View metadata, citation and similar papers at core.ac.uk
brought t
provided by Elsevier -

ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT®

Science of Computer Programming 53 (2004) 143–164

Science of Computer Programming

www.elsevier.com/locate/scico

# Strider: a black-box, state-based approach to change and configuration management and support

Yi-Min Wang[a,*], Chad Verbowski[a], John Dunagan[a], Yu Chen[b], Helen J. Wang[a], Chun Yuan[b], Zheng Zhang[b]

[a] *Microsoft Research, Redmond, USA*
[b] *Microsoft Research Asia, Beijing, China*

Received 27 December 2003; accepted 27 December 2003

Available online 2 July 2004

**Abstract**

We describe a new approach, called Strider, to Change and Configuration Management and Support (CCMS). Strider is a black-box approach: without relying on specifications, it uses state differencing to identify potential causes of differing program behaviors, uses state tracing to identify actual, run-time state dependencies, and uses statistical behavior modeling for noise filtering. Strider is a state-based approach: instead of linking vague, high level descriptions and symptoms to relevant actions, it models management and support problems in terms of individual, named pieces of low level configuration state and provides precise mappings to user-friendly information through a computer genomics database. We use troubleshooting of configuration failures to demonstrate that the Strider approach reduces problem complexity by several orders of magnitude, making root-cause analysis possible.
© 2004 Elsevier B.V. All rights reserved.

## 1. Introduction

Change and Configuration Management (CCM) refers to the task of monitoring configuration changes and maintaining systems in healthy configuration states. Change and Configuration Support (CCS) refers to the task of performing troubleshooting and repairs to bring systems back to healthy configuration states, after configuration failures have occurred.

* Corresponding address: Microsoft Research, Systems Management Group, Redmond, USA.
*E-mail address:* ymwang@microsoft.com (Y.-M. Wang).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2003.12.009

CCMS of computer platforms with large install bases and large numbers of available third-party software packages have proved to be daunting tasks [16]. Ideally, a white-box approach could greatly simplify the tasks: the developers of every OS component and every application would accurately and fully specify the set of configuration data that their programs use, the health invariants that subsets of these configuration data must satisfy, and the dependencies among the OS components and applications. Such information could then be used to compose machine-wide dependency information and golden configuration states [22] (or ideal states), in which all OS components and applications function correctly.

In practice, there are several difficulties. First, the large number of possible hardware and software combinations and their dependencies and interactions make it difficult to fully specify golden states for individual machines because each machine has essentially a unique, customized configuration. Also, studies have shown that [14,21] it is very difficult to declare and maintain accurate and conflict-free cross-application dependencies. Second, as a user makes intended changes to the configuration settings of her machine, the machine's configuration state moves from one good state to another, so there may not be a golden state in which the machine must stay. Third, the "correct operations" of programs are often defined with respect to user-expected services. An incorrect program behavior perceived by one user may look perfectly fine to another user, so absolute goldenness is sometimes hard to define. Finally, white-box specifications will not be available for the majority of existing legacy applications.

This motivates the Strider black-box approach to CCMS. In Strider, we use state differencing (or *diffing*) to identify deviations from known-good configuration state, use state tracing to discover relevant, run-time state dependency information, and use statistical behavior modeling for noise filtering. At the core of Strider is a *computer genomics database* [10] that can accommodate specifications based on white-box knowledge as well as those derived from black-box experiments using state diffing and tracing.

The main contributions of Strider are as follows, which also serves as the outline of the paper. First, we identify three *Strider principles* as the key to handling complexity in CCMS: *State-Based Analysis, Attack the Mess with the Mass*, and *Complexity–Noise Filtering*. Applying these principles allows us to decompose seemingly intractable CCMS problems into sub-problems, each of which is solved by a *Strider component*. Second, we introduce *Strider processes* as conceptual uses of various combinations of the Strider components to solve different problems, including troubleshooting, configuration certification, and change audit.

Third, we describe the *Strider toolkit* that implements the Strider components. Finally, we present the Strider troubleshooter that strings together components from the toolkit to implement the troubleshooting process. We evaluate the performance of the troubleshooter and discuss its limitations. To simplify our presentation, we will focus our discussion on a particular type of important configuration data—the Windows Registry [20], which provides hierarchical persistent storage for named, typed entries. The principles and techniques are generally applicable to other types of configuration stores and other platforms; we will discuss such applications at the end of the paper.

## 2. The Strider principles

We begin by describing the three Strider principles, and use troubleshooting of configuration failures (i.e., errors resulting from mis-configuration) as the primary example to illustrate problem decomposition.

### 2.1. State-based analysis

A configuration failure occurs when a program modifies a piece of configuration data and, some time later, that same program or another program reads that modification and experiences a persistent failure that cannot be repaired by application restart or machine reboot. The failure can exhibit symptoms in the form of a program crash, program hang, error dialog box, or simply not delivering user-expected service.

In particular, configuration problems caused by data sharing through persistent stores present a great challenge. Such shared stores may serve many purposes: they may contain system-wide resources that are naturally shared by all applications (e.g., the file system); they may allow applications installed at different times to discover and integrate with each other to provide a richer user experience; they may allow users to install new applications to customize default handlers or appearances of existing applications; they may allow individual applications to register with system services to reuse base functionalities; or they may allow individual components to register with host applications that provide an extensibility mechanism (e.g., toolbars in browsers).

Computer users (or support engineers) typically perform *symptom-based analysis* to troubleshoot configuration problems. Based on their knowledge and past experiences with similar problems, the users try to search the Web or a support-article database using search strings constructed in an ad hoc way in an attempt to describe the symptoms. Such search is highly imprecise and often results in a large number of irrelevant articles. Furthermore, there is no guarantee that the repair actions suggested in these articles would actually modify the configuration data relevant to the failure in question.

In Strider, we propose *state-based analysis* as the primary approach to troubleshooting. Given a configuration failure, we represent it as a high dimensional state vector of all configuration data. For example, Windows XP machines typically have around 200,000 Registry entries; a configuration failure due to a faulty entry can be represented as a 200,000-dimensional vector that contains the entry. The main challenge is to narrow down the problem to that entry.

To reduce the dimensionality to the level that can be handled by humans, we develop mechanical techniques to exclude those entries that are irrelevant to the current failure, and develop statistical techniques to filter out those entries that are relevant but less likely to be the root cause. Once we narrow down the potential candidates to a small subset, we perform a precise lookup in a computer genomics database for each entry in the subset to identify potential fixes. Optionally, we can use the imprecise symptom descriptions at this later stage to help rank the importance of the candidates by matching the descriptions against information retrieved from the database.

Given the name of a Registry entry, the genomics database answers the following two questions:

(1) *What is the function of this entry?* This provides any higher level information that can help users understand the function of the entry. It may associate the entry with the application(s) or OS component(s) that are mainly responsible for updating it and therefore can potentially be used to correct problems caused by the entry. It may also identify the entry as "noise" [5] from the viewpoint of configuration management because the entry is unlikely to cause configuration failures. (Labeling and filtering of noise will be discussed in more detail when we introduce the third Strider principle.)

(2) *Are there known problems associated with this entry?* This quickly points to support articles on known problems caused by the entry, if any. Such information is useful for troubleshooting, but it can also be applied to correct Registry problems before they cause application failures.

The computer genomics database can be populated today through troubleshooting experiences and black-box experiments (e.g., we have recorded the Registry access traces of most of the Windows XP configuration actions), as well as through application-provided specifications in the future.

### 2.2. Attack the mess with the mass

Applying the first principle allows us to decompose the problem into three parts: mechanical, statistical, and database. We now develop the second principle to provide further decomposition of the mechanical part.

Every Windows XP machine starts with approximately 77,000 Registry entries from the CD installation process. The majority of users are given the freedom and flexibility to grow the Registry any way they want by configuring their machines differently and installing different sets of applications. Such freedom and flexibility helped create a large install base, but also created *"the mess"*—every machine has a unique configuration and applications on each machine can interact in a unique way. For example, the default handlers for file extensions or the behavior of an extensible browser may depend on the particular combination of software components installed on a system. When a configuration failure occurs, the lack of a golden state vector particular to the unique configuration at hand typically presents a major obstacle to troubleshooting.

In Strider, we make the observation that full-size, absolutely golden state vectors may not be necessary for CCMS problems. When a program fails due to a configuration problem on a particular machine at a particular time, it suffices to find a state vector either from another machine or from the past on the same machine, where the program is/was working. In the space domain, *"the mass"* (i.e., the large install base) offers a high probability that one can find a healthy machine for cross-machine analysis. In the time domain, a periodic state snapshot feature such as Windows XP System Restore [25] can often provide a good state vector from the past for cross-time analysis.

Given a good state vector and a bad state vector, the mechanical part of Strider operates as follows. First, it performs a *state diffing* operation on the two vectors to obtain a sub-vector consisting of only the differences, which must capture the root cause for the difference in program behavior. Second, it asks the user to re-execute the failed program action and performs *state tracing* to record a sub-vector consisting of only those configuration data that are actually used as input to the current failed execution. Finally, it intersects

the two sub-vectors to identify those that are potential root-cause candidates for the current failure. (An illustration of these three components is shown in Fig. 1.) Preliminary results from our experiments show that, since the diffing sub-vector and the tracing sub-vector are mostly orthogonal, the number of candidates in the intersection is often several orders of magnitude smaller than the full vector describing the bad state.

Once we have decomposed the mechanical part of Strider into state diffing, state tracing, and state intersection, we can utilize various combinations to further take advantage of the mass. For example, if a good state vector is available both from the past on the same machine and from one or more other machines, we can use multiple state diffing sub-vectors in the intersection to further reduce the size of the candidate set. Similarly, if the same application failure occurs on multiple machines, the state diffing and tracing sub-vectors from all these machines can be intersected together to further narrow down the candidate set. Even in situations where no state diffing sub-vector is available, intersecting multiple traces may help eliminate non-deterministic parts of execution traces due to other system activities and irrelevant to the deterministic application failure that is the troubleshooting target. A caveat is warranted here: in order for these more elaborate combinations to succeed, the root cause of the configuration failure must be a single entry or a fixed set of entries that differ between every sick/healthy pair. Fortunately, our experience has been that such root causes are indeed responsible for many configuration failures.

State diffing and state tracing distinguish Strider's black-box approach from the white-box approach: instead of relying on a *full specification* of *absolutely golden state* provided by software developers, we use state tracing to scope essentially a "*partial specification*" of the portion of configuration state actually accessed by the code path taken by the failed program execution, and use state diffing to take advantage of the "*good states relevant to this failure*" available in state snapshots from the past and/or from other machines where the program does not fail.

### *2.3. Complexity–noise filtering*

An immediate concern about the mechanical part of Strider is that a large class of Registry entries are both updated and read frequently, which means that they will appear in both the diffing and tracing sub-vectors. Because of this, they will also appear in the intersection with high probability, and thus they will consistently inflate the size of the final candidate set. Timestamps, usage counts, caches, seeds for random number generators, window positions, and MRU (Most Recently Used)-related information are such examples.

In Strider, we make the observation that such "high frequency" entries should be considered *"operational states"* instead of the "configuration states" that we are mostly interested in for troubleshooting configuration failures. If a machine has been healthy in the presence of these high frequency updates, then when a configuration failure occurs, these operational data are less likely to be the root cause. In contrast, configuration data that have not changed often in the machine's history but have changed recently since the application was last known to be working are more likely to be the root cause. This leads to the concept of *state ranking* based on *Inverse Change Frequency (ICF)*: we assign each candidate in the intersection a score that inversely depends on its change frequency, and prioritize the

troubleshooting effort according to the score ranking; optionally, entries with scores below a threshold can be filtered out as noise and ignored. More sophisticated statistical analysis techniques that additionally take into account abnormal data content [8,12] should further improve troubleshooting effectiveness.

The same observation can be applied to cross-machine analysis. Clearly, there are classes of Registry entries that always contain different data on different machines; for example, the data may be a function of computer names, user names, user security IDs, Globally Unique IDs (GUIDs), hardware IDs, IP addresses, etc. These entries constitute the "natural biological diversity" among machines and are less likely to be root causes of configuration failures. These differences are much like the human genes that are simply responsible for the natural diversity in human appearances and that are not thought to be the cause of any genetic disease even though they frequently appear as genetic differences between sick and healthy people.

In summary, the state-based approach starts with a large and complex problem: the Registry contains many entries, many of them changing. Fortunately, when we apply the complexity–noise filtering principle, these sources of complexity tend to filter themselves out, allowing us to focus on the fewer and simpler Registry entries that are most likely to be significant. This again distinguishes Strider's black-box approach from the white-box approach: instead of relying on a specification of operational data versus configuration data, we use behavior monitoring and statistical modeling to derive this distinction. Similar statistical techniques can also be used to predict potential failures by analyzing a large number of state vectors and flagging those that deviate from the "normal majority" as problematic ones that require special attention.

We note that, for any Registry entry that Strider filters out as noise, one can always construct a counterexample in which the entry is in fact the root cause; such a trade-off between false negatives and false positives is inherent in any statistical techniques. Our empirical results so far have indicated that noise filtering is essential for dealing with complexities and it allows successful troubleshooting of a large class of failures. We will discuss the limitations of noise filtering in more detail in a later section.

## 3. The Strider processes

Applying the three Strider principles allows a decomposition of the troubleshooting problem into five Strider components: state diffing, state tracing, state intersection, state ranking, and the computer genomics database. In this paper, we use the term "Strider process" to refer to a conceptual use of some or all of the Strider components as building blocks in a specific way for a specific type of CCMS problem.

Fig. 1 illustrates the Strider process for troubleshooting. In the narrow-down phase, the state diffing result and the failed application trace are intersected to produce a candidate set, which is then ranked and filtered by the state ranking module. As more and more troubleshooter reports are gathered, entries that are known to cause failures can be emphasized and entries that have repeatedly appeared in the reports as false positives can be de-emphasized.

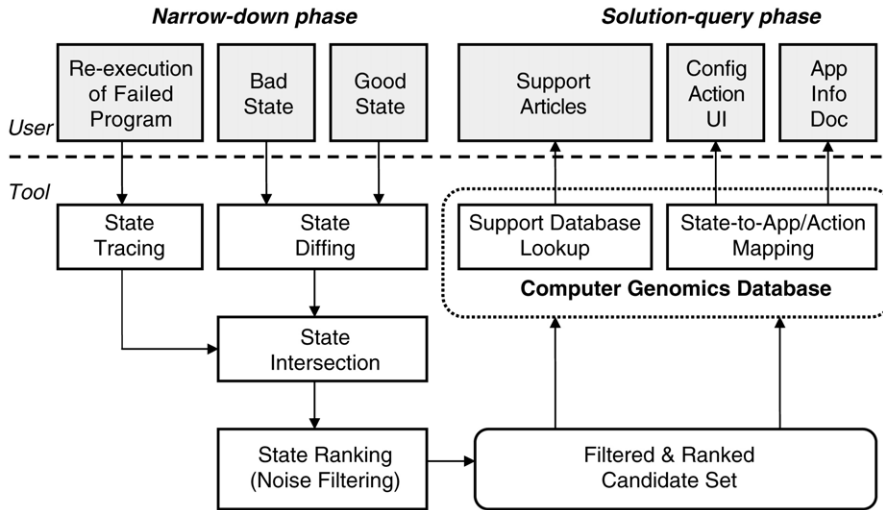


Fig. 1. The Strider process for troubleshooting.

In the solution-query phase, a genomics database lookup is performed for each entry in the candidate set, to yield one or more of the following three types of information: (1) support articles that describe known fixes of problems related to the entry; (2) a user interface for performing configuration actions that can potentially correct the data content of the entry; and (3) information about the application that owns this entry.

We next describe two other Strider processes for different CCMS problems to demonstrate the flexibility provided by Strider's componentized approach. The "configuration certification" process addresses an important CCMS scenario. In this scenario, we would like to answer the question of whether an operational machine still conforms to a certified configuration and so is eligible for product support service. The Strider process would involve state diffing between the operational machine and a certified machine, followed by noise filtering of known entries unrelated to certification. State ranking with an adjustable threshold could provide a trade-off between the time spent in determining the conformance and the time wasted in providing support for non-conforming machines falsely determined to be in conformance. The genomics database could store information regarding commonly installed, unsupported hardware or software to speed up the determination of non-conformance.

Next we describe the "change audit" process. In the scenario targeted by this process, we would like to answer questions in the form of "*what has changed on my machine since last week?*" This Strider process would involve always-on state tracing of all write operations with always-on noise filtering to control the size of the audit file. (State tracing captures the additional information of which process in what context made the changes, information which is typically not available from state diffing.) State ranking would distinguish significant configuration changes from the lesser ones. The genomics database would store mapping information that translates groups of changes into higher level, user-friendly descriptions for better presentation.

## 4. The Strider toolkit

We have implemented the full functionality of the first three Strider components in the Strider toolkit. A limited form of the state ranking component and part of the computer genomics database are also included in the toolkit.

The state diffing tool by default takes two System Restore checkpoints as input and produces an XML file containing Registry entries that exist in both checkpoints but have different data as well as those that exist in only one of the checkpoints. System Restore is a standard feature on Windows XP machines. It automatically saves a checkpoint of the Registry, selected files, and other configuration stores approximately every 24 hours. The number of available checkpoints depends on the maximum amount of disk space allocated for System Restore, which is set to 12% of each hard drive by default.

The tool also supports diffing of only selected Registry hives. For example, if a configuration failure occurs under one user account but does not occur under another user account on the same machine, then the root cause cannot reside in machine-wide Registry hives. Diffing only the per-user hives of the two users takes less time and reduces the number of false positives in the report.

The state tracing tool is implemented as a kernel-mode driver that, by default, intercepts and records every Registry call made by any application or OS component. It supports Include and Exclude filters for logging only those trace lines that contain or do not contain, respectively, specific sub-strings. It also supports efficient logging of only certain call types; for example, it can perform always-on logging of only write-related call types to provide a comprehensive change audit.

The state intersection tool uses a generic tree data structure to maintain a set of hierarchical names. It can take multiple state diffing files and/or multiple state tracing files as input. Each state entry in each of the input files is inserted into the tree and marked by the ID of its source file. Entries that are marked by the IDs of all input files are reported in the intersection result. As we extend the functionality of the state diffing and tracing tools beyond Registry to include other configuration stores such as files and application-specific XML configuration files, the same data structure can be used to compute the intersection.

Ideally, on each machine running the Strider tool, the state ranking component should compute the ICF scores based on a customized "change frequency dictionary" for the local machine because each machine is configured and used in a different way and so the change behavior of configuration state may be different. Building such a customized dictionary at troubleshooting time would not be feasible because it would involve invoking the state diffing operation on every pair of consecutive checkpoints, which could take several hours (with five minutes per pair in today's implementation).

Currently, we include a "static dictionary" in the Strider executable and use the static scores in the dictionary for all state ranking operations. The dictionary was built from analyzing the change frequencies on the main desktop machine of one of the authors. Our troubleshooting experience so far has indicated that such a dictionary appears to be effective in ranking commonly updated Registry entries, but may miss many application- or machine-specific changes. We plan to replace it with another one built from multiple

machines to increase its coverage and make it more representative. In the long run, we would like to have an always-on Windows service running on every machine, continuously updating a local, customized dictionary.

As an optimization, well-known Registry entries and sub-hierarchies that change very frequently and/or repeatedly appear as false positives in troubleshooting reports are filtered out in a *keyword-based noise filtering* step inserted right before the intersection; it is invoked after the intersection code reads an entry from an input file and before it inserts the entry into the tree structure. A second *threshold-based noise filtering* step is invoked after the intersection: it grays out entries with ICF scores below the (conservative) default threshold, which corresponds to a change frequency of 10% in the static dictionary. In addition to the ICF ranking, *order ranking* is also applied to assign more weight to entries that appear earlier in the trace, based on the intuition that later part of the trace may simply be a result of execution divergence caused by a bad value of an earlier entry.

Currently, part of the state-to-app/action mapping information of the genomics database is built into the executable. In a one-time experiment, we performed all commonly used Windows XP configuration actions and recorded their corresponding Registry update operations using the tracing tool. The reverse mappings can then be used to provide state-to-action mappings at troubleshooting time. As more state-to-app mapping information is obtained through experiments and actual troubleshooting experience, we plan to build a Web service for entering and querying such information. The same Web service will also be used to implement the support-article lookup part of the genomics database, which is currently compiled as a list on a Web page with pointers into a trouble-ticket database and a support-article database.

## 5. Experimental results

Clearly, the Strider approach would not work if the following worst case were the norm: a large percentage of the Registry entries change every day and a large percentage of them are used by every application action, resulting in a large candidate set that no human could reasonably handle.

We present empirical results in this section to show that the above worst case is not the norm. We first present measurements of Registry change frequencies from five machines to study the typical size of the state diffing set. Then we present results from troubleshooting experiments to evaluate the effectiveness of additional state tracing, intersection, and ranking.

We use the ten cases listed below in our experiments. They were all real-world failures that troubled some users. To allow parametrized experiments, we reproduced these failures on machines in our group and ran Strider to produce the results. We used configuration user interface (e.g, Control Panel applets) to inject the failures whenever possible, and used direct editing of the Registry for the remaining cases. All the chosen machines were desktop machines used by their owners on a daily basis. This is important because they would exhibit “regular” Registry change behaviors; using test machines from our lab (that have little installation/configuration activity) would have produced better but

invalid results. We also study the sensitivity of the results with respect to the choice of machines to inject the failures. Preliminary results from cross-machine troubleshooting are also discussed.

1. **Systems Restore:** no available checkpoints are displayed because the calendar control object cannot be started due to a missing Registry entry.
2. **JPG:** right-clicking on a JPG image and choosing the *Send To* → *Mail Recipient* option no longer offer the resize option dialog box due to a missing Registry entry.
3. **Outlook:** user is always asked upon exiting Outlook whether she wants to permanently delete all emails in the Deleted Items folder, due to a hard-to-find setting.
4. **Printing:** printing to a duplex-named printer always produces single-sided printing, due to a hard-to-find setting.
5. **IE Passwords:** Internet Explorer (IE) browser no longer offers to automatically save passwords; the option to re-enable the feature is difficult to find.
6. **Media Player:** Windows Media Player *"Open URL"* function would fail if the EnableAutodial Registry entry is changed from 0 to 1 on a corporate desktop.
7. **IM:** MSN Instant Messenger (IM) would significantly slow down if the firewall client is disabled on a corporate desktop.
8. **IE Proxy:** IE on a machine with a corporate proxy setting would fail when the machine is connected to a home network.
9. **IE Offline:** IE *"Work Offline"* option may be automatically turned on without user knowledge; the user would then be presented with a cached offline page instead of the default start page when launching IE.
10. **Taskbar:** IE windows would be unexpectedly grouped under the Windows Explorer taskbar group, due to the addition of a Registry entry.

### *5.1. Registry change behavior*

The common perception of the Windows Registry is that it contains an enormous amount of undocumented configuration information that is accessed frequently by various applications and OS components. To our knowledge, the study that we present in this section is the first quantitative study of Registry change behavior. In addition to providing insights for the troubleshooting problem, the study should serve as a useful guide for the general CCM community as well.

We studied the Registry from two perspectives. First, we looked at the aggregate change behavior of the Registry over a long period of time, ranging from 77 to 84 days. (These numbers are roughly determined by the number of available System Restore checkpoints per machine.) Next, we looked at the daily behavior of the Registry over the same observational period. We expect that Strider troubleshooting will most frequently be applied to a good checkpoint and a bad checkpoint that are close together in time, and therefore we expect the daily behavior of the Registry to be a good guide to the performance of the state diffing part of the Strider toolkit.

The machines in our study consisted of four developer workstations and one knowledge worker's machine, each of them in daily use. Fig. 2 shows the Registry change statistics

| Machine | Number of Registry Values | Days Observed | % Never Changed | % Operational | % Remaining |
|---|---|---|---|---|---|
| 1 | 139,458 | 84 | 95.3% | 2.6% | 2.1% |
| 2 | 213,574 | 84 | 90.4% | 1.9% | 7.7% |
| 3 | 232,890 | 84 | 89.6% | 5.6% | 4.8% |
| 4 | 237,622 | 77 | 79.3% | 1.2% | 19.5% |
| 5 | 200,812 | 84 | 86.8% | 1.9% | 11.3% |

Fig. 2. Registry change statistics.

we observed across the five machines over the entirety of the observational periods. We present the number of Registry values at the end of the period for each machine—these vary from just under 140,000 to almost 240,000.

On four of the five machines (#1, #2, #3, and #5), only 4.7%–13.2% of the Registry ever changed. Applying the noise filtering techniques (i.e., keyword-based filtering and change frequency threshold-based filtering by excluding any Registry entry that changed more than 10 times during the period) to these machines yielded that the number of Registry changes that were classified as non-operational and so potentially configuration-related ranged from 2.1% to 11.3%.

On machine #4, 20.7% of the Registry changed. Looking at this machine's history in more detail, we found that the majority of the changes were due to a single large installation: on this one day, 16% of the Registry changed. If we exclude this single large installation when calculating the Registry change percentage, we find that the changes drop to around 4% of the total Registry size. This number suggests that if we needed to troubleshoot this machine, state diffing of any pair of checkpoints on the same side of the large installation (i.e., either both before or both after) would likely result in a number of potentially significant entries that is comparable to the number found by state diffing on one of the machines with a small change percentage over the entire period. If the state diffing period must cover the large installation, then we need to rely on the intersection component to reduce the complexity, which will be discussed shortly.

Now we turn our attention to the daily behavior of the Registry. Fig. 3 illustrates the daily behavior across all five machines. Because checkpoints may be taken for multiple reasons (by the users manually, by System Restore-aware installers prior to installations, or by System Restore service periodically), we were careful to ensure that we only included one checkpoint per 24-hour bucket in our analysis. Therefore the diff sizes shown in Fig. 3 correspond to a gap of slightly more than 24 hours on average. The spike in machine #4's diff size due to the single large installation (mentioned in the previous paragraph) is clearly visible near the beginning of the observational period.

Across all five machines, the median number of changing Registry values on any given day is 302. After applying the Strider noise filtering, the median number drops to only 29. This demonstrates the additional power of noise filtering when applied to changes between checkpoints taken on consecutive days. This has the following simple explanation: although the percentage of operational Registry entries as shown in Fig. 2 may seem low (between 1% and 6%), these entries changed frequently and so appeared much more often

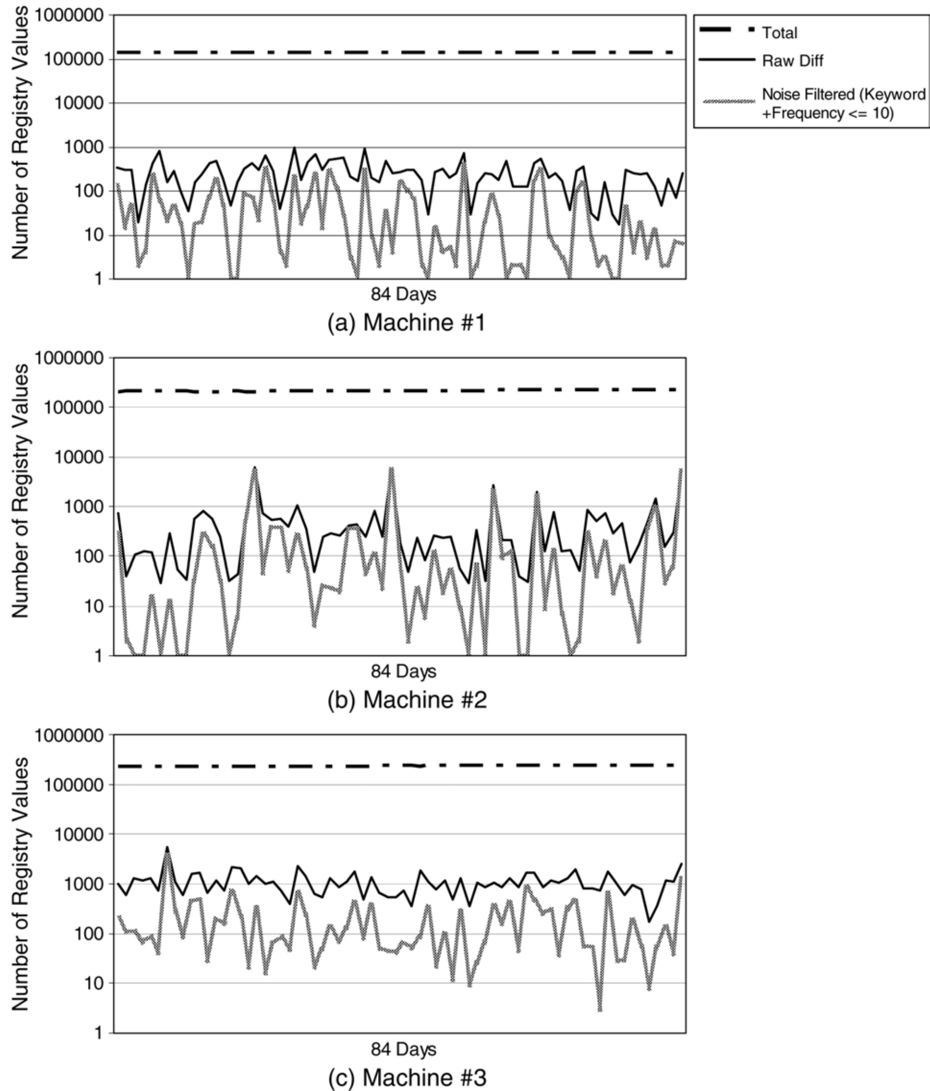


Fig. 3. Registry daily changes with and without noise filtering.

in daily diff results. Noise filtering effectively identifies these entries, which comprise a large portion of any daily diff, as unlikely to reflect significant configuration changes.

### *5.2. Same-machine, cross-time troubleshooting*

#### *5.2.1. Troubleshooting effectiveness*

Fig. 4(a) and (b) present our experimental results on the effectiveness of Strider troubleshooting for the ten cases, all with checkpoints that are approximately seven days apart.

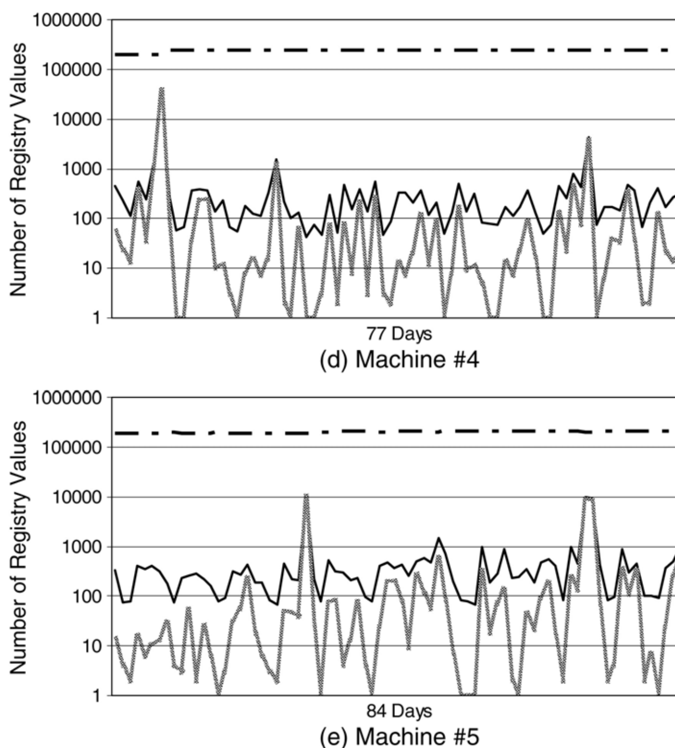


Fig. 3. (continued)

Along the horizontal axis, "Registry Size" is the average number of Registry values of the two checkpoints; "Diff" is the number of Registry values in the state diffing result; "Intersection" is the total number of Registry values appearing in the report, which consist of all the entries in the intersection of keyword-filtered diff and trace; "ICF" excludes those entries in the report whose ICF scores are below the default threshold; "Rank" is the order ranking of the root cause in the ICF-filtered list.

The effect of each step in the Strider troubleshooting process is evident from the figures. Typically, state diffing reduces the dimensionality by two orders of magnitude (from 200,000 to roughly around 2,000) and diff–trace intersection reduces it by another two orders of magnitude (from 2,000 to below 20). Even in the three cases where state diffing could provide only one-order-of-magnitude reduction because the seven-day period covered some significant installation events, the intersection still effectively brought down the number of candidates to below 20.

The ICF threshold-based noise filtering provided additional help for the three cases with more than 10 entries in the intersection (in Fig. 4(b)): it reduced the numbers from 17, 15, and 13 to 14, 11, and 7, respectively. The final ranking summarizes the overall effectiveness of Strider troubleshooting: the actual root cause was identified as the number 1 candidate in six of the ten cases, as number 2 in two cases, and as number 3 in one case. The root-cause

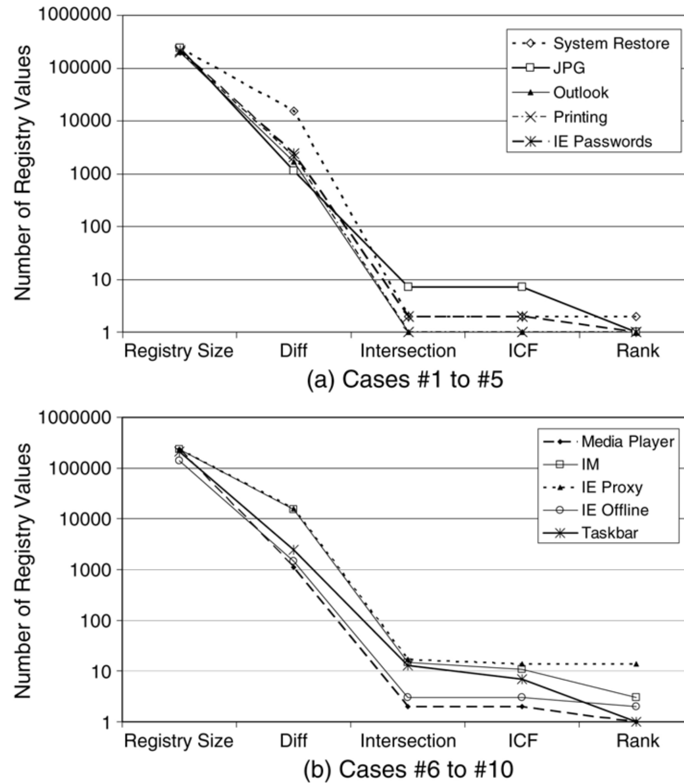


Fig. 4. Same-machine, cross-time troubleshooting effectiveness.

rank for the IE Proxy case was 14, which would require more manual analysis effort to filter out the false positives. We are currently investigating ways to group together relevant final entries to aid manual analysis.

*5.2.2. Sensitivity analysis*

We performed additional experiments to study the sensitivity of the troubleshooting results to variation in the machine being examined and the time interval of the diff. We let the time between the good checkpoint and the bad checkpoint vary among 3, 7, and 14 days. We varied the machine under consideration across all five machines in our study, and we examined four cases: System Restore (case 1), JPG (case 2), Media Player (case 6), and IM (case 7). The final ranking results are presented in Fig. 5.

We found the Strider troubleshooter to be robust to both factors being studied, although varying the factors did have some impact. In three of the four cases, the choice of machine affected the root-cause ranking, although the final rank remains number 3 or better in every case. In the case of machine #5 with case 6, we found that varying the offset in time from 7 to 14 days caused the root-cause rank to drop from 1 to 2.

| | System Restore (case 1) | | JPG (case 2) | | Media Player (case 6) | | IM (case 7) | |
|---|---|---|---|---|---|---|---|---|
| Machine #1 | 3 days | 1 | 3 days | 1 | 3 days | 1 | 3 days | 1 |
| | 7 | 1 | 7 | 1 | 7 | 1 | 7 | 1 |
| | 14 | 1 | 14 | 1 | 14 | 1 | 14 | 1 |
| Machine #2 | 3 days | 2 | 3 days | 1 | 3 days | 1 | 3 days | 1 |
| | 7 | 2 | 7 | 1 | 7 | 1 | 7 | 1 |
| | 14 | 2 | 14 | 1 | 14 | 1 | 14 | 1 |
| Machine #3 | 3 days | 2 | 3 days | 1 | 3 days | 2 | 3 days | 3 |
| | 7 | 2 | 7 | 1 | 7 | 2 | 7 | 3 |
| | 14 | 2 | 14 | 1 | 14 | 2 | 14 | 3 |
| Machine #4 | 3 days | 2 | 3 days | 1 | 3 days | 1 | 3 days | N/A |
| | 7 | 2 | 7 | 1 | 7 | 1 | 7 | N/A |
| | 14 | 2 | 14 | 1 | 14 | 1 | 14 | N/A |
| Machine #5 | 3 days | 2 | 3 days | 1 | 3 days | 1 | 3 days | 1 |
| | 7 | 2 | 7 | 1 | 7 | 1 | 7 | 1 |
| | 14 | 2 | 14 | 1 | 14 | 2 | 14 | 1 |

Fig. 5. Sensitivity analysis of same-machine, cross-time troubleshooting. (Numbers are final root-cause ranks.)

### *5.3. Cross-machine troubleshooting*

Although the current version of the Strider toolkit is primarily targeted at same-machine, cross-time troubleshooting, we have conducted some preliminary experiments and found that it can be useful for cross-machine troubleshooting as well. We used the same ten cases, but with checkpoints from two different machines: the configuration failure was introduced into one machine to make the target program action fail, while the same action succeeded on the other one.

Fig. 6 shows the results. First, we observe that the current state diffing tool is less effective in the cross-machine scenario; it reduced the number of entries by about two thirds, in contrast with the two orders of magnitude in the same-machine case. There are at least two factors that contributed to this: (1) different machines can simply have very different sets of installed programs; (2) the "same" Registry entries can appear to be different on different machines because their names contain machine-specific information. We are currently investigating a set of mapping rules to eliminate the latter.

Fortunately, the intersection operation remained effective and reduced the number to below 100 in all cases. The ICF noise filtering is only slightly useful for half of the cases because the static dictionary built from cross-time diffing analysis may not be suitable for the cross-machine scenario. We expect that a separate dictionary based on cross-machine diffing analysis of each Registry entry among a large number of checkpoints would improve the filtering.

The final step of applying the order ranking heuristics was still mostly effective: in eight of the ten cases, the root cause ranked number 10 or better. But for the IM case and the IE Proxy case, the root cause ranked 36 and 33, respectively.

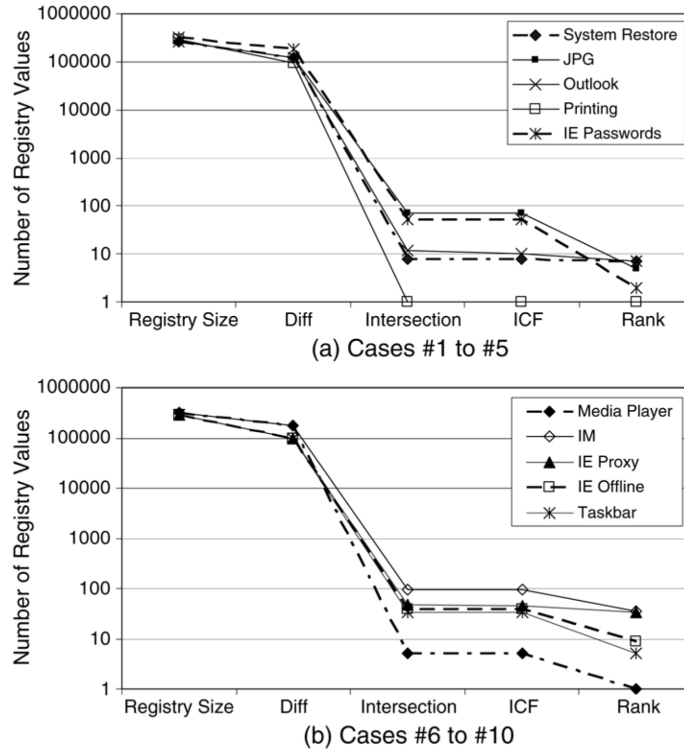


Fig. 6. Preliminary results of cross-machine troubleshooting effectiveness.

## 6. Discussions and future work

In this section, we discuss additional issues and factors that can potentially impact the effectiveness of Strider troubleshooting and were not covered by our performance evaluations presented in the previous section. We also discuss several types of problems for which the current version of Strider cannot successfully provide diagnosis, and we outline our future work directed at addressing these problems. In general, the challenge is to ensure that the mechanical operations capture the root cause, to understand the limitations of our current noise filtering techniques, and then to further exploit the mass to facilitate the final step of root-cause analysis.

*Capturing the root cause*

In most cases, it is fairly clear which application's execution should be traced. For example, in the Media Player case, we traced wmplayer.exe; in the IM case, we traced msmsgs.exe. Per-process traces were used for all the cases except for the "Taskbar" case, in which traces for both iexplore.exe (IE) and explorer.exe (Windows Explorer) were included because it was difficult to determine from the symptom which one was the offending application.

Although they usually achieve good root-cause ranking, per-process traces capture only “direct dependencies” (i.e., Registry accesses made by the target process) and may miss the root cause contained in “indirect dependencies”. For example, in a case where a pop-up stopper designed to stop pop-up ads interfered with the normal operations of a Web site, the root-cause Registry entry was accessed from a separate process, rather than from the browser process itself. We plan to enhance Strider with process dependency tracking so that it can capture indirect dependencies without resorting to using all-process traces.

In addition to indirect dependencies, “asynchronous dependencies” pose another challenge. Strider implicitly assumes that the root-cause entry must be accessed synchronously during the user-selected time interval for state tracing. However, it is possible that the traced application had read the root-cause entry before the tracing was started. For example, while in most of the ten cases the application actions that the user should trace were well defined and did contain the root causes, the remaining cases required tracing of application launching as well as the application action that led to the observed failure.

To study the effect of less-experienced Strider users always using the longer traces (i.e., since application launch) to avoid missing the root cause, we performed further experiments by replacing the action-only traces with the longer traces in cases 1, 3, 4, 5, and 6. This would drop the root-cause ranking from (2, 1, 1, 1, 2) to (3, 12, 1, 9, 17), respectively, which is still acceptable but may require significantly more troubleshooting effort depending on whether the additional false positives are easy to filter manually. Our long-term direction is to develop efficient, always-on tracing and logging to relieve users of the responsibility of specifying when to start and stop tracing.

Similarly, users may specify incorrect good states due to either incorrect memory or latencies between state corruption and application failure. This would cause the state diffing results and thus the intersection to miss the root cause. A near-term solution is to encourage users to be conservative in selecting good states. Our long-term direction is to develop statistical techniques that automatically analyze multiple good and bad checkpoints to relieve users of the burden.

*Limitations of noise filtering*

As discussed previously, statistical techniques such as Inverse Change Frequency ranking for filtering out false positives naturally introduce the possibility of false negatives. Although it is difficult to provide conclusive arguments that noise filtering does not introduce significant false negatives without a large number of failure cases, our experience has shown that it works well in practice and has allowed successful root-cause analyses of tens of cases.

We now describe several types of problems that could potentially defeat Strider’s current noise filtering strategy. Our plan is to refine the filtering rules as we encounter false negatives, and revisit the design if we gain concrete evidence that a significant number of real-world root causes actually fall into the false-negative category.

- **Usage counters:** for example, the behavior of a trial software package may change when its usage count exceeds a certain threshold.
- **Window positions:** for example, a corrupted entry that is supposed to remember the last position of an application window may cause display problems.

- **MRU and cache-related information:** for example, "last server connected" may be the root cause of a client application failure; "last file opened" may cause a document processing program to fail upon launch; a cached Web page may cause undesirable behavior in a browser.
- **Per-session data:** some application data may be updated on a per-session basis and have dependencies on the current environment; failures may occur when a user tries to restore such per-session data. Current Strider noise filtering would have mistakenly filtered out such data.
- **Data coupling:** a single Registry entry may contain both operational data and configuration data. For example, we have encountered a case where Word was used as the default email editor for Outlook and a certain document navigation option could not be turned off. The option was, unfortunately, controlled by some Registry entries containing binary blobs of data, and these binary blobs apparently contained operational data as well and so had low ICF scores. These entries were incorrectly filtered out as noise originally (i.e., grayed out in the report), but later determined to contain the root cause through further investigation. Once a false negative is discovered, the change frequency dictionary built in to the Strider executable is updated to assign the entry a very high score, reflecting the fact that the entry has been identified as the root cause of an actual configuration failure.

*Exploiting the mass*

The four orders of magnitude in dimensionality reduction typically achieved by the mechanical steps of Strider was a significant starting point for us for handling the complexity of Registry problems. However, we have encountered cases in which ICF noise filtering and order ranking failed to offer the final reduction of another order of magnitude and the users were left with tens of candidates to investigate. In some cross-machine cases, the final reports still contained hundreds of candidates and root-cause analysis remained very difficult.

We plan to address this challenge by further exploiting the mass. We are collecting a large number of Registry snapshots in our "GeneBank" and plan to generalize the diff-based techniques to statistical analyses across multiple snapshots. In particular, root-cause candidates containing data that clearly deviate from the "normal majority" will be ranked higher. We are also enhancing the tracing and noise-filtering techniques to enable always-on logging and analysis on a large number of machines for building and reporting known-good behavioral models.

## 7. Beyond the Windows Registry

Although we have focused on troubleshooting Windows Registry-related problems, the Strider techniques are generally applicable to any shared, persistent configuration store on any operating system platform.

We are currently extending the Strider implementation to provide troubleshooting of configuration problems due to changes in files and directories/folders. The implementation will utilize the file change log information from System Restore to detect which files

have been changed, use a filter driver to trace which files are being accessed as part of an application action, reuse the tree structure for computing the intersection, use file change frequency for state ranking, and rely on information from the genomics database to identify directories containing temporary files as known noise, to provide mappings of which files belong to which applications or OS components, and to point to support articles documenting known problems with certain files.

Similar techniques can also be applied to Unix machines. Unix configuration generally appears in files under /etc/. For example, user account information is in /etc/passwd, the IP address of the DNS server is in /etc/resolv.conf, and the many parameters for the X server are typically in /etc/X11/. Although many configuration files are used by a single program or OS facility, quite a few well-known configuration files are shared by multiple programs and so are subject to similar configuration problems as those in the Window Registry.

The most notable example is /etc/mailcap, which contains a system-wide mapping from MIME types to commands for handling them. Any software that displays or edits a particular type of MIME file may want to install entries in the mailcap file, so that other programs can use it to display or edit that file type. Therefore, applications that can handle common file types could write conflicting entries into the mailcap configuration file.

Another example is /etc/inetd.conf. Rather than having a separate daemon for each type of connection (for example, finger, telnet, rlogin, rsh, smtp, ftp) listening on its own port for incoming connections, the meta-daemon inetd listens on all the ports and starts an instance of the appropriate daemon on demand as each connection comes in. Each of the individual daemons is required to add an entry to inetd.conf when it is installed and remove that entry upon uninstallation. If two daemons use the same port, their entries in inetd.conf may conflict [14].

In addition, ill-written Unix applications may modify the environment variables in a user's .cshrc configuration file for their own operations. Such practices may result in conflicts in environment variables. When these conflicts result in faulty application executions, users typically resort to application reinstallation to repair the problem. Strider troubleshooting can help identify the root cause to potentially provide a less disruptive repair and avoid future occurrences of the same problem.

## 8. Related work

The body of work related to systems management through specification is quite large [2,3,9,15,18,23]. The general approach is to provide languages and tools to allow developers or system administrators to specify "rules" of proper system behavior and configuration for monitoring, and "actions" to correct any detected lack of compliance with a given rule to enable the system to converge with the specified requirements. Strider complements the specification-based approach by adopting a black-box approach to discover unspecified rules of proper system operation and gradually build up a genomics database of known-good requirements and known-bad issues.

Burgess [4,6] proposed a general "diffing" concept of adaptive, statistical, long-term anomaly detection for systems management, which is implemented into the configuration agent system *cfengine* for the Unix environment. The Strider Inverse Change Frequency

ranking applies a similar concept to the Windows environment. Specifically, a "statistical quantifier" in the summary form of change frequency is maintained for each Registry entry to approximately characterize its "normal" behavior; operational data exhibiting long-term, high frequency behavior is then de-emphasized at troubleshooting time even though it appears in the state diffing result. The garbage collection operation of System Restore defines a natural sliding window for Strider; contributions from data changes corresponding to the past, garbage-collected period are degraded to allow the ranking algorithm to adapt to progressive behavioral changes due to newly installed software or changes in user usage patterns.

Strider's basic approach of classifying changes as unimportant (noise) or important based on their statistical properties is in the same spirit as the statistical mechanics section of Burgess's paper on Computer Immunology [5]. In contrast to his work on Computer Immunology, we have not explored self-repairing behavior. We made this choice for two reasons. First, the current focus of Strider is on desktop machines, for which reliable measures of normality are harder to obtain compared to the case for heavily loaded server machines running widely used services, as was similarly observed in [6]. Second, the "state" considered by Strider consists of "raw" pieces of configuration data, such as a Registry entry, as opposed to the "resource usage variables" commonly used in the literature. It is more difficult to define anomalies and invoke self-repairing actions in the former category because most of the changes in configuration data are likely to have been intentional; they should be considered potential anomalies only when the user complains about the system or an application no longer delivering user-expected services.

Gossips [13] provides an extensible, object-oriented framework for monitoring distributed systems in an IT environment. Each Gossips process running on a participating client gathers and analyzes system state-related data, and reports any interesting state changes to a central server. By including cfengine, Gossips could be extended into an automated repair tool. Although Gossips also maintains a knowledge base of known problems indexed by state-related information, the "states" refer to the condition of a system or service (such as *working/broken*), which are quite different from the lower level, more precise "configuration data" states in the Strider genomics database.

In a recent position paper, Redstone et al. [19] described a vision of an automated problem diagnosis system that automatically captures aspects of a computer's state, behavior, and symptoms necessary to characterize the problem, and matches such information against problem reports stored in a structured database. In the Strider project, we have focused on developing actual root-cause analysis technologies for configuration failures by using state diff and trace information to characterize them. Symptom descriptions are used as secondary information and are still provided by the user because many Registry-related "problems" can only be defined against user expectation. An earlier version of Strider provided automatic search of support database for high ranking root-cause candidates [24]; but we have observed that such an approach can only be effective after a large number of support articles are written in a structured, machine-readable format and stored in the genomics database.

The concept of problem identification as deviant behavior from a "normal majority" by applying statistical techniques to a large number of samples has emerged in several areas in recent years. Engler et al. [11] described techniques that automatically extract correctness

rules from the source code itself (rather than the programmers) and flag deviations, and that use statistical ranking to prioritize the inspection effort. Liblit et al. [17] proposed a sampling infrastructure for gathering information about a large number of actual program executions experienced by a user community, based on which predicate guessing and elimination are used to isolate deterministic bugs and statistical modeling is used to isolate non-deterministic errors by identifying correlation between behaviors and failures. Apap et al. [1] presented a host-based intrusion detection system that builds a model of normal Registry behavior through training and showed that anomaly detection against the model can identify malicious activities with relatively high accuracy and low false positive rate. The PinPoint root-cause analysis framework [7] applies data clustering analysis to a large number of multi-tier request–response traces tagged with perceived success/failure status to determine the subset of components that are most likely to be the cause of failures.

## 9. Summary

We have proposed the Strider state-based approach to Change and Configuration Management and Support, and built and evaluated a system based on this approach. The approach allows a decomposition of complex problems into five Strider components that can be used as building blocks in various scenarios. In the context of our primary example, troubleshooting of configuration failures, we have demonstrated that combining the black-box techniques of state differencing, tracing, intersection, and ranking can effectively narrow down the list of root-cause candidates for many real-world cases. As we continue to build up the computer genomics database, where we provide precise mappings from configuration state items to their known functions and/or problems, more knowledge will be captured in a structured format, enabling even more effective root-cause analysis. Our future work includes providing differencing and tracing of more types of configuration state to increase coverage, collecting a large number of state snapshots and program traces to enable advanced statistical analysis and reduce Strider's dependence on manual steps, and evolving the current Strider toolkit for troubleshooting into a systems management framework for self-monitoring and self-healing.

## Acknowledgements

We would like to express our sincere thanks to our shepherd Alva L Couch for his valuable feedback, to Jidong Wang and Ji-Rong Wen for recording the UI-to-Registry mapping data, to those colleagues who provided Registry snapshots for change behavior analysis, and to those people who contributed the configuration failure cases for our experiments.

## References

[1] F. Apap, A. Honig, S. Hershkop, E. Eskin, S.J. Stolfo, Detecting malicious software by monitoring anomalous windows registry accesses, in: Proc. of the Fifth International Symposium on Recent Advances in Intrusion Detection, RAID, 2002.

[2] E. Bailey, Maximum RPM, 1997.

[3] M. Burgess, A site configuration engine, Computing Systems 8 (1995) 309.

[4] M. Burgess, Automated system administration with feedback regulation, Software Practice and Experience, vol. 28, 1998.
[5] M. Burgess, Computer immunology, in: Proc. of LISA, 1998, pp. 283–297.
[6] M. Burgess, Two dimensional time-series for anomaly detection and regulation in adaptive systems, in: Proc. IFIP/IEEE 13th International Workshop on Distributed Systems: Operations and Management, DSOM, 2002.
[7] M. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, Pinpoint: problem determination in large, dynamic, internet services, in: Proc. Int. Conf. on Dependable Systems and Networks, IPDS Track, 2002.
[8] K.W. Church, W.A. Gale, A comparison of the enhanced good-Turing and deleted estimation methods for estimating probabilities of English bigrams, Computer Speech and Language 5 (1991) 19–54.
[9] A. Couch, M. Gilfix, It's elementary, dear Watson: applying logic programming to convergent system management processes, in: Proc. of LISA, 1999.
[10] C. Dennis, R. Gallagher, The Human Genome, Nature Publishing Group, 2001.
[11] D. Engler, D.Y. Chen, S. Hallem, A. Chou, B. Chelf, Bugs as deviant behavior: a general approach to inferring errors in systems code, in: Proc. ACM Symp. on Operating Systems Principles, October, 2001.
[12] I.J. Good, The population frequencies of species and the estimation of population parameters, Biometrika 40 (1953) 237–264.
[13] V. Götsch, A. Wuersch, T. Oetiker, Gossips: system and service monitor, in: Proc. of LISA, 2001.
[14] J. Hart, J. D'Amelia, An analysis of RPM validation drift, in: Proc. LISA, 2002.
[15] A. Keller, C. Ensel, An approach for managing service dependencies with XML and the resource description framework, Journal of Network and Systems Management 10 (2) (2002).
[16] M. Larsson, I. Crnkovic, Configuration management for component-based systems, in: Proc. Int. Conf. on Software Engineering, ICSE, May, 2001.
[17] B. Liblit, A. Aiken, A.X. Zheng, M.I. Jordan, Bug isolation via remote program sampling, in: Proc. Programming Language Design and Implementation, PLDI, 2003, pp. 141–154.
[18] R. Osterlund, PIKT: problem informant/killer tool, in: Proc. LISA, 2000.
[19] J.A. Redstone, M.M. Swift, B.N. Bershad, Using computers to diagnose computer problems, in: Proc. HotOS, 2003.
[20] D.A. Solomon, M. Russinovich, Inside Microsoft Windows 2000, 3rd edition, Microsoft Press, 2000.
[21] Y. Sun, A.L. Couch, Global analysis of dynamic library dependencies, in: Proc. of LISA, 2001.
[22] S. Traugott, J. Huddleston, Bootstrapping an infrastructure, in: Proc. LISA, 1998.
[23] Tripwire, http://www.tripwire.com/.
[24] Y.M. Wang, C. Verbowski, D.R. Simon, Persistent-state checkpoint comparison for troubleshooting configuration failures, in: Proc. Int. Conf. on Dependable Systems and Networks, DSN, 2003.
[25] Windows XP System Restore, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwxp/html/windowsxpsystemrestore.asp.