Speeding up Cross-Platform Testing – A Case Study

Jeffrey Weston
Jeffrey.Weston@microsoft.com

Abstract

Microsoft Office for Mac has a very efficient developer process for porting code across platforms (from Windows to Mac). Unfortunately for testers, it creates many new bugs across our entire suite of applications, requiring a large amount of resources and time to test and find them all. Because the bugs appear at random each time we port code, test cost reductions are incredibly difficult.

To overcome this problem, we investigated a sample size of bugs to find the root cause of the defect. We identified the exact code change that introduced the defect, collected data on the process the developer was following and on the code context in which the bug existed. Categorizing the bugs by common attributes produced bug patterns that we could then act upon.

This technique provided valuable information that allowed us to extinguish (or at least severely limit) an entire category of defects, scope our testing to only the riskiest areas, and identify which test tools and techniques would give us the most value in terms of finding the bugs after future merges.

This paper describes an overview of the project as well as a list of requirements, pre-conditions and a template that other teams can use to reproduce this root cause analysis method, enabling teams to identify their own bug patterns and make investments to help find the bugs more efficiently.

Using this root cause analysis method is an effective means to help test organizations reduce costs while increasing engineering and product quality.

Biography

Jeffrey Weston is a Senior Software Tester with over nine years of testing experience. He currently works in the Office for Mac group at Microsoft, where he is architecting a new test strategy to testing ported code. Previously he has architected and driven performance testing for Microsoft Office for Mac 2011, as well as tested and shipped three versions of Microsoft Word for Mac. His interests include software metrics, root cause analysis, and improving engineering quality.

Jeffrey Weston has a B.Sc. in Computer Science from the University of Victoria in British Columbia, Canada.

1 Introduction

A Microsoft Office product like Word or Excel has existed on the Macintosh platform since the beginning. Up until 1997, Office for Windows and Office for Mac were co-developed at the same time by the same developers. In 1997, the Macintosh Business Unit was formed and a separate group of Mac-specific engineers took over.

In order to stay compatible with its Office for Windows counterpart, Office for Mac ports code written by Office for Windows engineers and integrates that new code into the existing Office for Mac code base. This enables new features and maintains compatibility across the suites for our customers. Internally, this porting and updating process is called “merging”.

The merging process is a very efficient means of getting code changes from Office for Windows into the Office for Mac codebase. The majority of the changes are automatically merged; the remaining work includes manually updating and integrating the platform-specific needs. Using this process, tens of thousands of code check-ins can be merged quickly.

However, just because a process is efficient for developers does not mean that it is efficient for testers. The bugs that result from this process can appear, from a black box perspective, completely at random. They can appear across the Office Suite while varying widely in severity.

Using a root cause analysis approach modeled on airplane crash investigations, I was able to understand why certain merge bugs occur, and come up with recommendations on how to prevent and find them more efficiently.

This paper will consist of the following:

- An overview of the merge process
- The test problems the merge process creates
- An overview of the root cause analysis approach used
- The bug patterns found from the analysis
- Recommendations on how to prevent or find those bugs more efficiently

2 General merging process

To understand where and how bugs come from, I thought it was vitally important to understand the process that can produce them.

The merging process works as follows:

1. Office for Mac applications have an established set of “core” source files that their developers always try to keep in sync with the Office for Windows codebase.
2. At a point in time, a “snapshot” of the Windows Office code is taken and the matching set of core files are merged into the set on the Macintosh side.
3. This merging produces two outcomes:
    a. Code changes that occur automatically
    b. Code changes that result in a conflict. (A conflict occurs when an Office for Windows developer has updated code in the same location as an Office for Mac developer. Conflicts must be resolved manually.)
4. At this point, the code often doesn’t compile due to missing or changed APIs, usage of the language that is unsupported by the Macintosh compiler, and missing code that was not included in the merge.
5. Developers then work to resolve conflicts, update integration points, and work though compiler errors. This process can take anywhere from days to weeks, and occasionally months.

6. Once the code compiles, developers run some basic build validation tests and fix any bugs those tests reveal.
7. Once the build passes the build verification tests, it is finally handed over to Testers.

3 “Random” bugs

Although the merge process is efficient for developers, it poses problems for testers. Developers can work on the code for weeks or months until a build is testable. Over that period of time, developers change hundreds of files, editing, deleting and adding thousands of lines of code. This code churn is not isolated to specific features, but occurs across the entire project.

Due to the breadth of churn, developers lose track of key risk areas or are unaware of high-risk changes, resulting in testers having to do a full black-box test pass in order to find defects.

Looking at churn statistics and summaries from the source control tool can also be problematic. There are two types of churn in the merge process:

1) Automatically changed
2) Manually changed code

Historically, code that is changed automatically to match the Windows code introduces fewer defects than code changed manually. Unfortunately, the source code change details do not differentiate between these two types of churn.

This large amount of churn makes bugs appear at random across the product rather than isolated on a single feature; for example, inserting a hyperlink could cause a crash, a memory could leak in a text styles component, or widespread yet subtle document corruption could be introduced.

Because of this perceived randomness of bugs, testers resort to running as many tests as possible across a wide breath of features. Blindly running all the tests isn’t an efficient process. An early internal study I did showed some teams spend over a day-and-a-half of testing for every one bug they find. When teams have exhausted their test cases, or simply run out of time, it can still be unclear whether they have covered all the key areas.

4 Software Crashes and Airplane Crashes

At the time, I was sure bugs did not get introduced randomly into the product. They are commonly introduced due to initial conditions, circumstances and actions. If I could understand how bugs got introduced into the product from the merge process, I could become better at predicting where these bugs will likely occur in the future and then provide a set of recommendations to prevent or find them quicker.

The technique I chose to deploy was root cause analysis, and I decided to model it based on how airplane crash investigators do their job.

Airplane crashes are not very predictable, either: they seemingly happen at random, but, like software bugs, they don’t. They occur due to specific events and circumstances, and it is the airplane crash investigator’s job to find out what those were. They have to determine how an extremely complicated system of parts and people failed; they have to understand what went so horribly wrong.

I found several parallels between finding the root cause of a plane crash and finding the root cause of a software bug that helped me through the process of understanding how merge bugs occur.

4.1 Not a single cause but a chain of events

An airplane is a complex system, just like a software product. When a fault in the system results in a problem, there are often multiple reasons for why that fault occurred. Take, for example, the crash of the Concord, also known as Air France Flight 4590. There were several probable causes of the disaster. (BEA 2002)

1) During takeoff, the tire of the Concord ran over a piece of debris from a DC-10.
2) The fuel tank became damaged from a blown piece of tire hitting it.
3) Broken wires in the landing gear ignited the leaking fuel.

A chain of events can also cause software bugs. For example, a bug where images failed to appear in a Word document had the following causes:

1) After the initial merge, the source file required many additional small changes.
2) During the tedious process of updating these changes, a piece of critical Macintosh code was deleted.
3) Execution of the code resulted in a failure when a document was opened with images.

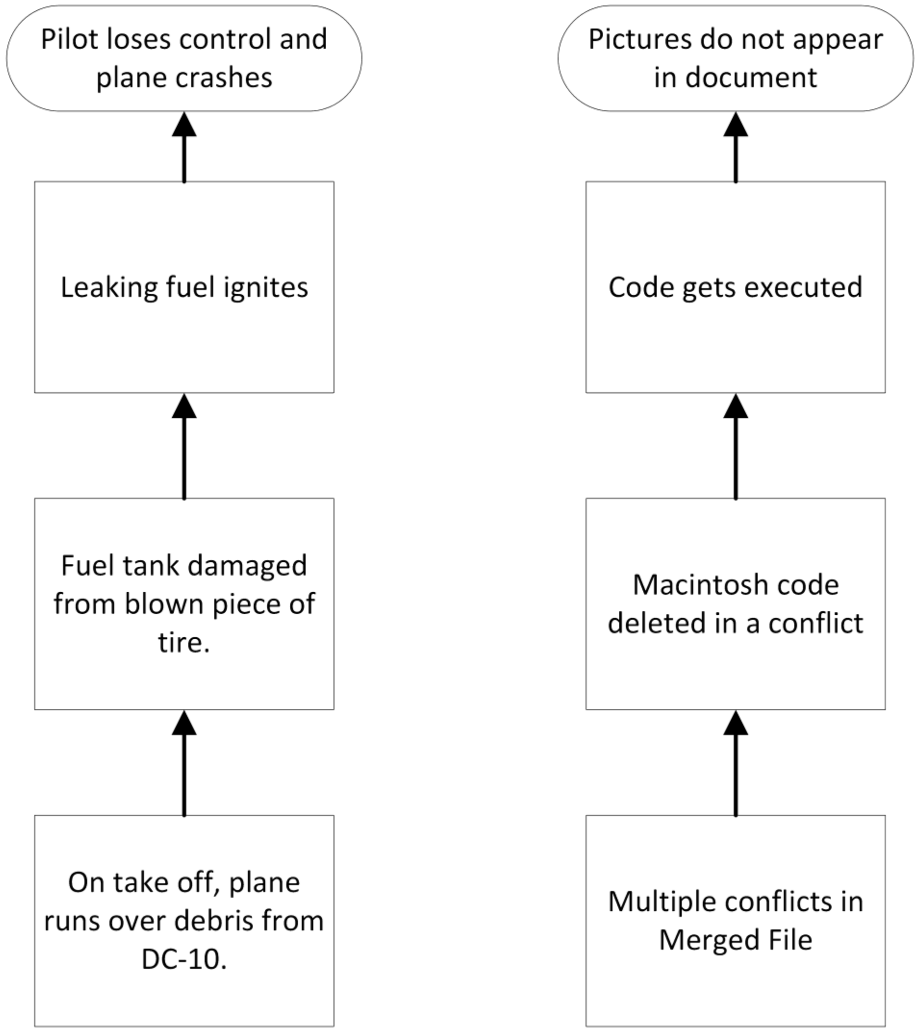
Pilot loses control and plane crashes
Leaking fuel ignites
Fuel tank damaged from blown piece of tire.
On take off, plane runs over debris from DC-10.
Pictures do not appear in document
Code gets executed
Macintosh code deleted in a conflict
Multiple conflicts in Merged File

Figure 1 Chain of events leading to a plane crash and software bug

Understanding the full chain events allowed me to find solutions that could break the chain in multiple places instead of only preventing the initial cause.

4.2 Using logs to create a history of the event

Airplanes often have flight data recorders and cockpit voice recorders. The purpose of these tools is to help the investigators understand what was happening before and during the events in question. For example, the investigators of the Concord crash used the flight data records and cockpit voice records to know when the fire alarm first went off and how the pilots responded to it. (BEA 2002)

At Microsoft, we have a source control system that records everything we change in the code base.

Using the source control system, I started with the change that fixed the bug and the developer's comments associated with that change to go back through the history and find out when the bug was first introduced into the system.

The source control system could also identify the phase of the merge process in which the bug was first introduced by using the dates, the developer comments, and the type of changes being made.

1. Automatic merging of the code
2. Resolving a conflict
3. Fixing a compiler issue
4. Fixing a build verification test failure

4.3 Collecting information on the context of the bug

Flight data recorders and source control logs tell only one side of the story. Since my goal was to develop recommendations to improve the merge process, I needed to understand a lot more context and wider knowledge of the environment to make the most effective recommendations.

For example, crash investigators analyzed many parts of the Concord crash, all the way from the management structure of the Concord division of Air France and right on down to the position of the seats in the cockpit. (BEA 2002)

In software, I wanted to collect as much information as I could within the time constraints I had.

Some of the information I collected included:

The context of the code that the bug was in:

- Did the bug exist in code that was identical to Windows code?
- Did the bug exist in Macintosh code?
- Did the bug exist in code that was heavily mixed between shared Windows lines and Macintosh lines?

Tools that could have helped find the bug during the testing phase:

- If it was found manually, could it also have been found via automation?
- Could it have been found via code coverage in that area (i.e. by executing that code)?
- Could it only have been found via a human doing manual verification (i.e. a visual problem like pictures not showing up)?

4.4 Reconstruction

Sometimes, in order to understand what happened, investigators will reconstruct the plane from pieces of the wreckage. In the Concord disaster, they reconstructed the damaged wing to understand the cause of the fire. (BEA 2002)

In the software case, when I did not understand how a developer could make such a mistake, I reconstructed that part of the merge. This involved:

1. Learning from the developer how they personally performed the merge, such as the specific tools they used and the commands they executed.
2. Using the source control history to revert the source files back to the pre-merge state.
3. Executing that part of the merge myself.

The reconstruction process allowed me to see what the developer saw and have a better understanding of how the developer could have introduced the error. For example, I found that when some files are merged, they produce many conflicts that need to be manually resolved. However, the diffing tools some developers used did not provide all the information needed to resolve the conflict.

5 Findings

From the investigation, I identified four patterns of bugs introduced by the merge process.

1. **Incorrect conflict resolutions**: Manual conflict resolution often resulted in Macintosh code being hidden amongst matching Office for Windows code that was inadvertently removed.
2. **Churn when integrating Office for Windows changes into Macintosh code**: Our code has areas specifically marked as Macintosh code. Usually this code is within the same source file, or it is closely dependent on Office for Windows code. If the Office for Windows code changes, the corresponding Macintosh code must be manually updated to match.
3. **Churn when integrating Office for Windows changes into older legacy code:** Teams often have source files that were originally ported from Windows Office, but they are not merged regularly. These files become out-of-date and must be manually updated to conform to Windows Office changes.
4. **Ported Office for Windows bugs:** Not all the Office for Windows code we merge is fully complete. It can still contain bugs in incomplete features, and these bugs can simply be copied into Macintosh code.

The following sections explore the four patterns identified by the defect analysis process in order to explain how bugs in the merge process occur.

5.1 Incorrect conflict resolutions

A conflict can occur in a three-way merge. Word for Windows and Word for Mac start off from the same code (ORIGINAL file). WinWord makes a change in the code (WINDOWS DEV). Word for Mac makes a change in the same place (MAC DEV). The code is then merged together. A conflict requires the developer to choose which changes to accept or rewrite it to make it work correctly.

However, during conflict resolution, the developer may not see a critical line of Macintosh code. If the developer doesn’t see any reason to keep the Macintosh code, they will often choose WINDOWS DEV to ensure the Office for Mac codebase matches Windows Office as much as possible.

This leads to Macintosh code getting deleted.

Example

```
        if (FPrinting() && list.Last < 0)
                EnumList(fFalse);

>>>> ORIGINAL ++++++++++++++++++++++++++++++++++++++++++++++++
        Assert(cache.isEmpty == fTrue);
        lserr = displayLine(theText, CLIPPED);;
        Assert(err == errNone);
==== WINDOWS DEV ++++++++++++++++++++++++++++++++++++++++
        NewAssert(cache.isEmpty == fTrue);

        if (FInvalid(cache)
                err = errNone;
        else
                err = displayLine(theText, CLIPPED);
        NewAssert(err == errNone);
==== MAC DEV ++++++++++++++++++++++++++++++++++++++++++++
        Assert(cache.isEmpty == fTrue);
        err = displayLineMac(theText, CLIPPED);
        Assert(err == errNone);
<<<< END ++++++++++++++++++++++++++++++++++++++++++++++
        destroyGroup(&theGroup);
        if (err != errNone)
```

In the above example code, a Windows Office developer has changed their assert API and added an additional `if` statement. The developer introduced a bug by accepting the WINDOWS DEV changes to resolve the conflict. This had the effect of changing the `displayLineMac(...)` API in the MAC DEV block back to `displayLine(...)`, which introduced a bug in Office for Mac.

Traits

These bugs tend to be:

- The result of simple human error (also known as pilot error).
- They are not specific to the feature being worked on.
- Increased risk of occurrence as the number of conflicts in the file increases.

Quite often the correct resolution is to accept the WINDOWS DEV block. If developers have to do this over and over, they can get fatigued and start missing some cases of Macintosh code hiding within Office for Windows code changes. This particular file had over sixty conflicts.

5.2 Churn in Macintosh code causes simple mistakes

Changes in Office for Windows code may require us to manually modify other parts of our code base. For example, Office for Windows modified an API called `DrawMark` to add another parameter. This function was also called in a Macintosh code block. After the merge, the code wouldn't compile because the function signatures no longer matched.

```
#ifdef MAC_ONLY_CODE_BLOCK

...
if (fDisplay)
     {
<        DrawMark(pFoo, pBar, pMark, pLimit, pStart, pParam, pClip, pObj);
>        DrawMark(pFoo, pBar, pMark, pLimit, pStart, pNew, pParam, pClip, pObj);
     }
     else
     {
         ValidateEndmark(hwwd, prcw, emk, dypAltSpacing);
     }

#endif // MAC_ONLY_CODE_BLOCK
```

The developer had to manually update this function call to use the newly added parameter. However, the developer made a simple mistake and inserted the parameter into the wrong position; the `pNew parameter` should be the second from last parameter, rather than the third from the last. Because the parameter types all still matched the signature, the code compiled and the developer assumed the change was correct.

5.3 Churn when integrating Office for Windows changes into older legacy code

Office for Mac contains source files that exist in Office for Windows and have not been freshly ported over in a very long time. Over the years, these files can diverge significantly from the Windows source and can no longer be merged automatically.

```
Original                        Changed

hbr = CreateBrush(pToolTip);  rtbr = NewCreateBrush(rtNil, pToolTip);
FillRect(hdc, prc, hbr);      NewFillRect(rt, prc, rtbr);
DeleteBrush(hbr);             NewDeleteBrush(rt);
SetTextColor(hdc, pToolTip);  NewSetTextColor(rt, pToolTip);
```

In the above example, the `DeleteBrush` function was manually updated to use new API calls that the Office for Windows team implemented. This required manual changes that inserted a bug because the developer used the `rt object` in the `NewDeleteBrush(...)` function instead of `rtbr`. As a result, the program crashed executing the next line because the `rt` object was deleted by `NewDeleteBrush` invocation.

5.4 Porting over of Office for Windows bugs

When we merge code, we will undoubtedly port over bugs that have yet to be found or fixed in Office for Windows code. This pattern is the only pattern I discovered where there is no interaction between Windows and Macintosh code in Office. These bugs will most commonly occur when porting over large blocks of new code that are tightly coupled.

6 Recommendations to break the bug chain

Once I identified our defect insertion patterns, I developed recommendations for breaking the chain of events leading to discovering bugs late in the development cycle.

At each stage in the process, I tried to determine how to prevent the bugs being generated and what our teams could do to find them more efficiently—except, of course in the test phase, where the bug is already checked in to the code and thus cannot be prevented anymore.

The table below shows an example of the conflicts pattern.

Table 1 Bug prevention and targeting

Stage	How to *prevent* bugs from getting checked-in?	How to *find* bugs more efficiently?
Design Phase (Pre-Merge)	Prepare the code to reduce the number of conflicts before the merge starts.	Mark up unmarked Mac code so developers can clearly see the Mac code.
Implementing (During Merge)	Review each conflict resolution specifically looking for dropped Mac code.	Have Test Code Review the conflicts after check-in, looking for dropped Mac code.
Testing Phase (Post Merge)	N/A	Use code coverage analysis to determine whether conflicts resolutions were covered.

6.1 Reduce the possibility of conflicting code

Resolving merge conflicts is a manual and largely tedious process, which can easily result in simple mistakes.

One possible way to avoid the tedious work is to prevent it from being required in the first place. If one knew—in advance of the merge—of conflicting changes, such as changing the assertion API from "`Assert(...)`" to "`NewAssert(...)`", one could manipulate the files before the merge to prevent conflicts from occurring.

The following process shows how to edit the ORIGINAL file to match the WINDOWS DEV and MAC DEV and thus avoid the tedious conflicts:

1. Find the baseline file (ORIGINAL).
2. Find the new Office for Windows file (WINDOWS DEV).
3. Find the current Office for Mac file (MAC DEV).
4. Bulk-edit the MAC DEV file, changing `Assert` to `NewAssert` to match the WINDOWS DEV file.
5. *Temporarily* bulk-edit the ORIGINAL, changing `Assert` to `NewAssert` so all three versions now match.
6. Merge.
7. Revert the change to the ORIGINAL.

This process has already been integrated into our toolset and has shown to prevent up to fifty percent of the conflicts.

6.2 Proactively markup the unmarked Macintosh code before the merge

Bugs occurred during conflict resolution because the Macintosh code was removed erroneously, often because it was embedded in Windows Office code that was updated and not identified as Macintosh code.

If we were to clearly mark the code as Macintosh code—by using compiler `#ifdef` flags or comments, for example—it could increase the likelihood a developer would see this code and preserve it when the conflict is removed.

Marking all Macintosh code, however, would be costly. It could take several hours per file to do this and there could be hundreds of files. For projects that have a lot of source files that get merged, this could become very expensive.

6.3 Concentrate code reviews on conflicts

Code reviews can be effective at preventing bugs. However, the volume of changes that occur during a merge is far too large to be reliably reviewed. This has resulted in only cursory reviews or selective reviewing of specific changes.

Reviewing only conflicting change resolutions could prevent defects repeatedly being inserted during conflict resolution.

Code reviews could be very effective here, particularly if reviewers were instructed to focus on looking for deleted Macintosh code.

If reviewers were watching for only one thing, they might be more likely to detect it.

6.4 Target Macintosh code in specialized test suites

Based on the bug context data collected, this study revealed that bugs tend to be introduced where developers must manually update the code after a merge, which is required whenever Macintosh code is intermixed with Windows Office code.

Having a more concrete theory on where bugs tend to be injected gives us a better idea of where to focus testing efforts. With this knowledge, testers could mark existing tests or create new tests to specifically target merge testing.

Code coverage can tell us whether or not tests exist for these higher risk areas of code. Testers could create sets of merge regression suites to target the higher-risk Macintosh code blocks. Developers could then run these larger more targeted sets of tests in addition to their build verification tests.

Developers running and finding their own bugs is more efficient than having a test engineers find, reproduce, and log a bug and then verify the fix afterwards, particularly if the bugs are simple basic functionality issues (as is the case with most merge bugs).

6.5 Use bug lists to identify bugs in unfinished Windows Office code ported to Macintosh

Bugs can exist in code we port over from Windows Office when it is incomplete.

It is not realistic to try and prevent these bugs since their creation is out of the Mac Office engineer's control.

To find them more quickly, a Mac Office tester could look through the bug database on the Windows Office side to see what bugs are still active at the time the code was protected.

7 Conclusion

Before the merge study, our team did not know how to optimize our testing or improve the quality of newly merged code, but we now have a clearer idea of how to tackle those issues.

Root cause analysis revealed that a chain of events could cause bugs. By leveraging our source control logs, collecting data on the context in which the bug occurred, and reconstructing the events that caused the bug, I was able to identify bug insertion patterns. I used these patterns to propose recommendations for both preventing bugs from being generated where possible, and to detect other bugs earlier in the merge process.

Based on these findings and recommendations, our development and test teams are now investing in:

- educating the teams about the patterns and recommendations,
- changing manual procedures to make conflicts more accessible for reviews,
- automated tools such as code coverage and scripts to detect deleted Macintosh code.

By treating software bugs in a similar way to airplane crashes, I was able to turn a seemingly hopeless situation (random bugs) into an actionable one for our team. The recommendations I developed provide a clear direction on where to invest our time and energy in order improve the quality of our code base.

References

BEA. *Accident on 25 July 2000 at La Patte d'Oie in Gonesse (95) to the Concorde registered F-BTSC operated by Air France.* Investigation, Bureau d'Enquêtes et d'Analyses pour la Sécurité de l'Aviation Civile, Paris: Bureau Enquêtes-Accidents, 2002.