to test for and uncover such subtle incompatibilities between these USB driver stacks.

To discover such incompatibilities due to dependences on observable behaviors (and ensure compatibility), many software shops employ field testing by providing customers with pre-release versions of their software (e.g. Windows 8 by Microsoft, Firefox by Mozilla). The success of field testing in uncovering compatibility issues depends both on the number of customers participating in such testing and the extent to which customers use and exercise various behaviors of the software. When pre-release versions of software have low adoption and usage, compatibility issues can go undiscovered until the release of the software. Hence, upon release and wider adoption of the software, latent compatibility issues can surface causing reliability issues to users and maintenance costs to software vendors.

Proposed Approach

Inspired by the above problem, we devised a simple data-driven differential approach to test for compatibility. Given two programs with identical published interfaces, the approach relies on clients interacting identically with these programs via their published interfaces, i.e. the clients submit same requests in the same order. These interactions (observable behaviors) are traced and the resulting traces are compared to detect possible compatibility issues resulting from both the presence of new unobserved behaviors and the absence of previously observed behaviors. Consequently, our approach can uncover issues that do not affect the observed (current) executions but could affect yet unobserved (future) executions.

For purpose of comparison, traces are abstracted as sets of structural and temporal patterns (based on existing notions of patterns that can be mined using existing algorithms) and these sets of patterns are compared using simple set operations, i.e. set union, intersection, and difference. This is the key characteristic of our approach.

In terms of guarantees, our approach is *unsound* — every detected deviation/difference (compatibility issue) need not be a bug; hence, it entails manual effort to examine detected deviations and classify them as either benign deviations or bugs. On the other hand, our approach is *complete* — all deviations that can be represented by a pre-defined class of patterns used to abstract traces will be detected.

As evaluation, during the development of Windows 8, we used our approach to test compatibility between USB 2.0 and USB 3.0 bus drivers in Windows 8. When used within an appropriate work flow, the approach uncovered 25 compatibility bugs in USB 3.0 bus driver by analyzing a pair of traces from only 14 USB 2.0 devices that were functioning without errors with both USB bus drivers.

From this effort, we make the following key contributions in this manuscript.

1. We propose a differential approach to compatibility testing based on patterns-based comparing of execution traces. This approach can uncover compatibility issues stemming from both the presence of unobserved behaviors and the absence of observed behaviors.

2. We demonstrate the effectiveness of the proposed approach in an industrial setting by using it to test compatibility between USB 2.0 and USB 3.0 bus drivers during Windows 8 development cycle. We also demonstrate that the approach can detect compatibility issues from succeeding executions.

3. We illustrate that sets of structural and temporal patterns observed in traces can serve as effective trace abstractions to enable software engineering and maintenance tasks, e.g. compatibility testing.

The rest of this manuscript is organized as follows. Sections 2 and 3 describe compatibility testing and patterns-based trace comparison. Section 4 provides a detailed exposition about our experience using patterns-based trace comparison to test compatibility between USB 2.0 and USB 3.0 bus drivers in Windows 8. Section 5 discusses related efforts. Section 6 presents future possibilities.

2 Compatibility Testing

As in all forms of testing, compatibility testing is used to check if a program P_t consumes a given input x and produces the expected output y, i.e. $P_t(x) = y$.

The distinction of compatibility testing is that the expected output is the output produced by another program P_r upon consuming x, i.e. $P_t(x) = y = P_r(x)$.

The above simple view of compatibility testing suffices when we are testing if the observed output is identical to the expected output, e.g. for argument -2, does function abs_1 return the same value as function abs_2? In many cases, we are interested in testing if the observed value is *similar (identical modulo certain differences)* to the expected output, e.g. upon consuming input x, do programs P_1 and P_2 output log files that contain the same messages but not necessarily in the same order? To admit such notions of similarity into compatibility testing, we define compatibility testing as follows.

Definition 1 Given an input x and two programs P_r and P_t, P_t and P_r are (α, ψ)-compatible (denoted as $P_t \sim_{\alpha,\psi} P_r$) if $\psi(\alpha(P_t(x)), \alpha(P_r(x)))$ holds where α is a transformation function over program outputs and ψ is a binary test predicate over transformation values.[2]

With this definition, different forms of compatibility testing can be described using appropriate combinations of transformation functions and test predicates. For example, the simplest form of compatibility testing based on equality of output (i.e. $P_t(x) = P_r(x)$) can be described by $\sim_{id,=}$ with identity function as α and equality predicate as ψ.

In this vein, we can describe compatibility testing of programs that output traces (sequences).[3] Consider programs that consume an input x and produce a trace π as output. By the above definition, given two programs P_t and P_r that output traces π_t and π_r upon consuming test input x, P_t is (α, ψ)-compatible to P_r if $\psi(\alpha(\pi_t), \alpha(\pi_r))$ holds (as $P_t(x) = \pi_t$ and $P_r(x) = \pi_r$). Hence, the problem of compatibility testing based on traces reduces to the problem of trace comparison under α and ψ.

In many situations, we want to test programs that consume input sequences and produce output traces. The above definition of compatibility testing can be applied in such situations by conditioning the output traces produced by programs as follows: when a program P consumes an input sequence $X = x_1, x_2, \ldots x_n$ produces an output trace π, X should be a subsequence of π and, for every input $x_k \in X$, if the corresponding output y_k of P exists in π, then y_k follows x_k in π, i.e. $\pi[j] = y_k \implies (\exists i. \pi[i] = x_k \wedge i < j)$.

The notion of trace similarity determined by α and ψ can range from simple equality to subsequence equivalence (similar to stuttering equivalence [16]). Further, the notion of similarity determines the kind of issues that can be uncovered via compatibility testing.

In summary, under a notion of similarity determined by α and ψ, the problem of testing compatibility between programs based on their output traces reduces to the problem of trace comparison.

3 Patterns-based Trace Comparison

In this section, we describe two notions of trace similarity. Both these notions of similarity use set equality as the test predicate ψ. The transformation function α in these notions are based on event abstractions and binary linear temporal patterns proposed by Lo et al. in [17].[4]

From here on, an event $e = \{a_1 = c_1, a_2 = c_2, a_3 = c_3, \ldots, a_n = c_n\}$ is a map from attributes to values and a trace $t = (e_1, e_2, \ldots, e_n)$ is a sequence of events.

3.1 Structural Patterns-based Similarity

The most common notion of trace similarity is based on the presence/absence of events in traces (while ignoring the order of events), i.e. consider traces as sets of events and compare these sets. We refer to this notion as *event based trace similarity.*

While this notion is simple, it can be ineffective when different attributes of an event have different

[2]Regression testing can be viewed as a form of compatibility testing where P_r and P_t are two consecutive versions P_i and P_{i+1} of the same program P. Further, this definition can be generalized to other forms of testing.

[3]We will use the terms trace and sequence interchangeably.

[4]For more details about these patterns and the corresponding pattern mining algorithms, please refer to [17].

relevance in different scenarios. For example, when comparing two execution traces in terms of invoked functions, it might suffice to consider views of events limited to the attribute capturing function names, e.g. consider {*fun=“fopen”*} view of the invocation event {*fun=“fopen”, arg1=“passwd.txt”, arg2=“r”, return=0x21*}.

From this observation, we propose a notion of trace similarity based on the presence/absence of *(event) abstractions* of events in traces where *any non-empty subset of an event e is an abstraction (view) of e*. We refer to this notion as *event abstraction based trace similarity.*[5]

In many situations, it is useful to consider data constraints spanning multiple attributes. For this purpose, we propose using the notion of *event abstraction with quantification — given an event abstraction* $e = \{a_1 = c_1, a_2 = c_2, a_3 = c_3, \ldots, a_n = c_n\}$, $e' = \{a_1 = v_i, a_2 = c_2, a_2 = v_i, \ldots a_n = c_n\}$ *is a quantified abstraction of e if there exists a substitution* θ *(a non-empty map from variables to values) such that* $\forall a_i . e[a_i] = e'[a_i] \vee e[a_i] = \theta(e'[a_i])$ *where* v_i*s are free variables.* Consequently, an attribute is *quantified* if it is associated with a free variable (as opposed to a value).

These event abstractions are patterns of event structures observed in a trace; hence, we refer to these abstractions as *structural patterns*. Further, structural patterns with and without quantification are referred to as *quantified structural patterns* and *unquantified structural patterns*, respectively. Finally, we define the notion of *structural patterns-based trace similarity* as comparing the sets of structural patterns observed in traces via set equality.

In other words, compatibility testing based on traces can be realized with a transformation function that transforms a trace into a set of observed structural patterns and a test predicate that checks for set equality.

[5]In the presence of full domain knowledge, this notion is not required as the exact data fragments can be extracted from events; however, full domain knowledge is seldom available in real-world settings.

3.2 Temporal Patterns-based Similarity

Structural patterns-based trace similarity will be ineffective in situations where traces are identical in terms of the structural patterns but differ in the order of structural patterns. For example, traces $t_1 = (a, b, c)$ and $t_2 = (a, c, b)$ are identical under structural patterns-based trace similarity as both traces result in the same set of structural patterns $\{a, b, c\}$.

To address this drawback, we propose transforming a trace into a set of temporal patterns composed of structural patterns. Of the numerous forms of temporal patterns, we consider the following four binary linear temporal patterns defined in [17].

Given structural patterns A and B are observed in events of a trace,

1. $A \overset{*}{\rightarrowtail} B$ ($B \overset{*}{\leftarrowtail} A$) is observed in the trace when an event e_i matching A is followed (preceded) by an event e_j matching B.[6] These are *eventually patterns.*

2. $A \overset{a}{\rightarrowtail} B$ ($B \overset{a}{\leftarrowtail} A$) is observed in the trace when event e_i matching A is followed (preceded) by an event e_j matching B and no event between e_i and e_j matches A. These are *alternation patterns.*

In the above patterns, either both A and B are quantified or none are quantified. Further, when A and B are quantified, the associated substitutions resulting from matching events are identical, i.e. $\theta_A = \theta_B$. For example, the pair of event abstractions {*fun=“fopen”, return=0x21*} and {*fun=“fclose”, arg1=0x21*} match the temporal pattern {*fun=“fopen”, return=*v_1} $\overset{*}{\rightarrowtail}$ {*fun=“fclose”, arg1=*v_1} under the substitution $\theta = \{v_1 \mapsto 0x21\}$. However, the event abstractions {*fun=“fopen”, return=0x21*} and {*fun=“fclose”, arg1=0x23*} do not match the same temporal pattern as the event abstractions do not share a common substitution.

[6]Given an structural pattern C, an event e *matches* C if C is an abstraction of e.

We shall refer to temporal patterns with and without quantification as *quantified temporal patterns* and *unquantified temporal patterns*, respectively. Finally, we define the notion of *temporal patterns-based trace similarity* as comparing the sets of all temporal patterns observed in traces.

In other words, compatibility testing based on traces can also be realized with a transformation function that transforms a trace into a set of observed temporal patterns and a test predicate that checks for set equality.

3.3 Discussion

Statistical Similarity The above notions of trace similarity is insufficient when the traces being compared exhibit the same set of patterns but differ in terms of the statistical properties of the observed patterns. In such cases, we need to consider patterns along with their statistical properties and employ notions of similarity sensitive to statistical properties. For example, each pattern in a trace can be associated with its frequency in the trace. So, while comparing two traces, a pattern common to both traces is deemed as a difference if its frequency is below a preset significance threshold in one trace and above the same threshold in the other trace.

Similarly, statistics about the distance between events matching temporal patterns can be used to enable patterns-based performance debugging.

We refer to this notion of similarity based on statistical properties of patterns as *statistical similarity.* While we have explored performance debugging based on this notion of similarity, we will not describe it in this manuscript.

Textual Comparison Given two traces, we can calculate the longest common subsequence (LCS) of these traces and identify events from traces that are absent from the subsequence as the difference between the traces. This would be similar to serializing traces into text files (e.g. with one line per event) and using your favourite textual differencing tool. While this technique can detect deviations captured by missing events, it will be inaccurate in identifying deviations stemming from temporal orderings of events that are not part of the longest common subsequence. Further, it is unclear if and how can event abstractions be effectively and efficiently considered in LCS-based comparison.

Graphical Comparison Given a trace, we can generate a succession graph in which nodes represent unique events in the trace and directed edges capture the immediate successor relation between events. Hence, given two traces, we can generate and compare their succession graphs to detect missing nodes and edges as the difference between these graphs. (This would be identical to diffing sets of all bigrams in given traces.) While this technique can detect issues captured by missing nodes (events) and edges (bigrams), it will fail to detect issues that can be only captured by event subsequences, e.g. the technique will detect presence of (a, b) and (b, c) in trace (a, b, c) but will fail to detect the presence of subsequence (a, c). However, this can be remedied by considering graph reachability information. Even so, as with textual comparison, it is unclear how to perform graph diffing with event abstractions.

4 USB Driver Compatibility Testing in Windows 8: An Experiment

In this section, we describe our experience using patterns-based trace comparison to devise an approach to test compatibility between USB 3.0 driver stack and USB 2.0 driver stack in Windows 8. This was a joint effort with USB team in Windows organization within Microsoft.

We describe this experience in detail to present the nuances involved in using patterns-based trace comparison for compatibility testing. With these details, other researchers and practitioners should be able to easily reproduce our experience in other contexts.

4.1 Windows Driver Subsystem

Most of the kernel-mode device drivers on Windows Vista and Windows 7 conform to Windows Driver

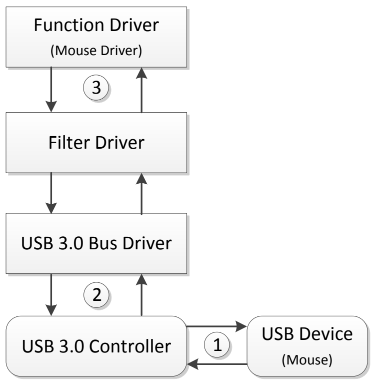

Figure 1: An illustration of how various types of WDM drivers are stacked when a USB device is plugged into a USB 3.0 port on Windows 8 PC.

Model (WDM). This model supports the following kinds of drivers.[7] (Please refer to Figure 1 for an illustration of how various kinds of WDM drivers are connected.)

- A *bus driver* services devices that can have child devices, e.g. bus controllers, adapters, and bridges. As these are necessary drivers, Microsoft generally provides these drivers for each type of bus, e.g. USB and PCI. All communication to devices on a specific bus goes through the corresponding bus driver.
- A *filter driver* extends the functionality of a device or intercepts and possibly modifies I/O requests and responses from drivers.
- A *function driver* exposes the operational interface of a device to the system, e.g. the device driver provided with Microsoft Comfort Curve 3000 keyboard.

In WDM, most communications with and between drivers is packet-based. Typically, an I/O request is *dispatched* to a driver by invoking `IoCallDriver` routine with an *I/O Request Packet (IRP)* (a structure in C language) embodying the request. The IRP is delivered to the I/O manager which then forwards the request to the appropriate driver. Upon *completing* a request, the servicing driver modifies the corresponding IRP (e.g. updates status fields or copies data into buffers in the IRP) and signals the completion of the request to the I/O manager by invoking `IoCompleteRequest`. I/O manager then signals the requesting driver about the completion by invoking the *IoCompletion* routine registered for the IRP. The fields of the IRP both define the type of requests and the data (both input and output) pertaining to requests.

[7]We use the terms driver(s) and device driver(s) interchangeably.

4.2 Problem

As mentioned in *Building Windows 8* blog [2], Windows 8 supports USB 3.0 protocol with a new USB 3.0 driver stack that provides a bus driver dedicated to USB 3.0 controller. Since USB 3.0 driver stack is a clean room implementation, it does not borrow any code and, hence, any behavior from existing USB 2.0 driver stack in Windows 8. Further, USB 3.0 driver stack exclusively supports devices controlled by USB 3.0 controller (connected to USB 3.0 port) while USB 2.0 driver stack exclusively supports devices controlled by USB 2.0 controller (connected to USB 2.0 port).

In the rest of this exposition, we focus on the USB bus drivers provided by the USB driver stacks as they control the underlying USB controller. Also, we shall refer to USB bus driver as USB driver.

Consider the situation where a user plugs in a USB 2.0 device into a USB 3.0 port on a computer running Windows 8. Since USB 3.0 protocol is backward compatible with USB 2.0 protocol, the user expects the device to behave as if the device was plugged into a USB 2.0 port and serviced by USB 2.0 driver. In other words, *the observable behavior of a device plugged into a USB 3.0 port should be identical to the observable behavior of the same device plugged into a USB 2.0 port.*

To enable the above scenario, USB 3.0 driver needs to support USB 2.0 protocol to guarantee behavioral equivalence with USB 2.0 driver. However, it is possible that existing function drivers could depend on unpublished yet observable behaviors of USB 2.0 driver, e.g. USB 2.0 driver zeroes out the `PortStatus` bits in the IRP upon failing to service I/O control code `IOCTL_INTERNAL_USB_GET_PORT_STATUS`.[8] Hence, for compatibility, USB 3.0 driver should support any unpublished yet observable behaviors of USB 2.0 drivers, and we need to test these USB drivers for equivalence of such behaviors.

A naive approach to test for such equivalence is to exercise USB 3.0 driver with every device and its function driver. However, this approach is prohibitive as there are over 10 billion USB devices in the world today.

To identify an alternative, we observed that every function driver exposes the functionality of a device to the system by interacting with the device via the bus driver. So, it is likely that any deviation in interactions between a function driver and the bus driver could lead to deviations in the observable behavior of the device. Hence, we tested compatibility between USB drivers by testing for equivalence of interactions between function drivers and the USB drivers (at point 3 in Figure 1), i.e. *for every request from a function driver, is the response from USB 3.0 driver similar to the response from USB 2.0 driver?*[9]

[8]Such behaviors can stem from decisions while implementing weakly specified parts of USB 2.0 protocol.

[9]Here are two alternative approaches to test compatibility of USB drivers.

Approach 1 Check equivalence of on-the-wire interactions between the USB controller and the USB device (at point 1 in Figure 1). While this form of checking can be highly accurate, it can be brittle due to controller specific nuances stemming from weakly specified parts of USB protocol. Also, since the USB driver does not have direct control over these interactions, it is unclear if this approach will help uncover compatibility issues directly stemming from the implementation of USB driver.

Approach 2 Check for equivalence of command-level interactions between the USB driver and the USB controller (at point 2 in Figure 1). This form of checking can be brittle due to nuances stemming from the combination of the flexibility of USB command language and the implementation of both the USB controller and the USB driver.

Independent of these details, similar interactions between the bus driver, the controller, and the device do not guarantee similar interactions between the function driver and the bus driver. Hence, we chose to test the interactions between the function driver and the bus driver, at the highest level in the stack.

For purpose of simplicity, we limited our focus to test compatibility between USB drivers during *enumeration* and *rundown* phases during the lifetime of a device on Windows, i.e. recognition of a device by Windows and cleanup following the ejection of a device, respectively.

4.3 Solution

Our solution to this problem uses patterns-based trace comparison (described in Section 3) with the workflow outlined in Figure 2. In the following sections, we describe various steps in this workflow.

4.3.1 Trace interactions between drivers

Given a USB 2.0 device, we enabled tracing, plugged in the device to a USB 2.0 port, waited for the device to be recognized by Windows, ejected the device, waited for the device to be unavailable in Windows, and disabled tracing. We then repeated these steps with the same device plugged into a USB 3.0 port.

For tracing, we used a customized filter driver (developed by USB team) to capture the interactions between functions drivers and USB drivers and log these interactions into ETW traces via Event Tracing for Windows (ETW) [1].

4.3.2 Mine patterns from traces

The traces collected in the previous step contained 13 simple types of events. Of these simple event types, few captured the invocations of driver routines (e.g. `IoCallDriver`) that enable inter-driver communication, few captured the completions of driver routines, and others captured the arguments to these routines. Hence, we combined these 13 simple event types into 6 compound event types that represent the invocation of driver routines along with their arguments and the completion of driver routines with their return values. Consequently, we preprocessed the traces to coalesce simple events into compound events. During preprocessing, to ease further processing, we synthesized certain information (e.g. I/O control codes (IOCTL)) and captured them in few (< 5) synthesized event attributes. Since the events in ETW are structured, we also flattened access paths to fields. After preprocessing and flattening of access paths, a total of 361 attributes existed (including few synthesized attributes) across 6 compound event types.

To curb the explosion of structural patterns during mining, we employed domain knowledge by consulting a developer from the USB team. Specifically, out of 361 attributes, we identified 108 attributes that could be ignored. Of the remaining 253 attributes, we identified 29 attributes as necessary (i.e. they should occur in all structural patterns of an event) and 224 attributes as optional (i.e. not necessary). Also, we identified 75 attributes that should not be quantified; of these, 23 attributes were identified to be abstracted as either NULL or non-NULL.

In terms of quantification, we identified 150 attributes that should always be quantified. For example, since we were interested in checking if similar IRPs are processed similarly by both driver stacks, we need not have to mine patterns involving event spanning different IRPs; hence, `irpId` attribute that uniquely identifies the source IRP of an event was always quantified. Further, when quantification of attributes is used to capture data flow between events participating in a temporal pattern, non-existent data flow can be captured due to representational equivalence as opposed to semantic equivalence. To curb this noise, based on domain knowledge, by way of configuration, we considered only data flows between 17 attribute pairs involving 26 different attributes in addition to data flow between same attributes occurring in different events.

With the above setup, we mined every structural and temporal patterns (of the forms described in Section 3) occurring in the collected traces. In unquantified form, we considered only patterns involving necessary attributes. In quantified form, we considered only patterns involving all necessary attributes and up to three optional attributes.

For mining, we used Tark, a toolkit to mine the patterns described in Section 3 [3]. Since Tark implements pattern mining algorithms described in [17], please refer to [17] for details about these mining algorithms.

Note While using domain knowledge does incur additional cost, this is a one time cost incurred when the approach is tailored and deployed for a specific context. Further, the benefits of using domain knowledge can be huge, e.g. knowledge about the fields that can be ignored reduced the ceiling on the number of structural patterns from 2^{361} to 2^{253}.

4.3.3 Calculate unique patterns (deviations)

For each USB 3.0 trace, we calculated two sets of unique patterns (or deviations) based on all USB 2.0 traces in our corpus.[10]

The first set was composed of *unique USB 2.0 patterns* observed in every USB 2.0 trace but not observed in the given USB 3.0 trace — the difference between the intersection of pattern sets of every USB 2.0 trace and the pattern set of the given USB 3.0 trace. *These patterns identify (always) observable behaviors of USB 2.0 driver that were not exhibited by USB 3.0 driver.*

The second set was composed of *unique USB 3.0 patterns* observed in given USB 3.0 trace but in none of the USB 2.0 traces — the difference between the pattern set of given USB 3.0 trace and the union of pattern sets of every USB 2.0 trace. *These patterns identify extraneous observable behaviors exhibited only by USB 3.0 driver.*

When executed with USB 3.0 driver, a function driver can fail due to both these patterns — a driver dependent on unique USB 2.0 patterns could fail due the absence of a pattern while a driver not capable of handling unique USB 3.0 patterns could fail due the presence of a pattern.

4.3.4 Shrink unique pattern sets

Given the number of patterns mined from each trace was huge (as shown in Table 2), we employed the following techniques to shrink the sets of patterns by eliminating redundant patterns.

Partitioning When a trace contains a unique structural pattern, the trace will also contain numerous unique temporal patterns that involve this unique

[10]We did this to reduce the number of false positives.

structural pattern. From the perspective of detecting unique deviations, such temporal patterns do not identify deviations that are different from the deviations detected by the contained unique structural pattern. Hence, we removed such temporal patterns from the pattern sets.

Simplification By definition of the temporal patterns in Section 3, the presence of an alternation pattern (e.g. $A \overset{a}{\rightarrowtail} B$) in a trace implies the presence of corresponding eventually pattern (e.g. $A \overset{*}{\rightarrowtail} B$) in the trace. Hence, we removed eventually temporal patterns from a pattern set if their alternation counterparts were present in the pattern set.

In a similar vein, by way of construction, the existence of a complex pattern (e.g. $A \wedge B \overset{*}{\rightarrowtail} C$) implies the existence of simpler constituent patterns (e.g. $A \overset{*}{\rightarrowtail} C$ and $B \overset{*}{\rightarrowtail} C$). Hence, either complex or simple patterns can be presented without any loss of information. Favoring simplicity, we removed complex patterns from a pattern set if all of their simpler constituent patterns were present in the pattern set.

Compaction If temporal patterns of the form $A \overset{a}{\rightarrowtail} B$ and $A \overset{a}{\leftarrowtail} B$ were present a pattern set, we replaced them with a single temporal pattern of the form $A \overset{a}{\longleftrightarrow} B$ with the meaning "an event matching A will be followed by an event matching B with no intervening events that match A or B."

4.3.5 Report deviations to the developer

As the final step, for each device, a developer from USB team examined the resulting unique patterns and classified them as either benign deviations or bugs. All bugs were entered into the Windows bug repository for triaging purposes.

Reducing False Positives To curtail false positives (due to recurring benign deviations), we applied user-defined filters (e.g. ignore patterns in which `IOCTLType` field is equal to `URB_FUNCTION_SELECT_CONFIGURATION`). These filters were often based on patterns observed while examining previous test results. The user-defined filters were saved and reused while examining results from subsequent tests. In addition, we suppressed patterns that were observed in previous tests as they were classified as either benign deviations or bugs. This is depicted by the dashed line in Figure 2. We refer to these patterns as *known patterns*.

Aiding Diagnosis For all unique patterns, we identified events that matched unique patterns along with events that did not match unique temporal patterns. The developer then started diagnosing the issue starting at these events.

4.4 Evaluation

For this evaluation, we collected a pair of traces — one with USB 2.0 driver and another with USB 3.0 driver — from 14 different USB 2.0 devices and compared these traces as described in the previous section as a test of compatibility.

Based on this data set, we evaluated the effectiveness, precision, and cost of our approach. Table 1 provides the breakdown of deviations and bugs uncovered in this evaluation. Table 2 provides the breakdown of mined patterns and costs of testing as observed in this evaluation.

4.4.1 Effectiveness

In this data set, a developer from USB team identified 25 deviations as representing compatibility bugs between USB 2.0 driver and USB 3.0 driver. Of these 25 bugs, 14 bugs were based on unique structural patterns and 11 bugs were based on unique temporal patterns (involving only common structural patterns). Following is a description of few of the detected bugs.

- When USB 2.0 driver fails to service `IOCTL_INTERNAL_USB_GET_PORT_STATUS` request, the driver zeroes out bits of `PortStatus` field. However, USB 3.0 driver does not zero out these bits. This bug was based on a unique structural pattern.

- Upon completing an isochronous transfer request, USB 3.0 driver set the `status` field of the isochronous packet to `0xFFFFFFFF`. However, this was not the case with USB 2.0 driver. As in the previous case, this bug was based on a unique structural pattern.

- USB 2.0 driver completed isochronous transfer requests at `DISPATCH_LEVEL` interrupt request level. However, USB 3.0 driver completed similar requests at `PASSIVE_LEVEL` interrupt request level. This bug was based on a unique structural pattern.

- A USB device can have multiple operational configurations along with corresponding interfaces, and one of the configurations is selected while enumerating the device. When an I/O request to select a configuration for a device was submitted, USB 3.0 driver failed to communicate the corresponding interface in its response. This deviation was based on a unique temporal pattern with `interfaceHandle` field (attribute) remaining unchanged across the two events supporting the pattern.

- When a USB device is not is use, its function driver can notify the USB driver that the device is idle and the device can be suspended or put in low power state. Upon completing such an I/O request corresponding to such a notification, USB 3.0 driver did not change `PendingReturned` field in the IRP. This deviation was based on a unique temporal pattern.

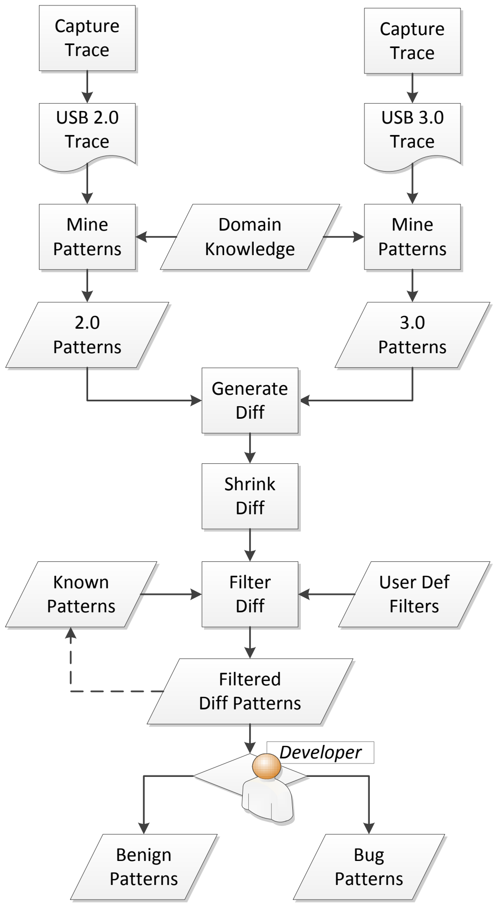

Figure 2: Work flow to perform compatibility testing using patterns-based trace comparison.

4.4.2 Precision

Our solution started out with a high number of false positives — out of 478 deviations reported for device 1, 465 deviations were false positives (see *False +ve* and *Reported* columns in Table 1). So, as we tested more devices, we collected and saved false positives to filter them out from subsequent test results (as described in Section 4.3.5). Consequently, the number of false positives dropped to less than 100 in tests corresponding to devices 2 thru 14 and to less than 10 in 8 out of 13 tests. Hence, we conjecture that the false positives reported by our approach will decrease as the number of devices used for compatibility testing increases.

In addition, we collected a total of 7(=2+3+2) user-defined filters while testing with devices 2, 5, and 14. In retrospective, few of these filters could have been injected as domain knowledge during pattern mining.

In terms of curtailing the number of deviations presented to the developer, simplification helped reduce the number of detected deviations by at least a factor of 10. Similarly, compaction helped reduce the number of simplified deviations by a factor of 2. (See columns *Simplified* and *Compaction* in Table 1.)

Revisiting the issue of number of false positives, let us consider the cost of compatibility testing. Observe that the bugs were detected from traces of devices that functioned without errors with both bus drivers. Instead, if we wanted to detect the same bugs by observing devices failing due to these bugs, then we would need to test both USB bus drivers with every USB device in the world. This would amount to testing with a fraction of more than 10 billion USB devices!! In contrast, with our approach, the developer spent less than 2 hours in many cases to examine a non-empty set of deviations resulting from a test (device); in very few cases, the developer spent up to a day to examine a set of deviations. So, comparing the cost of testing with every unique USB device to the cost of developer spending 2 hours to sift through less than 100 false positives per test, the number of false positives is insignificant.

4.4.3 Cost

While the cost of capturing a pair of traces for a device was in the order of few minutes, the time to mine quantified patterns from these traces (ranging from 200K to 500K patterns per trace) varied from 10 minutes to 100 minutes depending on the length of the trace and the average number of attributes per event (see *Time* and *Patterns* columns in Table 2). Considering only the time taken to difference pattern sets and to simplify the difference, the cost for automatically detecting deviations from a pair of traces was 2-3 minutes for unquantified patterns (see *Diff Time* columns in Table 2). However, the cost of detecting deviations based on quantified patterns was 5-12 minutes for most traces with few exceptions of 15, 20, and 45 minutes.

Given that our approach is automated and there are no alternative approaches/techniques to detect deviations that do not affect the device under test, we believe the cost of approach is reasonable.

	Number of Deviations						Number of Bugs	
Device	Known	Detected	Simplified	Compacted	Reported	False +ve	Structural	Temporal
1	0	9844	932	478	478	11+454	6/9	4/4
2*	932	2545	121	63	15	0+11	1/1	1/3
3	965	743	41	21	4	1+0	0/0	1/3
4	965	1372	67	34	2	1+1	0/0	0/0
5*	965	26118	1114	571	55	26+29	0/0	0/0
6	2141	26126	1054	541	0	0+0	0/0	0/0
7	2141	2320	84	44	0	0+0	0/0	0/0
8	2141	27804	1185	608	2	1+0	1/1	0/0
9	2141	34985	413	217	115	2+96	2/14	2/3
10	2141	51556	429	231	59	15+41	1/1	2/2
11	2141	695	35	18	0	0+0	0/0	0/0
12	2141	1372	67	34	0	0+0	0/0	0/0
13	2141	3315	122	72	24	19+4	1/1	0/0
14*	2141	9299	103	54	3	0+0	2/3	0/0

Table 1: Breakdown of deviations detected by our approach (in order). For a device/test, each row provides the number of known deviations, detected deviations, reported deviations, false positives, and bugs of various sorts. X+Y denotes X structural patterns and Y temporal patterns. A/B denotes A bugs were unique out of B bugs. Devices/Tests at which new filters were collected are marked with (*). Known deviations are all simplified deviations identified as difference in previous tests.

4.4.4 Limitations

As demonstrated, the approach can only detect deviations that can be represented as structural patterns or binary linear temporal patterns both in unquantified and quantified forms. However, this limitation can be addressed by mining and using richer patterns, e.g. longer temporal patterns.

4.5 Threats To Validity

While the above results are promising, they are based on a single experiment. Further, we neither considered nor controlled the effects of various latent factors on the experiment. Few latent factors are the size of the interface (e.g. number of functions), data flow properties at the interface, value domains of the data consumed/produced by the interface, and (non-)existence of a well-defined protocol governing the observable behavior at the interface. Also, from the available data, it is hard to discern if and to what extent did the effectiveness of the approach depend on the domain knowledge. More experimentation will help (in)validate these threats.

5 Related Work

The number of efforts in the space of software testing is huge — even to fairly cite a few efforts here — to the extent that there are venues dedicated to software testing, e.g. ICST [4], ISSTA [5], TAP [6]. Most of these efforts rely on software tests that embody well-defined expected outcomes to automatically provide conclusive test verdicts. In comparison, our approach relies on execution traces of both the system under test and the reference implementation to automatically detect a class of behavioral (and possibly performance) deviations. In other words, our approach automatically hoists observable behaviors in the reference implementation as well-defined expected outcomes.

In terms of using event sequence patterns as a core idea to enable software testing, Dallmeier et al. [10] used method call sequences as trace features to predict and localize defects by comparing traces from passing and failing executions of test cases. However, they employed method call sequences based on n-grams [18] while we used patterns spanning across events. In terms of fault localization, Dallmeier et al. use differing method call sequences to identify and rank likely defective classes while we present events that match deviating patterns as likely symptoms.

Recently, Beschastnikh et al. [9] used unquantified temporal invariants/patterns satisfied by logs to automatically construct a graph model of a system. In comparison, we have considered both unquantified and quantified variants of temporal patterns to model behaviors captured in traces.

Beyond software testing, there have been numerous efforts [14, 15, 20, 21, 22, 23, 25] to mine various features from system logs and then monitor live systems for absence of these features. Similar to software testing efforts, few of these efforts have used n-grams and state machines as trace features while other efforts have employed features based on statistical properties of traces such as relative frequency and correlation of events. Ignoring the specific classes of patterns and the corresponding mining techniques, our effort is similar to these efforts in terms of using patterns and pattern mining techniques as features and feature extraction techniques, respectively.

In a similar vein, Barringer et al. used human specified quantified (parameterized) temporal patterns with logs to enable postmortem runtime verification of flight software for NASA's recent Mars rover mission [8]. In comparison, we use automatically mined quantified temporal patterns to detect both the presence of new behaviors and the absence of previously existing behaviors when testing (compatibility of) programs.

In terms of trace comparison, data mining community has proposed and used various edit distances (e.g. Hamming, Levenshtein) and sequence-based patterns to compare sequences [11]. Miranskyy et al. [19] identified differences between traces by iteratively differencing various inter-event abstractions of traces (e.g. set of function calls, caller-callee relation, sequence of function calls). In comparison, our approach is similar to these efforts with the difference being the choice of class of structural and temporal patterns [17] used to abstract and compare traces.

While we were pursuing this effort, Yang and Evans [24] also explored the use of temporal patterns (properties) observed in program logs to identify behavioral differences between programs. Besides the operational differences in terms of the underlying mining algorithms and related cost-precision trade-offs, they mined nine types of unquantified patterns to characterize the traces in their evaluation while we used only four types of quantified patterns. In addition, we also devised a simple yet efficient feedback based work flow to effectively deal with false positives.

6 Future Work

Traditional regression testing relies on passing/failing of existing tests. It does not detect behavioral deviations that do not affect the outcome of tests. However, with execution traces of regression tests, our technique can be applied to detect such behavioral deviations. Also, in case of failing executions, our approach can be used to detect deviations between passing and failing executions to aid fault localization and debugging of failures.

Given the effectiveness of the proposed approach to test compatibility between USB bus drivers, we conjecture that the approach can be used in the context of other kinds of software, e.g. desktop and web applications, libraries (similar to [10]). Consequently, it will be interesting to explore the generality of the approach. Also, it will interesting to identify the characteristics of the software under test that make the approach effective, e.g. will the approach be effective only when the interface level interaction with the software under test can be described by a protocol?

For USB compatibility testing, we used a notion of trace similarity based on *all* temporal patterns satisfied by traces (while considering limited sorts of event abstractions). While this notion sufficed for short USB traces (most often less than 10,000 events), it might not suffice for longer traces, e.g. traces with more than 100,000 events. Depending on the application scenario, the cost of mining patterns required by this notion of trace similarity can be prohibitively high. In such cases, the cost can be reduced by adopting various constraints (e.g. limiting number of event abstractions) to identify a small number of patterns (e.g. only frequent patterns) as opposed to all patterns. Consequently, we will need new notions of trace similarity that are sensitive to these constraints and are helpful in reasoning about traces.

In this effort, equality constraint was used to relate attributes and values while defining events and event abstractions. However, many scenarios may benefit from a richer set of constraints (e.g. $<, \leq$) in terms of characterizing the traces both succinctly and accurately using patterns. Such scenarios can be enabled by extending the definition of event abstractions described in Section 3 and [17] with richer constraints. One of the key challenges with such extensions will be to deal with the explosion of event abstractions.

The proposed approach relies on sets of binary linear temporal patterns. However, it could be trivially modified to use sets of longer linear temporal patterns or sets of small finite state machines that collectively describe the event patterns in traces.[11] In such a setting, it would be interesting to explore state machine mining techniques (similar to [7]) that can admit event abstractions as described in Section 3.

As more and more systems embrace service-oriented architecture and operations engineers rely on log analysis as a primary tool to reason about runtime behavior of such systems, the need for online techniques for anomaly detection will increase. While we have explored an approach to off-line anomaly detection by comparing sets of patterns mined from traces, it would be interesting to extend the proposed approach and explore new approaches for online anomaly detection.

Moving beyond anomaly detection, the set of structural patterns and linear temporal patterns satisfied by traces can be used detect similarity between traces and answer clustering and classification questions — *Can a set of traces be partitioned into subsets of similar traces? Can a trace be classified as belonging to a specific set of a partition of traces?* These questions can be trivially answered by using patterns observed in traces as features and employing existing clustering

[11] N-ary linear temporal patterns can be easily constructed by combining m-ary linear temporal patterns under certain rules of combination and then checked against traces to calculate their statistical properties.

algorithms such as hierarchical clustering or k-means clustering [11, 12]. At this time, we are exploring this direction to enrich recommendation systems by using logs and to improve test prioritization/selection; thus far, the initial results have been promising.

7 Summary & Conclusion

We have proposed an approach to test compatibility between two programs with identical published interfaces in terms of their observable behaviors at these interfaces — test for both the absence of previously observed behaviors and the presence of previously unobserved behaviors. The approach abstracts traces of observable behaviors as sets of patterns and these sets are compared for differences. Existing pattern mining techniques are used to abstract traces into sets of structural patterns and linear temporal patterns. We have demonstrated the effectiveness of the approach by using it to test compatibility between USB 3.0 and USB 2.0 bus drivers in Windows 8 and identifying 25 compatibility bugs.

Based on this experience, we believe that structural and temporal patterns based trace abstraction is an invaluable tool to tackle common yet relevant software development and maintenance problems involving trace/log data.

Acknowledgments

We thank Randy Aull, Robbie Harris, Jane Lawrence, and Eliyas Yakub from the Windows USB team in Microsoft for their help and invaluable feedback during this effort. We thank Tom Ball and Peter Shier for initiating this effort. Lastly, we thank Ankush Desai and Jithin Thomas for their feedback on the manuscript. During this effort, Pradip Vallathol was a developer at Microsoft Research, India.

References

[1] Event tracing for windows. `http://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx`, 2000.

[2] Building robust USB 3.0 support. `http://blogs.msdn.com/b/b8/archive/2011/08/22/building-robust-usb-3-0-support.aspx`, 2011.

[3] Tark: Mining linear temporal rules. `http://research.microsoft.com/en-us/projects/tark/`, 2011.

[4] International Conference on Software Testing, Verification and Validation. `http://icst2012.soccerlab.polymtl.ca/`, 2012.

[5] International Symposium on Software Testing and Analysis. `http://crisys.cs.umn.edu/issta2012/`, 2012.

[6] Tests and Proof. `http://lifc.univ-fcomte.fr/tap2012/`, 2012.

[7] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002.

[8] Howard Barringer, Alex Groce, Klaus Havelund, and Margaret Smith. Format analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7, 2010.

[9] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE'11: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of Software Engineering*, 2011.

[10] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *Proceedings of 19th European Conference on Object-Oriented Programming*, 2005.

[11] Guozhu Dong and Jain Pei. *Sequence Data Mining.* Springer Science+Business Media, LLC, 2007.

[12] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification.* Wiley-Interscience, 2000.

[13] Martin Fowler. Public versus published interfaces. *IEEE Software*, 19:18–19, 2002.

[14] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of Ninth IEEE International Conference on Data Mining*, 2009.

[15] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of ACM/IEEE 32nd International Conference on Software Engineering*, 2010.

[16] Edmund M Clarke Jr., Orna Grumberg, and Doron A Peled. *Model Checking.* The MIT Press, 1999.

[17] David Lo, Ganesan Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Mining quantified temporal rules: Formalism, algorithms, and evaluation. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, 2009.

[18] Christopher D. Manning and Hinrich Schuetze. *Foundations of Statistical Natural Language Processing.* The MIT Press, 1999.

[19] A.V. Miranskyy, M. Davison, N.H. Madhavji, M. Wilding, M.S. Gittens, and D. Godwin. An iterative, multi-level, and scalable approach to comparing execution traces. In *Proceedings of the the Sixth joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering (FSE)*, 2006.

[20] Adam J. Oliner, Alex Aiken, and Jon Stearley. Alert detection in system logs. In *Proceedings of Eighth IEEE International Conference on Data Mining (ICDM)*, 2008.

[21] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Salsa: Analyzing logs as state machines. In *Proceedings of First USENIX Workshop on the Analysis of System Logs (WASL)*, 2008.

[22] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the 17th USENIX conference on System administration*, 2003.

[23] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[24] Jinlin Yang and David Evans. Automatic inference and effective application of temporal specifications. In David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu, editors, *Mining Software Specifications: Methodologies and Applications*, chapter 8. CRC Press, 2011.

[25] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. In *Proceedings of the First ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.

A Patterns and Log Analysis

While log analysis has been and is a common tool used by system administrators and operation engineers, it is lately becoming a common tool for support engineers and even developers to diagnose errors in remote applications, e.g. server running at a customer site.

Errors in server applications can be transient (e.g. they may be dependent on the workload) and the exact point in time at which the error occurred is hard to track. Further, the cause of the error may be temporally distant from the error; hence, the execution history of the application is required to diagnose such errors. Under these circumstances, traditional debugging would amount to running a good chunk of the application under the supervision of and interruption from a debugger. This can considerably hamper the performance of live systems (and possibly suppress the error).

As an alternative, support engineers work with customers to collect various logs from a failing system. Then, engineers analyze these logs to identify log entries corresponding to both symptoms and possible root causes of failures. Logs collected from live systems are often huge, and the task of analyzing such logs turns into "a search for the needle in the haystack".[12]

All of the above observations, reasons, and approaches are equally applicable to diagnosing errors in cloud applications.

This situation can be addressed by leveraging machine learning techniques such as clustering and classification [12]. Specifically, given a partition of logs based on similar errors (causes), classification can be used to detect likely errors (causes) in a new log by classifying the log as belonging to a set of the partition. In the presence of set of unpartitioned logs, clustering can be used to identify a partition of logs based to similar errors (causes). As with compatibility testing, trace comparison can be used to detect anomalies by comparing logs from successful executions to logs from failing executions.

To enabled the above solutions, features need to be extracted from logs. Most often, events along with their statistical properties both in isolation (e.g. presence/absence, frequency) and in combination with other events (e.g. relative frequency, correlation) can be used as features. In the same vein, structural patterns along with their statistical properties as described in Section 3.1 can be used as features of logs.

While features independent of the ordering of events are cheaper to extract and easy to understand, they most often have relatively lower discriminatory power than features involving the ordering of events. As the accuracy of machine learning techniques depends on the discriminatory power of features, in the context of log analysis, it would be better to consider features that are based on ordering of events and are easy to comprehend. In this vein, linear temporal patterns along with their statistical properties as described in Section 3.2 can be used as features of logs.

By representing logs as a set of patterns, distance measures such as Jaccard distance [12] can be used to measure distances between logs (and sets of logs) for the purpose of clustering and classifying logs.

In short, both structural patterns and linear temporal patterns used in this exposition can be used to enable automated log analysis via traditional machine learning techniques such as classification and clustering.

[12]Of course, we are assuming that the engineer knows the needle he/she is looking for. In many cases, this is not true.

	USB 2.0 Traces					USB 3.0 Traces					Diff Time	
Id	No. of Events	Unquant Mining		Quant Mining		No. of Events	Unquant Mining		Quant Mining		Unquant Patterns	Quant Patterns
		Time	Patterns	Time	Patterns		Time	Patterns	Time	Patterns		
1	243	94.75	22108	673.77	208084	411	84.51	20468	690.07	205212	106.14	393.52
2	163	40.26	20804	827.47	205592	211	39.41	22172	851.41	207244	106.27	404.59
3	163	31.92	18972	843.49	205064	2363	67.57	24812	3305.26	506772	105.50	635.97
4	*18017*	*102.32*	23112	4308.67	504332	183	18.10	15628	1583.36	479052	103.39	481.53
5	535	27.94	20542	1845.06	430108	763	28.16	24068	1129.92	406492	142.88	663.97
6	1329	51.52	*24542*	1951.91	437012	1779	29.84	21980	1508.28	402836	141.51	744.67
7	153	29.90	13900	1826.59	472732	239	26.36	15044	1091.75	298080	102.20	413.46
8	187	28.72	17092	1667.78	473448	*13027*	*89.39*	25756	*5121.93*	410092	*170.66*	*2729.10*
9	1009	27.35	15236	1719.23	468040	1441	37.14	*26606*	1247.51	438384	147.17	900.30
10	12511	83.35	23918	*5842.77*	436108	177	21.47	20592	1141.77	512104	104.85	1214.38
11	183	31.16	15044	1213.28	297280	153	14.33	13900	1061.33	473540	103.88	426.54
12	1751	34.21	20334	1628.79	429100	191	18.26	17092	1099.02	479736	103.12	452.37
13	181	22.32	20726	1382.10	*536980*	793	19.06	15236	1472.68	475920	105.16	541.82
14	383	22.92	19496	1439.67	362416	9411	56.23	23112	3100.73	*513532*	105.77	763.63

Table 2: Data about mining and diffing traces in our experiment. Each row provides data for both a USB 2.0 trace and a USB 3.0 trace of a device. For each trace, its length (in events), mining time (in seconds), the number of mined patterns of various sorts, and pattern differencing time (in seconds) are provided. The largest number in each column is italicized.