Compatibility Testing using Patterns-based Trace Comparison

Venkatesh-Prasad Ranganath
Kansas State University
rvprasad@cis.ksu.edu

Pradip Vallathol
University of Wisconsin-Madison
pradip16@cs.wisc.edu

Pankaj Gupta
Microsoft Corporation
Redmond, USA
pankaj.gupta@microsoft.com

ABSTRACT

When composing a system from components, we need to ensure that the components are compatible. This is commonly achieved by components interacting only via published and well-defined interfaces. Even so, it is possible for client components to learn about and depend on unpublished yet observable behaviors of components. To identify and support these situations, compatibility testing should uncover such observable behaviors.

As a solution, we propose a patterns-based approach to test compatibility between programs in terms of their observable behaviors. The approach compares traces of behaviors observed at identical published interfaces of programs and detects incompatibilities stemming from both the presence of previously unobserved behaviors and the absence of previously observed behaviors. The traces are compared by transforming them into sets of structural and binary linear temporal patterns.

During Windows 8 development cycle, we applied this approach to test compatibility between USB 2.0 and USB 3.0 bus drivers. Upon testing 14 USB 2.0 devices that were functioning without errors with both USB bus drivers, we uncovered 25 previously unknown incompatibilities between the bus drivers.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids; Testing tools*

Keywords

Observable Behaviors, Patterns, Sequential Data, Testing, Trace Similarity

ASE'14, September 15-19, 2014, Vasteras, Sweden.

ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642942.

1. INTRODUCTION

In its simplest 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 to 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 are mutually compatible. To this end, most software take proactive measures to allow only the use of compatible components.

For example, consider the scenario of a user upgrading her favourite web browser. Upon upgrading her browser from version V_1 to V_2, the installer program checks if any of the previously installed browser plug-ins are incompatible with version V_2 of the browser. For each incompatible plug-in, the installer will offer to upgrade the plug-in to a compatible newer version (if available) or to disable the plug-in.

The incompatibilities (deviations) between the browser and the plug-in can arise from syntactic or semantic changes to the published interfaces of the browser.[2]

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

As for incompatible 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 the plug-in against version V_2 of the browser. For most semantic changes, the plug-in will need to be executed with version V_2 of the browser to exercise the change and uncover the incompatibility. 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 by version V_2 of the browser. On the other hand, if a decision in plug-in A depends on the value *Monday*, then plug-in A will most likely exhibit a new behavior when executed with version V_2 of the browser; possibly, leading to malfunction of both plug-in A and the browser.

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

[1]The notion of published interfaces was introduced by Martin Fowler [12].
[2]We will use the terms deviation and incompability interchangeably.

```
s.q = c;
f(&s);
s.q = c;
g(&s);
```

(a) Correct

```
s.q = c;
f(&s);

g(&s);
```

(b) Incorrect

Figure 1: Ways to use functions f and g from library L.

For example, consider a C language library L that publishes two functions f and g that accept a pointer to a structure with a field q as their only argument and require field q to contain value c upon invocation. (The correct way to use these functions is shown in Figure 1a.) 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 (about the implementation of f) can be exploited to optimize clients of version V_1 of L — when invoking f and g in sequence with a common structure argument s, assign c to $s.q$ and then invoke f and g without any intervening assignment of c to $s.q$ (as shown in Figure 1b). However, in a subsequent version V_2 of L, if the implementation of f modifies $s.q$, then clients optimized against version V_1 of L will fail when executing with version V_2 of L.

A Real-world Example. During the development of Windows 8, the USB team built a new USB 3.0 bus driver from scratch to support USB 3.0 protocol in Windows 8 [5]. Since USB 3.0 protocol is backward compatible with USB 2.0 protocol, the USB 3.0 bus driver needed to support USB 2.0 devices along with their device drivers that were built against the existing USB 2.0 bus driver. Hence, when servicing USB 2.0 devices, the USB 3.0 bus driver was required to mimic the observable behavior of USB 2.0 bus driver. In this context, an example of incompatibility was that the USB 3.0 bus driver could complete isochronous transfer requests at `PASSIVE_LEVEL` interrupt request level when the USB 2.0 bus driver would always complete such requests at `DISPATCH_LEVEL` interrupt request level. While such incompatibilities may not affect any of the USB 2.0 devices used to test the USB 3.0 bus driver, they could affect untested USB 2.0 devices, which could be numerous given the vast number of unique USB devices in the world.

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

Proposed Approach

To address the above real-world scenario, we devised a simple differential approach to compatibility testing. 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 (until any failure occurs). These interactions are traced as observable behaviors and the resulting traces are compared to detect incompatibilities resulting from both the presence of previously unobserved behaviors and the absence of previously observed behaviors.

In addition, since the approach uncovers issues independent of the success (or failure) of the execution yielding the traces, it can uncover issues that can affect yet unexplored executions.

As for comparing traces in our approach, 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., union, intersection, and difference. This is the key feature of our approach.

In terms of guarantees, our approach is *unsound* — every detected incompatibility need not be a bug. It entails human effort to examine detected issues and classify them as either benign issues or bugs. On the other hand, our approach is *complete* w.r.t. given traces — every incompatibility that can be represented by the pre-defined classes of patterns (used to abstract traces) and are present in the given traces will be detected.

We successfully used our approach to test compatibility between USB 2.0 and USB 3.0 bus driver stacks in Windows 8, during the development of Windows 8. When used with an appropriate work flow, the approach uncovered 25 previously unknown compatibility bugs in the USB 3.0 bus driver by analyzing 14 pairs of traces from 14 USB 2.0 devices that were functioning without errors with both USB bus drivers.

Contributions

The key contributions of this manuscript are as follows.

1. We propose a differential approach to compatibility testing based on patterns-based trace comparison. This approach can uncover compatibility issues stemming from both the presence of previously unobserved behaviors and the absence of previously observed behaviors. Unlike traditional testing, our approach can detect compatibility issues independent of the success and failure of program executions.

2. We demonstrate the effectiveness of the proposed approach in an industrial setting by using it during Windows 8 development cycle to test compatibility between USB 2.0 and USB 3.0 bus drivers.

3. We show that 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 how to realize it via 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

Given two programs P_r and P_t, compatibility testing checks if P_r and P_t produce identical outputs upon consuming identical inputs, i.e., $\forall x.P_t(x) = y = P_r(x)$.

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 formally 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.[3]

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., $P_r(x) = P_t(x)$) can be described by $\sim_{id,=}$ with identity function (id) as α and equality predicate ($=$) as ψ.

Similarly, we can describe compatibility testing of programs that output traces (sequences).[4] Consider programs that consume an input x and produce a trace π as output. From the above definition, given two programs P_r and P_t that output traces π_r and π_t upon consuming test input x, P_r *is* (α, ψ)*-compatible to* P_t if $\psi(\alpha(\pi_r), \alpha(\pi_t))$ holds with $P_r(x) = \pi_r$ and $P_t(x) = \pi_t$. Hence, the problem of compatibility testing based on traces reduces to the problem of trace comparison under α and ψ.

Now, consider programs that consume a sequence of inputs and produce an output trace. If we want to test such programs based solely on the sequence of outputs, then we can perform compatibility testing via trace comparison as described above. If we want to consider both the sequence of inputs and outputs, then we can perform compatibility testing as described above by conditioning the output trace as follows: when a program P consumes an input sequence $X = x_1, x_2, \ldots x_n$ and produces an output trace π, X is 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., $\pi[j] = y_k \implies (\exists i.\pi[i] = x_k \wedge i < j)$.

In summary, under a notion of similarity determined by the combination of 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. These notions of similarity use set equality as the test predicate ψ. As for transformation functions α, they are based on event abstractions and binary linear temporal patterns proposed by Lo et al. [15].

From here on, an event $e = \{a_1 \mapsto c_1, a_2 \mapsto c_2, a_3 \mapsto c_3, \ldots, a_n \mapsto c_n\}$ is a map from attributes (a_is) to values (c_js) 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 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* $\mapsto$ *"fopen"*} view of the invocation event {*fun* $\mapsto$ *"fopen", arg1* $\mapsto$ *"passwd.txt", arg2* $\mapsto$ *"r", return* $\mapsto$ *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 of e.* We refer to this notion as *event abstraction based trace similarity.*

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* $e = \{a_1 \mapsto c_1, a_2 \mapsto c_2, a_3 \mapsto c_3, \ldots, a_n \mapsto c_n\}$, $e' = \{a_1 \mapsto v_i, a_2 \mapsto c_2, a_3 \mapsto v_j, \ldots a_n \mapsto c_n\}$ *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 variables.* Consequently, we say an attribute is *quantified* if it is associated with a 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 the equality of the sets of structural patterns observed in traces.

In other words, compatibility testing based on traces can be realized with a function that transforms a trace into a set of structural patterns as the tranformation function α and set equality as the test predicate ψ.

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 contain the same set of events $\{a, b, c\}$ and will result in the same set of structural patterns.

To handle such cases, 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 [15].

Given structural patterns A (referred to as *anchor*) and B are observed in a trace,

[3]Observe that 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.

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

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 (where an event e *matches* a structural pattern C if C is an abstraction of e.) 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 both A and B are unquantified. Further, when A and B are quantified, the associated substitutions resulting from matching events should be identical, i.e., $\theta_A = \theta_B$. For example, event abstractions $\{fun \mapsto$ *"fopen", return* $\mapsto$ *0x21*$\}$ and $\{fun \mapsto$ *"fclose", arg1* $\mapsto$ *0x21*$\}$ match the pattern $\{fun \mapsto$ *"fopen", return* $\mapsto v_1\} \overset{*}{\rightarrowtail} \{fun \mapsto$ *"fclose", arg1* $\mapsto v_1\}$ under the substitution $\theta = \{v_1 \mapsto 0x21\}$. However, event abstractions $\{fun \mapsto$ *"fopen", return* $\mapsto$ *0x21*$\}$ and $\{fun \mapsto$ *"fclose", arg1* $\mapsto$ *0x23*$\}$ do not match the same pattern as there are no substitutions common to both event abstractions.

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

In other words, compatibility testing based on traces can be realized with a function that transforms a trace into a set of temporal patterns as the tranformation function α and set equality as the test predicate ψ.

3.3 Statistical Similarity

The above notions of trace similarity are insufficient when traces exhibit identical set of patterns but the patterns have different statistical properties, e.g., frequency, distance between events matching temporal patterns. In such cases, the notion of trace similarity can be extended to include the statistical properties of patterns. For example, two traces are similar if they exhibit the same set of patterns and the frequency of each pattern in both traces is greater than a given threshold.

While we explored such notions of trace similarity for performance debugging, we will not explore it further in this manuscript.

3.4 Alternative Trace Comparison Techniques

In contrast to patterns-based trace comparison described above, here are two alternative trace comparison techniques.

LCS-based Comparison: The longest common subsequence (LCS) of two traces can be used to identify the differences between traces — events that are not part of the common subsequence. However, it will fail to identify incompatibilities due to temporal orderings of events. Further, it is unclear if and how can event abstractions be effectively and efficiently considered in LCS-based comparison.

Graph-based Comparison: Similar to LCS-based comparison, two traces can be compared by diffing their *succession graphs* — nodes represent events and directed edges capture successor (or reachability) relation between events. While graph-based comparison seems similar to patterns-based comparison, it is unclear how to incorporate event abstractions into graph-based comparison.

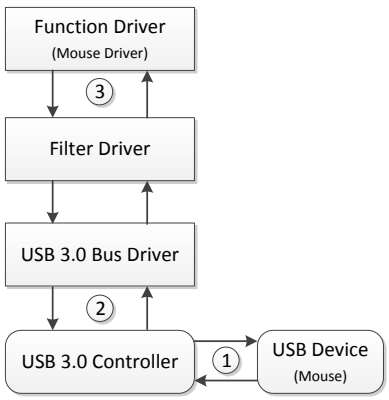

Figure 2: An illustration of how various types of WDM drivers are stacked when a USB device is plugged into a USB 3.0 port on Windows 8 PC.

4. TESTING COMPATIBILITY BETWEEN USB DRIVERS IN WINDOWS 8

In this section, we describe our experience using patterns-based trace comparison 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. These details should enable other researchers and practitioners 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 Model (WDM). This model supports the following kinds of drivers.[5] (Please refer to Figure 2 for an illustration of how various kinds of WDM drivers are connected.)

- A *bus driver* services devices that support child devices, e.g., bus controllers, adapters, and bridges. As these are necessary drivers, Microsoft 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 of the communication 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)

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

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.

4.2 Problem

As mentioned in *Building Windows 8* blog [5], 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. So, 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 USB 2.0 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 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`.[6] Hence, for compatibility, USB 3.0 driver should exhibit 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 USB 2.0 device and its function driver and check for observable failures. However, this approach is prohibitive given the vast number of unique USB devices in the world.

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 chose to test compatibility between USB drivers by the checking for equivalence of interactions between function drivers and the USB drivers (at point 3 in Figure 2), 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?*[7]

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

4.3 Solution

Our solution to this problem uses patterns-based trace comparison (described in Section 3) within a simple workflow outlined in Figure 3. In the following sections, we describe how we realized the approach via 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) [4].

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 the rest captured the arguments to these routines. We used this knowledge to combine 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. In addition, 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 these transformations, there were 361 attributes (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 in 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.

[7]Here are two other alternatives that we considered to test compatibility between USB drivers.
Approach 1 Check for equivalence of on-the-wire interactions between the USB controller and the USB device (at point 1 in Figure 2). This form of checking 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 was 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 2). 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.

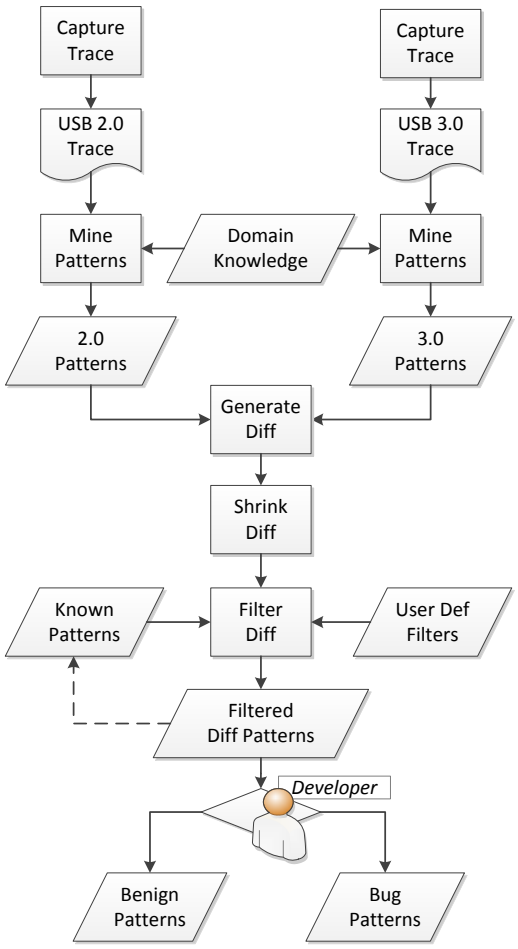

Figure 3: Work flow to perform compatibility testing using patterns-based trace comparison.

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, i.e., two fields with different semantics can use the same data type. To curb this noise, based on domain knowledge, by way of configuration, we considered only 17 data flows between 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 [6]. For details about the pattern mining algorithms implemented in Tark, please refer to [15].

While the patterns are mined from traces of every device, the domain knowledge about attributes is elicited once for the USB domain (and merely reused when patterns are mined from traces).

4.3.3 Calculate unique patterns (deviations)

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

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 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 such patterns while a driver not capable of handling unique USB 2.0 patterns could fail due the presence of such patterns.

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 can contain numerous unique temporal patterns that involve this unique 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 ignored such temporal patterns.

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 only complex patterns or only simple patterns can be presented without any loss of information. Favoring simplicity, we removed complex patterns from a pattern set if all of their simple constituent patterns were present in the pattern set.

Compaction. If a pattern set contains patterns of the form $A \overset{a}{\rightarrowtail} B$ and $A \overset{a}{\leftarrowtail} B$, we replaced them with a single 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 matching either 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. Subsequently, detected bugs were entered into the Windows bug repository.

Reducing False Positives. As the approach is conservative (in considering all possible patterns), benign deviations can lead to false positives. For example, unique patterns stemming from fields/values that do not impact the behavior of function drivers will lead to false positives (due to incomplete domain knowledge).

To curtail such false positives, 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 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 already classified as either benign deviations or bugs. This is depicted by the dashed line in Figure 3. We refer to these patterns detected in previous tests as *known patterns*.

Aiding Diagnosis. For all unique patterns, we identified matching events. In addition, for unmatched temporal patterns, we identified events that matched the anchor of the temporal pattern. The developer then started diagnosing the issue starting at these matching events.

4.4 Evaluation

For this evaluation, we collected 14 pairs 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 to test for 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 previously unknown incompatibilities 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 the 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 all 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 sets the `status` field of the isochronous packet to `OxFFFFFFFF`. However, this was not the case with USB 2.0 driver. 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 these 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 in 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 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.

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.) To alleviate this problem, as we tested more devices, we collected and saved false positives and then filtered them out of subsequent test results (as described in Section 4.3.5). Consequently, the number of false positives dropped to less than 100 in subsequent tests corresponding to devices 2 through 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 reduced the number of detected deviations by at least a factor of 10. Similarly, compaction reduced the number of simplified deviations by a factor of 2. (See *Simplified* and *Compaction* columns in Table 1.)

Revisiting the issue of number of false positives, consider the cost of compatibility testing. Observe that the bugs were detected from traces of devices that functioned without errors with both bus drivers. 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 unique USB device in the world. This would amount to testing with thousands of unique 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 few cases, the developer spent up to a day to examine a set of deviations. So, when we compare

	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 simplified deviations from previous tests.

the cost of testing with every unique USB device (including the cost of tracking and procuring devices) to the cost of developer spending 2 hours to sift through test verdicts (with less than 100 false positives per test), the number of false positives becomes a non-issue.

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 (on a 16 core server with 32GB of RAM) depending on the length of the trace and the average number of attributes per event. (See *Time* and *Patterns* columns in Table 2.) The time taken to difference pattern sets and to simplify and report the difference for a pair of traces was 2-3 minutes for unquantified patterns and 5-12 minutes for quantified patterns with few exceptions of 15, 20, and 45 minutes. (See *Diff Time* column in Table 2.)

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

4.4.4 Comparison

At the time of our experiments, none of the related approaches discussed in Section 5 could be immediately applied in our setting. Hence, we could not directly compare our approach with the related approaches.

Regression testing can uncover the reported incompatibilities by testing USB 3.0 driver against USB 2.0 driver with every unique USB device in the world under all possible usage scenarios. However, since test labs have access to a subset of all the unique USB devices in the world (and are not aware of all possible usage scenarios), regression testing in its naive form — parity of device failures on both drivers — will not suffice. This observation is supported by the fact that our approach detected incompatibilities in the version of USB 3.0 driver that was regression tested and all the issues reported from regression testing were fixed.

Nevertheless, regression testing can be adapted to uncover the reported incompatibilities by employing test criteria that are based on observable behaviors. Such test criteria can be defined using the approaches to trace comparison described in Section 3.

4.4.5 Limitations

As demonstrated, the approach can detect only deviations that can be captured as structural patterns or binary linear temporal patterns. However, this limitation can be addressed by mining and using richer patterns, e.g., longer linear temporal patterns or finite state machines.

4.5 Threats To Validity

Since the approach was effectively applied to one non-trivial problem instance, the approach and its effectiveness cannot be immediately generalized to other problem instances.

In our experiment, the effects of various latent factors such as the size of the interface (e.g., number of functions), data flow properties at the interface, domains of values consumed and produced by the interface, and existence of a well-defined protocol governing the observable behavior at the interface were not considered. Hence, the influence of the latent factors on the effectiveness of the approach cannot be immediately ruled out.

However, both of the above concerns can be addressed by more experimentation.

4.6 Lessons Learned

If domain knowledge is available, use it. As the results suggest, the approach would have provided useful results independent of the domain knowledge. However, the number of false positives would have been high in such results and, consequently, the cost of identifying anomalies would have been high. Also, domain knowledge helped the underlying algorithm to focus on relevant patterns and reduce computation costs.

If a feedback loop can be established, set it up. With user feedback (filters and decisions), the approach was able to filter out both irrelevant and previously seen results and bubble up previously unseen deviations; hence, helping the user focus on relevant deviations.

Presentation matters. While the sorts of patterns presented as deviations were simple, few presentation tweaks

Id	USB 2.0 Traces					USB 3.0 Traces					Diff Time	
	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.

based on the information needs of the user helped reduce the number of reported deviations (without any loss of information). In turn, this made the approach and the results more accessible to the user.

5. RELATED WORK

The number of efforts in the space of software testing is huge — it is impossible to to fairly cite a few efforts here — to the extent that there are venues dedicated to software testing, e.g., ICST [1], ISSTA [2], TAP [3]. 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. [9] 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 [16] while we used patterns spanning across non-consecutive events [15]. In terms of fault localization, Dallmeier et al. used differing method call sequences to identify and rank likely defective classes while we presented events that match deviating patterns as likely symptoms/causes.

Recently, Beschastnikh et al. [8] 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 [13, 14, 18–21, 23] 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 monitoring efforts have used n-grams and state machines as trace features while other monitoring 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 [7]. In comparison, we use automatically mined quantified temporal patterns to detect both the presence of new behaviors and the absence of old 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 [10]. In software community, Miranskyy et al. [17] 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 structural and temporal patterns [15] used to abstract and compare traces.

While we were pursuing this effort, Yang and Evans [22] 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 four types of unquantified and quantified temporal patterns along with structural patterns. In addition, we also used a simple yet efficient feedback based work flow to effectively deal with false positives.

6. POSSIBILITIES

Traditional regression testing relies on pass-fail outcome of existing tests. It does not detect behavioral deviations that do not affect the outcome of tests. However, with execution traces from regression tests, our approach can be employed to detect such behavioral deviations. Further, similar to alternative approaches mentioned in Section 5, the approach

can be used to detect deviations between passing and failing instances of a test to aid fault localization and debugging of failures.

When responding to a customer incident, support engineers often sift through system/application logs to identify if the reported incident is similar to any previously observed incident; possibly, resolved incidents. In such scenarios, structural and temporal patterns observed in logs can be used as features with off-the-shelf clustering and classification algorithms (e.g., hierarchical clustering, n-nearest neighbor classification [10, 11]) to automatically recommend a ranked list of similar incidents. Consequently, organizations can harness institutional knowledge and decrease turn around time for customer incidents.

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, Jithin Thomas, and other anonymous reviewers for their feedback on the manuscript. During this effort, Pradip Vallathol was a Developer at Microsoft Research, India, and Venkatesh-Prasad Ranganath was a Researcher at Microsoft Research, India.

7. REFERENCES

[1] International Conference on Software Testing, Verification and Validation (ICST). `http://icst2012.soccerlab.polymtl.ca/`.

[2] International Symposium on Software Testing and Analysis (ISSTA). `http://crisys.cs.umn.edu/issta2012/`.

[3] Tests and Proof (TAP). `http://lifc.univ-fcomte.fr/tap2012/`.

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

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

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

[7] H. Barringer, A. Groce, K. Havelund, and M. Smith. Format analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7, 2010.

[8] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Foundations of Software Engineering (FSE)*, 2011.

[9] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[10] G. Dong and J. Pei. *Sequence Data Mining*. Springer Science+Business Media, LLC, 2007.

[11] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley-Interscience, 2000.

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

[13] Q. Fu, J. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *International Conference on Data Mining (ICDM)*, 2009.

[14] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *International Conference on Software Engineering (ICSE)*, 2010.

[15] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Mining quantified temporal rules: Formalism, algorithms, and evaluation. In *Working Conference on Reverse Engineering (WCRE)*, 2009.

[16] C. D. Manning and H. Schuetze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[17] A. Miranskyy, M. Davison, N. Madhavji, M. Wilding, M. Gittens, and D. Godwin. An iterative, multi-level, and scalable approach to comparing execution traces. In *Foundations of Software Engineering (FSE)*, 2006.

[18] A. J. Oliner, A. Aiken, and J. Stearley. Alert detection in system logs. In *International Conference on Data Mining (ICDM)*, 2008.

[19] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Salsa: Analyzing logs as state machines. In *Workshop on the Analysis of System Logs (WASL)*, 2008.

[20] Y. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *USENIX Conference on System administration (LISA)*, 2003.

[21] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Detecting large-scale system problems by mining console logs. In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[22] J. Yang and D. Evans. Automatic inference and effective application of temporal specifications. In D. Lo, S.-C. Khoo, J. Han, and C. Liu, editors, *Mining Software Specifications: Methodologies and Applications*, chapter 8. CRC Press, 2011.

[23] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *European Conference on Computer Systems (EuroSys)*, 2006.