Evaluating the Infrastructure of Software Applications
Cherry Keahey Owen
Texas Tech University
owen_c@utpb.edu

1. Research Area

This research will focus on Interface Design and Evaluation.

2. Technical Problem to be Solved

With the popularity of Internet applications comes an increase in distributed systems and increasingly complex software. With the added complexity comes an increased risk of errors. Developers and researchers have traditionally addressed the complexity issue with modular design and information hiding [1]. These methods definitely have their advantages, but also have some frequently overlooked disadvantages. Modular design requires interfaces between the modules, but “module interface complexity is a prime cause of software errors” [1].

The purpose of this research is to investigate the interfaces between user programs and the infrastructure on which they depend. The primary goal is to find and handle undocumented or “hidden” interfaces [2]. Since the infrastructure is specifically built to hide the cumbersome details of managing the operating system and networking functions, the developers of the user programs do not usually know all of these details. Indeed, it should not be necessary for the developers to concern themselves with such details. The infrastructure applications provide a set of interfaces for interaction with the user programs. The developers should be able to “plug in” through the defined interfaces without knowing all the inner workings of the infrastructure. However, when some of the infrastructure functions have defects or do not work as documented, the interfaces might no longer be clearly defined. Unexpected signals and return values can be sent to the user programs. Since they are unexpected, the user programs are not prepared to handle them, and the results can be unpredictable.

Hidden interfaces make defects much more difficult to find and fix. The solutions to software problems do not always address the sources of the problems because the sources are so well hidden in the infrastructure. Instead a quick fix is applied to fix the problem for one particular instance or application. If this fix does not address the source of the problem, the problem is likely to reoccur as different inputs are applied to the application or as the function causing the problem is called by other applications. As an example, consider a situation where the details of memory allocation are hidden within the infrastructure. Most of the time enough memory is available and can be allocated as needed by the user program. However, when the required amount of memory is not available, the allocation might fail and an error code might be returned to the calling function. If the user program does not check for this code and handle the situation, the results can be unpredictable. Hundreds of different error codes can be returned by Windows operating systems. The other infrastructure applications can add to the number of error code possibilities. A user program cannot practically check for every possible code. The interface definition should clearly state which codes can be returned for each function that can be called. However, in some cases a function could fail to provide this information. Then other
functions that call the poorly documented function could fail to check for the undocumented return code. If the return code indicates a problem, then all of the functions in the call chain leading to the current function are subject to unpredictability resulting from failing to handle the problem [3]. The user program might be in the call chain, but the poorly documented function might be buried in the infrastructure and not visible to the application developer.

3. Justification of the Importance of the Problem

One of the findings of the “NSF Workshop on a Research Program for the 21st Century” was that more no-surprise software is needed [4]. The current software has too many surprises. This seems to contradict the statement that “software is deterministic” [2][5]. Starting from a given state and providing a given sequence of inputs should result in the same response every time. In reality, a given sequence of human user inputs is altered in many ways by non-human user inputs. These non-human inputs include return codes from the operating system, changes in shared memory, input from files that may have been altered by unknown sources, and so on. If these other inputs are not handled correctly, then many surprises can result. The only way to have no-surprise software is to handle all inputs to every function, including those from other functions within the application, those from human user interactions, those from machines, and those from non-human external interfaces. As long as multiple processes are allowed to run at the same time in a multiprocessing or multitasking environment, messages passed from one process or thread to another must be considered as input. Not only are the inputs important, but the sequence and timing of the inputs can also affect the results of the processing. Therefore, both static and dynamic analyses of the infrastructure applications are needed.

Many hours are being spent by developers and end users trying to work with “buggy” software. Often hours of work have to be redone due to a hanging process. A novice user might try the same thing over and over because the user manual says it will work, and not realize that a bug is causing the failure. If the time of all software users is considered, and each user spends even a few minutes a week on problems resulting from buggy software, then any time spent on the root causes of the software problems would be worth while. Due to time-to-market pressures, it is not always practical for developers to do the in-depth analysis of the infrastructure interfaces that is needed. Researchers should take the responsibility to address this problem.

4. Justification that prior Research Has not Solved the Problem

It is evident from looking at the Microsoft Knowledge Base (MSKB) that the problem of buggy infrastructure applications has not been solved. The “List of Bugs Fixed in Windows 2000 Service Pack 2” shows that over 500 bugs have been found and fixed for Windows 2000 alone [6]. This might cause one to think other bugs exist that were not found and fixed in service pack 2. In fact, since the service pack 2 list was published in May of 2001, at least 565 new bugs have been fixed with hotfixes [7]. In addition to new bugs being found on an almost continuous basis, the old bugs that have been fixed with hotfixes frequently reappear in subsequent versions of Windows [8]. Due to the complexity of Windows operating systems and other infrastructure products and the constantly changing environment in which they run, developers are fighting a constant battle to prevent or fix the problems that can arise. There is never enough time to sufficiently evaluate the infrastructure as a whole. Even if more time were available, the
infrastructure would change before the evaluation could be completed. The integrity of the infrastructure is a moving target. Dedicated research is needed to improve the situation.

5. Research Hypothesis

The null hypothesis: “The inputs coming from the infrastructure are all documented.” Based on preliminary results, the research has a high probability of showing that the null hypothesis is false.

6. Proposed Solution

Track infrastructure interface problems to their source. This takes a significant amount of time compared to supplying a quick fix. However, in the long run much time will be saved by fixing the source of the problem, since this source is possibly causing other problems. The source of the problem should be traced to the function level. Sometimes the source of the problem might be a combination of several functions, their interfaces and the timing of their execution. A given set of functions might work fine for some period of time, and then suddenly fail because the sequence of inputs is altered by timing and the instruction being executed when a specific input is applied. As far as the user is concerned, the sequence of inputs is the same. However, those “hidden” inputs from the operating system are not always considered, and are very sensitive to timing and coordination with the rest of the operating system.

Just identifying those functions used in the infrastructure which have the potential to cause problems, and documenting the potential problems and how to avoid them should make working with the infrastructure more predictable.

For some situations, filters can be written to handle problems before they propagate to the user programs. The Microsoft Visual C++ language provides a filter option to allow developers to write filters to check for return codes and throw exceptions to handle the problems specified by the return codes [9]. This option could be used or separate code could be written for the filter.

7. Expected Contributions

One or more filters will be written to show how a filter can handle specified types of problems related to interfacing with the infrastructure.

Results of experimentation will be documented in the dissertation and other publications, and made available for use in future research efforts. These results will include a list of functions used in the infrastructure which have undocumented interfaces that are causing errors or have the potential to cause errors. Correct documentation and a list of problems found that are caused by each function will also be provided.

8. Methods

This research focuses on infrastructure applications that are mostly closed source. Although much of the research will also apply to open source, issues specific to open source will be reserved for future research. Some of the infrastructure applications to be included are the
Windows 2000 operating system and C and C++ libraries. Since many operating systems and other infrastructure applications are written in C or C++, any research on the C and C++ libraries will apply to much of the infrastructure.

The Microsoft Knowledge Base [10] has been searched for known problems with the Windows 2000 operating system and applications running on this platform. Problems that relate to the interface between application programs and the infrastructure have been selected and considered. The problems have been classified, and one class, “problems resulting in access violations,” has been chosen to be used in the research. The other classes will be reserved for future study.

The access violation problems will be analyzed as follows:
- Look at the problem report to see if it mentions any functions or other causes of the problem.
- Search the Internet for any available information on the problem.
- Try to reproduce the problem.
- Look at the stack to see what processes and threads were running.
- Look at the processes and threads to see what functions were involved.
- Try to determine which of the functions were likely to produce this type of problem.

This analysis should produce a list of functions that could possibly be causing the problems. For each of these functions, the following questions will be asked:

1. Does this function return any undocumented return codes or values?
2. Does this function throw any undocumented exceptions?
3. Does this function require any undocumented resources such as data files and include libraries?
4. Does this function produce results different from what is documented when the documented preconditions are met?
5. Does this function produce results different from what is documented when the documented preconditions are not met?

Metrics will be collected in the form of the number of functions that have yes answers to each of the 5 questions. Each function will also be documented individually. The name of the function will be recorded, along with the yes or no answer to each of the questions. For yes answers, an explanation will be given of how the function execution differs from the documentation.

Various methods will be used to test and analyze the functions in order to answer the questions. These methods include using available debuggers, looking at any available source code, reviewing documentation, and using fault injection [12].
After sufficient information has been gathered, one or more filters will be written to address the problems found. Testing will be done with and without the filters to see if a problem that exists without a given filter is eliminated with the filter applied.

8. Summary

Current infrastructures on which most end user applications depend are very complex. These infrastructures are composed of various heterogeneous commercial-off-the-shelf (COTS) products from different vendors [13]. In many cases the COTS products are not designed to interoperate with each other [14]. In addition, the software in these products is often buggy, as evidenced by the over 500 hotfixes for Windows 2000 alone in the year 2001 [7]. The current research is designed to make the infrastructure more dependable. The behavior of a Windows based infrastructure is being studied, and filters are being developed for more reliable interfacing between user applications and the infrastructure. In addition, problems with the infrastructure are being documented, specifying the source of each problem and how to avoid or solve it.

9. References