Workshop on Reliable Embedded Systems, in conjunction with the 20th IEEE Symposium on Reliable Distributed Systems
October 28, 2001, New Orleans, LA.

Testing the Portability of Desktop Applications to a Networked Embedded System

Michael W. Bigrigg and Joseph G. Slember
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh, PA 15213
bigrigg@ices.cmu.edu, jslember@ece.cmu.edu

Abstract

Applications that were engineered for desktop environments are often ported to networked embedded systems and mobile environments which have a higher rate of errors due to variable and intermittent connectivity. In embedded systems there is a lack of additional hardware resources, which then requires the software to handle far more and to be increasingly robust. This paper examines the ability of common desktop applications to gracefully handle error conditions when ported to an unreliable networked embedded system. The focus of the testing is the ability of the GNU binutils and textutils to catch and properly handle error return values from the Standard C I/O library.

1. Introduction

In reducing time to market and in an effort to not reinvent the wheel, software reusability plays an important role. Much effort has been placed on the reuse of software components as well as porting entire applications to new architectures. We are addressing the porting of desktop applications to an unreliable networked embedded system.

In a desktop environment, while unreliability is possible, it is not often accounted for. The loss of network connectivity, for example, is an exception rather than the rule. The disruption of service due to an unplugged cable or an overloaded network are all possibilities, yet remain in the realm of unlikely. In addition, a desktop environment has enough resources to make allowances for transient failures, such as caching file system data. An embedded system does not have those abundant resources to throw at the problem and the transient failures are much more commonplace.

The PARIS project is an effort at Carnegie Mellon University to harden embedded applications by providing a way to manage its availability, performance, and security.

We will show the problem of directly porting an application from a desktop environment to a networked embedded environment by testing the effectiveness of an application to handle the propagation of error conditions from called routines.

Our initial focus is the ability of the GNU binutils and textutils to catch and properly handle error return values from the Standard C I/O library.

2. Test Harness

We first developed a testing harness to allow us to selectively manipulate the behavior of the function calls in the application source code. The test harness modifies the application at the source code level. It consists of a source-level translator, a run-time library, and a control driver.

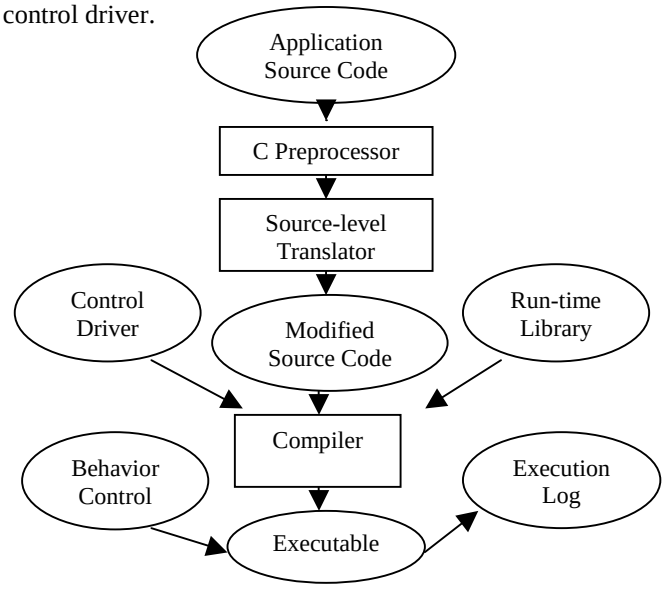

Figure 1. Test harness phases

The source-level translator is the front end of a C compiler. The C preprocessor is first run on all source code before being passed to the translator as shown in Figure 1. The translator builds a parse tree to be used to modify the source code and add in conditional statements to give control over specific function calls [1]. For example a normal fread function call in the sort text utility looks like:

```
cc = fread (buf ->buf +buf ->used, 1,
      buf->alloc-1-buf->used, fp ));
```

At the point of the function call a conditional is inserted around the function call to give control to the driver over the execution of the fread function. This is shown below:

```
cc =(control_fread ( 14262 )
  ?paris_fread ()
  :fread (buf->buf+buf->used, 1,
      buf->alloc-1-buf->used, fp));
```

The control_<function>(<line number>) implements a call to the driver to determine whether to execute or fail the function. It takes in the line number where the function is found in the code to help in analysis of the results when several function calls are being made. The control driver can fail functions on a specific instance of a call or for all call instances. For example, fread can be failed at a given point in a program by returning the error code rather than executing the function. A behavior control input file allows for functions to be specified when to fail: always fail, fail consecutively, or fail on a certain line number.

A log is generated that tracks the progress of the application and tracks the calling pattern of the application.

The paris_<function> returns the appropriate error value based on the specified function. For example, the C standard I/O library fread call will return up to but no more than the number of bytes that have been requested. A return value of 0 does not signify an error condition, just that no data is currently available [2]. A return value of -1 signifies an error and is what will be returned by paris_fread.

The use of program analysis to identify potential sources of problems has been previously presented in such systems as lint [3] that uses the approach to check for common portability errors. All problems are reported to the application programmer to take appropriate action. The errors are those typical when an assumption is made about the processor architecture. LcLint[4] extends the lint system to allow for specifications allowing a more customizable portability check.

There are also tools that attempt to classify the robustness of an application and present mechanisms for the automatic generation of run-time test coverage. For example, the Ballista project [5] stresses the API of a module by making calls to a module using a series of extreme values. The behavior of the module is examined and reported to the application programmer.

Our system focuses on the proper handling of error conditions using a combination of program analysis and run-time testing.

3. Test Cases

We applied the testing harness to a specific class of applications that are I/O based and use the C standard I/O library. In the case of networked embedded systems, especially that of a mobile environment, the situation is not always that the data cannot be obtained, but may be that a timeout has occurred in the request for data. This timeout is placed by the network communication protocol in the operating system so that the system returns to the calling application within a reasonable amount of time, but also results in the generation of an error.

Our error handling testing was performed on the GNU binutils and textutils. There are 26 text utility and 15 bin utility applications written by several authors. They are all written in C and rely heavily on the C standard I/O library. They have been ported to numerous platforms and are a widely accepted and used set of utilities. Our experiments were performed using the Linux platform.

The test harness was applied as described in section 2. The source files for the utilities as well as supplemental libraries were run through the C preprocessor, test harness source translator, and then compiled. We first ran the applications in their normal operating mode and then in our test mode. As displayed in the chart, several of the applications were unable to be successfully modified by the test harness. These utilities were not built due to incompatibilities between cpp and our translator. The majority of utilities were built successfully and are a significant representation of the package source code.

4. Results

After analyzing all of the utilities for how they handle different error situations, four different failure categories became apparent. These four are: handles correctly (HC), handles incorrectly (HI), silent failure (S), and silent failure (S2) when failed in conjunction with another function.

Handles correctly (HC) is when an application recognizes that an error has occurred, an error return value

Text Utilities	fread()	fopen()	fclose()	fwrite()	read()	close()	putchar()	ungetc()	printf()	fprintf()	vfprintf()
cksum	S	HC	HI						S	S2	S2
comm		HC	HC	S						S2	S2
csplit		HC	HC	S	HC	HC				S	
cut		HC	HI				S	S		S2	S2
expand		HC	HC				S			S2	S2
fmt		HC	HI				S			S2	
fold		HC	HC	S						S2	
head		HC	HC	HC	HI	HI				S2	S2
join		HC	HI	S			S			S2	
md5sum	S	HC	HC				S		S	S2	S2
nl		HC	HC	S					S	S2	S2
paste		HC	HC	S						S2	S2
pr		HC	HI				S			S/S2	S2
split		HC			HI	HI				S2	S2
sum		HC	HC				S			S/S2	S2
tr		HC		HC	HC					S2	S2
tsort		HC	HC						S	S2	S2
unexpand		HC	HC				S			S2	S2
uniq		HC	HC	S						S2	S2

Figure 2. GNU Text Utilities

has been acknowledged by the calling function and an appropriate error display is given. This is correct behavior.

Handles incorrectly (HI) is similar to handles correctly in as much as the application acknowledges that an error has occurred and an error return value has been received, but an accurate error display is not given.

Silent failure (S) occurs when an application does not crash or acknowledge any error, even though an error value was returned by the function. The application continues its execution as if no error occurred.

The final error category is also a silent failure (S2), but the function in question was used to produce an error message because another function has failed. The primary error is being acknowledged, but the function used to produce the error message itself is not checked for errors.

The test results are summarized in Figures 2 and 3. The primary observations are that the most common case is silent failures. The silent failures do not always produce no output, but very often corrupted data. Additionally, output is less likely to be checked for an error condition than input.

In a silent failure, a function call would fail to read to or write from a file and either report nothing at all or report false data. For example when failing fread in the cksum utility, the output is 0 for the number of bytes in the file and an incorrect checksum is produced as well. No error was reported even though the correct error condition was returned.

Csplit is a utility that splits a file into two separate files. When fwrite is failed in csplit, two files are created and the csplit even displays what the correct sizes should be, but the files themselves are empty. No error is given and the application exits normally.

If an fclose fails, an error is almost always given. However, the program does not always exit gracefully nor does it give clear information about what error occurred or where it occurred. For example, in the join utility when fclose() fails the error given is: "./join_paris: k: ,ðÿ¿,ðÿ¿ ".

5. Conclusion and Future Work

Common desktop applications, even those that have been ported to multiple platforms, do not appear to be easily ported to an unreliable networked embedded environment. Our results show that the majority of problems resulted in silent failures that lead applications to produce results that are incorrect and inconsistent with no indication of any problem. Output was assumed to work correctly, especially as outputting an error message was the common way of acknowledging an error. We focused on the Standard C I/O library for its wide use, but the testing harness can be applied to other functions as well.

Bin Utilities	fwrite()	fclose()	fopen()	putchar()	rename()	fprintf()	vfprintf()	fputs()	fgets()	fflush()
ar	S	HI	S/HI							
nm	S			S						
objdump	S			S						
ranlib					HC	S	S			
size	S			S						
strings				S				S		
strip					HC	S	S			
addr2line	S								S	S

Figure 3. GNU Bin Utilities

We are extending this work to allow for an automated way to harden the application by inserting an adaptation layer to isolate the application from the underlying unreliable system. The applications that were written for a desktop environment do not have to be structurally changed, and instead the adaptation layer can make the necessary modifications. This decouples the reliability need from the functional need of the program.

6. References

[1] Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.

[2] P. J. Plauger. The Standard C Library. Prentice Hall, 1992.

[3] S. C. Johnson, lint, a C Program Checker, *Computer Science Technical Report Number 65*, 1978.

[4] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. SIGSOFT Symposium on the Foundations of Software Engineering, December 1994.

[5] Philip Koopman. Toward a Scalable Method for Quantifying Aspects of Fault Tolerance, Software Assurance, and Computer Security. *Computer Security, Dependability, and Assurance: From Needs to Solutions (CSDA'98)*, November 1998.