Instrumenting Static Analysis Tools on the Desktop

Nathaniel Ayewah
Dept. of Computer Science
Univ. of Maryland
College Park, MD
ayewah@cs.umd.edu

Yue Yang, David Sielaff
Analysis Technologies Team
Microsoft Corporation
Redmond, WA
{jasony,dsielaff}@microsoft.com

ABSTRACT

At Microsoft we use a number of static analysis tools to ensure the quality of the code we produce. Over several years, we have solved problems associated with deploying these tools in a large development environment, including problems of performance, policies for using tools, and methods for encouraging their usage. One challenge is getting appropriate feedback from users about the effectiveness of these methods. In particular, we do not get feedback about errors and warnings that are found and resolved on the desktop and do not make it into the code repository. To address this problem, we have developed an instrumentation framework called ATMetrics, which allows us to collect usage metrics that we can use to analyze how static analysis tools are used in the field. In this paper, we discuss our experiences putting together this metrics system in a complex industrial setting and shed light on how it can help to guide key business decisions around the deployment of static analysis tools.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis; D.2.8 [**Metrics**]: Product metrics

General Terms

Measurement, Reliability, Security

Keywords

static analysis, software defects, managed code, software quality, software instrumentation, cost benefit analysis

1. INTRODUCTION

Software organizations rely on multiple approaches to provide quality assurance, including code review, testing and static analysis. Static analysis offers the opportunity to find bugs automatically, though it also potentially raises many false alarms. To minimize these false alarms, some tools encourage developers to add annotations which provide more semantic information to the analysis. Many tools also aim to bring warnings to the attention of developers soon after they write the code, by interrupting them with alerts or warning markers. All things considered there is a cost to using static analysis, and a need for a clear benefit to make it worthwhile.

The Analysis Technologies team at Microsoft manages a number of static analysis tools and provides support to thousands of developers who use these tools. We recently started an effort to better understand how developers are interacting with these tools, and learn from their experiences. Our ultimate goals are to improve the tools and associated processes, and demonstrate the value of using tools.

In particular, in this project we focus on getting information that was previously unavailable to us: the developer's activities on the desktop. We can easily observe static analysis warnings in source files that have been checked into the source repository, but warnings that are fixed soon after they are introduced never make it to us. With many of our tools now integrated into the build system, developers are regularly alerted about problems in the code. Furthermore any problems checked in can potentially break the build or prevent code integrations (see Section 2). These incentives make it likely that many problems are fixed before the code is checked in. Prior to this, such "transient" activities would be lost.

To capture this missing information, this project aims to setup lightweight instrumentation on developer workstations to capture metrics on which warnings occur, which ones are fixed or suppressed, and other details about user interaction with static analysis. We also aim to explore any connection between behaviors observed on the desktop and other measures of the quality of the underlying component.

In this paper, we describe our experiences and the challenges of putting together a system like this in an industrial context. The primary challenges were correctly and robustly inferring the actions of developers with very little overhead, and supporting many heterogeneous static analysis tools. One constraint was that the static analysis viewer containing much of our instrumentation was still under development, so we had limited opportunities to test and deploy it.

A key contribution of this paper is the lightweight mechanism that enables us to track and infer state transitions for static analysis warnings. To the best of our knowledge, this is the first effort in developing and deploying a pragmatic metrics system for static analysis tools in a complex industrial context.

We start by describing the processes we have developed to maximize the use of static analysis at Microsoft in Section 2. Then we overview some of the key questions we want to answer in Section 3. We describe our experiences, implementation and challenges in Section 4, our test frameworks in Section 5, and present some qualitative feedback from users in Section 6. Going forward, we expect to use this system to support critical business decision making about which tools to modify and rules to prioritize.

2. BACKGROUND

Over the last several years, we have focused our efforts on pushing defect detection tools based on static analysis into the regular software development process of the largest product groups at Microsoft, involving thousands of developers working on tens of millions of lines of code against strict deadlines.

We use different processes for triaging warnings from our global and local tools. Inter-procedural analysis tools such as PREfix [3] and Global Esp [4] are based on heavyweight global static analysis; these tools are run periodically in a centralized manner, and the defects identified by the tools are filed automatically into the defect database of the product. Intra-procedural tools, such as PREfast [11] based plugins, are lightweight and more suitable to be run on the developer's desktop while the code is being constructed. A wide range of PREfast plugins have been developed for tackling critical problem areas such as security, concurrency, performance, internationalization issues, and device driver issues. These tools typically analyze one function at a time based on function contracts and field invariants specified in the SAL [14] annotation language.

Today, many of these lightweight tools are enabled by default on the desktop machines of every programmer in the organization, using the Microsoft Auto Code Review (OACR) build infrastructure [13]. OACR integrates static tools into a common and automated build environment which runs the checkers in the background. Developers are notified with a pop up message about the warnings. Warnings are grouped into warning numbers and warning numbers are classified with severity levels. When developers review the warnings, they have the opportunity to fix the code or suppress the warnings. One of our goals is to understand when issues are fixed and when they are suppressed.

Another level of quality control is through the "quality gates" that are applied when moving code from one branch to a higher branch (called reverse integration). A class of critical checks form the "minimum bar". Reverse integration is prohibited until all warnings from the minimum bar are fixed. This mechanism ensures that the most serious issues can be caught and fixed early in the development process. For big legacy code bases, adding a new check to the minimum bar may introduce a large number of warnings triggered by pre-existing bugs. We apply a *baselining* mechanism to "mask" these warnings in order to avoid a sudden disruption to the development schedule. Typically these pre-existing bugs are fixed during a concerted cleanup effort at the early stage of a product cycle.

3. KEY QUESTIONS

There are hundreds of data points we could potentially collect as part of an instrumentation effort, ranging from high level metrics about how often warnings are generated and shown to developers, to low level details about which issues are clicked or which code edits are made. But one of our important goals is to make this system minimally intrusive, as developers engage in their primary development activities. To limit our instrumentation effort, we identified the key business questions we want to answer with this project, some of which are described in this section.

3.1 Which warnings occur and are fixed or suppressed?

The most basic question is: which warning numbers occur on the desktop, which ones are fixed and which ones are suppressed? Here we are interested in the absolute counts as well as the proportional rates. The absolute counts tell us which issues developers encounter the most. If a warning number occurs often but is generally suppressed or ignored, then this may indicate that the associated analysis needs to be tweaked or the warning deprioritized.

Collecting data points to answer this question presents some challenges. To maintain correct counts, we need to keep track of issues from build to build and session to session. Using the warning number and line number is not sufficient because line numbers change often. We use a common approach that relies on contextual information surrounding a warning [16]. In addition to keeping track of issues, we need to remember what state the issue was in previously, because this can change multiple times. For example an issue can go from being ignored to being suppressed, and then back to being ignored a few days later. This example raises a third challenge related to our goal to keep instrumentation lightweight. Sending updates to a central server every time an issue changes state might be too much overhead if there are many issues, or if they change state often, so we need a solution that minimizes the overhead by using summaries. Our design decisions for tackling these challenges are described in more detail in Section 4.

With these data points, we expect to observe that high priority issues are fixed at a higher rate, but we want to look for those warning numbers that buck the trend. This may influence our decisions about what severity levels to assign to warning numbers. In particular, we want to identify high priority issues that tend to be suppressed. To facilitate this, we need to collect more data about these suppressed issues to allow us to trace them in the code repository and throughout the development cycle.

These data points may also point us to trends that need to be investigated more closely. For example, we may observe that a particular class of issues occurs much less frequently than we expect. This could be because the analysis is limited and we need to improve our tools to find more issues. Or it could be that the problem really does not happen in the code base and we can spend less time working on the analysis. This instrumentation effort can help us identify these classes of issues for closer investigation.

3.2 How do tools impact developers?

While many problems identified by static analysis tools occur because the developer made a logical error, some problems represent best practices or code conventions the developer may be unaware of. In the latter case, we expect developers to change their development styles upon learning better coding practices, and hence introduce fewer instances

of this class of issues. If this is the case then it potentially represents a significant benefit of using our tools. An alternative outcome is that developers consistently ignore or suppress these issues. In this case, the number of new issues added per session may not change over time. Or we may observe different patterns in different groups or at different parts of the development cycle. All these trends can inform the policies we recommend to groups and the way we promote tools. Our hypothesis is that most higher priority issues will be introduced at lower frequencies as the developer becomes more experienced.

To answer this question, we need data points that can be used to distinguish individuals and groups, as well as information on trends observed in each user session. In our design discussion in Section 4, we describe data points that refer to overall counts, as well as counts just for warnings introduced in the active session. We also need data points about how much development time is spent interacting with tools and issues.

Another possible impact of tools on developers comes from the presence of issues in legacy code. In general, older code is not analyzed unless the developer checks out the source or explicitly runs the analysis on legacy code. This means that if a developer has to make even small changes to a legacy file, they may be confronted with any warnings that were in that file. This is not a problem for many groups that have put in a concerted effort to remove serious issues from legacy files, but our instrumentation should allow us to observe those scenarios where this does happen.

3.3 What is the effect of baselining?

One way to deal with large numbers of warnings in legacy code is to *baseline* them, i.e. to temporarily hide them from view. The rationale is that warnings in older code are less likely to be serious since this code has undergone extensive quality assurance testing. By baselining these issues, we encourage developers to focus on newer issues. One of our hypotheses is that developers will fix more issues sooner if they are only shown new ones.

Our instrumentation allows us to detect whether an issue has been baselined or not, and whether it is visible. The static analysis viewer allows developers to turn on or off filters that determine if baselined issues are invisible or visible respectively. In addition to baselined issues, we can examine whether developers keep only high priority issues visible, or they display all issues, and how this affects the likelihood that fixes are made.

3.4 How do our observations correlate with business metrics?

Our ultimate goal is to understand how usage of static analysis affects the quality of the final software product. This is in general a hard question to answer since there are many factors that may affect component quality. Hence our goal is simply to look for trends and correlations between static analysis usage patterns and existing business metrics. We can use internal metrics about components such as the number of reported crashes or security flaws, and compare these metrics with data from our instrumentation including fix and suppress rates, new issue introduction rates, and whether baselining is used or low priority issues are filtered out. We can also compare our instrumentation data with the policies and practices we observe in different groups.

To facilitate this investigation, we group issues that come from the same component or binary as part of our instrumentation. In addition, we will need to separately collect information about the policies and practices of different groups, and identify business metrics that are collected uniformly across many groups and organizations. Ultimately, this investigation should inform our efforts to encourage best practices for using our tools.

3.5 Other Questions

There are several other questions and goals we aim to support with our instrumentation. One question is how long different groups of warnings tend to stay active. In some cases, an issue may be fixed soon after it is flagged by the build and the developer is alerted. In other cases, developers may wait until just before a code check-in to fix all serious issues. Other developers may wait for issues to be flagged during overnight builds or by quality gates. One expectation is that lower priority issues will stay active longer because they do not affect the overnight builds or quality gates. We also expect to see different trends for different individuals and teams, and different parts of the development cycle.

We are also interested in how users interact with the static analysis viewer and the warnings. Our instrumentation will show us how often users run builds and launch the viewer, how frequently issues are clicked and which issues are clicked. A click indicates that the user is investigating an issue including possibly looking up the help pages. If a warning number has a low fix rate, but its issues clicked fairly often, this may suggest that users are choosing to ignore this warning number. We could also observe that some warning numbers are fixed at a high rate though their issues are rarely clicked. This may indicate that those issues are being fixed during normal software development without much guidance from the viewer.

4. IMPLEMENTATION AND CHALLENGES

We have designed and implemented ATMetrics (Analysis Technologies Metrics system), an instrumentation system that builds upon existing platforms and processes used in Microsoft. Specifically, the instrumentation was designed to be a lightweight add-on to a static analysis viewer which is driven by the build system. Our custom data points are transmitted and aggregated using Microsoft's Software Quality Metrics (SQM) [12], a platform for collecting remote data from thousands of volunteers, used in many Microsoft products. One hope is that in the future our architecture will be adapted to support other types of analysis technologies including code coverage and binary instrumentation.

In this section, we provide a broad overview of our design and go into more details about some of the challenges and our solutions.

4.1 Architecture Overview

Figure 1 illustrates some of the key aspects of our architecture. Our instrumentation is designed as a library which manages the processing and counting of issues and sends data to a central database. This library also maintains a persistent state for each unique user configuration enabling us to track issues from session to session. We added very few lightweight calls (less than 20 lines) to an existing (though relatively new) static analysis viewer that is being deployed as part of the build system to thousands of developers. Our

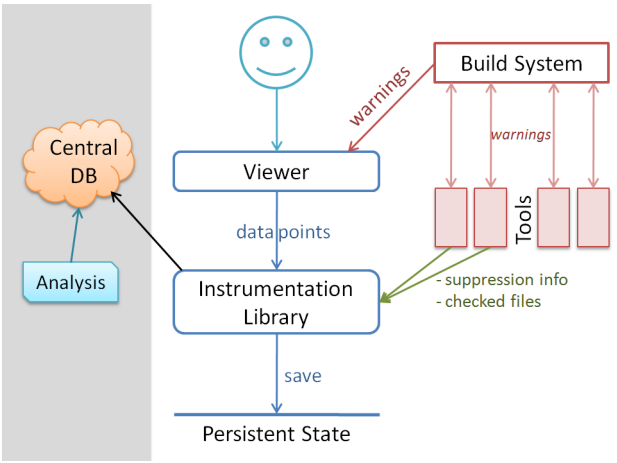

Figure 1: Architecture Overview

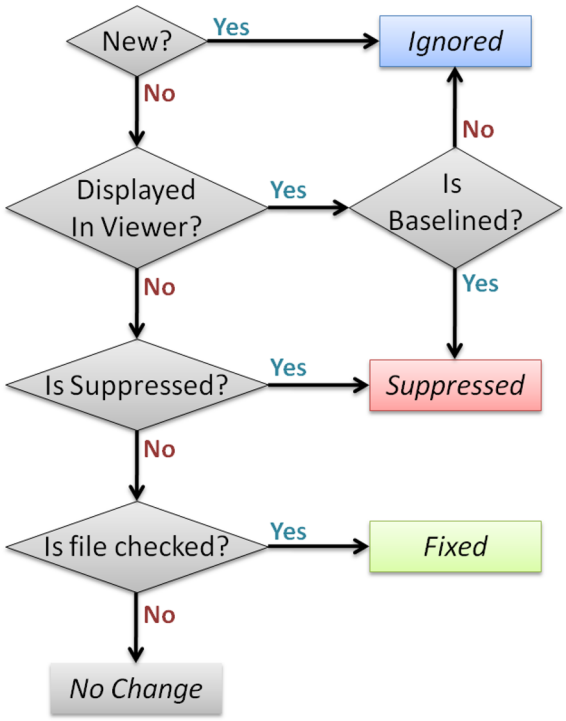

Figure 2: Classifying Issues into Groups

hook points call the library when certain important events occur in the viewer, such as when issues are added or clicked. This arrangement facilitates the possibility that our instrumentation can be adapted in the future to other scenarios or analysis technologies.

As we mentioned earlier, our static analysis tools run automatically with each build. This means we can update most of our instrumentation state using just the information from the build system. However, some of our data points rely on data that is not readily available from tools. To solve this, we made small modifications to some of our tools to enable them to send data directly to the instrumentation system. We describe this in more detail in Section 4.4.

One important concept in our instrumentation is the notion of a session. A session represents the period of time for which observations are grouped. For example, all issues that are first seen during a particular session are considered *new* for that session, and *old* for any subsequent sessions. In our current implementation, we define a session as the period from when the viewer is opened to when it is closed, but the notion of a session is flexible enough to be defined as "every 24 hours", or the periods when the viewer is not idle.

At the end of each session, we use SQM to transmit an update to a central database, based on the activities that have occurred in that session. The contents of the update are described in more detail in Section 4.3. SQM uses this update to aggregate counts of the issues that have been fixed, suppressed or ignored. SQM also provides services to allow us to pull down the data and analyze it.

4.2 Inferring the State of Issues

One of the primary challenges in constructing this instrumentation was inferring whether issues are being fixed, suppressed or ignored. We are not parsing the source code or even monitoring every key stroke as this would be too much overhead. All we can see is when issues appear and disappear. Based on this, we have to classify issues into one of the three groups: *Fixed*, *Suppressed* or *Ignored*.

Figure 2 summarizes the conditions that determine which group an issue is classified into. We can generally always assume that when an issue appears in the viewer, it should be treated as *Ignored*. An exception is when the issue has been baselined: baselined issues can appear in the viewer if the developer turns off the *Baseline* filter. We add baselined issues to the *Suppressed* group. All the information needed to make these two decisions is readily available from the viewer.

If an issue that was previously displayed in the viewer disappears, there are several inferences that we could make. An issue could disappear because it was fixed, because it was suppressed, because code churn put the issue out of reach of the analysis or even because the containing source file was not analyzed. We do not have enough information in the viewer to make all these inferences, so we need to refer to the static analysis tools to get more information. This is another example of information that is only available to us on the desktop.

For example, we cannot tell which issues have been suppressed because we are not parsing suppress tags or pragmas (used for source line suppression). However, we can query tools for the full list of all issues generated before suppression is applied. Any issues in this full list that do not appear in the viewer can be inferred to be suppressed. This strategy is limited by the fact that we have many heterogeneous tools and is not implemented for all tools (Section 4.4). So in some cases, we may not be able to detect if an issue has been suppressed.

Another scenario is when the source file containing an issue is skipped during the most recent analysis run. This could happen because the developer chooses to build only a subset of files (usually done to get a faster build). It could also be because the file is not checked out of the code repository; such files are built but not analyzed by default so that developers are not presented with warnings in source files

they are not actively working on. Our strategy in these scenarios is to retain the last decision about the issue under question. In other words, we do not move issues into the *Fixed* group if its containing source file was not analyzed. Again we run into the problem that different tools have different processes for indicating which files were checked, and so this strategy is not implemented for all tools. For those tools that do not provide information about which files were checked, we treat all issues that disappear as fixed.

An issue may also move from one of the groups to another. For example, if an issue that was previously marked as fixed reappears in the viewer, it will be marked as ignored. As the next section explains, we account for these movements in the updates we send to the central database.

4.3 Making Instrumentation Lightweight

The last section presented some scenarios in which we may make an inaccurate inference. For example, an issue may be incorrectly marked as *Fixed* if the containing file is not checked and the issue was generated from a tool that is not modified to provide this information. This problem may be corrected in a later session when the file is checked again. This example raises the possibility that an issue may change state spuriously many times. Most of our key questions are focused on the final state of an issue, or more specifically of a group of issues with common properties. This allows us to adopt an aggregation algorithm that maintains and updates counts of the number of issues in each group. Motivated by the architecture of the SQM pipeline which encourages aggregation on the server, our approach reduces the amount of data sent to the central database, since we are concerned with groups of issues, not individual issues.

In this approach, we group issues based on some properties that are common to many issues, such as warning number, the warning priority, and the binary being built. We then keep track of all the issues in each group on the local machine and send only summaries and updates to the central database.

Figure 3 illustrates this approach for a group of high priority issues that have warning number 100 and that occur in code used to build *shell.dll*. When the developer launches the instrumented viewer for the first time (a), all issues are included in the *Overall Ignored* count. If any issues are added during the session (b), they are included in both the *New Ignored* and *Overall Ignored* counts. Eventually issues are fixed or suppressed (c) according to the heuristics described in previous sections, and those issues are subtracted from the *Ignored* counts.

In the example in Figure 3, two of the new issues are fixed. Those issues will never make it into the code repository, and without this desktop based instrumentation, this information would be lost. At the end of the first session (d), the snapshot sent to the central database contains all the final counts for this group.

This approach accounts for the possibility that changes may occur between sessions. At the start of each subsequent session, it identifies any issues that changed state or were added between sessions and reflects this in the *Overall* counts (e). The *New* counts are also reset to zero. During this second session, it is possible that some issues that were previously marked as *Fixed* reappear in the viewer (f). This effectively reduces the total number of fixed issues. This change is reflected in the next update to the server at the

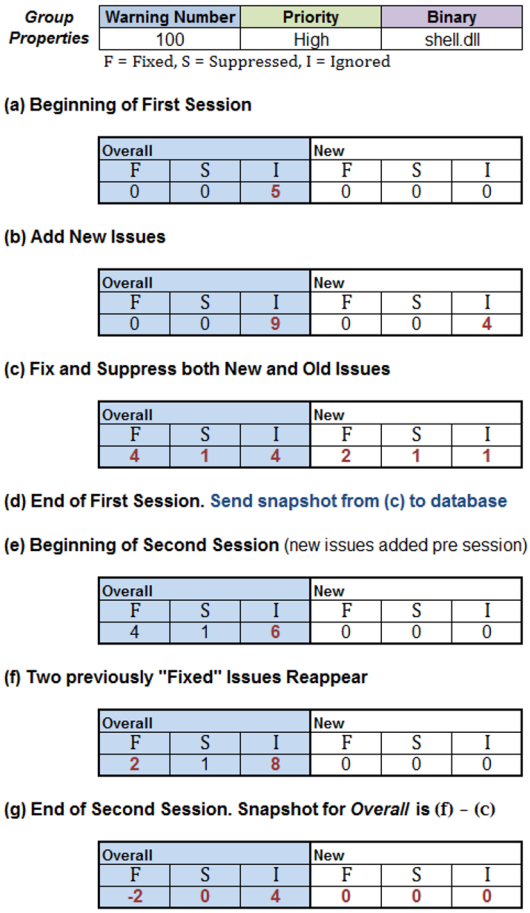

Figure 3: Counts for a Group of Issues

end of the second session (g), which includes a negative number for the *Overall Fixed* count. The update is computed by subtracting the snapshot at the end of the first session (c) from the snapshot at the end of the second session (f). In this way the central database reflects the state of the desktop after each update.

4.4 Supporting Heterogeneous Tools

Since our instrumentation is built on the static analysis viewer that is deployed with the build system, we can receive issues from many different tools including some that we do not control. For most functions in our instrumentation, this is not a problem since the build system provides common information about all issues irrespective of the tool it came from. For example, all issues have an associated warning number, and the path to the containing file. However tools sometimes used different conventions. For example some tools provided the path to the source file containing the issue, while others provided the path to a compiled binary.

Even more importantly, since we do not own all the tools, we cannot modify some of them to get the extra information we need to determine which issues were suppressed or which files were checked (as described above in Section 4.2). Even if we could modify them all, we would not want to come up with a complicated custom solution for each one.

To help deal with this, we created a small library that could connect an arbitrary tool to our instrumentation. Tools can simply include this library and call its high level functions to pass the information needed to the instrumentation. In addition, we made small modifications to some utilities shared by many PREfast plugins to collect the data we need so that we do not need to modify the plugins themselves.

Ultimately, we had to design our instrumentation to be robust to the fact that some features would not be supported by all tools. Specifically, the heuristics for determining which issues were fixed or suppressed would not be as accurate for these tools. This means that during our analysis we will need to focus on the data points and aggregations that come from tools that are fully supported. Fortunately many of the important tools including many PREfast plugins are among the tools providing full information and accurate inferences.

4.5 Other Implementation Considerations

In Section 4.1, we briefly mentioned that we use a few hook points in our static analysis viewer to add lightweight calls to the ATMetrics library. This viewer was being actively developed by another group and we did not want to disrupt their process or do extensive redesign. So we inserted our instrumentation in parts of the code where we can collect lots of data with minimal disruption and little overhead. Specifically, we instrumented the points where the viewer is started and shut down, where the list of warning is updated, and various event handlers. Our instrumentation is not a critical function of the viewer so we ensured that under no circumstances will these calls cause it to crash.

While most of our instrumentation focuses on collecting counts summarizing groups of warnings, there were some questions that required more detailed information. Specifically we wanted to capture details about individual issues in some cases. For example, we would like to collect instances of warnings from new analysis tools to better understand the contexts in which they occur. We would also like to track suppressed issues to see if they cause problems later in the development cycle. We use random sampling to choose a subset of issues with these characteristics, and transmit details about where they occur with the rest of our data.

5. TEST SUITES AND PRELIMINARY RESULTS

Prior to deploying our instrumentation, we created a test framework to establish expected outcomes in different scenarios and validate our implementation. We also constructed a micro-benchmark that we can use to configure ATMetrics for optimal performance, and created unit tests to validate the correctness of its components. We describe these test suites and some preliminary results in this section. The instrumented static analysis viewer is currently being deployed to some teams as part of a pilot. We expect to start receiving data after it is widely deployed.

5.1 ATMetrics Test Framework

The test framework simulates all the scenarios we hope to support and computes expected counts at different parts of the instrumentation process. These counts represent the number of warnings in the *Fixed*, *Suppressed* or *Ignored* groups for each scenario. Table 1 shows some of the scenarios in our test framework.

FIRST Session Scenarios
No new or moved warnings
Warning Fixed in Session
Warning Suppressed in Session
Warning Added In Session
Warning Added then Fixed In Session
Warning Added then Suppressed In Session
SUBSEQUENT Session Scenarios
Warning Fixed pre Session
Warning Suppressed pre Session
Warning Added pre Session
Previously Fixed Warnings Reappear pre Session
Previously Suppressed Warnings Reappear pre Session
Warning Added then Suppressed In Session
Previously Fixed Warnings Reappear in Session
Previously Suppressed Warnings Reappear in Session

Table 1: Some Scenarios in the ATMetrics Test Framework

There are three parts of the process used in this framework: the initial state, the pre-session activity and the session activity. The initial state specifies counts of warnings at the beginning of the process. Before a user's first session, these counts are all zero, but for subsequent sessions, these counts are the values recorded at the end of the previous session. For example the following table represents sample counts for a user who has used the static analysis viewer before[1]:

`InitState`

F	S	I	Sum
5	5	5	15

We represent counts from this part of the process with the `InitState` variable. The counts from the initial state will affect counts in other parts of the process.

The pre-session activity specifies counts of issues that were added or moved since the end of the last session, but before the beginning of the next session. Here we break up this part into two variables: `PreAdded` and `PreMoved`. `PreAdded` represents the number of issues added to the code base before the next session. Of course, we cannot add issues to the *Fixed* group because only previously seen issues can be fixed. Issues can be added to the *Suppressed* group if they are added to the baseline. In the following table, three warnings are added to the code base before the session, with one of them added to the baseline. This represents the scenarios "Warning Suppressed pre Session" and "Warning Added pre Session" from Table 1.

`PreAdded`

S	I	Sum
1	2	3

`PreMoved` represents previously seen issues that move to a different group before the next session. We use `PreMoved`$^-$

[1]The counts are for a group of issues with the same warning number, priority and binary as in Figure 3

to represent the values subtracted from each group, and `PreMoved`$^+$ to represent the values added to each group. We also give the counts in `PreMoved`$^-$ a negative sign so that all the counts add up to zero. Of course the values in `PreMoved` are constrained by the values in `InitState`. In the following table, one of the five fixed warnings (from `InitState`) is moved to the *Ignored* group. Similarly, one suppressed warning is moved. This represents the scenario "Previously Fixed/Suppressed Warnings Reappear pre Session".

`PreMoved`$^-$			`PreMoved`$^+$			
F	S	I	F	S	I	Sum
-1	-1	0	0	0	2	0

Finally, the session activity represents the counts observed during the session. Here we separate the counts for previously seen issues from those for new issues. Previously seen issues can be moved from one group to another (represented by `InMoved`), while new issues are first added (represented by `InNewAdded`) and then potentially moved to a different group later in the session (represented by `InNewMoved`). The counts in `InNewMoved` are constrained by the corresponding values in `InNewAdded`, while the counts in `InMoved` are constrained by the corresponding values from the sum of `InitState`, `PreAdded` and `PreMoved`. In the following tables, four old issues are fixed and two are suppressed during the session. Meanwhile four new issues are added, three of which are resolved (one fixed, two suppressed).

`InMoved`$^-$			`InMoved`$^+$			
F	S	I	F	S	I	Sum
0	0	-6	4	2	0	0

`InNewAdded`	
I	Sum
4	4

`InNewMoved`$^-$		`InNewMoved`$^+$			
S	I	F	S	I	Sum
0	-3	1	2	0	0

With this arrangement of counts into three parts, we can now easily compute the expected counts at the end of the session. Each count in the following table is computed by adding the corresponding values in all the tables above. For example, the final number of suppressed warnings is $5+1-1+2+2=9$.

F	S	I	Sum
9	9	4	22

With this final count information, we can also compute the expected update transmitted at the end of the session similar to the way this was computed in Figure 3. In other words, we compute the *Overall* counts by subtracting `InitState` from the final counts above, and compute the *New* counts by adding the values from corresponding groups in `InNewAdded` and `InNewMoved`.

Overall			New		
F	S	I	F	S	I
4	4	-1	1	2	1

We can use this test framework to represent different scenarios by changing the non-computed counts in the preceding tables. We use these counts as inputs to a test suite that exercises our implementation such that these values are observed at the different parts of the process. We can then compare the values transmitted by the implementation to the computed expected outcomes. In this way, we were able to validate that our implementation matches the outcome in the framework.

5.2 Other Test Suites

We created a micro-benchmark that contains configurable performance tests used to exercise our implementation under various loads. These tests simulate the critical developer activities, i.e. they generate warnings from a build and auxiliary information from tools, load issues into the viewer, and fix or suppress some issues. Based on the performance of the implementation, we can set thresholds on the number of issues that can be instrumented. Since this instrumentation is best effort, we prefer to turn it off and lose information if there are too many warnings than to allow the user to experience performance degradation. Our goal is for ATMetrics to cost less than 1% in overhead to the static analysis viewer.

We also created unit tests to validate the correctness of the implementation. The tests help to verify the robustness of the implementation. This means that even if a component fails, or the persistent state containing counts is corrupted, the viewer should not crash.

6. USER FEEDBACK

While our instrumentation focuses on accurately identifying some quantitative trends, we also need to complete the picture by talking to users. We conducted informal interviews with some users to get qualitative information about their experiences and perspectives on static analysis tools. We interviewed six senior developers who each have several years of experience using our tools. These interviews and the opinions stated are not representative of all the developers we support, but provide us with useful ideas that we can consider when making business decisions and that we can validate using our instrumentation. We overview some observations from these interviews in this section.

6.1 On Fixing Issues

Most users reported that they usually addressed all the high priority issues, and one user working with a security team aimed to fix all issues, including low priority warnings. When working on new code, users usually fixed issues just before checking code into the source repository. But many of our users also worked on code owned by someone else; in this case, they would wait for issues to be flagged by the overnight build and focus on those issues, to minimize changes to someone else's code.

Some users pointed out that close to milestones, the emphasis is usually on minimizing code changes, so only the most serious issues are fixed. Alternatively, during development cycles dedicated to cleaning up code (often after a major release), teams usually devote resources to wade through lower priority issues and warnings flagged in legacy code.

Obviously the type of warnings that interest users depend on the nature of the code they work on. Many of our interviewees worked with unmanaged C and C++ code that often

included large legacy components. Hence they were most interesting in problems related to potential buffer overflows. They also reported that many of the warnings pointed to missing annotations and unused variables.

Users perceived that most issues were worth fixing, though they mentioned that sometimes it was necessary to suppress issues or rewrite the code to make the warnings go away. One user mentioned that this would often happen when code conventions in legacy code did not match the expectations of the tools, and refactoring would be burdensome and potentially error prone. For example, different legacy components may have different conventions for dealing with error states including returning status codes or throwing exceptions.

In general, care was needed to effectively use tools on legacy code. One user reported that anytime a legacy routine was touched, the developer was expected to clean up any old warnings that may be present. But in general, users preferred to address issues in legacy code as part of a dedicated cleanup cycle. Some users credited an "auto-fix" feature in some tools (used to automatically correct some problems) as one property that made the cleanup process feasible. One user cautioned that assigning the task of cleaning up issues in legacy code to junior developers or contractors can sometimes lead to regressions because they are not as familiar with the code. Outside the cleanup cycle, any new legacy issues (i.e. issues found by new or modified static analysis techniques) need to be added to a baseline so developers can focus on problems in new code.

6.2 On the Importance of Static Analysis

All our interviewees felt that using static analysis was worthwhile, though most emphasized the relative importance of code review and testing. These different quality assurance methods find different kinds of problems and so all are necessary. Static analysis can be exhaustive, and increases the confidence users have in their code. Some users also reported changing their programming style to avoid static analysis warnings, leading to more maintainable code. Even with these sentiments, one user still expressed the importance of reducing the "noise" or false positives in tools, saying that tools with less noise are taken more seriously.

7. RELATED WORK

One popular framework for instrumenting software development activities is Hackystat [8, 7]. Hackystat is a general purpose framework that enables software projects to define, collect and analyze a wide variety of metrics. The data collection system we used, SQM, was designed for robust lightweight collection from millions of customers, not just software teams. We chose to use SQM because it is supported within Microsoft and widely used in many products. But we still had to make many of the same considerations and tradeoffs Hackystat users make including assuring data correctness, distinguishing files and projects, making the system configurable, and scaling to potentially millions of data points.

We believe this is the first project to focus on collecting metrics on static analysis usage from the desktop in an industrial context. In earlier work, Ayewah used Hackystat to collect metrics in a controlled lab environment where many of the challenges we describe do not come up [1]. Here the goal was to monitor the activities of user study participants who were asked to review a few static analysis warnings. The instrumentation enabled the authors to capture the user reviews, how long they spent on each issue, and which files and IDE resources they used when making their decisions.

Several related studies have explored various ways of figuring out which static analysis warnings are important, or understanding how users interact with static analysis. Some studies have users review warnings and provide feedback directly to researchers. Ayewah and Pugh observed in lab studies that users review issues fairly quickly and consistently, indicating a low cost associated with their usage [1, 2]. The studies also identified bug patterns that users viewed as most important. Lab studies are useful for focusing on a narrow scope of issues but results may not generalize to all scenarios. Other studies identify the important warnings indirectly by seeing which ones developers fix. Ruthruff et al. [15] create models to predict which warnings will be fixed based on the characteristics of warnings fixed in the past. Kim and Ernst examine open source projects that did not necessarily use static analysis tools to see which warnings were removed as a result of other quality assurance activities as a measure of the importance of the issues [10, 9]. These studies only capture warnings that make it into the code repository but we believe that many serious issues are caught on the desktop using other quality assurance processes. In future work, we hope to apply some of the techniques in these studies on the warnings we observe on the desktop.

Ultimately, we would like to more concretely demonstrate the cost benefits of using our tools. We can learn from the experiences of other researchers evaluating tools and unit test order. Jaspan et al. measure the return on investment of using a static analysis tool by comparing it to the costs and benefits of manual testing [6] and conclude that static analysis is very worthwhile. Do et al. use cost benefit analysis to change the order in which unit tests are run to ensure that the most important tests are run first [5].

8. CONCLUSIONS AND FUTURE WORK

We have successfully built and deployed an instrumentation framework to collect more metrics on how our static analysis tools are used. In particular we can now collect data on activities on the desktop that would previously have been lost. This is of particular value to us because we believe that many of the warnings that our users receive are fixed in the developers' workspaces before they commit any code to the source repository. Through this project, we learned that instrumenting analysis tools on the desktop is feasible, but can be challenging because of the need to infer developer activities with limited information and because of heterogeneous tools.

Going forward we plan to use this data to identify issues that occur a lot, and those that are fixed or suppressed at high rates. We can use this information to tweak our analysis or change the severity levels of warnings. We also plan to look for trends among developers and groups, and compare our observations with internal business metrics used to measure the quality of components. This information will enable us to make a stronger case for the return on investment from using static analysis. In the future, we expect that this work can be adapted to instrument some of the other analysis technologies we support including code coverage and binary analysis.

9. ACKNOWLEDGMENTS

The authors would like to thank Sunny Chatterjee for providing technical advice and supporting the deployment of ATMetrics.

10. REFERENCES

[1] N. Ayewah and W. Pugh. A report on a survey and study of static analysis users. In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5, New York, NY, USA, 2008. ACM.

[2] N. Ayewah and W. Pugh. Using checklists to review static analysis warnings. In *DEFECTS '09: Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, pages 11–15, New York, NY, USA, 2009. ACM.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[4] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, New York, NY, USA, 2002. ACM.

[5] H. Do, G. Rothermel, and A. Kinneer. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Softw. Engg.*, 11(1):33–70, 2006.

[6] C. Jaspan, I.-C. Chen, and A. Sharma. Understanding the value of program analysis tools. In *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 963–970, Montreal, Quebec, Canada, 2007. ACM.

[7] P. M. Johnson. Requirement and design trade-offs in hackystat: An in-process software engineering measurement and analysis system. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[8] P. M. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita. Improving software development management through software project telemetry. *IEEE Softw.*, 22(4):76–85, 2005.

[9] S. Kim and M. D. Ernst. Prioritizing warning categories by analyzing software history. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 27, Washington, DC, USA, 2007. IEEE Computer Society.

[10] S. Kim and M. D. Ernst. Which warnings should i fix first? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54, New York, NY, USA, 2007. ACM.

[11] J. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. Rajamani, and R. Venkatapathy. Righting software. *Software, IEEE*, 21(3):92–100, May-June 2004.

[12] Microsoft. Software Quality Metrics, July 2005. http://www.microsoft.com/windowsvista/privacy/privacy_b1.mspx#EAFAC.

[13] Microsoft. Microsoft Auto Code Review (OACR). MSDN Library, October 2009. http://msdn.microsoft.com/en-us/library/dd445214.aspx.

[14] Microsoft. SAL Annotations. MSDN Library, July 2009. http://msdn.microsoft.com/en-us/library/ms235402.aspx.

[15] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2008. ACM.

[16] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136, New York, NY, USA, 2006. ACM Press.