Software Design Based on Operational Modes

by

Alan Albert Jorgensen

Bachelor of Science
Electrical Engineering
University of Arizona
1963

Master of Science
Computer Science
Arizona State University
1990

A dissertation submitted to the Graduate School of
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

Melbourne, Florida
December, 1999

We the undersigned committee hereby recommend that the attached document be accepted as fulfilling in part the requirements for the degree of Doctor of Philosophy of Computer Science.

"Software Design Based on Operational Modes,"
a dissertation by Alan Albert Jorgensen.

_____

James A. Whittaker, Ph.D.
Associate Professor and Chairman, Software Engineering
Dissertation Advisor

_____

William D. Shoaff, Ph.D.
Associate Professor and Computer Science Chair

_____

Muzaffar A. Shaikh, Ph.D.
Professor, Management Science and Engineering Management,
School of Business

_____

Walter P. Bond, Jr., Ph.D.
Associate Professor of Computer Science

_____

Frederick B. Buoni, Ph.D.
Professor Emeritus, College of Engineering

_____

J. Richard Newman, Ph.D.
Dean, College of Engineering

Abstract

Title: Software Design Based on Operational Modes

Author: Alan Albert Jorgensen

Committee Chair: James A. Whittaker, Ph.D.

The use of software is ubiquitous despite its reputation for low reliability. This dissertation experimentally verifies this reputation and then proposes changes to the development process to prevent certain classes of failures. I begin by presenting numerous examples of software failures from modern, professionally tested, software products. The root cause of each of these failures can be traced to incorrect partitioning of internally stored data.

I propose a new design technique based on a recently developed testing concept called "operational modes." Operational modes allow correct decomposition (abstraction) of software states defined by storage constraints and describe the cause of a large class of software failures. Operational mode design is influenced by four constraining software features: input, output, computation, and data storage. From this understanding, four classifications of failure are derived from this improved definition of operational modes: 1) improperly constrained input, 2) improperly constrained output, 3) improperly constrained computation, and 4) improperly constrained internal data. Illustrative examples of these failure classes are presented from a number of published programs.

I propose changes to the software design process to eliminate these four identified categories of defects by proper identification and implementation of system constraints, i.e., operational modes that correctly partition program data. This new theory provides developers a methodical mechanism to prevent a large class of software faults and provides software testers a roadmap to the broad class of software behaviors that must be tested.

I demonstrate the application of this design process modification with a small example that, though proven to be correct in the literature, fails due to lack of proper constraint checking. The resulting example program no longer contains these defects as a direct result of the improvements to the design process. The process is further verified by redesigning an example program from a modern software development text. Not only does the technique correct a defect in that example, but results in a function that is now clearly specified and eliminates the need to rely on "clever" design to achieve the desired results.

Table of Contents

List of Keywords

Software Defects

Software Design

Software Requirements Analysis

Operational Mode

Model Based Software Testing

State Machine

State Model

Software Constraint Design

List of Figures

List of Tables

Acknowledgement

The members of the Spring 1999 semester class of Software Testing Methods contributed to this work by establishing that software defects are ubiquitous and can be located with a minimum of software test training. In particular, Mazin Al-Shuaili, Steven Atkins, Jeremy Babb, Cibel Castillo, Rahul Chaturvedi, Arun Chitrapu, Adam Duccini, John Grant, Pi-Yu Lee, Roby Matthew, Kay Michel, Florence Mottay, Luke Nowak, Luis Rivera, Giovanna Giovanna Scaffidi, Keyur Shah, Brian Shirey, and Sharma Vanterpool provided well documented results of their work.

Florence Mottay collected and organized the material that appears in Appendix B and Appendix C. In addition she expanded, improved, and completed the work in Table 2, Table 3, and Table 4 which contains detailed requirements and design information for the running average example of the improved design process. Thank you, Florence. This is only the beginning of your contribution to this field of endeavor.

I particularly wish to thank those who provided detailed critical reviews: Jane Davis, Dr. William (Bill) Shoaff, Dr. Shirley (Annie) Becker, Edwin Mallette, Tom Engler, and Dr. James Whittaker.

My committee provided support and encouragement. I thank them for the time, energy, and good advice: Dr. William D. Shoaff, Dr. Muzaffar A. Shaikh, Dr. Frederick B. Buoni, and Dr. Walter P. Bond.

I hold a very special appreciation for my committee chairman, Dr. James A. Whittaker, who has read and reread this dissertation and commented with consummate skill as a wordsmith as well as providing encouragement and technical insight. His dedication to all of his students and to this one in particular is remarkable. Thank you, James.

Dedication

This work is dedicated, with love, to my parents:

Ione Towner Jorgensen

Albert Henry Jorgensen

whose dedication to me has never wavered.

This is for you.

Chapter 1. Software Fails

Software development, like many other creative endeavors, is prone to failure. Even with today's best software development techniques, well-designed and thoroughly tested software sometimes, and even frequently, behaves improperly due to defects introduced during development. Software development is now necessarily a craft but must become an engineering discipline before software consistently produces reliable behavior. This dissertation presents advances in software development technology in support of the transition toward engineering rigor.

There is much to do before software development can become an engineering discipline. My focus is to identify and correct a particular class of failures: those that escape the capabilities of today's test and development technology. This might not make software perfect, but it is an important step in the software development maturation process. Refinement of the design process starts by determining the root cause of design failures. Not only should we correct the design flaws, but we must also correct the method by which the product was designed [Paulk, *et al*., 1993]. We begin, then, by discussing the nature and consequence of software defects. Examination of defects and searching for common causes will lead to a better design process.

1.1 Software Defects

Public concern over the presence of defects in software has deepened in recent years due to the proliferation of personal computers and high-profile defects such as the so-called "Y2K bug." Moreover, as the number of people using computers steadily increases, the impact of software defects on the performance of our day-to-day routine is potentially enormous. Defects cause down time and force rework when data is lost. In addition to these disruptions and frustrations, software defects have the potential for significant loss of life, wealth, and property. Our safety and our security depend on the correct operation of software.

Peterson [Peterson, 1995] describes the real and potential impact of software defects on those who are unaware that their lives depend on the correct operation of computer programs. Defective software can shred luggage [Glass, 1998], over radiate cancer victims [Leveson, 1995], and destroy rockets with their payload [Lions, 1996, Baber, 1997]. The importance of the quality of software and the effect of that software on our daily lives cannot be overstated. We use software unknowingly every day; in our microwaves, in our automobiles, and at the bank teller. How frequently must we hear, "I'm sorry, the computer is down right now?"

In this year, 1999, there is considerable public concern and discussion of the so-called "Y2K" bug. Legacy software, developed some time in the past and with older languages and development techniques, frequently represents the year with a two-digit number. After 1999, the representation will transition from "99" to "00."

This representation will be interpreted as the year 1900 instead of the year 2000. Interest rates, programmed delay periods, and many other time dependent functions will behave irrationally. Understanding the reasons that software fails is of deep concern to software professionals *and* the general public, even when they are not aware of how dependent they are on the correct operation of software.

The specific definition of software failure is equivocal. Definitions range from undesirable operation to catastrophe. We will work from IEEE definitions as follows:

> Failure: The inability of a system or component to perform its required functions within specified performance requirements [IEEE, 1991].
>
> Defect: A product anomaly. Examples include such things as (1) omissions and imperfections found during early life cycle phases and (2) faults contained in software sufficiently mature for test or operation. ... [IEEE, 1994]
>
> Anomaly: Any condition that deviates from expectations based on requirements specifications, design documents, user documents, standards, etc., or from someone's perceptions or experiences.
>
> Anomalies may be found during, but not limited to, the review, test, analysis, compilation, or use of software products or applicable documentation. [IEEE, 1994]
>
> Fault: Any change in state of an item that is considered to be anomalous and may warrant some type of corrective action. Examples of faults include · · ·,

> out-of-limits conditions on sensor values, · · ·, software exceptions (e.g., divide by zero, file not found), rejected commands, measured performance values outside of commanded or expected values, an incorrect step, process, or data definition in a computer program, etc. Faults are preliminary indications that a failure may have occurred [IEEE, 1991].

Within these definitions there are some important distinctions to be made. In particular, there is a clear difference between the symptom as viewed by an observer of the system, and the erroneous code that lead to the symptom. However, these definitions do not clearly indicate the lack of one-to-one mapping of symptoms to "anomalies." A single defect in the software may result in a multitude of varying symptoms (and vice versa).

Modern software is highly vulnerable to failure. I demonstrate below that "fully tested" retail software can be forced into repeatable failure situations. I show that software quality advocates have "proven" buggy programs to be "correct" without exposing embedded defects. There is a common characterization for these defects and simple training in that characterization allows novice software testers to quickly uncover defects not found by experienced testers after months or years of accumulated testing and use.

1.2 Defects in Released Software

My research in this area began simply enough while playing with the calculator program released with Microsoft® Windows® 95. I wondered if it were possible to put the calculator into a state such that it would calculate incorrect results. I began by investigating limiting or "boundary" values, such as the largest number that could be entered (9999999999999e+289). I found, however, that this number could be increased computationally. So I searched for the largest value that could be computed (1.797693134862e+308). I obtained this number by taking the inverse log of 308.2547155599. Starting with this value (and other apparent limitations of the calculator), I was able to achieve numerous erroneous results such as the following:

- A number can be computed that cannot be divided by two. 1.797693134862e+308 divided by 2 results in the error message, "Result is too large." Apparently there is a check on the maximum value computed for some functions but not for others, thus allowing a value to be computed and stored that is larger than this (unspecified) allowable limit.
- The value 1.797693134862e+308 can be copied to the clipboard but cannot be pasted back into calculator. Values can be calculated that cannot be reentered. It seems reasonable to assume that if the calculator

prevents the user from entering a value, it should also prevent the same value from being generated by calculation.

Appendix A contains a list of fifteen such input sequences that cause anomalies in the calculator from Windows® 95.

Similar, yet different, defects are found in Windows® 98 and Windows® NT calculators. One such problem in NT calculator, for instance, causes the program to terminate and abruptly close its window. This same sequence performed in Windows® 95 or Windows® 98 calculators does not cause the program to abort. Apparently there are significant differences in design between the various versions of the calculator. Windows® 98, for instance, uses a significantly different internal representation of floating point numbers and can represent extremely large numbers that require a significant amount of time to compute.

These defects appear harmless, but it isn't a large stretch of the imagination to see them leading to more serious problems when they occur in more important applications. However, my interest in these failures is that they give insight into the very nature of software failure and from this insight we can derive techniques to improve the overall quality of software intensive systems.

Software failures do have a direct impact on our everyday lives. During the writing of this dissertation, Microsoft® Word® 97 (Service Release 1) produced many persistent anomalies that hindered my productivity. Some were merely inconvenient but some caused the word processor application to fail

catastrophically, such as the following sequence: Select Outline View, Select Level 1 outline display, expand a single level by double clicking on the "+" in the outline view.

In order to study this failure phenomenon closely, we conducted a study to systematically investigate a number of software products for potential defects. On the first day of the Spring 1999 class in Software Testing Methods at the Florida Institute of Technology, the students were given a thirty-minute synopsis of this chapter and a preliminary theory on the causes of software defects. The challenge to these students, who were previously untrained in software testing, was to identify repeatable anomalies in both released software and published source code. The students had only two days to complete this assignment. We obtained the following results:

- Every student submitted unique and successful results.
- Defective published code was found in both programming and software engineering texts.
- Defects were found in a variety of software from a variety of vendors, including, but not limited to, operating systems, web browsers, and desktop applications.

Twenty six defects were found in sixteen software packages and nineteen code defects were found in eighteen different programming texts. A table summarizing the application defects appears in Appendix B and the summary of the published

code findings appears in Appendix C. These examples of defects were found in commercially available products that should be presumed to be of the highest quality available, reviewed and tested by experts in the field, and then subject to years of use in the field.

The fact that untrained students could so easily find failures outside the capability of the current testing methods of software suppliers indicates that something is fundamentally wrong with the way software is developed and tested.

1.3 Seeking a Solution

This dissertation studies this very problem and presents a solution for preventing large classes of software defects introduced during development. (The Software Engineering Institute at Carnegie Mellon University, in its Capability Maturity Model, recommends that knowledge of defects be used to improve the software development process [Paulk, *et al*., 1993]. When I attempted to insert this parenthetical statement as a footnote, the word processor inserted the footnote outside the lower margin.)

I take the following approach. I 1) search for software failures, 2) study the cause and effect, searching for root causes, 3) classify the causes of defects, 4) test this classification theory in a software testing laboratory, and 5) discuss possible changes to the software design process that will alleviate these defects.

In Chapter 2, I perform root cause analysis of the software failures and present a theory about why software fails. Chapter 3 describes what needs to be done to fix this problem with reference to why other methods have not corrected the problem. Chapter 4 describes how to modify requirements definition to capture the information necessary to correct the problem. An example problem is introduced to illustrate the methodology. Chapter 5 continues with the impact on design techniques; the example is continued and completed. Chapter 6 is a case study. A well-crafted programming example from a modern program development text is examined, found defective and inadequately specified, and then re-designed to be of higher quality and more completely specified.

Appendices provide detailed implementation and test details.

Chapter 2. Why Software Fails

Two basic testing techniques were employed to identify the anomalies described in the previous chapter. The first technique was to stress test software using extreme values that might not have been anticipated by the developer. The second technique was by model based testing [Whittaker, 1997b]. Using either technique, failures could be attributed to erroneous state transitions. This realization led to a careful study of the relationship between state machines and software failures. The search for the basic cause of software defects begins by examining the fundamental nature of software systems when viewed as a state machine.

2.1 Software Systems as State Machines

The applicability of state machine modeling to mechanical computation dates back to the work of Mealy [Mealy, 1955] and Moore [Moore, 1956] and persists to modern software analysis techniques [Mills, *et al*., 1990, Rumbaugh, *et al*., 1999]. Introducing state design into software development process began in earnest in the late 1980's with the advent of the cleanroom software engineering methodology [Mills, *et al.*, 1987] and the introduction of the State Transition Diagram by Yourdon [Yourdon, 1989].

A deterministic finite automata (DFA) is a state machine that may be used to model many characteristics of a software program. Mathematically, a DFA is

the quintuple, $M = (Q, \Sigma, \delta, q_0, F)$ where M is the machine, Q is a finite set of states, Σ is a finite set of inputs commonly called the "alphabet," δ is the transition function that maps $Q \times \Sigma$ to Q,, q_0 is one particular element of Q identified as the initial or stating state, and $F \subseteq Q$ is the set of final or terminating states [Sudkamp, 1988]. The DFA can be viewed as a directed graph where the nodes are the states and the labeled edges are the transitions corresponding to inputs.

When taking this state model view of software, a different definition of *software failure* suggests itself: "The machine makes a transition to an unspecified state." From this definition of software failure a *software defect* may be defined as: "Code, that for some input, causes an unspecified state transition or fails to reach a required state."

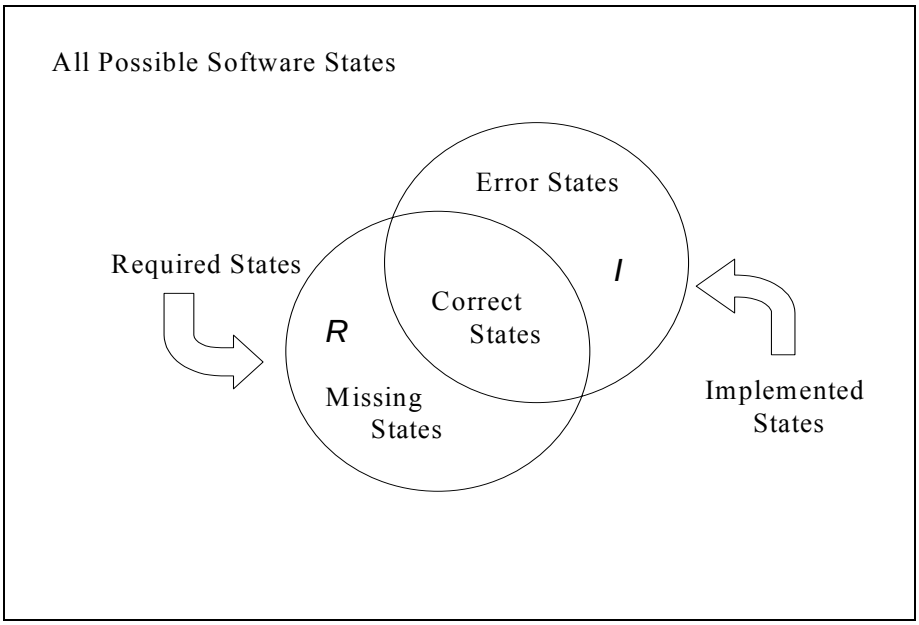

Figure 1 -- Required and Implemented States

Figure 1 is a Venn diagram of the software state space showing two subsets of states. One subset is the states fulfilling the needs of the system (R) and the other (I) is the subset of states actually implemented in the developed product. The subset $M = R \cap \sim I$ are those states that should have been implemented but were not; $C = R \cap I$ is the subset correctly implemented; $E = \sim R \cap I$ are the states implemented that should not have been. When developers test their own product by testing only the states they know they implemented, then only the states in C would be tested. Using the standard testing practices of today, when test engineers test states against requirements, then only those states in R would be tested. The difficult and subtle software problems occur in the subset defined by E. These are the states reached by improper state transitions. Software reaching these states behaves in an

unpredictable manner. These are the class of failures that are the focus of this work.

For a computer program to behave reliably, the states of the program and the transitions between them must be clearly and completely defined and implemented in the code. Therein lies a problem, however, for even simple programs can have a vast number of states. Consider a 32 megabyte computer: there are approximately 2^{28} bits of memory. This means that the computer could have $2^{2^{28}}$ states. In fact, a program with only ten 32-bit variables can be in 2^{320} different states (2^{10*32} different possible value combinations), more than the estimated number of atomic particles in the universe [Suber, 1998]. Clearly, a mechanism is needed to simplify this complexity.

This simplification can be accomplished by determining some property of states that cause certain states to be equivalent in some sense, and identify the states within these "equivalence classes" as a single state [Hopcroft & Ullman, 1979].

One method of defining equivalence classes is to base states on input sequences. This method of identifying states has been described as a stimulus history [Mills, *et al*., 1987] or input history [Prowell, 1996]. A state is identified by the set of input sequences that reach that state from the starting state. Input sequences reaching a particular state are said to belong to the same equivalence

class. Since the number of possible input sequences is infinite, determining the states of a system by this method is problematic. One of the problems is the determination of the equivalence of input sequences. If input and input sequences cannot be found to be equivalent, new states are identified and a proliferation of states occurs. This is known as "state explosion." Another problem is completeness; how can it be determined that all of the states (and corresponding input sequences) have been identified? Though theoretically useful, representing software states as input sequences is impractical and a better representation is required. Fortunately there is a relatively new concept for representing states called "operational modes."

2.2 Operational Modes Decompose States

Recent developments in software system testing exercise state transitions and detect invalid states. This work, [Whittaker, 1997b], developed the concept of an "operational mode" that functionally decomposes (abstracts) states. Operational modes provide a mechanism to encapsulate and describe state complexity. By expressing states as the cross product of operational modes and eliminating impossible states, the number of distinct states can be reduced, alleviating the state explosion problem.

Operational modes are not a new feature of software but rather a different way to view the decomposition of states. All software has operational modes but

the implementation of these modes has historically been left to chance. When used for testing, operational modes have been extracted by reverse engineering.

Whittaker provides the following definition of an operational mode:

> "An operational mode is a formal characterization (specifically a set) of the status of one or more internal data objects that affect system behavior." [Whittaker, 1997b, p. 120].

A similar concept is described by Heitmeyer, Kirby, and Labaw in [Heitmeyer, *et al*., 1997]:

> "A *mode class* [operational mode] is a partitioning of the system states. Each equivalence class is called a *system mode* (or simply *mode*) [operational mode value]."

Using these definitions we can explain the failures described in the first section in terms of the operational modes. A calculator failure occurred, for example, when an attempt was made to divide a particular result by two and the result was too large. The divide-by-two calculator failure occurs because there are operational modes associated with the prior result and with the value entered such that the calculator will refuse to compute the division result. A state view of the calculator can be very complex. This complexity can be simplified, however, by considering only specific portions of the state: those having to do with the previous result, the value entered, and the operator entered. When the operator entered is divide, and the previous result divided by the value entered exceeds some particular value, the calculator is in a state such that depressing the equal key (or some other

function that causes the divide to occur), will produce an error message instead of computing the result. There are many different states of the calculator where this is true. But by considering only particular characteristics of the state, we can determine the behavior of the calculator for a particular input. These characteristics of the state are some of the operational modes of the calculator.

From the description of the problem, we can define the relevant operational modes of the system as, the last operation key entered, *CurrentOp,* the result from a prior calculation, *PriorResult,* and the last value entered, *EntryValue*. Each of these operational characteristics may take on many values and the domain of these variables is a set as follows: *Domain(CurrentOp) = { ...,+,*,-, /, ...}, Domain(PriorResult) = { ... ,0,1,* (*PreviousResult > SomeMaximum / ValueEntered*)*, ...}, Domain(EntryValue) = { ...,0,1, ValueEnteredTooSmall, ... }*. These operational characteristic variables are some of the operational modes of the calculator system. When these operational modes have the particular values, *CurrentOp = /, PriorResult =* (*PreviousResult > SomeMaximum / ValueEntered*)*,* and *EntryValue = ValueEnteredTooSmall,* the equal key will cause the error message to appear. Note that in some cases a single operational mode value may represent many values in the calculator, such as *EntryValue = ValueEnteredTooSmall*. This operational mode value is uniquely defined by the partitioning of the values that can be entered.

A similar situation occurs for the copy-paste failure. The operational mode for the contents of the clipboard can assume values that will cause paste to operate incorrectly. There is an operational mode associated with the copy buffer and an operational mode value corresponding to a numeric string that cannot be pasted into the calculator. Such a string can be stored into the copy buffer from the calculator output and therefore the calculator enter a state wherein it cannot accept paste input correctly.

2.3 Operational Mode Values Partition Stored Data

Normally, an operational mode is associated with a single persistent storage element. *Persistent storage* is memory referenced by a software component between any two successive inputs. A single persistent storage element contains a set of values (though it may range from a single bit to multiple, disjoint words of main memory). Temporary storage, on the other hand, that is reinitialized as a result of input is not independent from the input and will not cause the system to behave differently at the next input and therefore need not be considered part of the state of the system. This dissertation extends and formalizes the definition of an operational mode value by identifying it as a partitioned set of persistent storage values as follows:

An *operational mode value* is an exclusive subset of instantaneous values of elements in persistent storage. An *operational mode* is a set of operational mode

values. Note the distinction between an operational mode value and a stored value that defines the operational mode. The operational mode value is an abstraction of one or more stored values. The operational mode value combines in a single variable the values of elements of persistent store that are equivalent in the sense that the variation of values does not affect the behavior of the system. Mathematically we can describe the domain of an operational mode as a set of operational mode values and each operational mode value as a set of storage values as follows:

$$\text{Domain}(M) = \{V \mid V = \{V_1, V_2, V_3 \ldots V_i \ldots V_n\}\}$$

$$\text{Domain}(V_i) = \{X_i \mid X_i = \{x_{i1}, x_{i2}, x_{i3} \ldots x_{imi}\}\}$$

Where M is an operational mode, V is the set of operational mode values, V_1, V_2, $V_3 \ldots V_n$; n is the number of operational mode values in the domain of M; V_i is the ith operational mode value X_i where X_i is the specific set of storage values, x_{i1}, x_{i2}, $x_{i3} \ldots x_{imi}$, and mi is the number of storage values in this ith operational mode value. The relation, $X_i = \{x_{i1}, x_{i2}, x_{i3} \ldots x_{imi}\}$, determines the set of storage values belonging to the operational mode value. (This relation could also appear as a partitioning, such as $X_i = \{x \mid y \leq x \leq z\}$). Operational mode values are mutually exclusive, i.e.,

$$\forall i \; \forall j, \; 1 \leq i \leq n, \; 1 \leq j \leq n, \; i \neq j, \text{ and } V_i \cap V_j = \varnothing$$

The storage values that define an operational mode value are equivalent in the sense described by Whittaker: e.g., storage values are in the same equivalence

class if there is no difference in externally observable system behavior associated with the values within that class.

As an example, we might define an operational mode, *Count,* as *Domain(Count) = {New, In Range, Max, Invalid}* where *Count* is an operational mode that is associated with a storage location containing the value designated by the variable name, "*C*." Thus *M* is *Count*, V_1 = *New*, V_2 = *In Range*, V_3 = *Max*, and V_4 = *Invalid*. The operational mode values may then be defined as:

Count = *New* iff $C = 0$; Thus $X_1 = \{0\}$

Count = *In Range* iff $C > 0$ and $C <$ *Maximum Value*;

$X_2 = \{x \mid 0 < x <$ *Maximum Value*$\}$

Count = *Max* iff $C =$ *Maximum Value*;

Count = *Invalid* iff $C < 0$ OR $C >$ *Maximum Value*.

The set of operational mode values encompasses the complete domain of the storage value(s) represented by the operational mode.

Each operational mode value is defined by one or more partitions of stored data. For example, a partitioning of *x* could be $\{x < 0, x \geq 0\}$. If this partitioning describes equivalence classes on *x*, then these partitions represent operational mode values.

The partitions that define operational mode values need not be constant, however, and may be dependent on variable conditions. We accommodate this with the following definitions: A *static partition* is a relation between the stored value and a constant. A *dynamic partition* is a relation between the stored value

and a function including at least one independent variable. This means that the set of memory values in an operational mode value may change over time. The example above is a static partitioning. If x were partitioned as $\{x < z, x \geq z\}$, then the partitions may still define equivalence classes, but the values in each class depend on the variable value, z. This is an example of a dynamic partition.

As a practical matter, partitions may be described based on the nature of the limiting values of stored data. The partitioning may take place naturally due to the finite nature of computers or the limitations may be imposed by requirements. I define the partitions imposed by the physical computation environment as *natural partitions* and those imposed by requirements as *artificial partitions*. This is an important distinction because the bounds on a problem solution may be imposed in such a way that requirements may not be met.

Another more rigorous example is the operational mode for an arbitrary integer variable in persistent storage. A signed, twos-complement 16-bit integer may be represented by the following operational mode and associated operational mode values:

$$\mathit{Domain(\ INT)} = \{\mathit{Min, Negative, Zero, Positive, Max}\}$$

Where:

$$\mathit{Min} = \{\text{-}32768\},$$
$$\mathit{Negative} = \{\text{-}32767 \mathinner{..} -1\} = \{x \mid \{\text{-}32767 \leq x \leq -1\},$$
$$\mathit{Zero} = \{\ 0\ \} = \{x \mid x = 0\},$$

Positive = { 1 .. 32766 } = {$x \mid 1 \leq x \leq 32766$},

Max = {32767},

-32768 is the smallest 16-bit twos-complement integer, and

32767 is the largest 16-bit twos-complement integer.

The partition, *P* = { x = -32768, -32767 ≤ x ≤ –1, x = 0, 1 ≤ x ≤ 32766, x = 32767}, completely defines the operational mode values and hence the operational mode, *INT*. The constraints defining the operational mode values *Min* and *Max* are natural since they define the limiting properties of 16-bit twos-complement integers. All of these partitions are static because all of the values in the partitions are constants. Appendix D contains additional examples of operational modes demonstrating artificial and dynamic partitions. In addition, these examples also demonstrate an important characteristic of the constraints that partition operational mode values. The underlying influence on the partitions determining operational mode values are the constraints imposed by input, output, storage, and computation. Constraints of all four types may be imposed by user requirements (artificial constraints) or they may be imposed by the characteristics of computation on a finite device, the computer (natural constraints). The key issue is that *all* input, output, data storage, and computation is constrained either by requirements or by finite computational resources. For software to be truly robust, it must test all such constraints and respond appropriately.

2.4 Partition Constraints

There are four predominant influences that determine operational mode partition values. These influences form the basis for a theory explaining software failure. All observed failures can be explained by noting that an anomolous state led to the failure. The anomalous state was caused by an unplanned operational mode value and this occurred because of one of the following software defects:

- Improperly constrained input.
- Improperly constrained output.
- Improperly constrained computation.
- Improperly constrained stored data.

The following examples of software failure are from code published in college texts and commercial retail software. (There is an example to illustrate each of these fault classes.)

2.4.1 Improperly Constrained Input

There is nothing new about input constraint errors. Most good programming texts advise protecting against invalid input. However, these same texts provide us with coding examples that do not properly constrain inputs. Software developers are taught to check inputs before processing. Few do, however, even those who teach new developers how to program. Consider the following program taken from [Kernighan & Ritchie, 1988, p. 62] as an example.

```
/*shellsort: sort v[0]…v[n-1] in order */
void shellsort(int v[], int n)
{
  int gap, i, j, temp;
  for (gap = n/2; gap > 0; gap /= 2)
    for (i = gap; i < n; i++)
     for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap){
        temp = v[j];
        v[j] = v[j+gap];
        v[j+gap] = temp;
        }
}
```

To expose a bug in this program, we can simply pass the routine a "bad" parameter. There are at least two choices for doing so. First, we can incorrectly specify the length of the array. The following driver code causes shellsort to fail because of an array out-of-bounds error.

```
main()
{
  int in[] = {10, 20, 30, 40};
  int length = 5;
  shellsort(in, length);
}
```

In practice, the storage for the "in" array will contain four locations followed by the "length" value which will be included in the sorted array. After calling shellsort, the array n will contain (5,10,20,30) and length will have the value 40. Most languages permit passing reference to an array to a function, though some languages can prohibit array access out of bounds.

Second, we can pass *shellsort* an invalid array address and the program will fail due to an invalid pointer. The following driver performs this task:

```
main()
{
  int *in = main;  /* Sort the program code! */
  int length = 50;
  shellsort(in, length);
}
```

In this case shellsort will rearrange the code for the program itself and cause the program lock up or otherwise terminate with prejudice.

One can argue that it is the task of the calling program to avoid passing unacceptable parameters. This argument is weak because it requires redundant inclusion of check code at each call to the routine.

Such examples also may be justified as merely illustrative. However, in writing a routine that can be broken so easily is dangerous for any developer who might eventually incorporate that routine into a new software product. Such coding examples are seldom described as being incomplete because they fail to perform required bounds checking.

One important example code omission is the return of error information to the calling program. There are numerous error conditions that could be detected by the shellsort routine. Is the data to be sorted valid, i.e., is there a linear order or is it in the required range? Is there data to be sorted (is $n = 0$)? This are two common

themes of constraint errors: 1) constraints are not checked and 2) no provision is made to report violation of constraints.

2.4.2 Improperly Constrained Output

Just as there are limitations on the data that the system can accept for processing, there are also limitations on a computer's ability to present information. Such an example can be found in a retail version of Microsoft® Money® 98. By entering large, but acceptable, values in a field for entering dollar amounts, we can elicit an output constraint error when the application adds dollars signs and decimal points to the numbers for display. Figure 2 shows a manifestation of this bug. At the bottom right of the screen, a display field has overflowed causing the display of invalid format characters instead of the correct value.

This problem may be reproduced with Microsoft® Money® 98 Financial Suite Version 6.0 with the following input sequence.

1) Invoke Microsoft® Money®.
2) Click on "Planner."
3) Click on "Get Out of Debt."
4) Click on "Next."
5) Click on "New Account."
6) Type "asdf" enter.
7) Type enter 3 more times.
8) Type "999999999999" enter. (12 '9' digits)

9) Type enter.

10) Type “999999999999” enter. (12 '9' digits)

11) Type enter 4 more times.

Figure 2 -- A Broken Output Constraint from Microsoft® Money® 98

Output constraints are probably the easiest to define because they appear, at first, to be unconstrained; that is to say, if the number is too big to fit the output field, then the output field could be extended to accommodate the required value. In terms of the operational mode value affected by the output constraint, changing the output field width increases the bound on the presentation value. This value is then constrained by something other than the output field width. This is a legitimate design technique for minimizing operational mode values (and therefore minimizing states). In any case, however, the output constraining value must be considered as a limitation on the output and hence on any data and computation driving that output value.

A check could have been performed to determine that the data could not fit in the field provided. But how should this fact be reported back to the originator of that data? Here again are the common themes: the checking of a constraint and the reporting of the constraint violation.

2.4.3 Improperly Constrained Computation

The design for a running sales average is presented in [Mills, *et al*., 1990, p. 11] and "verified" later in the same text [Mills, *et al*., 1990, p. 117], including verification for "improper use." Nevertheless, this program fails when forced to overflow its stored data through a simple calculation.

The running average is computed by :

$$R(i) = \frac{S(i) + S(i-1) + ... + S(i-11)}{12} \qquad \text{[Mills, } et\ al\text{., 1990]}$$

where *S* represents an input (called a stimulus), *R* the output (called a response), and *i,* the index representing the order of arrival of the inputs. When this program is implemented, the storage set aside for the running sum will overflow when submitted two or more inputs that, when added together, are larger than the maximum allowed integer.

Such a test exploits the fact that this particular computation is unconstrained. In fact, there is no check to ensure that the result will fall within an acceptable range. A solution to this problem is to check the values by subtracting

the sum from the maximum allowable integer and detecting that the result is less than the next input: (if $MaxSum - S(i\text{-}k) < R(i)$ then *error*.)

However unlikely these circumstances may be, the result of overflow is almost certainly a severe defect. For real-time software, this is a dangerous situation no matter how rare. The aborted maiden flight of the *Ariane 5* is an example of this phenomenon [Baber, 1997], and caused by a failure similar to the running average problem above. Aboard the Ariane 5 a floating point to integer conversion computation produced a result that fell outside the allowable integer range (a violation of a natural constraint). This caused the guidance software to malfunction resulting in the rocket veering off course. The self destruct software (which worked perfectly) destroyed the rocket in flight. Improperly constrained computation has serious consequences.

In both of these examples, what was the course of action open to the programmer? Mills' monthly average procedure should report to the data originator that the computation cannot take place. In the case of the Ariane 5, however, some other course of action should have been required. Computing the incorrect result and steering the rocket off course was not the correct option. Perhaps the software should have halted to relinquish control to manual operation. Again the common themes appear: check a constraint and report the violation.

2.4.4 Improperly Constrained Stored Data

Even if inputs, outputs, and computation are constrained, programs sometimes store bad data as a result of internal processing. In addition, sometimes a system corrupts its own internally stored data. As a case-in-point, a commercial spreadsheet package can be tricked into just this situation after entering a formula that is longer than the allowed limit.

When Microsoft® Excel® 97 receives a formula composed of a large number of characters, it displays a message indicating that the formula is too long (see Figure 3).

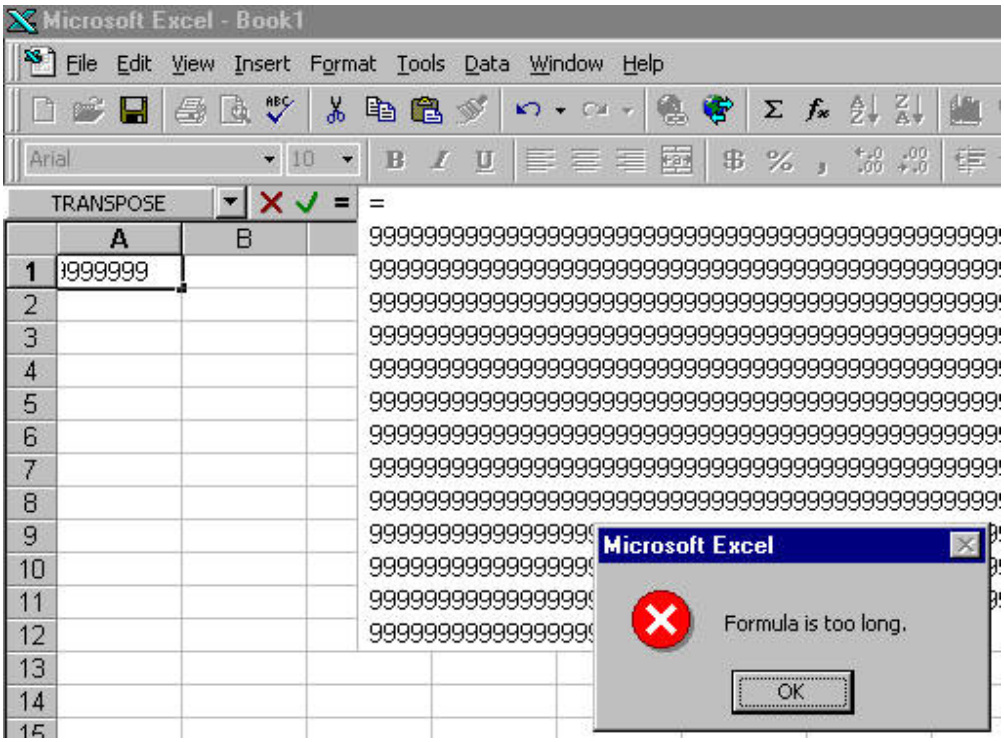

Figure 3: -- Microsoft® Excel® 97 Correctly Constrains Input

Thus, it successfully constrained the input and avoided processing the bad value. However, in doing so, it managed to corrupt its internal memory. Hitting the "Enter" key in the cell in which the formula was attempted causes the spreadsheet to completely crash (see Figure 4).

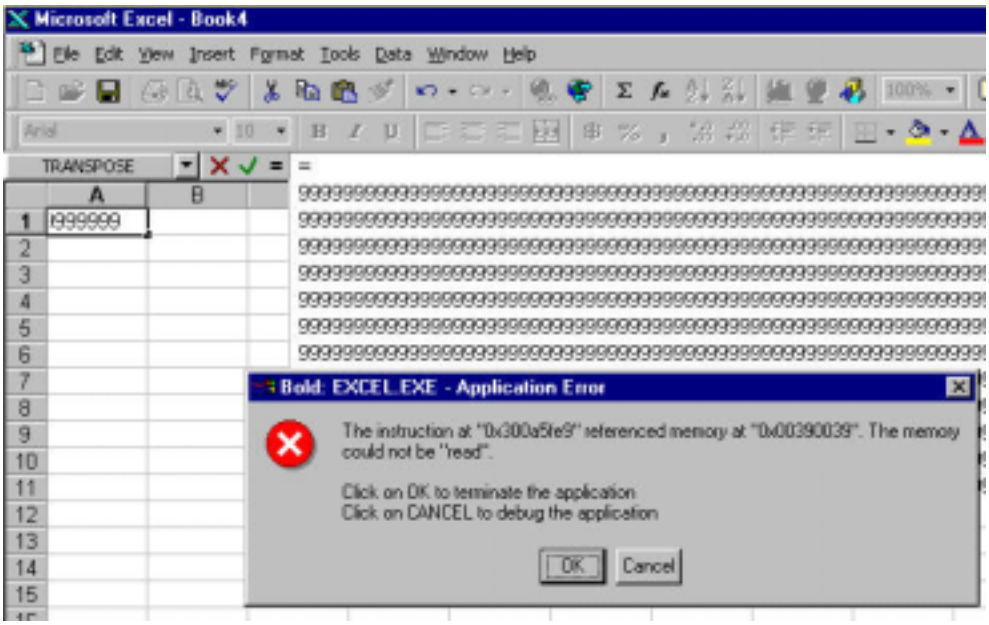

Figure 4: -- Microsoft® Excel® 97 Crashes Due to Corrupt Data

Three inputs in sequence are required to cause the spreadsheet failure: the entry of the excessive length formula had to be followed by pressing the "OK" button to clear the dialog box, and finally, pressing the "Enter" key at the appropriate cell location. Some failures occur only after long, complex input sequences. This makes such defects difficult to diagnose and reproduce.

Though these examples of failure may seem unimportant because they are in software features not commonly used, there are reasons they must be taken

seriously. One, mentioned earlier, is for study. We learn the most from our mistakes and by studying these failures we learn more about the very nature of software failure. A second reason that these failures are important is more subtle. The examples shown are but a single instance of failure brought about by the software defect. A defect has been detected by using extreme values that commonly would not be used. The range of input values and sequences that exercise the defect have not been explored. For example, further study of the spreadsheet defect described above reveals that the input sequence required to exceed the input constraint exception is 1024 characters. This seems to be a sufficiently long equation for the spreadsheet. However, the buffer overflow is detectable at 383 characters and a failure will occur with an input stream that does not produce the "Formula is too long" error message. This is a much shorter equation than actually occurred at the first appearance of the failure. Perhaps there is a different and commonly used sequence that will exercise this same defect. Until the problem is precisely diagnosed, the significance of the defect cannot be determined. The harm that may be caused by this defect is also not obvious. This type of defect, a buffer overrun, is the type commonly exploited by those who would maliciously attack systems. It is conceivable that this defect could be exploited by sending a file via e-mail. The file could contain a macro that exploits this defect whenever the file is opened and control of the computer could be relinquished to the malicious user.

In this example, an input constraint was checked and error information returned to the originator. There was a disconnect, however, since that input constraint failed to match some internal storage constraint and that internal storage constraint was not tested and therefore not reported. It is doubtful that an error reporting mechanism even exists for reporting the storage constraint violation other than the "Application Error" message shown above.

The "Y2K" bug described in the first chapter is another example of a storage constraint defect. When the year is represented as the last two digits of the year, 1999 is that last year that the storage media can represent. Attempting to store the year 2000 or later will resort in a storage constraint violation. The two digit representation of the year assumes a common century. The values of that representation may be considered to be in an equivalence class, *Twentieth Century*. Care must be taken to avoid storing a value that is not in the equivalence class.

2.5 Properties of Constraints

Constraints may take on a variety of characteristics depending on the type of data involved. Different data structures present different constraint properties. Integer values and calculations, for example, can be bounded simply by upper and lower limits. Other data structure representations, however, introduce many other constraint considerations, such as floating point precision, character string length, character string alphabet, array length, record size, etc. In addition to data and

computation constraints, there are also performance constraints, which are not addressed here. Table 1 provides a summary of the properties of constraints of data types and the following paragraphs describe some of these constraint properties.

Table 1 -- Summary of Constraints on Data Types

Data Type	**Constraint Properties**
Integers	Range Precision
Real	Range Precision
Enumerations	Discrete Values
Characters	See Enumerations
Pointers	See Enumerations
Arrays	Element Constraints Syntactic Semantic
Complex Structures	Element Constraints Semantic

2.5.1 Integer and Real

Integer and real data are constrained by range and precision. Ranges, in general, involve upper and lower bounds where values are constrained to be between (or outside) the range of boundary values.

Integer precision arises because of rounding issues: consider the integer divisions $(A+B+C)/3$ versus $A/3 + B/3 + C/3$) when A, B, and C are integers.

Assuming that integer division rounds down (as is normally the case), when A, B, and C each have the value 1 the first expression result is 1, but the second result may be 0. The evaluation of these expressions is language dependent.

Another example is documented in the IEEE Floating Point Standard [IEEE, 1990]. Denormalized representations are used for very small values. This creates the situation that $(1/x)$ is not represented for all (very small) representations of x. For example, the range of values for single precision is 2^{-149} to $(2-2^{-23})\times 2^{127}$. The reciprocal of 2^{-149} is 2^{149} and this value is not in the range of values represented.

2.5.2 Enumerations and Characters

Enumerations are constrained by specifically allowed values. The ASCII character set [ANSI, 1997] can be thought of as integer data because it is continuous in the range [0..127]. Sometimes, however, an alphabet from the ASCII character set must be considered an enumeration because the system behavior may depend on input of specific characters and not ranges of characters. For example, the `scanf( )` functions in standard C libraries parse strings based on the standard white space characters space (`' '`), tab (`'\t'`), and, new line (`'\n'`). All other byte values (including null, `'\0'`) are considered "letters" and are returned as part of the string that has been parsed by `scanf( )`. The `scanf( )` function

implements constraints on input and output by partitioning the input characters into two classes: white space and non-white space.

2.5.3 Pointers

Pointers are often treated as integers when, in fact, they should be treated as enumerations, i.e., pointers should be permitted only certain, specific values. Pointers are a dangerous software feature and can lead to an extremely large number of partitions when erroneous values and their effects are considered. When passing a pointer as an input to a software component, the value of the pointer usually cannot be tested for correctness and therefore violates the requirement for constraining input values. (See the shellsort example above.)

2.5.4 Arrays

Arrays inherit the constraints of the constituent elements as well as their own properties of length and content.

Arrays of enumeration, such as character strings, may also have syntactic and semantic constraints. Consider a numeric string representing a decimal integer. The syntactic rules are:

```
<Number> ::= <Digit><Digit>*  ;
<Digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

and the semantic rule is:

```
if (Number  < (MaxValue / 10))
then  if (Digit <= (MaxValue – (Number * 10)))
      then valid: Number := ((10 * Number) + Digit)
      else invalid
else invalid
```

where *MaxValue* is an upper bound on *Number* and also on the multiplication (“*”) and addition (“+”) functions. That is to say, *Number*, as a character string, is constrained syntactically to digits and semantically constrained such that it cannot represent a value larger than *MaxValue*.

2.5.5 Complex Structures

Complex data structures also inherit the constraints of their constituent elements. There may be, however, constraints on the structure in addition to those on the components. A simple example might be a structure consisting of a variable length array with storage for the array elements and another variable indicating the current number of elements in the array. The number of elements is bounded by the size of the array and the element values are bounded by the constraints on those elements. As a structure, however, elements outside the bound imposed by the number of elements are not constrained since they are, in effect, not elements technically in the array. Software constraints should be imposed to ensure that these non-elements are not referenced, but this is a property of the structure, not just the elements themselves.

The operational modes of a structure are affected in a similar fashion since the operational modes are derived from the constraints. For example, a structure composed of two integers is constrained by both integers.

The operational modes for the entire structure may be more useful than considering the operational modes of each of the integers. When the only constrains on a structure are those imposed by the individual components, the operational mode values of the structure may be computed as the cross product of the operational modes of each component.

The following example illustrates this point by creating a structure with two integers that are each constrained differently. The integers may establish separate operational modes, each with its own operational mode values, but by combining the constraints, we can arrive at a single operational mode for the structure and fewer total number of operational mode values.

```
struct
{
int V1; /* 0 .. 200 */
int V2; /* 100..300 */
} S;
```

We can consider separate operational modes associated with the values within the structure, *V1* and *V2*,

$$Domain(OpModeV1) = \{V1.Valid,\ V1.Invalid\},$$

where the operational mode value $V1.Valid$ is the set of data values

$$\{S.V1 \mid 0 \le S.V1 \le 200\},$$

$$V1.Invalid = \{S.V1 \mid S.V1 < 0 \text{ or } S.V1 > 200\}$$

and

$$Domain(OpModeV2) = \{\text{V2}.Valid,\ V2.Invalid\},$$

where

$$V2.Valid = \{V2 \mid 100 \le V2 \le 300\},$$

$$V2.Invalid = \{V2 \mid V2 < 100 \text{ or } V2 > 300\}.$$

This leads to four operational mode values for S:

$$\begin{aligned} Domain(OpModeS) = \{ & V1.Valid \wedge V2.Valid, \\ & V1.Valid \wedge V2.Invalid, \\ & V1.Invalid \wedge V2Valid, \\ & V1.Invalid \wedge V2.Invalid\}. \end{aligned}$$

The operational mode for the structure, *OpModeS,* may be reduced to two values by combining the constraints on the individual components of the structure as follows:

$$Domain(OpModeS) = \{\text{S}.Valid,\ S.Invalid\}$$

where

$$S.Valid = \{V1,V2 \mid 0 \le V1 \le 200 \wedge 100 \le V2 \le 300\}$$

and

$$S.Invalid = \{V1,V2 \mid V1 < 0 \vee V1 > 200 \vee V2 < 100 \vee V2 > 300\}.$$

Reducing the domain of *OpModeS* to only two values for *S* simplifies the application of this operational mode but loses the relationship between *OpModeV1*, *OpModeV2*, and *OpModeS*. Operational modes of more complex structures may be reduced in the same manner.

2.6 Testing Constraints

Knowledge of constraints is useful for software testing. An effective technique for locating software defects is to search for unanticipated operational mode values. In general, these attributes of software have not been designed nor implemented and consequently are pervasive in all software. The key to finding unanticipated states is by looking for improperly implemented input, output, computation, and storage constraints. Sometimes complicated input sequences are required to exercise these constraints.

2.7 Designing Constraints

What is most certain is that requirements specifications must identify each external variable and precisely specify the constraints of each of those variables. This is also true for design specifications: each internal variable must be specified in the same precise manner as the input and output. The constraint properties, described earlier, must be completely defined. For example, character array variables (or "string" structures) are constrained by length, alphabet, collation

sequence, syntax, and semantics. The range limitations of the target computational device, given the domain of the variables, must be examined to ensure that the computation is possible over the domain and range of the computation. Indeed, the implementation of a procedure must ensure that the calculation of a function is possible within the constraints of the data presented and the calculation capability available to the function.

The partitions that determine whether a calculation will be correct must also be defined. These partitions determine the constraints that specify the values of the operational modes. These constraints must be explicitly implemented in the software system. During test, these partitions determine the boundary values that must be tested to ensure that the procedure behaves correctly (computes the expected result) for every value of each operational mode.

In the next chapter, I examine prior work on constraint design and then I develop a process for specification and design based on operational modes.

Chapter 3. Constraint Design and State Modeling

There are many references in the literature that suggest a need for specifying, designing, and testing constraints and states. However, none goes so far as to dictate how such an endeavor is accomplished. The following section describes a few representative examples of the application of constraints to software development.

3.1 References to Constraint Design

Kernighan and Plauger [Kernighan & Plauger, 1978, p. 97, 118] have seventy-seven (77) programming style checklist items that include the following:

> "Check input data for validity and plausibility."
> "Make sure data cannot violate the limits of the program."
> "Test programs at their boundary values."

These properties are vital to proper system operation and require explicit definition and rigorous treatment. Indeed, they have not followed their own advice, for we find in [Kernighan & Plauger, 1981] essentially that same code for *shellsort* that is shown to be unsafe in our prior example. The earlier reference, [Kernighan & Plauger, 1978], was coded in Pascal and is intrinsically safer than the C language example of *shellsort*. The Pascal code cannot exhibit the same symptoms because the language does not allow the same constructs nor does it handle memory in the same manner as the C language.

[Shooman, 1983, p. 120] provides a table of "defensive programming" recommendations. This table of "typical items to be checked" includes array bounds, division by zero, input from devices, stack depth, output, data from other sources.

Discussing specification reviews in his modern software engineering text, Pressman briefly mentions the need to specify constraints [Pressman, 1997, p. 292].

> "Be sure stated ranges don't contain unstated assumptions (e.g., 'Valid codes range from 10 to 100.' Integer? Real? Hex?)"

Though Pressman describes software state with respect to the "Z" language (see below), very little connection is made between data constraints and states.

In addition to these references on designing software constraints, there is substantial literature on assertions and exceptions. However, these properties are not associated in the literature with state and the effect of external and internal events on state. Because these techniques generally divert program control, they can sometimes *cause* software failure and support fault tolerance inadequately. For example, consider the case of an exception instigated by a division by zero fault in a function containing two division operations. At which invocation of divide did the failure occur? This is important because an action may have taken place between the two division operations (such as reserving a system resource) that must be reversed for the function to proceed properly.

Recent work on the design of languages such as UML [Booch, *et al*., 1999; Jacobson, *et al*., 1999; Rumbaugh, *et al*., 1999], Eiffel [Meyer, 1992] and Java [Lambert & Osborne, 1999] place greater emphasis on assertions to provide bounds checking generally know as "preconditions," "postconditions," and "class invariants."

Sun Microsystems has developed "Assertion Design Language" (ADL) [Sun, 1996] that allows a developer to specify procedure post conditions. However, ADL does not require a full definition of post conditions, nor does it concern itself with stored state.

Spivey defined the "Z" specification language [Spivey, 1988] that allows specification of preconditions and postconditions. Full definition of these conditions, however, is not provided nor is it required in the language. Variables are assumed to be within the specified constraints.

All of these methods allow for constraint programming but none completely specify how and when to provide constraint checking. It is vital that constraints be detected as early as possible in the development cycle and that the characterization of constraints be imperative and not merely possible as in the "Z" language. A robust design language must *require* constraint specification and the process for utilizing the design language must ensure that the constraints are implemented in code including the appropriate response to constraint violations. Certainly, if

constraints are not specified in the design, it is unlikely they will appear in the implementation.

I propose a rigorous method of designing constraints based on the definition of operational modes and my theory of software constraint defects. Careful constraint design should provide a complete definition of all program boundary values under all conditions. Proper coding of these constraints will ensure that a system produces only correct output and behaves in a predictible manner under all operating conditions.

3.2 States and State Modeling

The work of Mills, Linger, and Hevner [Mills, *et al*., 1987] in the cleanroom method shows the importance of representing software as a state machine. States are determined by input (stimulus) history. Mills assumes that input sequence enumeration is not difficult during the 11-step box-structure expansion process:

> "Define the black box
>
> (1) Define black-box stimuli.
>
> Determine all possible stimuli for the black box.
>
> (2) Define black-box behavior.
>
> *For each possible stimulus*[1], determine its complete response in terms of its stimulus history

[1] Emphasis added.

```
"Design the state box
      (3)   Discover state data requirements
               For each response to be calculated,
               encapsulate its stimulus history into a
               state data requirement. …" [Poore &
               Trammell, 1996, p179].
```

In this extract, the entire process for determining state is encapsulated in the phrases, "… determine its complete response in terms of its stimulus history." And "… encapsulate its stimulus history into a state data requirement. " This is a non-trivial task since there are almost always an infinite number of stimulus histories that arrive at a single state. However, little, if any, guidance is provided.

Prowell's work [Prowell, 1996] (also associated with cleanroom) identifies the need to design the states of software but also provides few clues as to how to accomplish this. His work identifies states by input history and provides a procedure (declared an algorithm) for identifying the states by enumerating input sequences. The major difficulties with this procedure are the assumption that input sequences in the same equivalence class (state) are easily identified and that the procedure will terminate, thereby assuring that all states are identified. In fact, neither is necessarily the case. Assume that each input sequence identifies a state. The problem of determining whether two of those states are equivalent is to determine 1) whether the set of available inputs is identical, and 2) for each such input, whether the destination states equivalent. Since determining whether two states are equivalent requires that it be determined whether two (possibly the same)

states are equivalent, this is infinitely recursive. A different approach to the identification of states is required.

Whittaker [Whittaker, 1997b] provides a definition of state as a function of operational modes as used in software testing:

> "Definition 7. A *state* of the system under test is an element of the set *S*, where *S* is the cross product of the operational modes (removing the impossible combinations)."

In the application of this concept, operational modes have been determined by reverse engineering of operational software for testing purposes. By making decisions about operational modes early in the development lifecycle, we avoid the practices that create the defects we would otherwise uncover only during testing. A method for developing the state model from operational modes, inputs, and input properties is presented in [El-Far, 1999]. The state model is developed from the cross product of operational modes and the impossible states are eliminated by a process based on combinations of operational mode values that are not allowed due to input condition conflicts. The transition function is computed from the transitions between operational mode values.

Chapter 4. Constraint Specification

This chapter proposes modifications to the software engineering process to avoid constraint related defects. These modifications introduce constraint design to include properties generally overlooked by current design techniques. The technique is illustrated by means of a small example. Chapter 6 provides a longer and more complex example.

The basic engineering process is 1) *determine what* is to be engineered, 2) *determine how* it is to be engineered, 3) *perform* the engineering, and 4) *validate* that the engineering process was successful. This development process includes the classical waterfall software development processes, requirements analysis, design, implementation, and test. But rather than the waterfall or even the more modern iterative or spiral development models [Sorensen, 1995], the actual process is recursive in that as development progresses, information is uncovered that affects prior design decisions, driving process steps that influences the current process step as well as subsequent steps. Each such discovery must be incorporated into the requirements, design, implementation, and validation.

Operational modes are derived from constraints, which in turn are extracted from requirements and design. Figure 5 illustrates operational mode design data flow through the requirements analysis and design phases of product development. This diagram shows the contribution of system requirements analysis and design to the development of the constraints necessary for determining operational modes.

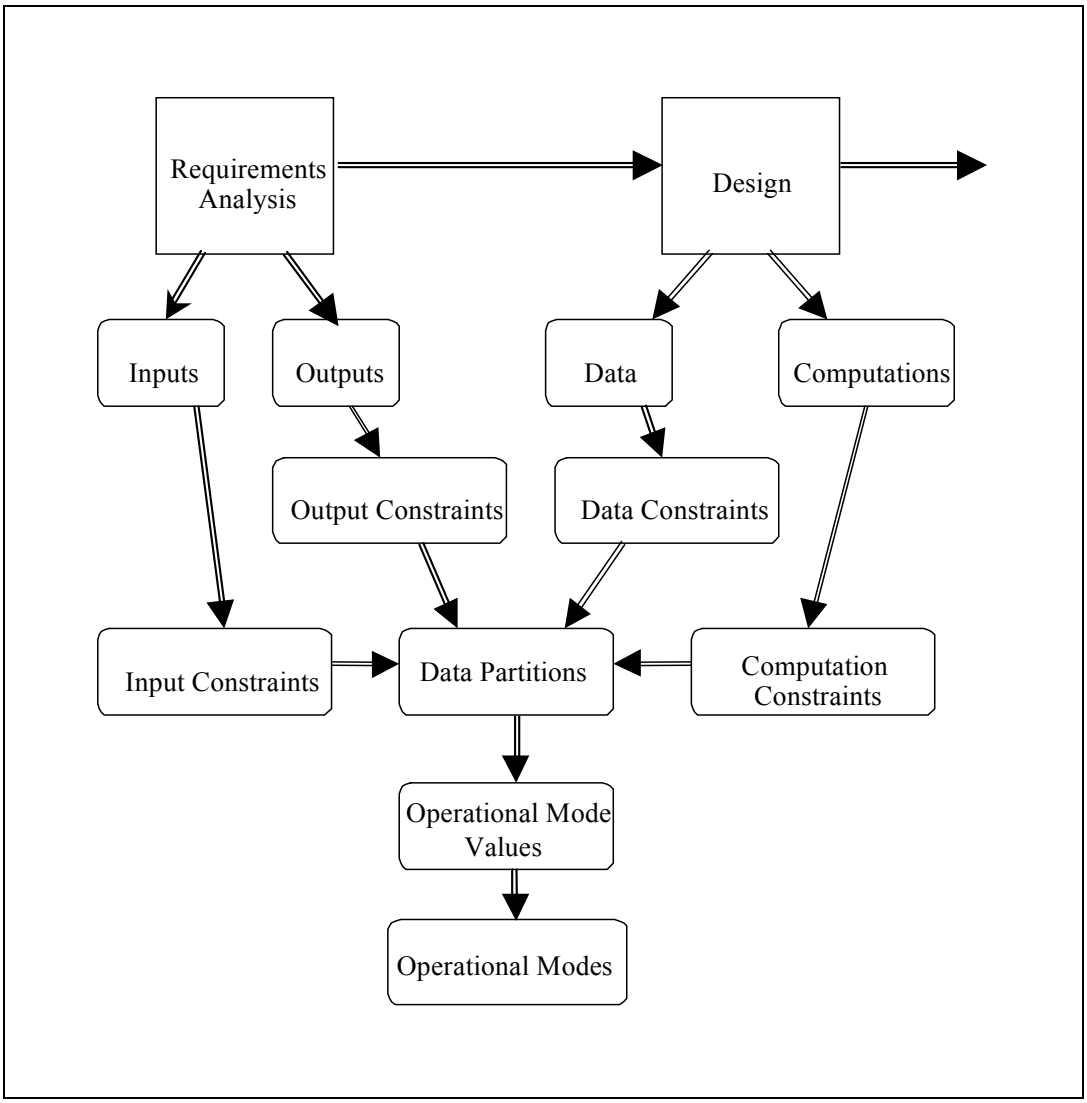

Figure 5 -- Operational Mode Design Data Flow

The modifications to design methodology will be illustrated using a variation of the running average problem. Presented in the form commonly found in software design texts, the running average computation might be expressed as follows:

```
Integer N, Sum, Value;
N = 0; Sum = 0;
While Input(Value)
        N = N + 1; Sum = Sum +Value;
        Print (Sum/N);
```

This code suffers from three of the four classes of failure to constrain:

1. failure to constraint input (because there is no provision for identifying and handling failures of the procedure "Input,")
2. failure to constraint output, (Was the procedure, "Print," successful?) and
3. failure to constrain computation. (The computation of N+1 and Sum + Value can generate computation constraint violations when they overflow.)

My implementation of this example using the proposed changes to the design process will redesign this procedure to eliminate these defects.

Starting with the requirements analysis, the statement of the base requirements is: 1) Input a sequence of positive integer character strings. 2) After each number string has been entered, print the average of all numbers entered thus far. The following analysis and design refines these requirements into a coded example that does not contain the constraint defects described above.

4.1 Requirements Analysis

The engineering process that determines *what* is needed, when applied to software, is called "requirements analysis" or "requirements discovery." In the large body of literature on this subject, a significant characteristic has been given scant attention or omitted entirely: adequate identification of constraints on input and output. Though sometimes mentioned as a desirable characteristic, current requirements analysis techniques do not demand input constraint specification. Output constraints are hardly ever mentioned. The improvements to the development process presented in this dissertation demand that these constraints be identified and specified. In fact, each requirement must be constrained and each such constraint results in an additional requirement. This recursive characteristic must be terminated by a common limiting factor, such as system failure; e.g., when it is not possible or not useful to report error information, a system failure has occurred. Requirements must specify the characteristics of system failure.

Though the proposed techniques are applicable to any of the current development methodologies, the following developmental model presents a design process modified from [Whittaker, 1998]. Two useful tools for requirements analysis are the system context diagram [De Marco, 1979] and transaction analysis [Prowell, 1997]. The method presented here for requirements discovery is, 1) develop a system context diagram, 2) perform transaction analysis, and 3) develop a preliminary data dictionary [De Marco, 1979]. From these work products a

complete requirements specification may be developed that will include the definition of all applicable external (artificial) constraints.

4.1.1 System Context Diagram

A system context diagram is a device for identifying system boundaries and defining the elements external to a system. This diagram provides a clear definition of those elements (called "users") that interact externally with the system and those objects that are internal to the system (called "components")[2]. A *user* is any object or person that provides system input or is affected by system output. De Marco calls a user a "source or sink."

> "A source or sink is a person or organization, lying outside the context of a system, that is a net originator or receiver of system data." [De Marco, 1979, p. 59]

Users have several important characteristics: 1) users cannot be constrained, 2) requirements cannot be placed on users, 3) users can do anything, and 4) it is okay to warn the user about system limitations.

---

[2] The external objects are identified here as "users," though this term (and the term "actors" used by [Jacobson, 1995]) are probably overly anthropomorphic since the objects that may interact with a system may be devices or operating systems as well as human "users."

There are some important classes of human users of the system that are often overlooked:

1. software development engineers are users,
2. software test engineers are users, and
3. software maintenance engineers are users.

The system context diagram also helps identify the data that flows across the system boundary. Figure 6 is an example of a system context diagram as applied to the running average example.

The large circle represents the entire system. Nothing is shown inside this circle since, at this stage of analysis, nothing is known about the internal characteristics of the system.

The smaller circles represent users. Users may be abstractions. In this example a keyboard is described as a user providing input. This could be less abstract by defining various keys as individual system users. Conversely, the display and keyboard could be combined to form a higher level abstraction representing a single user with both input and output capabilities. The arrows indicate the direction of data flow and the text associated with the arrows enumerates the types of data that flows between the users and the system.

Developing a system context diagram reveals a number of other requirements as was the case developing this example. The diagram shown is the final version since during various stages of development, several data items were

added such as “System Error Message” (the need to display diagnostic information when the system cannot perform the required task).

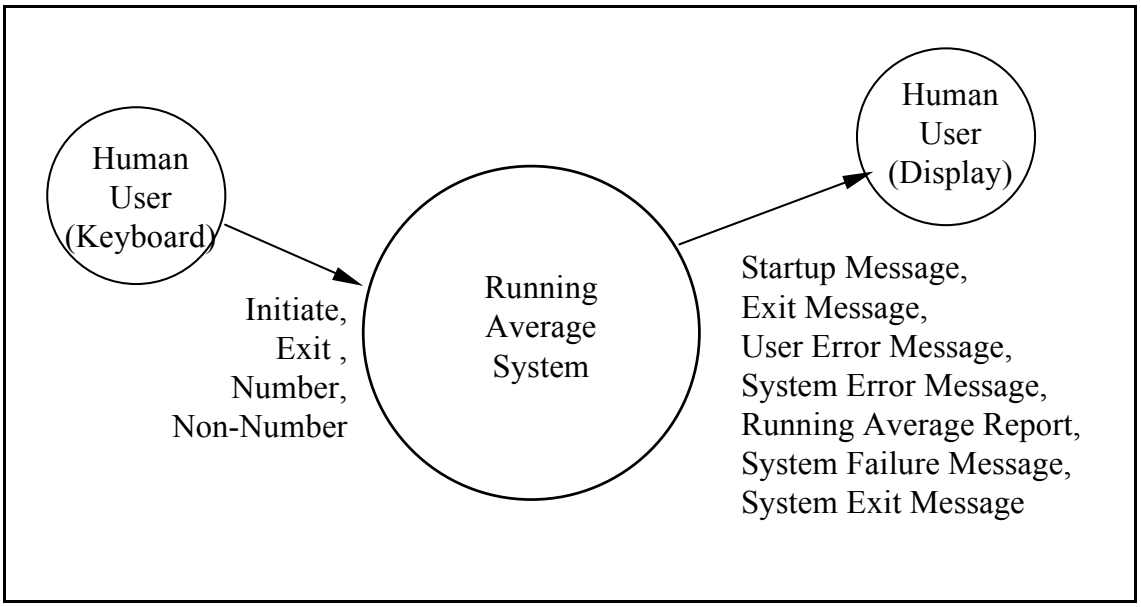

Figure 6 -- Running Average System Context Diagram

The only communication defined in this system is with the human operator. The operator has the ability to start the program, enter numbers, enter things that are not numbers, and to provide some sort of input to indicate that the program is no longer needed. The system will output a welcome message, the average for each valid entry, an error message for when the operator enters something invalid, an error indication when the system cannot perform the requested function, and a termination message.

4.1.2 Transaction Analysis

Transaction analysis is a structured design technique described in [Mills, *et al*., 1987 and Whittaker, 1998]. The requirements for inputs, outputs, and events are stated in high level terms called abstractions. These abstract requirements are refined into details called "derived requirements." Complex transactions are decomposed into sequences of simpler transactions. My modification to the transaction analysis process is to require that all transactions be constrained. As part of the expansion of transactions into requirements, these constraints must be identified and catalogued. A constraint is any limitation on the transaction. As a consequence of this process change, a new column has been added to the transaction analysis table to allow definition of transaction constraints.

During requirements discovery, transaction analysis identifies only those activities that are externally visible to the system. The information provided by transaction analysis is also used to refine the system context diagram as described above. Transaction analysis must ultimately define all interactions between users, the system, and the services to be provided by the system.

In the Running Average example, where user requirements are not specified, only token constraints are specified. Such token constraints should be highlighted for customer review since they are previously undefined requirements and should err on the side of conservatism, i.e., minimum cost, since customers are likely to approve generous product constraints. In this example, input and output

were first limited to the maximum value of a 32-bit twos-complement number. Later analysis established that this requirement was too liberal and it was restated to accommodate a limitation discovered later in the process. The original requirement could have been met by using a more complex computation, but would significantly increase development time. These new values must be reviewed by the end-user and must be specified as requirements. When not specified as requirements, any constraint imposed by the designer may be assumed to be valid. This is a common characteristic of software and many "bugs" are simply unspecified features or constraints.

A transaction analysis table consists of one row for each unique transaction, user, input, or output. There is a column for transaction item number (TR), transaction, newly identified user, newly identified input, newly identified output, and transactions constraints. Each transaction is numbered for ease of identification in later design steps and for traceability. One column contains a description of each transaction originated by either an external user or inside the system. The form of the description is a simple noun-verb-object. In this manner, unique users and new objects are identified. When a user or object not already on the system context diagram is identified, the diagram must be updated to contain these new features. The new attribute of the transaction analysis table is the column for the constraints on the transaction.

When the constraint is abstract, such as “message must fit in the display area,” the abstraction must be refined to more specific details. In this instance, “display area” will need more detail before this constraint is fully defined. As each constraint is defined, it is assumed that this constraint may be violated and a transaction defined to accommodate each such violation. A general transaction such as “system fails” provides for failures of the error reporting transactions.

Transaction abstractions are compositions of more detailed, underlying transactions and eventually must be fully elaborated. One method of ensuring that transactions are adequately elaborated is to examine the constraints for specificity. In our example, “Message must fit in the display area,” is not specific until the size of the display area is identified. Since all transactions must be constrained, all natural or artificial constraints must be recorded for each transaction. These constraints may be violated for many reasons including operator error and system failure and therefore for each transaction constraint there must be a corresponding error transaction describing how that error will be handled. The behavior of the system should be specified even when a failure transaction can not be completed.

The transaction analysis example creates a table with an entry for each transaction discovery. These transactions may be readily identified from the system context diagram in Figure 6 above. However, our new design process requires that we capture the constraints on each transaction. Because of the possibility of violation of each constraint, an additional transaction associated with

each constraint violation is provided. These are new transactions and thus new requirements. A system error terminates this recursive process. Table 2 is the transaction analysis table for the running average example.

For closure of the system context diagram with the transaction analysis table.

1. Each user on the system context diagram must appear in a least one transaction.
2. All users that appear in the transaction analysis table must appear on the system context diagram.
3. Each input on the system context diagram must appear in the appropriate context in the transaction analysis table, i.e., where an input is associated with a user on the system context diagram, a transaction must appear in the transaction analysis table associated with that user and as an input from that user.
4. Each input in the transaction analysis table must be associated with a user and that input must be associated with that user on the system context diagram and shown as an input.
5. Each output on the system context diagram must appear in the appropriate context in the transaction analysis table, i.e., where an output is associated with a user on the system context diagram, a

transaction must appear in the transaction analysis table associated with that user and as an output to that user.

6. Each output in the transaction analysis table must be associated with a user and that output must be associated with that user on the system context diagram and as an output to that user.

Table 2 -- Running Average Transaction Analysis

Item	Transaction	New User Identified	New Input Identified	New Output Identified	Constraints
1	User starts program.	Human User (screen)	-	Startup Message	Message fits display area.
2	User terminates program.	-	Exit Value	Exit Message	Message fits display area.
3	User enters positive integers.	Human User (keyboard)	Number	-	Number is a positive integer and no larger than the maximum 1,000,000. The number of values entered shall not exceed 1,999.
4	User enters a non-number.	-	Non-Number	Error message	Non-Number is an entry that is not a “Number” as defined above and is not the Exit entry. Error Message must fit in the display area.
5	System computes the average.	-	-	-	Running Average subject to the same constraints as “Number” above.
6	System fails to compute average.	-	-	System Error message	Message fits display area.

Item	Transaction	New User Identified	New Input Identified	New Output Identified	Constraints
7	System reports average.	-	-	Running Average Report	The Running Average Report shall identify the result, present the result as a 1 to 7 digit decimal number. The report shall fit the display area. The average shall be reported to the nearest integer (precision requirement).
8	System Fails	-	-	System Failure message	Message forced to fit in available space in the display area to avoid an additional error.
9	System terminates program.	-	-	System Exit Message	Message fits display area.

4.1.3 Preliminary Data Dictionary

The *preliminary data dictionary* enumerates each input, output, and other data item identified during transaction analysis with one entry for each item. Each entry provides an item index, an item identifier, a description, a list of all constraints on that item, and traceability information. The index is provided for ease of reference and traceability. For example, the constraint of item 1 of Table 3, identifies a new data item, "display area." Subsequent definition of this data item refers back to item 1 from which it was derived for traceability purposes. The item

identifier must be unique within the system and should be functional but brief. It is the intent that this identifier will be subsequently used in the design and implementation that will follow. The description, on the other hand, must be detailed and contain information concerning the purpose, scope, and intent of the item.

The requirements specification engineer must elicit important information during the requirements discovery phase. In particular, requirements definition must capture artificial constraints on inputs, outputs, and data structures identified during the analysis. This dissertation exposes the new meta-requirement that each data dictionary entry must identify all constraints on the data item identified by that entry. This new meta-requirement is a direct result of identifying the failure to properly constrain data elements during the development process. The decomposition and implementation of these constraints, when designed and implemented, will result in reduction of the types of failures identified in the opening chapter.

The "Origin" column is added to provide traceability of the origin of the requirement for the data. This column may also refer to the transaction number that placed the requirement for the dictionary entry.

A new concept is that each constraint requires an additional data dictionary entry for the constraint violation error indication. This error indication information

specifies system diagnostics. This is similar to the requirement for to add an error transaction for each normal transaction.

Closure with the preliminary data dictionary is accomplished with the following checklist.

1. Each data object appearing in the transaction analysis table must appear as an entry in the preliminary data dictionary. This includes objects referenced by the system as well as input and output objects. (The term "object" as used here is the object clause of the transaction statement, such as, "System computes average." "Average" is the object of the sentence.)
2. The constraints on each object must be specified.
3. Where the constraints are abstract, the object must be sufficiently refined so that the constraints are specific. In the example, the constraint "Must fit display area" is not specific and must be refined to "no longer than 79 characters."

Table 3 is the preliminary data dictionary for the running average example.

Table 3 -- Running Average Preliminary Data Dictionary

Item	Data Item	Usage	Description	Constraints	Origin
1	Startup Message	Output	Message to welcome user to the program.	Must fit in the display area.	TR 1…
2	Display Area	Output Constraint	User's view of the system.	A single line of output no more that *Line Length* characters in length.	PDD 1
3	Line Length	Output Constraint	Maximum Length of an output line.	79 characters.	PDD 2
4	Exit	Input	Input from operator indicating that termination is requested.	Must be distinguishable from *User Entered Number* below.	TR 2
5	User Exit Message	Output	Message informing user that the program is terminating.	Must fit in the display area.	TR 2
6	User Entered Integer	Input	Numeric value entered by the user.	Positive numeric string less than or equal to the *Maximum Integer*. String is terminated by a new line character. String may contain only decimal digits.	TR 3

Item	Data Item	Usage	Description	Constraints	Origin
7	Maximum Integer	Input Constraint	Maximum input value accepted by the system.	1,000,000.	PDD 6
8	Non-Number	Input	Any invalid character string entered by the user.	Any input string containing a non-decimal digit character. Terminated by a new-line.[3]	TR 4
9	User Error Message	Output	Message informing the user that entry is invalid.	Must fit in the display area.	TR 4
10	Average Value	Output	Value computed by the system. (Sum of Input values) / (Number of values entered).	Constrained by the User Entered Number constraints.	TR 5, PDD 7
11	System Error Message	Output	Indicates to the user that the system has made an error.	Must fit in the display area.	TR 6
12	Running Average Report	Output	Displays the computed running average, in context.	Must fit in the display area.	TR 7

[3] Any character string that is not the exit message nor a valid number string. This string may exceed the input display area size since the user may not be constrained and may type any length string. The operating system may also constrain the length of string that the user may enter.

Item	Data Item	Usage	Description	Constraints	Origin
13	System Failure Message	Output	Indicates that a message intended for the display area did not fit.	Should fit the display area. (Over length message may be truncated.)	TR 8
14	System Exit Message	Output	Indicates to the user that the program was terminated by the system.	Must fit in the display area.	TR 9

Transaction analysis identifies three types of data items: input, output, and constraint values. These are data values that are externally visible to the system. During the design phase internal data items will be identified and added to the data dictionary. The design phase will complete this dictionary as internal data structures are defined and constrained.

4.1.4 Preliminary Operational Mode Design

Once the preliminary data dictionary is complete it is then possible to create the preliminary design of operational modes. From this initial set of operational modes, the top-level state model may be constructed from the techniques described in [El-Far, 1999]. This state model will define the operation of the overall system and include interaction with the memory shared between the program and the rest of the system (or retained by the system between program executions). For the example provided, the only operational mode that can be defined is *Domain*(*OpMode Running Average*) = {*Not Invoked, Invoked*}. This is because there is no external memory modified by the system. The operational model of the example system is trivial since the state set is {*Not Invoked, Invoked*} and each input either causes the Running Average system to remain invoked or else exits.

4.1.5 Requirements Specification

From the details provided by the system context diagram, the transaction analysis table, and the preliminary data dictionary, a final requirements specification may be developed. Each transaction must result in one or more specific requirements statements ("shalls"). Each constraint identified in the preliminary data dictionary must appear as a limitation on a specific requirement.

The requirements specification may take any of the many accepted specification forms including that defined by the *IEEE Recommended Practice for Software Requirements Specifications* [IEEE, 1993]. The information captured in the transaction analysis table and the preliminary data dictionary should completely define the functional requirements of the system. All constraints identified during transaction analysis and development of the preliminary data dictionary must be included in the requirements specification. These requirements analysis features provide direct input to the design phase.

Chapter 5. Constraint Design

Software design describes in detail *how* each requirement will be implemented. Design is accomplished by continued decomposition of the transactions identified during requirements analysis. The transactions identify *what* must take place. Design determines precisely how that will be accomplished. This results in procedures to implement these transactions and the data structures necessary to support them. Procedures encompass decisions, data movement, and calculations; all of which are constrained either by hardware limitations or requirements. The new technique of constraint identification and operational mode design will provide a mechanism to define those limitations uniquely and rigorously. The result of the design phase is a data dictionary and a functional design specification.

Each input, output, calculation, and stored value must be described in detail. Again, as in requirements analysis, current techniques fail to completely identify data and calculation constraints. Newly recognized constraints must be checked to determine whether they are, in reality, new requirement constraints. Sufficient detail must be provided so that no additional design decisions are necessary during the implementation phase.

During design, decisions must be made about the allowable states internal to the system. This is simplified by considering only the design of the operational modes and mode values.

During design, the design engineer decomposes the system into two component types, procedures and data structures. In addition to the constraints on inputs and outputs defined during requirements analysis, the design phase introduces constraints imposed by the limitations on procedures and data structures. As before, constraints create error conditions and produce error data. An important characteristic dealt with poorly by current design technology is the flow of error information.

5.1 System Decomposition

System decomposition is based on the system context diagram and transaction analysis and defines the inner working of the system, i.e., from the system boundary inward. Decomposition results in the identification of procedures and storage shared between the procedures. It is during the decomposition of the system that critical decisions regarding system state occur.

Optimal system decomposition is a rich area for research. As development progresses, the system becomes more specific and less abstract and decomposition is complete when no abstraction remains. Choosing decomposition boundaries to minimize coupling and maximize cohesion seems to be an optimal strategy. See

[Berard, 1993] for example. I use the term "cohesion" here to describe the degree that the functions of a component refer to a single data object and "coupling" to describe the frequency that a component must call on the functions of another component to achieve its task. Minimizing coupling and maximizing cohesion tends to minimize operational modes and consequently minimize the number of states of the system. The study of operational modes may lead to a more optimal solution for this problem. More research is necessary.

[Parnas, 1972] has suggested decomposition based on data hiding and modules likely to change. I have selected this data hiding technique as the primary means of system decomposition because it seems intuitive that hiding data also hides operational modes, which, in turn, hides states thereby simplifying modeling, testing, and system understanding in general. In particular, a component is defined around a single data structure and all functions that modify that data structure are contained within that component. The data structure is not directly visible outside that component and access to the data in the structure is achieved only through access functions within the component. External requests for data are not trusted and must be constraint checked.

For the running average example, the two data structures required are *Number* and *History*. *Number* is the value entered by the user and may also represent the exit condition. To perform the average operation, there must be some

form of historical data retained by the system, (*History*). The choice of the form of the stored history is a design decision and such decisions often prove problematic.

What is the optimal way to store the history? For our example problem the choice of retaining the sum and count of the values entered is the classical solution. There are other choices, such as retaining each of the values entered in a list or retaining the current average and the count. Retaining the list of values places a more limiting constraint on the number of values that may be included in the average. Retaining the current average and count allows reconstruction of the sum of values but with severe precision constraint issues. I will not attempt to solve this problem here but identify this problem for future research.

In any case, the running average example has two components, one based on the *Number* data structure and one on *History*. Externally visible functions are *Input* for the *Number* component and *Initialize* and *Average* for the *History* component. The *Initialize* function establishes the initial values of *History*. Normally this should be to set the sum and count of values to zero, but to make our system more testable, these values are also parameterized and the values set to zero as the default condition. This allows us to set *History* to any value so that test cases can be constructed representing any possible point in the history.

The *Input* function must accept character input from a human user and therefore must anticipate all possible input sequences. It is necessary to

assign a special input sequence to represent the program termination sequence. The input function may accept only those sequences that represent either the termination sequence or a sequence representing a valid numeric input. The *Input* function must be decomposed into a simple state machine to process each successive input character and produce the results: *Valid Number*, *Termination*, and *Invalid Sequence*. This is an example of selecting the boundaries of a function to hide information and consequently state. As each character is entered, the accumulated value of the number being input may overflow the maximum value or be an invalid character (for the current state of the input function.) This information is all hidden, however, from the user(s) of the *Input* function.

The overall system flow: The *Initialization* function establishes the initial *History* and calls on the *Input* function to complete the task. The *Input* function will process input sequences and deliver valid numeric values to the *Average* function. *Input* must recognize the terminating input sequence and exit gracefully.

5.2 Specification of Procedure

Procedure specification provides a detailed description of executable functions. The specification should be sufficiently detailed that mapping the procedure to any language should be relatively effortless. This is, in fact, the goal of design. Sometimes the specification of procedure requires the identification of new data structures. These new data structures, including constraints, are added to

the data dictionary. For each procedure described, the data structures affected must be identified including any new constraints on those data structures imposed by the procedure. This identifies and creates the need for an additional component.

The entire running average system requires detailed design of the *Input*, *Output*, and *Average* functions. However, for brevity, the design example concentrates on the *Average* function, though the problems encountered are similar for each executable component.

It is necessary, though insufficient, to bound the input to the limiting condition, *Max Input*. Though the input component may limit the range of the values it delivers, *Average* must perform this check and deliver to the calling component the *Input Invalid* status. The calling component must be prepared to deal with the *Input Invalid* status. But the *Input Out-of-range* condition is not the only constraint that may be violated by a given invocation of *Average*. It is possible that the *Max Values* constraint may be violated as well. This constraint is not on the value of the input but rather on the number of values that may be entered. *Average* must detect this condition and provide an appropriate response (which must also be dealt with by the *Input* component).

Next, the computation of the average value itself may not be correct because of hardware or specified limitations. If *History* is designed to meet the requirements for the maximum value and maximum number of values, and *Average* is to accommodate the necessary computational range, we may well have escaped

the need for this particular constraint check. And lastly, the output of the average value may be constrained by the format of the output itself. The *Average* component must check the output value to ensure that it meets the constraints on output and if it does not, report this to the *Input* function which must be prepared to handle it.

The total data requirements determine the operational modes of a system. The data structures accessible across the system determine the system level operational modes. In all cases it is desirable to minimize the number of operational modes and the number of operational mode values within those modes. Data isolation is equivalent to operational mode isolation. That is, storage not visible to a component does not contribute to the operational modes of that component and hence does not contribute to the visible state. State minimization implies and is implied by operational mode minimization.

5.3 Final Data Dictionary

The design phase completes the data dictionary. At completion the dictionary contains the definition of data internal to procedures. The data dictionary rules that were previously defined still apply: all entries are constrained, data design constraints must be checked to see if they are previously unrecognized requirements, except for parameterized constants, and new constraints will require

additional data dictionary entries. Abstract data structures will have been decomposed until only constant values and base data types remain.

The final data dictionary is identical in form to the preliminary data dictionary show in Table 3. The distinction is that the final data dictionary will have additional entries for the intra-component data flow and internal data storage. Further, data design details are complete. For this running average example, the preliminary data dictionary specified the nature of error messages, but the final data dictionary specifies the exact content of the error messages. Table 4 is the final data dictionary for the running average example.

Table 4 -- Final Data Dictionary

Item	Data Item	Usage	Description	Constraints or Value	Origin
1	Startup Message	Output	Message to welcome user to the program.	Must fit in the display area. "Welcome to the Running Average program."	TR 1…
2	Display Area	Output Constraint	User's view of the system.	A single line of output no more that *Line Length* characters in length.	PDD 1
3	Line Length	Output Constraint	Maximum Length of an output line.	79 characters.	PDD 2
4	Exit	Input	Input from operator indicating that termination is requested.	Must be distinguishable from *User Entered Number* below. A Line with no characters.	TR 2
5	User Exit Message	Output	Message informing user that the program is terminating.	Must fit in the display area. "Thank you for using Running Average."	TR 2
6	User Entered Integer	Input	Numeric value entered by the user.	Positive numeric string less than or equal to the *Maximum Integer*. String is terminated by a new line character.	TR 3

Item	Data Item	Usage	Description	Constraints or Value	Origin
7	Maximum Integer	Input Constraint	Maximum input value accepted by the system.	1,000,000.	PDD 6
8	Non-Number	Input	Any invalid character string entered by the user.	Terminated by a new-line.[4]	TR 4
9	User error message	Output	Message informing the user that entry is invalid.	Must fit in the display area. “ Invalid Input.”	TR 4
10	Average Value	Output	Average value computed by the system.	Constrained by the User Entered Number constraints. (Sum of Input values) / (Number of values entered) rounded to the nearest integer.	TR 5, PDD 7
11	System Error message	Output	Indicates to the user that the system has made an error.	Must fit in the display area. And	TR 6, 8

[4] Any character string that is not the exit message nor a valid number string. This string may exceed the input display area size since the user may not be constrained and may type any length string. The operating system may constrain the length of string that the user may enter.

Item	Data Item	Usage	Description	Constraints or Value	Origin
12	Running Average Report	Output	Displays the computed running average, in context.	Must fit in the display area. “ The current average is xxxxxxx.” Where xxxxxxx is the average value, up to 7 digits.	TR 7
13	System Exit Message	Output	Indicates to the user that the program has terminated.	Must fit in the display area.	TR 9
14	Number of values entered.	Stored data.	An integer value representing the number of valid values the user has entered.	Must be less than or equal to the *Entry Count Limit*	TR 3, 4
15	Entry Count Limit	Stored data constraint	The maximum number of values that may be entered by the user.	Constrained by computation in DD 10 by DD 20: 2000.	TR 3, 4, DD 10
16	Entry Count Constraint Violation Message	Constraint violation	Message to output when Entry Count Limit Constraint is violated.	“Too many values have been received.”	DD 14, 15
17	Sum of Input Values	Stored Data	An integer values representing the sum of the values the user has entered.	Must be less than or equal to the *Input Sum Limit*.	TR 4

Item	Data Item	Usage	Description	Constraints or Value	Origin
18	Input Sum Limit	Stored data constraint	The upper bound on the sum of input values.	Must be less than or equal to the maximum 32-bit integer: 2,147,483,647	TR 4
19	Input Sum Constraint Violation Message	Constraint violation	Message to output when Input Sum Limit Constraint is violated.	“Sum is too large for this value.”	DD 17, 18
20	System Error Message	Constrain violation	Message to output when system fails to properly display error messages.	“ Error Message Overrun.”	

5.4 Operational Modes

The basic method of operational mode design is to identify all variables and computations in a program and to identify and document all natural and artificial bounds of those variables and computations. These bounds provide the partitions necessary to identify the values of the operational modes of the system. Every operation on an operational mode value must be assured to be correct dynamically at the bounds of that operational mode value.

Operational modes are designed in the following manner.

1) Select the variable or variables that contribute to a particular operational mode.
2) Create a functional name for the variable set that describes the operational mode.
3) Determine the set of partitions that affect those variables
4) Order the partitions in a manner that guarantees exclusivity, i.e., there is complete independence of the values within the sets defined by the partitions.
5) Identify with each partition a name depicting the equivalence class. This name should be appropriate for an operational mode value.

The operational characteristics of the running average system depend on the behavior of the Human User. Even at this level of abstraction, we can identify three different activities that the Human User can perform. We'll identify this as an

operational mode, *Domain(OpMode UserInput)* that will be partitioned into the set of values {*Number, Non-Number*, and *Exit*}. The details of the actual elaboration of the operational mode values can be left until later but an important point is to realize that regardless of how we implement these operational mode values, coded partitions will determine precisely the value of the operational mode, *UserInput*.

Another operational mode that exists for any system is the basic operational status, *System*. *System* has the values {Invoked, Not invoked}.

Clearly, there is yet another operational mode that cannot be defined in terms of the inputs since there are varying system behaviors without variation of inputs as they are defined at this point in the development cycle. These behaviors might be explained in terms of input histories, however, the identification of relevant input history is an arduous task. But, as explained previously, we can determine the operational modes of a procedure by examining the persistent storage visible to that procedure. The persistent storage for the *Average* component is *History*. The validity of a given invocation of the *Average* component is determined by the values of *History*, and the partitioning of *History* determines the operational modes of the running average system. In particular, the operational modes are *Domain(Count)* = {*Within Limits*, *At Limit*}, and *Domain(Sum)* = {*Within Limits*, *Current Input Causes Sum Limit Violation*}. Note that *Count* and *Sum* do not have values "*Out of Limits*." This is accomplished in the code by checking constraints and never storing an invalid value in the associated variables.

5.5 Design Methodology Summary

Software consists of the constituents inputs, outputs, storage, and procedures. All program constituents are constrained either naturally (hardware limitations) or artificially (requirements limitations). The complete set of constraints on storage imposed by natural and artificial limitations and by the mapping all of them to storage constraints creates the set of boundary conditions for each storage element. This, in turn, formally defines the operational mode values and hence the operational modes of a system.

Proper system design requires that, where artificial limitations exist, hardware must be selected to meet or exceed these limitations. This includes such characteristics as:

- Word size
- Adequate representation of variables such as floating point precision
- Processor with sufficient performance

The definition of the inputs, outputs, and procedures of a system occurs during the requirements definition and design phases. The complete definition of storage must be completed during the design phase.

5.6 Implementation

The coding phase should be a simple implementation of the design. All decisions regarding representation and function should have been made in the design phase. Any unknown representation must be referred back to the design to

ensure that no new constraints are introduced by coding. If new constraints are to be introduced, those constraints must be evaluated as system requirements.

Code must be provided to fully implement all constraints so that improper input is prohibited, the results of each computation is correct, and output is correct in appearance and value. By constraining the values stored in memory, the system will avoid entering improper states. By checking stored values against constraints, invalid states created by events outside the system are detected and controlled. In fact, this provides us with a useful system quality metric for robustness. We can define a system robustness metric as the percentage of total constraints that have been implemented in code. The number of constraints actually implemented can be estimated by the number of predicates (if statements, etc.) in the program and the number of constraints needed can be estimated by the number of variables (inputs + outputs + storage) plus the number of operators (computations) in the program. This is reminiscent of McCabe's Cyclometric Complexity [McCabe, 1976] divided by Halstead's Length metric [Halstead, 1975].

The code for the running average example appears in Appendix E. This program is claimed to be free any of the four types of constraint errors. To establish this, a mechanism for validation has been provided.

5.6.1 Validation

Validation ensures that the system requirements have been met. This is a final system check. Each requirement, including requirement constraints, should be tested, i.e., each operational mode value should be verified at its boundaries. The relations that define the operational mode values also define these boundaries. Among other tests, the system should be compared against the system state model to ensure proper response to each input for each state of the system.

With the addition of carefully defining constraints during the requirements analysis, it is now possible to define verification testing in terms of those constraints. Each constraint should be tested at the boundary conditions of that constraint. When the constraint is a range, for instance, a test should be performed at the outer limits within the range and at the immediate inner limits just outside that range. Where external means are not available for testing these limits, test probe code should be added to permit the testing of these limits. These testing techniques are illustrated with the test suite for the running average example.

The test suite for running average together with the test results are listed in Appendix F. Each constraint listed in the final data dictionary is tested. Where special test conditions were required for testing, such as testing for the maximum number of input values, special testing code was added to the running average program, and access to this code provided through special test mechanisms.

5.6.2 Test Verification

To ensure that the running average test program was a thorough test, a test verification mechanism was developed the performed the following functions:

1) Scan the running average source code and locate each numeric string
2) For each such string
    a) Create a version of the source program with that numeric string increased by 1.
    b) Compile that version
    c) If no compilation error, run the validation test noting failures.
    d) Repeat a)-c) with the original string decreased in value by 1.

This verification test produced interesting results. The first was that a condition could exist during code reuse that would cause the program to hang up by an inadvertent recursion if the length of the error message explaining that an error message was too long, was itself too long. That problem has been corrected in the final version presented here. The second was an interesting oversight of requirements definition. A user, the operating system, was not identified as a consumer of data from the program. The program environment, unix, expects an error code returned by programs. From habit, this author programmed such exit codes, however, they were never specified during requirements definition nor during design. Though they have been coded into the test for validation, they are

still not specified in the requirements and design. The third was trivial in that the fault injection software permuted numeric values that occurred in comments and, of course, such "defects" could not be detected by the run time test. The fault injection software was modified to ignore numeric values in comments. The forth concerned configuration information. Some configuration parameters do not affect the external operation of the system such that modification of one of these parameters would not cause the system to malfunction. For example, in the running average example, there is an internal parameter, Exit, that is the value returned by the Input to indicate that the operator has requested program termination. As long as this value is unique from the input values themselves, values are indistinguishable external to the program.

There are 46 non-comment numeric constants in the running average program resulting in 92 test validation cases. Of these cases, the program could not be compiled in 3 cases. With the correction of the identified defects in the code and test only 15 of the cases went undetected. The test validation procedure detect 74 of the 92 faults that were injected into the source code. The source for the test verification program together with the verification results are published in Appendix G.

5.6.3 Scaleability

This method of design has been tested on relatively small problems and the question of whether it be useful for large programming system still remains. This is the problem known as scalability. I claim that the suggested modifications to the design process are as scalable as the process itself. One manner that the technique shown is scalable is that as the system is decomposed into components, each of those components may be considered a system in itself and the same technique as applied to systems may be applied to components as though they were complete systems. It should be noted that the design methods shown here are concerned only with the software component of a system and assumes that the total system has already been decomposed into a hardware component and a software component. Other decomposition may have been applied to define what the "system" is that is under design.

Chapter 6. Case Study: Kernighan and Pike Markov Chain Algorithm

The best software available today is not engineered but is the product of skilled craftsmanship. Software development texts, in general, provide directions and examples for crafting software and honing craftsmanship skills. [Kernighan & Pike, 1999] is an example of such a text and Brian W. Kernighan is recognized as one of the best software development craftsmen in the business today. The implementation of the "Markov Chain Algorithm" found in this text is a typical example of coding examples in software development texts. This is a very well crafted piece of software. Never-the-less, this program "fails" when put to rigorous testing. Defining failure in crafted software is problematic, however, since there is seldom, if ever, a clear definition of the function of the software. For example, this Markov Chain algorithm processes "words." No clear definition of this term is provided, and, in fact, such a definition is required as the basis for the claim that the software fails. In general, this program inputs "words" and reproduces those "words" in a different sequence. The code provided does not do this in 100% of the cases because the *fscanf()* function utilized by this program does not produce proper C strings, but rather produces an array of *char*. This standard library routine, *fscanf()*, treats the null character (\000) as a valid input character which it reproduces in the output. This, however, is the C language string termination character and during string processing, including output, signifies the end of the

string. Thus characters following the null character in the same "word" are not reproduced when the character array is output as a string.

Using the Markov algorithm as a case study requires that the entire problem be re-engineered to meet some general requirements that are not met in the original example. Some of these requirements are 1) reusability, 2) robustness, 3) maintainability 4) reliability 5) fault tolerance., and 6) user friendliness (operability). The decomposition of these requirements result in such derived requirements as 1.1) Precise function definition, 1.2) Defined boundaries of operation, 3.1) Clear, concise, unambiguous error messages, 3.2) Functional modularity, 5.1) Corrects errors where possible. For reuse, the system documentation must make clear the operation performed so that a potential new user of the system or system code may determine easily whether the system will apply to a new application.

On the issue of reuse, when considering a software component that has already been designed and tested, such as the Markov Chain algorithm of this case study, the constraints of the component must be known to determine whether the component is useful for a new design under consideration. For this example, given that the Markov algorithm for random text generation based on a random walk through a given source text is the desired feature, is the capacity of this previously designed component sufficient for the new application? The answer to this question for the example in the cited text is not known. In fact, repeated executions

with the same input data may produce different results depending on how busy the system is that is executing the program. The reliance on *malloc*() to assign memory as needed by the program means that the results of running the program with the same set of data is not repeatable, i.e., the program may process an input text on one occasion, but will not process that same text on another occasion. The quantity of data that may be processed is dependent on the amount of free memory in the system at the time of execution. Memory requirements for the system are not specified.

6.1 System Context Diagram

We begin the redevelopment with a simple system context diagram as shown in Figure 7. This diagram indicates that the user initiates the program and that *Words, White Space*, and an *End of File* are processed from standard input and that *Words* are sent to standard output.

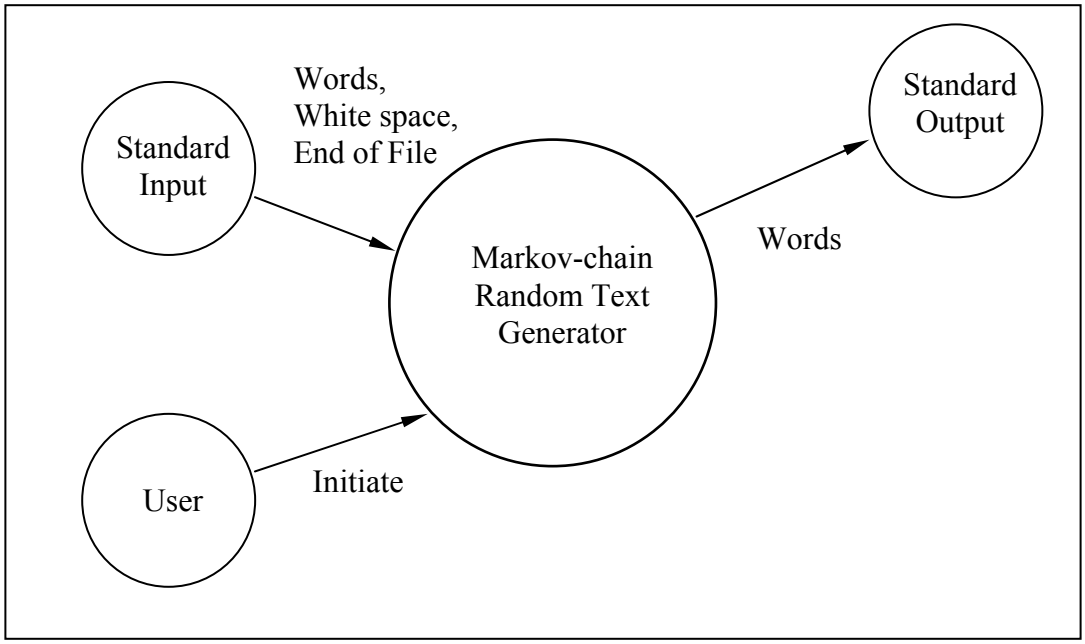

Figure 7 -- Markov-Chain Random Text Generator System Context Diagram A

6.2 Transaction Analysis

This information is copied into the transaction analysis table shown in Table 6. The important new feature of the transaction analysis table added by this dissertation is the constraints on the transactions. This is where important definitions are included in the design, such as the precise definition of "Word" that is lacking in [Kernighan & Pike, 1999]. Other important constraints are also identified, such as the number of unique "Words" that may be processed, the definition of "White Space," and the character set from which "Words" are composed. In addition, by the rule that each transaction must have a corresponding

transaction to process a violation of that constraint, transactions identifying error conditions are also defined (with associated constraints.) The special self referential transaction 12 provides for system errors corresponding to failure to process constraint violation transactions.

Table 5 -- Markov Algorithm Transaction Analysis

Item	Transaction	New User Identified	New Input Identified	New Output Identified	Constraints
1	User initiates program	User	Invoke		
2	Standard Input provides Word	Standard Input	Word		A word consists of one or more printable ASCII characters. Printable ASCII characters are valued 33..126 ('!' .. '~'). Words may be no more than 20 characters in length.
3	Standard Input provides White Space		White Space		White space consists of one or more white space characters (tab, space, newline).
4	Standard Input provides End of File		End of File		The total number of words may not exceed 50,000. The total number of unique words may not exceed 20,000.
5	System determines valid successors.				For each word, a valid successor is defined as the word following any identical word for which the prior word in the input stream is also identical. The number of valid successors may not exceed the total number of words.

Item	Transaction	New User Identified	New Input Identified	New Output Identified	Constraints
6	System emits random valid successors to Standard Output	Standard Output		Valid Successor Word	Starting with the first word from standard input, a randomly selected (even distribution) valid successor is output on each output line. A maximum of 10,000 words may be output.
7	System emits Invalid Character Message to Error Output	Error Output		Invalid Character Message	Message must fit error display area.
8	System emits Invalid Word Length Message to Error Output			Invalid Word Length Message	Message must fit error display area.
9	System emits Too Many Words Error Message to Error Output			Too Many Words Error Message	Message must fit error display area.
10	System emits Too Many New Words Error Message to Error Output			Too Many New Words Error Message	Message must fit error display area.
11	System emits Too Many Successors Error Message to Error Output			Too Many Successors Error Message	Message must fit error display area.
12	System emits Invalid Error Message Message to Error Ourput			Invalid Error Message Message	Message must fit error display area.

6.3 System Context Diagram Revisited

The expansion of the transaction table to include transactions for processing constraint violations has identified a new user, Error Output, with associated data structures identified external to the system. We redraw the system context diagram in Figure 8 to include this new user and the additional data structures.

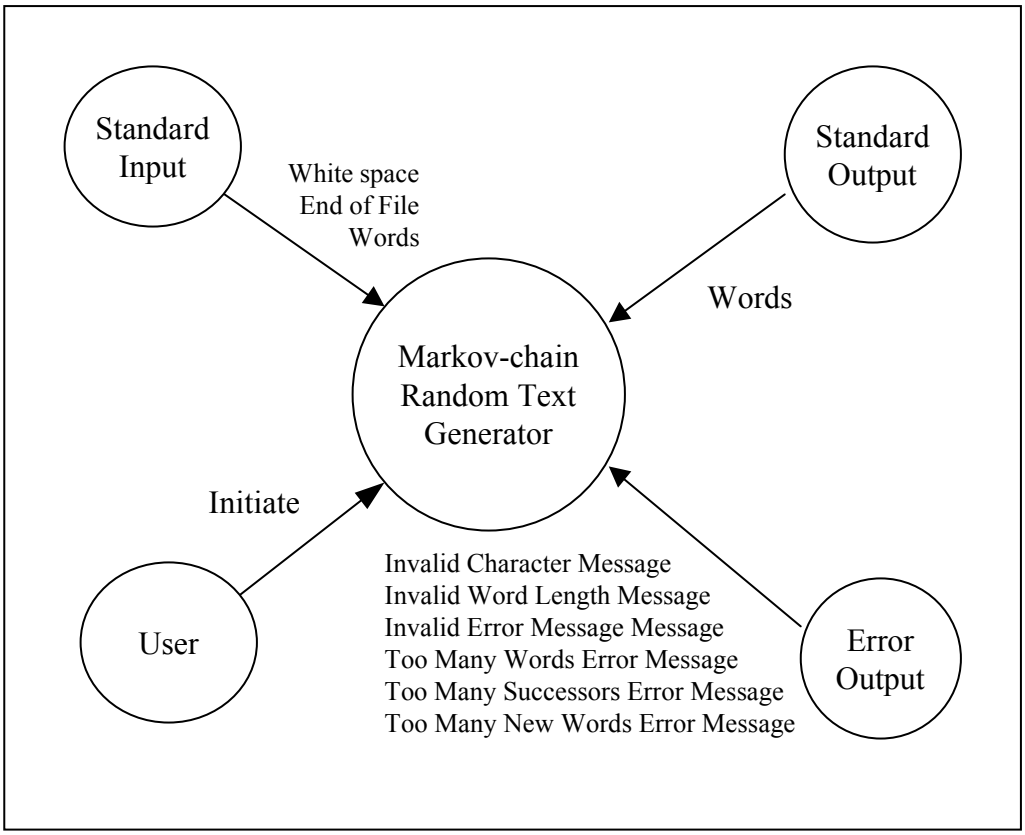

Figure 8 -- Markov-Chain Random Text Generator System Context Diagram B

6.4 Data Structures and Decomposition

We begin the design of our random text generating system by decomposing the system into smaller subsystems. The recommended method of decomposition is based on information hiding. By examination of the transaction analysis table we see that there are two fundamental data structures mentioned, *Words* and *Valid Successors*. Our arguments for storing only unique input words cannot improve on [Kernighan & Pike, 1999], however, there the similarity ends. We retain the original order of the text by maintaining a list of the words in input order. The text is maintained in a text pool as a "string" but instead of pointers, an index into the text pool identifies specific word text. Thus, the input word sequence is stored as a sequence of text pool indices. Our choice for identifying unique words is to maintain a binary tree with a node associated with each unique word. We also maintain a circular list of all of the occurrences of each unique word. This allows quick identification of the location of all identical words starting from any word position in the text. A word that occurs only once will refer immediately back to itself. We could use this structure with the Markov Chain concept to compute valid successors in the same manner as [Kernighan & Pike, 1999], however, this ingenious and clever idea is unnecessary. It is the opinion of this author that clever solutions should be avoided because of the cost of maintenance of such programs. In this case, we no longer need to deal with data of unknown length since we have a design requirement that constraints be specified. It seems a simple matter that if

we know the number of valid successors to any word position, we can easily randomly select one of them. A second data structure is required to identify the valid successors. As with the text, the valid successors may be kept in a successor pool and a pointer and count of successors will identify each valid successor list. Only one such list need be maintained for each unique word pair, so the list is associated with the first occurrence of the word pair for the successor list. This means that it will be necessary for each word in the original word list to identify the first occurrence of the word pair to which it belongs.

These data structures determine the need for subsystem components. The contents of a component is determined by the type of access to the data structure defined within the component. All storage modification references to a data structure are contained in the same component and the data structure is global to that component. External access to the data in that component is controlled by access functions that return the requested data, after validating that the request for data is valid. Functions within a component are "trusted." This means that the code in a given component is assumed correct within that component, i.e., a component does not corrupt its own data. This is avoided by rigorous checking of storage constraints to ensure that data as stored is correct.

In addition to the definition of the data structures, a complete definition of error conditions and messages is provided. All error message text is defined in a single file so they may be translated easily to additional languages. There is

another benefit to this technique, and that is that each error has a unique and testable identifier. This design follows the original requirement that the system terminate operation on a failure. In a more realistic system designed for fault tolerance, the nature of the error is passed to the user for error recovery. The knowledge of the specific error is usually necessary for this recovery to take place and this method of error identification supports that requirement

Table 6 -- Markov Chain Preliminary Data Dictionary

Item	Data Item	Usage	Description	Constraints	Origin
1	Word	Input	A Sequence of printable ASCII characters obtained from Standard Input	Printable ASCII characters range from 33 ('!') to 126 ('~'). A word may contain no more than 20 characters.	TR 2
2	Smallest Printable Character	Input Constraint	The smallest printable ASCII character.	'!' (value of 33)	PDD 1
3	Largest Printable Character	Input Constraint	The largest printable ASCII character.	'~' (value of 126)	PDD 1
4	Maximum Word Length	Input Constraint	Maximum number of characters in a Word	20 Characters.	PDD 1
5	White Space	Input	Word delimiters; the ASCII characters for tab, space, and newline.	Tab (9), space (32), or newline (10).	TR 3
6	End of File	Input	Standard unix end of file value.	Normally -1.	TR 4
7	Maximum Number of Words	Input Constraint	Maximum number of words that my be read from standard input.	50,000 words.	TR 4

Item	Data Item	Usage	Description	Constraints	Origin
8	Maximum Number of Unique Words	Input Constraint	Maximum number of different words.	20,000.	TR 4
9	Valid Successors	Storage	A list for each input word of valid successors for that word.	For each alternative successor, both immediate predecessor input words must be identical.	TR 5
10	Maximum Valid Successors	Storage Constraint	There can be no more successors that there are words.	Maximum Number of Words	TR 5, PDD 7
11	Valid Successor Word	Output	A random selection from the Valid Successors	Uniform distribution of selection from Valid Successors.	TR 6, PDD 9
12	Maximum Words Output	Output Constraint	The maximum number of words that may appear on the output.	10,000	TR 6
13	Invalid Character Message	Output	Error message when an invalid input character has been encountered.	Must fit in the display area.	TR 7

Item	Data Item	Usage	Description	Constraints	Origin
14	Invalid Word Length Message	Output	Error message when a word is too long.	Must fit in the display area.	TR 8
15	Too Many Words Error Message	Output	Error message when too many words have been input.	Must fit in the display area.	TR 9
16	Too Many New Words Error Message	Output	Error message when too many unique words have been input.	Must fit in the display area.	TR 10
17	Too Many Successors Error Message	Output	Error message when a too many unique words have been input.	Must fit in the display area.	TR 11
18	Invalid Error Message Message	Output	Error message when an error message of invalid length has occurred.	Should fit the display area. (Over length message may be truncated.) The display area is defined as 79 characters. (Line wrap, after the 80th character, causes an additional, unwanted new line.)	TR 12, PDD 13, 14, 15, 16, 17, 18

Rather than rely on the Markov chain to select from a list of unknown length, structures of known length are utilized. The requirement that arrays be of known length so that constraints may be checked allows the problem to be defined in a manner such that the number of valid successor words for each word of the source text may be computed. This makes the computation of a random successor trivial and easy to understand. To minimize the storage required for text, a text pool is created that stores only unique words. A list of words is maintained and the word list contains the index to the text of the word, word numbers for lesser and greater words of a binary tree for searching for new words, and a circular linked list of identical words. By searching each circular list of identical words, identical word pairs may be located to compile a list of valid successors. (If a word pair occurs only once, the only valid successor is the next word in sequence.) The second data structure contains the lists of valid successors for each word of the source text. Similar to the manner in which text is saved in a text pool, the lists of valid successors are kept in a pool of successor lists. If there is only one valid successor, it need not be stored in the pool since it is the next word in succession. Further, the list of successors only need be associated with one occurrence of the prefix word pair, so subsequent use of the valid successor list may simply refer to the first reference to the list. These data structures are defined in detail in the Final Data Dictionary in Table 7.

Table 7 -- Markov Chain Final Data Dictionary

Item	Data Item	Usage	Description	Constraints	Origin
1	Word	Input	A Sequence of printable ASCII characters obtained from Standard Input	Printable ASCII characters range from 33 ('!') to 126 ('~'). A word may contain no more than 20 characters.	TR 2
2	Smallest Printable Character	Input Constraint	The smallest printable ASCII character.	'!' (value of 33)	PDD 1
3	Largest Printable Character	Input Constraint	The largest printable ASCII character.	'~' (value of 126)	PDD 1
4	Maximum Word Length	Input Constraint	Maximum number of characters in a Word	20 Characters.	PDD 1
5	White Space	Input	Word delimiters; the ASCII characters for tab, space, and newline.	Tab (9), space (32), or newline (10).	TR 3
6	End of File	Input	Standard unix end of file value.	Normally -1.	TR 4
7	Maximum Number of Words	Input Constraint	Maximum number of words that my be read from standard input.	50,000 words.	TR 4

Item	Data Item	Usage	Description	Constraints	Origin
8	Maximum Number of Unique Words	Input Constraint	Maximum number of different words.	20,000.	TR 4
9	Valid Successors	Storage	A list for each input word of valid successors for that word.	For each alternative successor, both immediate predecessor input words must be identical.	TR 5
10	Maximum Valid Successors	Storage Constraint	There can be no more successors that there are words.	Maximum Number of Words	TR 5, PDD 7
11	Valid Successor Word	Output	A random selection from the Valid Successors	Uniform distribution of selection from Valid Successors.	TR 6, PDD 9
12	Maximum Words Output	Output Constraint	The maximum number of words that may appear on the output.	10,000	TR 6
13	Invalid Character Message	Output	Error message when an invalid input character has been encountered.	Must fit in the display area. “Invalid Character. Not a valid text file.”	TR 7

Item	Data Item	Usage	Description	Constraints	Origin
14	Invalid Word Length Message	Output	Error message when a word is too long.	Must fit in the display area. “Word too long.”	TR 8
15	Too Many Words Error Message	Output	Error message when too many words have been input.	Must fit in the display area. “Too many words encountered.”	TR 9
16	Too Many New Words Error Message	Output	Error message when too many unique words have been input.	Must fit in the display area. “Too many new words encountered.”	TR 10
17	Too Many Successors Error Message	Output	Error message when a too many unique words have been input.	Must fit in the display area. “System Error. Too many successors.”	TR 11
18	Invalid Error Message Message	Output	Error message when an error message of invalid length has occurred.	Should fit the display area. (Over length message may be truncated.) “System Error. Error message truncated.”	TR 12, PDD 13, 14, 15, 16, 17, 18
19	Text	Storage	Character storage for unique words.	Each word is preceded and followed by a null character. Only unique words are stored.	FDD 1

Item	Data Item	Usage	Description	Constraints	Origin
20	Stored Characters	Storage	The number of characters stored in Text	Must be less than or equal to the (Maximum Word Length + 1) * Maximum Number of Words + 1.	FDD 1, 4, 7
21	Word List	Storage	An array of information about each word ordered by input.	One entry for each word of source. Size is Maximum Number of Words	FDD 1
22	Words	Storage	The number of words in the source.	Must be less than or equal to the Maximum Number of Words	FDD 7
22	Text Index	Storage	Item in Word List. Character index in Text of the Word.	Greater that equal to 1, Less than or equal to the number of character stored in Text.	FDD 1
23	Lesser	Storage	Item in Word List. Binary tree branch for limb less than the current word. Contains a Word Number of Word List.	Must be less than or equal to Words.	FDD 8
24	Greater	Storage	Item in Word List. Binary tree branch for limb greater than the current word. Contains a Word Number of Word List.	Must be less than or equal to Words.	FDD 8
25	Next	Storage	Item in Word List. Word number of the next word of a circular list of equal words.	Must be less than or equal to Words. All Words in circular list will have equal Text Index values.	FDD 8

Item	Data Item	Usage	Description	Constraints	Origin
26	Successor List	Storage	The collective pool of valid successor lists. Each successor list is a list of Word Numbers of valid successors to the current word.	There are no more Successors than there are words.	FDD 9
27	Successors	Storage	The list of successor parameters. There is one entry for reach Word Index.	There are no more Successors than there are words. Will have the value 1 when there is only 1 valid successor or the current word is not the first occurrence of the matching word pair.	FDD 9,10
28	Successor	Storage	An item in Successors. The index to the Successor list for the list of valid successors for this word.	Must be less than or equal to the number of successors stored in the Successor List	FDD 10
29	Valid Successor Words	Storage	An item in Successors. This entry is the number of successor words that will be found at Successor.	Must be less than or equal to the number of successors stored in the Successor List.	FDD 10

Item	Data Item	Usage	Description	Constraints	Origin
30	First	Storage	An item in Successors. The successor list is associated with the first occurrence of second word of a set of matching word pairs. First is the Word Index of that first occurrence of the word pair.	Must be the word index of the first occurrence of duplicate word pairs.	

6.5 Implementation

Appendix H contains the program source for the Markov Chain system. This program does not have the "defect" of the original code in [Kernighan and Pike, 1999] caused by accepting non-printable (and string terminating) characters. But in addition to correcting the defect, the detailed properties of the entire program are defined and specified. The quantity of input data that the program can process is clearly defined as well as the consequences when the constraints are violated.

Summary

This dissertation has shown that modern software fails and fails frequently. By examining many failures of commercial, off-the-shelf software, assumed to be thoroughly tested before release and sale, a new method of categorizing failures has been identified. These failure categories are based on the incorrect or incomplete design of constraints for input, output, storage, and computation. These constraints are shown to be partition values for operational modes and that by design of operational modes, these constraints will be included in the design and these classes of defects can be eliminated. Improvements to existing design methodology have been recommended to develop the information necessary to design operational modes. A simple process is presented illustrating how to implement these methodological improvements. This process is applied to a small design example previously proven to be correct, yet containing defects within the defect classifications presented. By redesigning this example using the improved development methodology, the defects no longer exist.

The improvements in the design technique suggest a metric for robustness, i.e., the robustness of a system can be defined as the percentage of constraints implemented in the final code.

This development technique has been applied to a more serious case study and has proven to be scalable. A defect in the original code of the case study has been corrected as a matter of course using the defined improvements to the design

process. In addition to correcting the defect, the result is a clearly specified function the produces repeatable results.

Conclusions

Software has a reputation for unreliability that has been verified in this dissertation. Software is generally unreliable due to an underlying weakness in the methodology used to develop software.

Examination of failures produced in the laboratory with respect to operational modes reveals that there is an underlying reason for this unreliability. This reason is the failure to provide proper constraint checking and corresponding constraint violation processing.

This dissertation has developed a fault classification system. Faults can be classified into four general categories; failure to properly constrain inputs, outputs, stored values, and computations. All software failures examined by this research fell into one or more of these fault categories. For software to be robust, it must test all such constraints and respond appropriately. The "Garbage In Garbage Out" (GIGO) paradigm for software development does not lead to robust software.

I propose that this weakness can be strengthened by improvements to software engineering process. The specifics of the improvements are:

1. Mandate the definition of constraints on all transactions during requirements analysis, specifically on all system inputs and outputs.
2. Propagate those constraints into the design phase.
3. Require the definition of constraints on storage and procedures defined during design.
4. Implement all defined constraints in code.

There are many research issues still remaining. Some of the research areas identified include:

1. Constraints of floating point precision
2. Decomposition optimization
3. Data structure selection criteria

Very little testing technology exists for checking constraints on internally stored data and for finding input sequences that violate them. One area for additional research beyond this dissertation may be to construct models of sequences based on the relationship of inputs and the knowledge of how data is stored in the system. Such a model might then be used to verify the correct operation of the system for arbitrary input.

The constraint violation class of defects creates a very dangerous situation for users. Frequently, the system either crashes or corrupts internal data such that results are no longer trustworthy but the user is not so informed. This paper addresses only the design issues necessary to avoid constraint violation defects and does not address the testing technology necessary to detect all constraint violation defects. Since testing theory covers this subject meagerly, this area is ripe for additional research.

Future Work

This work has exposed many problems and questions that need resolution before the general technology of software engineering may be said to produce reliable software of a known quality.

A common reaction by a developer to a tester exposing an anomaly is that the problem is caused by the environment. A typical example is the problem introduced by floating point precision. In general, we accept the defects introduced by floating point precision errors, yet such errors may cause serious and unnoticed software failure. Floating point now detects and represents overflow and underflow as well as undefined values. Can it not detect and represent loss of precision? Floating point precision is a constraint that may be applied to the four categories of failure defined in this dissertation. Other environmental issues are created by defects in the operating system and common-use libraries. Should any subsystem "trust" any other subsystem?

Some effort has been expended to provide criteria for component decomposition and data structure selection. These design areas must now be re-examined in light of imposing constraints and processing constraint violations. Indeed, this dissertation has shown that all computer operations may produce errors and the current methods of dealing with errors, such as exceptions, do not provide sufficient flexibility to deal will all error processing paradigms. Current program

languages do not have effective standard methods of reporting errors between components nor represent erroneous information in data structures. There are no existing standards for error messages or reporting diagnostic information, such as component trace-back. These are fruitful areas for research.

Previous efforts have developed methods for the reverse engineering of operational modes and to employ those operational modes in the automated generation of software test cases. With the explicit definition of operational modes presented here, it should now be possible to provide operational mode characteristics directly to such test design automation tools. In addition the relationship between operational modes and system usability and between operational modes and general system reliability should be studied.

We infer that by using operational mode design we can eliminate false states that could be reached by erroneous software and that reaching these states will cause serious software failure. However, this reverse engineering of operational modes has proven to be somewhat ad hoc. The operational mode idea can be expanded to reverse engineer existing software operational modes and determine the false states with the associated vulnerability. Techniques must be developed to reliably acquire operational modes and values from existing software. This is required not only to determine system states that must be tested, but to provide the information necessary to re-engineer operational modes to correct defective operational mode design.

Initial examination of the development of the constraints and hence, the operational modes of a system, suggests that this is a highly repetitive process and should be ripe for automation. The recognition and implementation of operational mode (and mode values) must be thorough and automation would enhance this requirement.

State explosion can still occur making a full system state model impractical. It is possible to design a system in which all variables are global and the decomposition of modules is such that components access a large number of data structures in common with other components. This prolific use of high level access to shared data (low system cohesion) results in a large number of operational modes and since the states of a system are derived from the cross product of operational modes, state explosion will surely occur. One clear objective of operational mode design is to establish high system cohesion. Similarly, if the modules unnecessarily create unneeded partitions values, there will be an increase in the number of operational mode values and a corresponding increase in the number of states. The tried and true method of information hiding makes operational modes invisible outside the information access functions and consequently such variables do not contribute to system operational modes thereby reducing state explosion.

Since deterministic finite automata do not represent all reasonable computations, a DFA is not a sufficient model. More sophisticated modeling

techniques are required that may include grammars or other tools of formal language theory. Is there a relationship between operational modes and grammars?

Though the design methodology modification presented contributes to the accuracy of development and design, the question remains, "Are there software defects that can occur even though this process is followed with rigor?" Though this method improves and extends our capability to define and discover requirements, will all requirements be defined? Have we captured all data constraints?

As noted by Knuth in 1968 [Knuth, 1973] and again in 1999, software development is still an art. Before we can declare software development an engineering science, we will need the answer to these questions and to many others.

References

[Aitken & Jones, 1995] Aitken, Peter G., and Bradley L. Jones. Teach Yourself C Programming in 21 Days, Sams Publishing, 1995.

[ANSI, 1997] American National Standards Institute (ANSI), *Information Systems - Coded Character Sets - 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII)*. Document Number: ANSI X3.4-1986 (R1997), New York: American National Standards Institute, 1997.

[Baber, 1997] Baber, Robert L "The Ariane 5 explosion as seen by a software engineer ." http://www.cs.wits.ac.za/~bob/ariane5.htm, Johannesburg, South Africa: University of the Witwatersrand, 1997.

[Berard, 1993] Berard, E.V., *Essays on Object-Oriented Software Engineering*, Englewood Cliffs, NJ:Prentice-Hall, 1993.

[Booch, 1994] Booch, Grady. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings. 1994.

[Booch, *et al*., 1999] Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading Massachusetts: Addison-Wesley, 1999.

[Bergner, *et al*., 1997] Bergner, Klaus, Andreas Rausch, and Marc Sihling, "Using UML for Modeling a Distributed Java Application." Müchen: Mathematisches Institut und Institut für Informatik der Technisichen Universität, July 1997.

[Becker & Whittaker, 1997a] Becker, Shirley A., and James A. Whittaker. "An Overview of Cleanroom Software Engineering." *Cleanroom Software Engineering Practices*, Edited by Shirley A. Becker and James A. Whittaker, Harrisburg PA: Idea Group Publishing, 1997.

[Crawley & Miller, 1983] Crawley, Winston and Charles E. Miller *A Structured Approach to FORTRAN*, Prentice Hall, 1983.

[Cobb & Mills, 1990] Cobb, Richard, and Harlan Mills. "Engineering Software under Statistical Quality Control." *IEEE Software*, 7 (November 1990.): 44-54.

[Cosmo, 1994] Cosmo, Henrick. Black-box specification Language for Software Systems. Masters Thesis, Lund University, 1994. Sweden: Department of Communication Systems.

[Deitel & Deitel, 1994] Deitel, Harvey M., and Paul J. Deitel, *C++ How to Program, 2nd Edition,* Prentice Hall Engineering, 1994.

[Deck, 1997] Deck, Michael. "Development Practices." *Cleanroom Software Engineering Practices*, Edited by Shirley A. Becker and James A. Whittaker. Harrisburg PA: Idea Group Publishing, 1997.

[Dale, *et al*., 1997] Dale, Nell, Chip Weems, and John McCormick. *Programming and Problem Solving with Ada*. Boston, Massachusetts: Jones and Bartlett Publishiers, 1997.

[Dale & Whittaker, 1997a] Deck, Michael, and James A. Whittaker. "Lessons Learned from Fifteen Years of Cleanroom Testing." Presented at *Software Testing, Analysis And Review (STAR) '97*, Cleanroom Software Engineering, Inc., 1997.

[De Marco, 1979] De Marco, Tom, *Structured Analysis and System Specification*, Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

[El-Far, 1999] El-Far, Ibrahim Khalil Ibrahim. Automated Construction of Software Behavior Models. Master's Thesis, Florida Institute of Technology, May, 1999.

[Etter, 1993] Etter, D.M., *Structured Fortran 77 for Engineers and Scientists, 4th Edition.*, Menlo Park, California: Benjamin/Cummings Publishers, 1993.

[Ett & Trammell, 1995] Ett, William and Carmen Trammell. "A Guide to Integration of Object-Oriented Methods and Cleanroom Software Engineering." *STARS Task Final - Comment Draft*. Loral Federal Systems, December 22, 1995.

[Glass, 1998] Glass, Robert L. Software Runaways, Upper Saddle River, NJ: Prentice Hall, 1998.

[Garlan & Shaw, 1993] Garlan, David and Mary Shaw. *An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering*, Vol. 1. River Edge, NJ: World Scientific Publishing Company, 1993.

[Halstead, 1975] Halstead, Maurice H. "Toward a Theoretical Basis for Estimating Programming Effort." *Proceedings of the ACM Conference,*. Association of Computing Machinery, October 1975, pp. 222-224.

[Heitmeyer, *et al*., 1997] Heitmeyer, Constance L., James Kirby, and Bruce Labaw. "Tools for Formal Specification, Verification, and Validation of Requirements," Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS '97), June 16-19, 1997, Gaithersburg, MD.

[Hevner & Mills, 1993] Hevner, Alan and Harlan Mills. "Box-Structured Development Method for System Development with Objects." *IBM Systems Journal*. v. 32, no. 2 (1993): 232-251.

[Hopcroft & Ullman, 1979] Hopcroft, John E., and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Reading, Massachusetts: Addison-Wesley, 1979.

[IEEE, 1994] IEEE Computer Society. *IEEE Standard Dictionary of Measures to Produce Reliable Software, ANSI/IEEE Standard 982.1-1988*, New York: IEEE, 1994.

[IEEE, 1990] IEEE Computer Society. *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-1985(R1990)*. New York: IEEE, 1990.

[IEEE, 1991] IEEE Computer Society. *IEEE Standard Glossary of Software Engineering Terminology*, *IEEE Standard 610-12-1990*, Corrected Edition. New York: IEEE, 1991.

[IEEE, 1993] IEEE Computer Society. *IEEE Std 830-1993, Recommended Practice for Software Requirements Specifications*, New York: IEEE, 1993.

[IEEE, 1994] IEEE Computer Society. *IEEE Standard Classification for Software Anomalies, IEEE Standard 1044-1993*. New York: IEEE, 1994.

[Leveson, 1995] Leveson, Nancy G. *Safeware: System Safety and Computers*, New York: Addison-Wesley, 1995.

[Lions, 1996] Lions, Prof. J. L., Chairman of the Board, "ARIANE 5 Flight 501 Failure Report by the Inquiry Board." (http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html) Paris: 1996.

[Jacobson, 1995] Jacobson, Ivar. "Use-Case Construct in Object-Oriented Software Engineering." In *Scenario-Based Design: Envisioning Work and Technology in Systems Development*, edited by John M. Carroll. John Wiley and Sons, 1995.

[Jacobson, *et al*., 1999] Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Reading Massachusetts: Addison-Wesley, 1999.

[Jacobson, *et al*., 1993] Jacobson, Ivar, Magnus Christerson, Patrick Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach,* Revised Fourth Printing. Addison Wesley, 1993.

[Koskimies & Mäkinen, 1994] Koskimies, K. and E. Mäkinen. "Automatic Synthesis of State Machines from Trace Diagrams." *Software Practice & Experience,* 24,7 (July 1994), 643-658.

[Koskimies, *et al*., 1996] Koskimies, K., T. Männistö, T. Systä, and J. Tuomi. "SCED - A Tool for Dynamic Modeling of Object Systems." Report A-1996-4, Department of Computer Science, University of Tampere. (July 1996).

[Koskimies & Mössenböck, 1996] Koskimies, K.and H. Mössenböck. "Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs." *Proceedings of the 18th International Conference on Software Engineering*, *Berlin*. IEEE Computer Society Press. (1996):366-375.

[Kasner & Newman, 1940] Kasner, Edward and James Newman., *Mathematics and the Imagination*. New York: Penguin Press, 1940.

[Knuth, 1973] Knuth, Donald E. *The Art of Computer Programming, Volume 1/Fundamental Algorithms*, Second Edition, Reading, Massachusetts: Addison-Wesley Publishing Company, 1973.

[Knuth, 1975] Knuth, Donald E. *The Art of Computer Programming, Volume 3/Sorting and Searching*, Second Printing. Reading, Massachusetts: Addison-Wesley Publishing Company, 1975.

[Kochan, 1988] Kochan, Stephen G. *Programming in C*, Revised Edition. Indianapolis, Indiana : Hayden Books, 1988.

[Kernighan & Plauger, 1978] Kernighan, Brian W., and P. J. Plauger. *The Elements of Programming Style*, 2nd Edition, New York: McGraw-Hill, 1978.

[Kernighan & Plauger, 1981] Kernighan, Brian W., and P. J. Plauger. *Software Tools in Pascal*, Reading, Massachusetts: Addison-Wesley Publishing Company, 1981.

[Kernighan & Ritchie, 1978] Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*, Englewood Cliffs, New Jersey: Prentice-Hall, (1978).

[Kernighan & Ritchie, 1988] Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language: ANSI C Version*, Englewood Cliffs, New Jersey: Prentice-Hall, (1988).

[Kruchten, 1995] Kruchten, Philippe. "The 4+1 View Model." *IEEE Software*, 12,6 (November 1995): 42-50.

[Lezard & Johnson. 1992] Lezard, Tony, and Paul Johnson. "London Ambulance Service Fails Again." *Forum On Risks to the Public in Computers and Related Systems*. 14,2, (November 9, 1992). (http://catless.ncl.ac.uk/risks)

[Liberty & Keogh, 1996] Liberty, Jesse and Jim Keogh, *C++: An Introduction To Programming,* Que Books, 1996.

[Lambert & Osborne, 1999] Lambert, Kenneth A. and Martin Osborne. *Java: A Framework for Programming and Problem Solving*. Boston: Brooks/Cole Publishing Company, (1999).

[Lyu, 1996] Lyu, Michael R., editor, *Handbook of Software Reliability Engineering*, Los Alamitos: IEEE Computer Society Press, (1996).

[McCabe, 1976] McCabe, Thomas J. "A Software Complexity Measure." IEEE Transactions on Software Engineering, Volume SE-2, no. 6, December 1976, pp. 308-320.

[Mills, *et al*., 1987] Mills, H. D., M. Dyer, and R. C. Linger. "Cleanroom software engineering." *IEEE Software*. 4,5 (September, 1987): 19-24.

[Meyer, 1992] Meyer, Bertrand. *Eiffel: The Langua*ge. Englewood Cliffs, New Jersey: Prentice Hall, 1992.

[Mealy, 1955] Mealy, G. H. "A method of synthesizing sequential circuits." *Bell System Technical Journal*. 34,5 (1955) 1045-1079.

[Moore, 1956] Moore, E. F. "Gedanken experiments on sequential machines." Automata Studies. Princeton, NJ: Princeton University Press. (1956): 129-153.

[Musa, *et al*., 1996] Musa, John, Gene Fuoco, Nancy Irving, and Diane Kropfl. "The Operational Profile." *Handbook of Software Reliability Engineering*, Michael R. Lyu, editor, Los Alamitos: IEEE Computer Society Press, (1996): 167-216.

[Mills, 1988] Mills, H. D. "Stepwise Refinement and Verification in Box-structured Systems." *IEEE Computer*, 21,6, (June 1988): 23-36. (Also see [Poore & Trammell, 1996 169-197].)

[Mills, *et al*., 1990] Mills, Harlan, Richard Linger, and Alan Hevner. *Principles of Information Systems Analysis and Design*. Academic Press, 1990.

[Mills, *et al*., 1987] Mills, H. D., R. C. Linger, and A. R. Hevner. "Box structured information systems." *IBM Systems Journal.* 26 (1987): 396-413. (Or see [Poore & Trammell, 1996 137-167].)

[Mills & Poore, 1988] Mills, H. D., and J. H. Poore. "Bringing software under statistical quality control*." Quality Progress*, (November, 1988): 52-55.

[Mössenböck & Koskimies, 1996] Mössenböck, H., and K. Koskimies. "Active text for structuring and understanding source code." *Software Practice & Experience,* 26,7 (July 1996): 833--850.

[Myers, 1979] Myers, Glenford. *The Art of Software Testing*, , New York: Wiley & Sons, 1979.

[Parnas, 1972] Parnas, D. L. "On the Criteria to Be Used in Decomposing Systems into Modules." Classics in Software Engineering, Edward Nash Yourdon, Editor, New York: Yourdon Press, [1997]: 141-150. (Originally published in *Communications of the ACM*, Association for Computing Machinery, [December, 1972]: 1053-58.)

[Peterson, 1995] Peterson, Ivars. *Fatal Defect: Chasing Killer Computer Bugs*. New York : Times Books (Random House), 1995.

[Press, *et al*., 1988] Press, W., B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in C*. Cambridge University Press, 1988.

[Perry & Kaiser, 1990] Perry, Dewayne and Gail Kaiser. "Adequate Testing and Object-Oriented Programming." *Journal of Object-Oriented Programming*, 2, 5 (January-February, 1990): 13-19.

[Pressman, 1997] Pressman, Roger S. *Software Engineering, A Practitioner's Approach*, 4th Edition, New York: McGraw-Hill, 1997.

[Prowell, 1996] Prowell, Stacy J. Sequence-Based Software Specification, Ph.D. dissertation, University of Tennessee, Knoxville. 1996.

[Prowell, 1997] Prowell, Stacy. "Specification Practices." *Cleanroom Software Engineering Practices.* Edited by Shirley A. Becker and James A. Whittaker. Harrisburg, PA: Idea Group Publishing (1997):13-36.

[Poore & Trammell, 1996] Poore, J. H., and C. J. Trammell, *Cleanroom Software Engineering: A Reader*. Oxford, England: Blackwell Publishers, 1996.

[Paulk, *et al*., 1993] Paulk, Mark C., Charles V. Weber, Suzanne M. Garcia, Marybeth Chrissis, and Marilyn Bush. *Key Practices of the Capability Maturity Model, Version 1.1*. Technical report CMU/SEI-93-TR-25, Pittsburgh, Pennsylvania: Software Engineering Institute, Carnegie Mellon University, 1993.

[Rumbaugh, *et al*., 1999] Rumbaugh, James, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*. Reading, Massachusetts: Addison-Wesley, 1999.

[Schildt, 1995] Schildt, Herbert, *C the Complete Reference 3/E, 3rd Edition*. Berkeley, California: Osborne McGraw-Hill, 1995.

[Smith, *et al*., 1985] Smith, Douglas, Maruice Eggen, and Richard St. Andre. *A Transition to Advanced Mathematics*, Second Edition, Monterey, California:Brooks/Cole Publishing Company, 1985.

[Sexton, 1988] Sexton, B. C. "Statistical testing of software." Master's thesis, Department of Computer Science, University of Tennessee, 1988.

[Shooman, 1983] Shooman, Martin L. *Software Engineering - Design, Reliability, and Management*, New York: McGraw-Hill, 1983.

[Shaw, 1990] Shaw, Mary. "Toward Higher Level Abstraction for Software Systems." *Data and Knowledge Engineering 5*, , New York, NY: North Holland, 1990.

[Shlaer & Mellor, 1992] Shlaer, Sally and Stephen J. Mellor. *Object Lifecycles: Modeling the World in States*. Singapore: Yourdon Press, 1992.

[Sorensen, 1995] Sorensen, R. "A Comparison of Software Development Methodologies." CrossTalk, January, 1995, pp. 12-18.

[Spivey, 1988] Spivey, J. M., *Understanding Z: A specification Language and Its Formal Semantics*. Cambridge University Press, 1988.

[Stroustrup, 1997] Stroustrup, Bjarne, *The C++ Programming Language, 3rd Edition*, Addison-Wesley, 1997.

[Stavely, 1998] Stavely, Allan M. *Toward Zero-Defect Programming*, 1st Edition, Addison-Wesley, 1998.

[Sudkamp, 1988] Sudkamp, Thomas A. *Languages and Machines, An Introduction to the theory of Computer Science*, Addison-Wesley, 1988.

[Suber, 1998] Suber, Peter. "Infinite Reflections." *St. John's Review*, XLIV, 2, Santa Fe, New Mexico:St. John's College (1998): 1-59. Originally presented as an all-college address at St. John's College in October 1996.

[Sun, 1996] Sun Microsystem Inc. *ADL Language Reference Manual*. Palo Alto, California: Sun Microsystems, 1996. (http://www.smli.com/research/adl/lrm/adl)

[Thomason, 1990] Thomason, M. G. "Generating functions for stochastic context-free grammars." *International Journal of Pattern Recognition and Artificial Intelligence*. 4, 4 (April 1990) 553-572.

[Whittaker & Jorgensen, 1999] Whittaker, James A. and Alan A. Jorgensen, "Why Software Fails." ACM Software Engineering Notes, July 1999.

[Whittaker, 1997a] Whittaker, James A. "Certification Practices." *Cleanroom Software Engineering Practices*, Edited by Shirley A. Becker and James A. Whittaker, Harrisburg PA: Idea Group Publishing, (1997) 83-115.

[Whittaker, 1997b] Whittaker, James A. "Stochastic software testing." *The Annals of Software Engineering*. *4* (1997) 115-131.

[Whittaker, 1998] Whittaker, James A. *Introduction to Software Engineering*. Melbourne, FL: SES Press, 1998.

[Whittaker & Poore, 1993] Whittaker, James A., and J. H. Poore, "Markov analysis of software specifications." *ACM Transactions on Software Engineering and Methodology*. 3,1 (1993) 93-106.

[Whittaker & Thompson, 1994] Whittaker, J. A. and M. G. Thompson. "A Markov Chain Model for Statistical Software Testing." *IEEE Transactions on Software Engineering*, 20,10 (1994) 812-824.

[Yourdon, 1989] Yourdon,Edward. *Modern Structured Analysis* Englewood Cliffs, N.J.: Prentice Hall, (1989).

APPENDICES

Appendix A – Calculator Anomalies

This appendix enumerates tests and results performed on Microsoft® Windows 95® Calculator.

Microsoft® calculator may be operated by a sequence of mouse clicks. Each of these mouse clicks may also be performed with keystrokes and the examples presented in this appendix are expressed as a sequence of keystrokes. Table 8 describes the function of each keyboard character used in these examples.

Table 8 -- Keyboard to Mouse Click Equivalence

Key	Mouse Button	Key	Mouse Button	Key	Mouse Button	Key	Mouse Button
-	-	|	Or	<Delete>	CE	h	Hyp
!	n!	~	Not	<Enter>	=	i	Inv
#	x^3	+	+	<Esc>	C	<Insert>	Dat
%	%	<	Lsh	<F2>	Deg	l	log
%	Mod	0-9	0-9	<F2>	Dword	m	dms
&	And	A-F	A-F	<F3>	Rad	n	ln
(	(	<Backkspace>	Back	<F3>	Word	o	cos
)	)	<Ctrl+A>	Ave	<F4>	Byte	p	PI
*	*	<Ctrl+D>	s	<F4>	Grad	r	1/x
. or ,	.	<Ctrl+L>	MC	<F5>	Hex	s	sin
/	/	<Ctrl+M>	MS	<F6>	Dec	t	tan
;	Int	<Ctrl+P>	M+	<F7>	Oct	v	F-E
@	sqrt	<Ctrl+R>	MR	<F8>	Bin	x	Exp
@	x^2	<Ctrl+S>	Sta	<F9>	+/-	y	x^y
^	Xor	<Ctrl+T>	Sum				

The following test examples are described as keystroke sequences and these sequences are described with a notational convention. The notation also provides a method of showing the consequences of those sequences.

Abstractions for keystroke sequences appear as follows:

<Abstraction> ::= <Keystroke Sequence>

where the greater-than and less-than symbols bracket the name of an abstraction and the symbol "::=" reads "consists of" or "is equivalent to." This statement may then be read as "An abstractions consists of a keystroke sequence."

We provide a notation for repeating keystrokes and keystroke sequences:

<Repeated Keystroke Sequence> ::= <Key>^{N} | <Abstraction>^{N}

where N is a positive integer greater than one. Hence, 9^{6} means depressing and releasing the "9" key precisely 6 times. The symbol "|" stands for logical alternation and reads "or."

Compound keystrokes, such as simultaneously depressing "Ctrl" and another key, are treated as an abstraction such as <Alt+D> or <Ctrl+x>. Named keys will also be treated as abstractions such as <F2>, <Esc>, <Backspace>, etc.

For example: <MaxE> = <Esc>9^{13}x289 means the label <MaxE> represents the keystroke sequence: Depress "Escape," depress '9' 13 times, depress 'x', then '2', then '8', and then '9'. We have now defined an abstraction, <MaxE>, which represents the maximum value that may be entered into the calculator. As an additional example, <Invoke Scientific> = <Invoke><Alt+V>s represents starting the calculator, depressing the "Alt" and 'V' keys simultaneously (selecting the "view" dialog) and then depressing 's' (selecting scientific view). "Invoke" is

a special abstraction representing the initiation of the calculator program. This provides us with an abstraction for starting the calculator and ensuring that it is in the scientific mode of operation.

One more addition to this notational convenience:

<Sequence of Keystrokes> ➜ <Result>

Where < Result > is the information displayed in the calculator result window or some other result. When placed in quotation marks, the result is the information display in the calculator results window. The symbol "➜" reads "results in" or "produces."

Following are fifteen (15) sequences that produce anomalies. Note that in each of the following examples, it is possible to save values such as <MaxD> in memory. Memory recall will replace the results indicated for the abstraction without having to reproduce the entire sequence described. The full sequence provides rigor to the description. These sequences are not unique for the results obtained.

1. Maximum Entry <> Maximum Display

We discover that the maximum value that can be entered can be multiplied by 10^5 and therefore the maximum value that can be computed and displayed is larger than the maximum value that can be entered.

<Invoke Scientific><MaxE>*10^5= ➜ 9.999999999999e+306

The following sequence defines the maximum value that may be displayed:

<MaxD> = 1797693134862x289=r/10^{7}=r

<Invoke Scientific><MaxD> ➔ 1.797693134862e+308

2. Minimum Fixed Point Entry <> Minimum Fixed Point Displayed

In this example, the minimum fixed point value that can be entered is larger than the minimum fixed point value that can be computed.

<MinFE> = .0^{11}1 (The minimum fixed point value that my be entered)

< Invoke Scientific><MinFE>= ➔ 0.000000000001

< Invoke Scientific><MinFE>=/1000= ➔ 0.000000000000001

3. Fixed Point Copy Pastes Incorrectly

Both of the prior examples place the calculator in an anomalous state and an anomaly may occur after that. In this example, we find that we cannot paste back into the calculator a value computed by the calculatorF, the small fixed point number computed above.

< Invoke Scientific><MinFE>=/1000=<Alt+e>c<Alt+e>p

➔ 0.000000000000

4. Maximum Display not Divisible by 2

In this example, a number can be computed that cannot be divided by two. Since any number divided by two results in a number of smaller magnitude, the expectation is that the computation is possible.

<Invoke Scientific><MaxD>/2= ➔ “Result is too large.”

5. Valid Sum or Product Invalid

In a similar manner, it should be possible to multiply any number by one.

<Invoke Scientific><MaxD>*1= ➔ “Result is too large.”

Adding one a very large number should result in no change, since the value one is out of the precision range of a very large number.

<Invoke Scientific><MaxD>+1= ➔ “Result is too large.”

6. Negative Overflow

This result might be considered valid, however, the calculator is ill prepared to deal with “-1.#INF.” Clearly, the developers have not anticipated that result would produce a floating point overflow.

<Invoke Scientific><MaxD>-=^{3} ➔ -1.#INF

7. Float Copy Pastes Incorrectly

Since the maximum computable value is larger that the maximum value that may be entered, we would expect that we cannot copy and re-paste this value; however, paste is allowed but the result is incorrect.

< Invoke Scientific><MaxD>=<Alt+e>c<Alt+e>p ➔ 1.797693134862e+030

The result should be 1.797693134862e+308.

8. Load from Statistics Box Not Displayed

A value sent to the statistics box (STA) is incorrectly displayed when reloaded to the calculator.

< Invoke Scientific><MaxD><Ctrl+s>r<Insert><Esc><Ctrl+s>l ➔ 0

The correct result is stored in the calculator input. It is the display of the currently entered value (from the statistics box) that is incorrect.

9. Trig functions of Maximum Display Invalid

The sine and cosine functions should always return a value between –1.0 and +1.0. The tangent function might return "INF" but NANQ is unexpected and is not anticipated in further calculator processing.

<Invoke Scientific><F2><MaxD>s ➔ 3.137566414384e+306

<Invoke Scientific><F2><MaxD>o ➔ 3.137566414384e+306

<Invoke Scientific><F2><MaxD>t ➔ "1.#QNAN"

10. Trig functions Inaccurate

An error is introduced by even small multiples of 90 degrees.

The cosine of 990 degrees:

<Invoke Scientific><F2>990o ➔ 4.655537023598e-15

The tangent of 990 degrees:

<Invoke Scientific><F2>990t ➔ -2.147979910655e+14

The expected result in each case is zero.

11. Squaring Square Root Invalid

A value exists such that the square of the square root may not be calculated.

The square root of the maximum displayable value may be computed but this number may not be squared to arrive at the original value.

< Invoke Scientific><MaxD>i@@ ➔ "Result is too large."

Similarly for cube root and cube functions:

< Invoke Scientific><MaxD>i## ➔ "Result is too large."

And for natural log:

< Invoke Scientific><MaxD>inn ➔ "Result is too large."

But interestingly enough, however, for based ten log:

< Invoke Scientific><MaxD>ill ➔ 1.797693134862e+308

This gives us another sequence for <MaxD>:

308.2547155599il

This sequence is not equivalent to <MaxD>, however, because:

< Invoke Scientific><MaxD>-308.2547155599il= ➔ 6.91e+297

12. F-E Erases Display

Changing the display mode from fixed point to exponential in the midst of a calculation erases the current computation.

<Invoke Scientific>666vv ➔ 666.

However:

<Invoke Scientific>666v*v ➔ 0.

13. F-E Exponentiates Invalid Strings

Defining some additional abstractions plus and minus infinity and for not a number:

<-Inf> = <MaxD>-=[3]

<Inf> = <-Inf><F9>

<Nan> = <MaxD>t

The calculator treats these representations of invalid numbers as numbers. Clicking on the F-E button to change display modes causes these unusual display values to have exponents appended in the display.

<Invoke Scientific><Inf>v ➔ "1.#INFe+0"

<Invoke Scientific><Nan>v ➔ "1.#QNANe+0"

14. Factorial Function Accepts Invalid Input

Entering a real (non-integer) number should be invalid for the factorial function.

<Invoke Scientific>1.0[12]1! ➔ "Result is too large."

15. Dat Clears Invert Mode

The invert function precondition flag should not be reset by operations that do not perform functions. The DAT function (transferring data to the statistics box) clears the inverse function precondition flag.

<Invoke Scientific>i ➔ <Invert Box is Checked>

< Invoke Scientific>i<Ctrl+s><Insert> ➔ <Invert Box is not Checked>

Appendix B – Defects in Production Applications

Students from the Spring 1999 class in Software Testing Methods discovered the defects shown in Table 9 in production applications with a minimum of training in software testing but including the classifications of software defects presented in this dissertation.

Table 9 -- Production Software Defects

Report number	Severity	Type	Reported by
1	Crash WordPad NT	Input array overflow	Adam Duccini
2	Crashes Excel WinNT	Input constraint	Luis Rivera
3	Wrong output in Word 97	Memory?	Rahul Chaturvedi
4	Close a file without saving, Word7 Win95	Stored Data	Kay Michel
5	Improperly stored data constraint in Word 97 WinNT	unhandled input	Pi-Yu Lee
6	Wrong result in Money 97 WinNT	Computation constraint	Cibel Castillo
7	Incorrect Error Message in MS Paint Win98	unexpected result	Rahul Chaturvedi
8	Crashes Netscape 3.0	the browser stops responding	Rahul Chaturvedi
9	Crashes Internet Explorer 3.0	the browser stops responding	Rahul Chaturvedi
10	Crashes Microsoft 95/98	system freezes	Rahul Chaturvedi

Report number	Severity	Type	Reported by
11	Win NT takes an exception while starting Lotus Bean Machine	takes an exception	Steven Atkins
12	Unexpected result in WinTeX95 V2.01 and V2.02	unexpected result	Mazin Al-Shuaili
13	Unexpected result in WinTeX95 V2.01 and V2.02	crashes the program	Mazin Al-Shuaili
14	Unexpected result in WinTeX95 V2.01 and V2.02	crashes the program	Mazin Al-Shuaili
15	Unexpected result in WinTeX95 V2.01 and V2.02	unexpected result	Mazin Al-Shuaili
16	Crash Word 97/Win 95/98/NT	Stored Data	Florence Mottay
17	Bad result in Win CE	Rounding Error	Florence Mottay
18	Bad result in Win CE	Input Constraint	Florence Mottay
19	Different compiling errors with Borland Turbo C++ 3.0 and Visual C++ 6.0. In Borland: Crash	input/stored data/ computation constraints	Arun Chitrapu
20	Windows 95 failed after install and uninstall Win98	?	Sharma Vanterpool
21	Disable MS Word to work properly until reboot of the machine	memory overflow	Giovanna Scaffidi
22	WS_FTP V 951229 Non existent file transferred	stored data constraint	Keyur Shah
23	MS-Access. Closes the application	stored data constraint	Roby Mathew

Report number	Severity	Type	Reported by
24	Newtek's Lightwave 3D 5.5 Modeler. Numbers converted incorrectly	computation or/and stored data constraint	Luke Nowak
25	Newtek's Lightwave 3D 5.5 Modeler. Program crashes	Input constraint	Luke Nowak
26	Install Office 97, Visual C++ v. 5.0 are inaccessible	?	Kay Michel

Appendix C – Defects in Software Development Texts

This appendix contains the examples of coding defects found by students in the spring, 1999 Software Testing Methods class. Each entry identifies 1) the text, 2) the student name identifying the defect, 3) the code questioned by the student, and 4) the defect classification.

1. Text: *The C Programming Language: ANSI C Version,* Kernighan and Ritchie, [Kernighan & Ritchie, 1988, 62]

   Student: Steven Atkins

   Code: `void shellsort(int v[], int n)`

   Problem: Unconstrained input

2. Text: *C++ How to Program, 2nd Edition,* Deitel and Deitel, [Deitel & Deitel, 1994, 62],

   Student: Pi-Yu Lee

   Code:
   ```
   cin >> integer1
   cin >> integer2
   sum  = integer1 + integer2
   ```

   Problem: Unconstrained computation

3. Text: *Programming and Problem Solving with Ada*, Dale, Weems and McCormick, 1997, [Dale, *et al*., 1997, 361]

   Student: Cibel Castillo

   Code:
   ```
   Upper_Count : in out Natural
   Upper_Count = UpperCount + 1
   ```

   Problem: Unconstrained stored data

4. Text: *C++ How to Program, 2nd Edition,* Deitel and Deitel, [Deitel & Deitel, 1994, 438],

   Student: Keyur Shah

   Code:
   ```
   cin>>phone
   ```

   Problem: Unconstrained input (the format of the phone number is not checked. If you enter more than 10 digits, the program crashes)

5. Text: *Numerical Recipes in C*. Press, Flannery, Teukolsky, and Vetterling, 1988 [Press, *et al*., 1988].

   Student: Mazin Al-Shuaili

   Code:
   ```
   fv = (float) v[mn];
   Fu = 1.0/sqrt(fv);
   ```

   Problem: Unconstrained input

6. Text: *Teach Yourself C Programming in 21 Days*, Aitken and Jones. [Aitken & Jones, 1995, 71]

   Student: Arun Chitrapu

   Code:
   ```
   cin>>firstNumber
   cin >>secondNumber
   ```

   Problem: Unconstrained input (what if firstNumber is greater than the maximum allowed size for an integer?)

7. Text: *C the Complete Reference 3/E, 3rd Edition*. Schildt, [Schildt, 1995, 206]

   Student: Brian Shirey

   Code:
   ```
   char *p;
   printf("Enter an address:  ");
   scanf("%p", &p);
   ```

   Problem: Unconstrained input (if address is 0000:0000, it will crash)

8. Text: *Structured Fortran 77 for Engineers and Scientists, 4th Edition.* Etter,1993, [Etter, 1993, 64].

   Student: Jeremy Babb

   Code:
   ```
   READ*, CARBON
   AGE = (-LOG(CARBON))/0.0001216
   ```

   Problem: Unconstrained input (what if carbon is 0)

9. Text: *The C++ Programming Language, 3rd Edition*, Stroustrup [Stroustrup, 1997, 50]

   Student: John Grant

   Code:
   ```
   float x;
   cout <<"Enter length";
   ```

```
cin >> length;
```

Problem: Unconstrained input (what if we enter length = 0)

10. Text: *Programming in C, Revised Edition,* Kochan, 1988 [Kochan, 1988, 48].

    Student: Adam Duccini

    Code:
    ```
    int triangular_number
    For (n=1; n<= number; n++)
        triangular_number = triangular_number + n;
    ```

    Problem: Unconstrained computation

11. Text: *A Structured Approach to FORTRAN,* Crawley and Miller, 1983, [Crawley & Miller, 1983, 220]

    Student name: Roby Mathew

    Code:
    ```
    loop
    Total = total + age
    ```

    Problem: Unconstrained computation

12. Text: *Programming and Problem Solving with Ada*, Dale, Weems and McCormick, 1997, [Dale, *et al*., 1997, 655]

    Student: Luke Nowak

    Code:
    ```
    for Column in 1..Column_Length loop
        Total := 0;
        For Row in 1..Row_Length loop
            Total := Total + Total(Row,Column);
        end loop;
    end loop;
    ```

    Problem: Unconstrained computation

13. Text: *C++: An Introduction To Programming,* Liberty and Keogh, 1996

    [Liberty & Keogh, 1996,207].

    Student: Kay Michel

    Code: `Rectangle* pRect = new Rectangle;`

    Problem: Unconstrained input (set the pointer to NULL)

Appendix D – Partitions that Create Operational Modes

The main text provides some examples of the derivation of operational modes from constraints. This appendix provides additional examples of determining operational modes from natural and artificial partitions of persistent storage values imposed by input, output, other storage, and other computations.

For the following examples that x and y are16-bit, twos-complement integers. These examples all may be generalized by considering that x and y are members of the set, N, of twos-complement numbers (which, by definition, are always bounded), where $N = \{n \mid Min \le n \le Max$, Min is the minimum value represented and Max is the maximum value represented$\}$.

Operational Mode for Integer Division

As an additional example, consider the integer computation x / y. We assign to this computation the operational mode $Domain(Quotient) = \{Valid, Invalid\}$. Clearly, the operational mode value must be determined from X and Y. However, $Quotient$ may be computed from the operational modes of X and Y. Note that the operational mode values for X and Y have been determined for the constraints on the division operation only. Other operational mode values may also be introduced

by partitions of *X* and *Y* introduced by constraints on inputs, outputs, storage, and other computations involving *X* or *Y*.

Domain(OpMode Numerator) = {*Numerator.Nominal, Min*},

Domain(OpMode Denominator) =

{*Zero, Minus One, Denominator.Nominal*}.

Where:

Min = {-32768} = {$x \mid x = -32768$}

Numerator.Nominal = {-32767..32767}

= {$x \mid \{-32767\ x \leq 32767\}$

Zero = {0} = {$y \mid y = 0$}

Minus One = {-1} = {$y \mid y = -1$}

Denominator.Nominal = {-32767..-2,1..32767}

= {$y \mid \{-32767 \leq y \leq -2$ or $1 \leq y \leq 32767\}$

The complete cross product of the *X* and *Y* operational modes is:

(OpMode Numerator x *OpMode Denominator*) = *Quotient* =

{ (*Numerator.Nominal, Zero*),

(*Numerator.Nominal, Minus One*),

(*Numerator.Nominal, Denominator.Nominal*),

(*Min, Zero*),

(*Min, Minus One*),

(*Min, Denominator.Nominal*) }

Relating these states to the values of *OpMode Quotient*,

Valid = { (*Numerator.Nominal, Minus One*),

(*Numerator.Nominal, Denominator.Nominal*),

(*Min, Denominator.Nominal*) }

Invalid = { (*Numerator.Nominal, Zero*),

(*Min, Zero*),

(*Min, Minus One*) }

Note that the values of *OpMode Quotient* determine the partitions and hence the operational mode values of *X* and *Y*. Thus, the constraints on the computation imply constraint partitions for the operational mode values of the operands.

Dynamic Partitions, Integer Addition Operational Mode

The limiting (or constraining) values as illustrated by the previous example need not always be constant. Consider another integer computation, $x + y$. We can assign to this computation, the operational mode *Domain(OpMode SUM)* = {*Sum.Valid, Sum.Invalid*}. Clearly, the operational mode value must be determined from *X* and *Y*. However, *OpMode SUM* cannot be computed from the operational modes of *X* and *Y* independent of the specific values of x and y:

Domain(X.Mode) = {*X.Too Small, X.Nominal, X.Too Big*},

Domain(Y.Mode = { *Y.Too Small, Y.Nominal, Y.Too Big*}

X.Too Small = $\{x \mid x < \text{-}32768 - y\}$

X.Too Big = $\{x \mid x > 32767 - y\}$

X.Nominal = $\{x \mid \text{-}32768 - y \le x \le 32767 - y\}$

Y.Too Small = $\{y \mid y < \text{-}32768 - x\}$

Y.Too Big = $\{y \mid y > 32767 - x\}$

Y.Nominal = $\{y \mid \text{-}32768 - x \le y \le 32767 - x\}$

Now the operational mode values for *SUM* can be calculated easily as

SUM.Valid = *X.Mode.Nominal* ∪ *Y.Mode.Nominal*

SUM.Invalid = *X.Mode.Too Small* ∪ *X.Mode.Too Big*
∪ *Y.Mode.Too Small* ∪ *Y.Mode.Too Big*

Assignment Operational Mode

There is another, even simpler limiting case:

$x := y$ (assignment) ($x \in X, y \in Y; X = Y = N = \{n \mid -32768 \leq n \leq 32767\}$)

In this example, by implication, the constraints on x are necessarily constraints on y; that is to say, the operational modes of x and y have the same values and the same constraining partitions because of the assignment. We also introduce artificial constraints imposed by arbitrary requirements. For example, suppose that prior to the assignment we have operational modes:

$Domain(OP.X) = \{X.Valid, X.Invalid\}$ and

$Domain(OP.Y) = \{ Y.Valid, Y.Invalid\}$,

where $OP.X.Valid = \{x \mid 0 \leq x \leq 200\}$, (0 and 200 are arbitrary requirements costraints) and

$X.Invalid = \{x \mid x < 0$ or $x > 200)$ and

$Y.Valid = \{y \mid 1 \leq y \leq 400\}$ and

$Y.Invalid = \{y \mid y < 1$ or $y > 400)$.

After the assignment, there is a new set of operational mode values imposed:

$Domain(OP.X) = \{X.Y.Valid, X.X.Invalid, X.Y.Invalid, X.XY.Invalid\}$

where $X.XY.Valid = \{x \mid 1 \leq x \leq 200\}$,

$X.X.Invalid = \{x \mid 200 < x \leq 400\}$,

$X.Y.Invalid = \{x \mid x = 0\}$,

$X.XY.Invalid = \{x \mid x < 0$ or $x > 400\}$,

and

$Domain(OP.Y) = \{Y.XY.Valid, Y.X.Invalid, Y.Y.Invalid, Y.XY.Invalid\}$

where *Y.XY.Valid* = $\{y \mid 1 \le y \le 200\}$,

Y.X.Invalid = $\{y \mid 200 < y \le 400\}$,

Y.Y.Invalid = $\{y \mid y = 0\}$,

Y.XY.Invalid = $\{y \mid y < 0 \text{ or } y > 400\}$.

If this assignment statement must always be executed, y inherits the constraints on the memory location x and the operational mode *OP.Y* can be redifined as:

Domain(OP.Y) = { *XY.Valid, Y.Invalid* }

where *XY.Valid* = $\{y \mid 1 \le y \le 200\}$,

Y.Invalid = $\{y \mid y < 0 \text{ or } y > 200\}$.

Input and Output Operational Modes

Limitation on input or output can also partition storage values. One constraint may be the number of digits that may exist in certain input (or output) fields. For example, consider Input(x), where the limiting field width is four characters including a sign character. Then the range of values that may be entered is –999 to 9999 and we can define an operational mode for the input value, x:

Domain(OpMode INPUT) = {*Valid, Invalid*}

Where *Valid* = $\{x \mid -999 \le x \le 9999\}$

Invalid = $\{x \mid x < -999 \text{ or } 9999 < x\}$

The calculator copy/paste result described in Appendix A is a result of having different operational modes for the display value, the input value and the output value. The partition that is the upper bound for input is less than the upper bound for the partition for output. Consequently there is a range of values that can

be displayed (output) that cannot be input and the copy/paste function will not work as expected for all values of output.

Appendix E – Running Average Program

```
/*       This program was reformatted by prettyc     */
#define D EBUG

/*

    Running Average System

    Alan A. Jorgensen
    Wed Jun 16 12:14:42 EDT 1999

*/

#include <stdio.h>

/*   System Configuration Parameters */

#define MaxValues 2000
#define MaxEntry 1000000
#define MaxInt 0x7FFFFFFF
#define LineLength 80

/* Macro to set error message code */
```

```
#define SetError(Message) seterror(Message)

/* Macro to display the error message and exit process with error code. */

#define ErrorExit errorexit()

/* Macro to return an error message to caller */

#define Return(ErrorMessage) {ErrorCode = ErrorMessage; return;}

/* Macro to test for existence of error */

12345678901234567890123456789012345678901234567890123456789012345678901234567890
#define Error (ErrorCode != NULL)

/* Macro to display error message and reset error code. */

#define DisplayError displayerror()

/* LineLength, though 80 characters, can only accomodate 79 characters without
line wrap.

To constrain output to no more that 79 characters, for configuable line length we
need a dynamic format string as follows.
*/

char LineLengthFormat[35];
SetFormat()
```

```
    {
    sprintf(LineLengthFormat, “%%.%ds\n”, LineLength - 1);
#ifdef DEBUG
    printf(“%s”, LineLengthFormat);
#endif
    }

#if (MaxEntry + (MaxValues / 2)) > ( MaxInt / MaxValues )
X CONFIGURATION_ERROR - MaxValues * MaxEntry + MaxValues / 2 must be < MaxInt
#endif

/* Output component

This component displays all messages to the operator and constrains output to the
required display area.  */

Output(FILE *Destination, char Message[])
{   /* Output message but truncate to allowed
       length*/
    fprintf(Destination, LineLengthFormat, Message);

    /*
    ** Fatal error if Message is too long
    */
    if (strnlen(Message, LineLength) > LineLength - 1)
        {
        SetError(“Output Message Length Overrun.”);
        ErrorExit;
```

```
        }
    }


/* Global Error System */

char * ErrorCode = NULL;

/* Since C does not provide a strnlen function and strlen can address out of
bounds with an improperly terminated string, we provide a strnlen function that
allows bounding the string length search. */

int strnlen(char s[], int l)
    {
    int c;
    for (c = 0; c < l; c++)
        if (s[c] == 0)
            return c;
    return l;
    }


/* Definition of path for error messages */

#define ErrorDevice stdout
#define DisplayDevice stdout
```

```
/* Procedure to display the error message and exit
   process with error code. */

errorexit()
    {   /*  We do not use “Output” or “DisplayError”
            here because they call error  exit on an
            error. */
    fprintf(ErrorDevice, LineLengthFormat, ErrorCode);
    fprintf(ErrorDevice, “Fatal Error.  Program \
Terminates.\n”);
    exit(1);
    }


/* Procedure to set error message code */

seterror(char * Message)
    {
    ErrorCode = Message;
    }
```

```
/* Procedure to display error message and reset error code. */

displayerror()
    {
    Output(ErrorDevice, ErrorCode);
    ErrorCode = NULL;
    }


/* Definition of Average History */

int Values = 0;  /* Number of valid values entered. */
int Sum = 0;  /* Sum of the valid values entered. */

/* Procedure to set the history to known values.  Error
   if values not zero. */

SetHistory(int values, int sum)
    {
    if (Sum != 0)
        Return(“History Already Initialized.”);
    if (Values != 0)
        Return(“History Already Initialized.”);
    Values = values;
    Sum = sum;
    }
```

```
/* Procedure to perform normal initialization.  History
   (count of values and sum of values) are set to zero.
   The system welcome message is displayed. */

Initialize()
    {
    SetHistory(0, 0);
    Output(DisplayDevice, “Welcome to the Running Average program.”);
    }


/* Definition of unique value produced by the procedure
   “Input” to indicate that program termination has been
   requested. */

#define Exit (-1)

/* Procedure to read input characters and form them into
   valid integer numbers.  Only positive numbers are
   allowed.  Any other string will result in an error
   message being returned to the user. */

Input(int * Value)
    {
    int Number = 0;
    char Digit;
    Digit = getc(stdin);
```

```
if (Digit == '\n')
    {
    *Value = Exit;
    return;
    }
while (Digit != '\n')
    {
    if ((Digit < '0')
    ||(Digit > '9'))
        {
        if (!Error)
            {
            SetError(“Invalid Input.”);
            Number = 0;
            }
        }
    else if (!Error)
        {
        if (Number > MaxEntry / 10)
            {
            SetError(“Number entered is too large.”);
            }

        /*
           The control structure used here is
           necessary if the order of evaluation of
           logical expressions is not specified.
        */
```

```
            else if (Digit - '0' > MaxEntry - 10 *
                     Number)
                {
                SetError("Number entered is too \
large.");
                }
            else   /* The current value can accommodate
                      the next new digit value */

            /* The order of the following computation is
               important if overflow is to be avoided.
               If we add Digit and then subtract the
               representation of '0', we might cause a
               spurious overflow, though the computation
               would correct itself, generating another
               overflow, when we subtract '0'.
            */
                Number = 10 *Number - '0' + Digit;
            }
        Digit = getc(stdin);
        }
    *Value = Number;
    return;
    }


/* Procedure to compute the average of the valid values
   submitted. Boundary values of operational modes are
```

```
   checked to ensure that program does not reach an
   invalid state.  Error messages indicate unacceptable
   input.   System capacity errors are catestrophic and
   the program terminates gracefully.  On recieving the
   terminating input the exit message is displayed and
   the program terminates.  */

Average()
    {
    int Number = 0;
    if (Error)
        ErrorExit;
    while (Number != Exit)
        {
        Input(&Number);
        if (Error)
            DisplayError;
        else if (Number != Exit)
            {
#ifdef DEBUG
            printf("You entered %d\n", Number);
#endif
            if (Number < 0)
                SetError("Number entered is less than zero.");
            else if (Number > MaxInt - Sum)
                {
                SetError("Sum is too large for this value.");
                ErrorExit;
```

```
                }
            else if (Values > MaxValues - 1)
                {
                SetError(“Too many values have been received.”);
                ErrorExit;
                }
            else
                {
                Values = Values + 1;
                Sum = Sum +(long) Number;
                }
            if (Error)
                DisplayError;
            else
                {
#ifdef DEBUG
                printf(“%d values have been entered.\n”, Values);
#endif
                    {
                    char RunningAverageReport[80];
                    sprintf(RunningAverageReport,
                        “The current average is %d.”,((Sum +(Values / 2))
Values));
                    Output(DisplayDevice, RunningAverageReport);
                    }
                }
            }
        }
```

```
Output(DisplayDevice, “Thank you for using Running Average.”);
}
```

```
main(int Params, char * Parameter[])
    {
    int InitialCount;
    int InitialSum;

    /* Build the format message required for output
       constraint.
    */
    SetFormat();

    /* The following code is for test purposes only.
       Normally the program is used with no command line
       parameters; however, if the password “test” is as
       the first command line parameter, the next two
       parameters are values to initialize the count and
       sum. This avoids the necessity of using large
       input streams to test boundary conditions. A
       second password, “trial” tests the prohibition
       against multiple initialization, “long” tests the
       ability to detect excessively long error
       messages.
    */
    if (Params != 1)
        {
        if (!strcmp(Parameter[1], “test”))
            {
            Initialize();
```

```
            sscanf(Parameter[2], “%d”, &InitialCount);
            sscanf(Parameter[3], “%d”, &InitialSum);
            SetHistory(InitialCount, InitialSum);
            }
        else if (!strcmp(Parameter[1], “trial”))
            {
            sscanf(Parameter[2], “%d”, &InitialCount);
            sscanf(Parameter[3], “%d”, &InitialSum);
            SetHistory(InitialCount, InitialSum);
            Initialize();
            }
        else if (!strcmp(Parameter[1], “long”))
            {   /* Create an excessively long error
                   message */
            SetError(
“12345678901234567890123456789012345678901234567890\
12345678901234567890123456789012345678901234567890”
                );
            DisplayError;
            }
        else
            {
            SetError(“Invalid Test Command.”);
            ErrorExit;
            }
        }
    else
```

```
    /*
    ** End Test Code
    */
Initialize();
Average();
exit(0);
}
```

Appendix F – Running Average Program Test

The test for the running average program is a Korn Shell script that tests each of the constraints defined in the final data dictionary. Each test heading refers to the data dictionary entry number that is being tested (DD n). Test output is saved in a file, "ver.2" and compared against the file, "ver" which is the set of correct answers verified manually.

```
# Verification -- Test Script to verify the running
average program.

function Divider
    {
    echo
"_____________________________________________________
_________"
    }

function Log
    {
    Divider
    echo
    echo Test $Test
    echo $1 $2 $3 $4 $5 $6 $7 $8 $9
    echo
    Divider
    let Test+=1
    }

function Run
    {
Divider
echo "Test Script to verify the running average
program."
echo $(date)
Divider
echo
```

```
Test=1

Log DD 1, 4-6 Verify Startup and Exit

RunAvg <<.

.
echo Exit Code $?

Log DD 2, 3 Verify Message Length

RunAvg long <<.

.
echo Exit Code $?

Log DD 6, Verify Valid Input Integers

RunAvg <<.
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000001
0
1000000

.
echo Exit Code $?

Log DD 7, Range of Input Values

RunAvg <<.
-1
```

```
-0
0
1000000
1000001

.
echo Exit Code $?

Log DD 8, 9 Invalid Integer Inputs

RunAvg <<.

Blank Line
\0
-0
100000O
1000001

.
echo Exit Code $?

Log DD 10, Average of Maximum Input

RunAvg <<.
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
```

1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000
1000000

```
1000000
1000000
1000000
1000000

.
echo Exit Code $?

Log DD 10, Running Average

RunAvg <<.
0
2
4
10
24
56
128
288
640
1408
3072
6656
14336
30720
65536
139264
294912
622592
1310720
5767168

.
echo Exit Code $?

Log DD 10, Rounding, Zero Result

RunAvg <<.
0
0
1
0
1
0
1
```

```
0
1
0
0
1
1
0
0
1
1
0
0
0
1
1
1

.
echo Exit Code $?

Log DD 10, Rounding, Result 1

RunAvg <<.
1
0
1
0
1
0
1
0
1
0
1
1
0
0
1
1
1
0
0
0

.
```

```
echo Exit Code $?

Log DD 10, Average with a large number of inputs.
RunAvg test 1998 199800000 <<.
100999
100999
100999

.
echo Exit Code $?

Log DD 11, System Error Message

RunAvg test 1999 21474836470<<.
1
0

.
echo Exit Code $?

Log Reinitialize

RunAvg trial 1999 2000000000 <<.
0
1000000

.
echo Exit Code $?

Log Reinitialize with too many values

RunAvg trial 1999 0<<.
0
1

.
echo Exit Code $?

Log Reinitialize with New Sum too big

RunAvg trial 0 21474836470<<.
0
1

.
```

```
echo Exit Code $?

Log Initialize with Sum too big

RunAvg test 0 21474836470<<.
0
1

.
echo Exit Code $?

Divider
    }

Run >ver.2
diff ver ver.2 | sed “s/^< //” | sed “s/^> //”
```

Following is the set of correct answers for this test.

```
_____________________________________________________________________
_______
Test Script to verify the running average program.
Sat Oct 23 13:15:26 EDT 1999
_____________________________________________________________________
_______

_____________________________________________________________________
_______

Test 1
DD 1, 4-6 Verify Startup and Exit

_____________________________________________________________________
_______
Welcome to the Running Average program.
Thank you for using Running Average.
Exit Code 0
_____________________________________________________________________
_______

Test 2
DD 2, 3 Verify Message Length
```

```
_____________________________________________________________________
_______
12345678901234567890123456789012345678901234567890123456
7890123456789012345678 9
Output Message Length Overrun.
Fatal Error.  Program Terminates.
Exit Code 1
_____________________________________________________________________
_______

Test 3
DD 6, Verify Valid Input Integers

_____________________________________________________________________
_______
Welcome to the Running Average program.
The current average is 1.
The current average is 1.
The current average is 333334.
Thank you for using Running Average.
Exit Code 0
_____________________________________________________________________
_______

Test 4
DD 7, Range of Input Values

_____________________________________________________________________
_______
Welcome to the Running Average program.
Invalid Input.
Invalid Input.
The current average is 0.
The current average is 500000.
Number entered is too large.
Thank you for using Running Average.
Exit Code 0
_____________________________________________________________________
_______

Test 5
DD 8, 9 Invalid Integer Inputs
```

```
Welcome to the Running Average program.
Invalid Input.
Invalid Input.
Invalid Input.
Invalid Input.
Invalid Input.
Number entered is too large.
Thank you for using Running Average.
Exit Code 0
```

Test 6
DD 10, Average of Maximum Input

```
Welcome to the Running Average program.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
```

```
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
The current average is 1000000.
```

```
Thank you for using Running Average.
Exit Code 0
```

Test 7
DD 10, Running Average

```
Welcome to the Running Average program.
The current average is 0.
The current average is 1.
The current average is 2.
The current average is 4.
The current average is 8.
The current average is 16.
The current average is 32.
The current average is 64.
The current average is 128.
The current average is 256.
The current average is 512.
The current average is 1024.
The current average is 2048.
The current average is 4096.
The current average is 8192.
The current average is 16384.
The current average is 32768.
The current average is 65536.
Number entered is too large.
Number entered is too large.
Thank you for using Running Average.
Exit Code 0
```

Test 8
DD 10, Rounding, Zero Result

```
Welcome to the Running Average program.
The current average is 0.
The current average is 0.
The current average is 0.
```

```
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
The current average is 0.
Thank you for using Running Average.
Exit Code 0
```

Test 9
DD 10, Rounding, Result 1

```
Welcome to the Running Average program.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
```

```
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
The current average is 1.
Thank you for using Running Average.
Exit Code 0
```

Test 10
DD 10, Average with a large number of inputs.

```
Welcome to the Running Average program.
The current average is 100000.
The current average is 100001.
Too many values have been received.
Fatal Error.  Program Terminates.
Exit Code 1
```

Test 11
DD 11, System Error Message

```
Welcome to the Running Average program.
Sum is too large for this value.
Fatal Error.  Program Terminates.
Exit Code 1
```

Test 12
Reinitialize

```
Welcome to the Running Average program.
History Already Initialized.
Fatal Error.  Program Terminates.
```

```
Exit Code 1
```

```
Test 13
Reinitialize with too many values
```

```
Welcome to the Running Average program.
History Already Initialized.
Fatal Error.  Program Terminates.
Exit Code 1
```

```
Test 14
Reinitialize with New Sum too big
```

```
Welcome to the Running Average program.
History Already Initialized.
Fatal Error.  Program Terminates.
Exit Code 1
```

```
Test 15
Initialize with Sum too big
```

```
Welcome to the Running Average program.
Sum is too large for this value.
Fatal Error.  Program Terminates.
Exit Code 1
```

Appendix G – Running Average Program Test Verification

The running average test program was verified by generating permuted versions of the running average program and rerunning to test program to ensure that the defect injected into the permuted program was detected. Permutation was accomplished by locating numeric strings in the source and replacing them with values one larger and one smaller in sucessive permutations of the program source. The was accomplished with the following parsing program designed to emit a list of numeric string positions in the text when no input parameters are supplied or to emit a permuted copy of the input when a line number, character position, and replacement value are supplied as parameters. This parsing program was generated using the Useful Self Replicating Program (USRP). The source file for the parser is included as well as the generated parsing C code program.

USRP source:

```
(#include “Header.h”)
(int number;)
(int pos, ln;)
(int Pos, Ln, Val;)
(char NumberString[1000/];)
(main(int parms, char * Parm[]/))
({ sscanf(Parm[1],”%d”,&Ln/);)
(  sscanf(Parm[2],”%d”,&Pos/);)
(  sscanf(Parm[3],”%d”,&Val/);)
(printf(“//* Line %d Character Position %d Replaced with
%d. *//\n”,)
(Ln,Pos,Val/);)
(<File>)
(/})
```

```
<P> (if (Ln != 0/) printf(“%s”,MatchStr/);)

<Spacing> ::= <Space> | <Tab>

<White Space> ::= <Space> | <Tab> | <Line Break>

<Spaces> ::= { <White Space> }

<CM> (strcat(NumberString, MatchStr/);)

<Number> ::=
    (pos = cp; ln = LineNo; NumberString[0/] = 0;)
    <Digit> (number = MatchStr[0] - '0';) (<CM>)
    {<Digit> (number = 10*number + MatchStr[0] - '0';)
(<CM>) }

<Replace> (return Ln;)

<Empty>

<Save or Replace Number> ::= <Replace>
    (if ((Ln == ln/) && (Pos == pos/)/))
     (printf(“%d”,Val/);)
    (else printf(“%s”,NumberString/);)
    | <Empty> (printf(“Line %d Character %d =
%d\n”,ln,pos,number/);)

<Comment Character> ::= (if ((Line[cp] == '*'/) &&
(Line[cp+1] == '//'/)/) )
    (return FALSE;)
    <Any Character> (<P>)

<Comment> ::= //* (<P>) {<Comment Character>}

<Number or Character> ::=
    (if (EndFile/) return FALSE;)
    <Comment> *// (<P>)
    | <Number> <Save or Replace Number>
    | <Any Character> (<P>)

<Program> ::= <Number or Character> {<Number or
Character>}
```

```
<File> ::= (<Read Line>) [<Spaces>] <Program> <End of
File>
```

The C program:

```
/*       This program was reformatted by prettyc     */
#include “Header.h”
int number;
int pos, ln;
int Pos, Ln, Val;
char NumberString[1000];
main(int parms, char * Parm[])
    {
    sscanf(Parm[1], “%d”, &Ln);
    sscanf(Parm[2], “%d”, &Pos);
    sscanf(Parm[3], “%d”, &Val);
    printf(“/* Line %d Character Position %d Replaced
with %d. */\n”, Ln, Pos,
        Val);
    File();
    }

int P()
    {
    if (Ln != 0)
        printf(“%s”, MatchStr);
    return TRUE;
    }

int Spacing()
    {
    if (Space())
        {
        return TRUE;
        }
    if (Tab())
        {
        return TRUE;
        }
    return FALSE;
    }

int WhiteSpace()
    {
```

```
    if (Space())
        {
        return TRUE;
        }
    if (Tab())
        {
        return TRUE;
        }
    if (LineBreak())
        {
        return TRUE;
        }
    return FALSE;
    }

int Spaces()
    {
    while (WhiteSpace())
        ;
    return TRUE;
    }

int CM()
    {
    strcat(NumberString, MatchStr);
    return TRUE;
    }

int Number()
    {
    pos = cp;
    ln = LineNo;
    NumberString[0] = 0;
    if (Digit())
        {
        number = MatchStr[0] - '0';
        CM();
        while (Digit())
            {
            number = 10 *number + MatchStr[0] - '0';
            CM();
            }
        return TRUE;
        }
    return FALSE;
```

```
    }

int Replace()
    {
    return Ln;
    return TRUE;
    }

int Empty()
    {
    return TRUE;
    }

int SaveorReplaceNumber()
    {
    if (Replace())
        {
        if ((Ln == ln)
        &&(Pos == pos))
            printf("%d", Val);
        else
            printf("%s", NumberString);
        return TRUE;
        }
    if (Empty())
        {
        printf("Line %d Character %d = %d\n", ln, pos,
number);
        return TRUE;
        }
    return FALSE;
    }

int CommentCharacter()
    {
    if ((Line[cp] == '*')
    &&(Line[cp + 1] == '/'))
        return FALSE;
    if (AnyCharacter())
        {
        P();
        return TRUE;
        }
    return FALSE;
    }
```

```
int Comment()
    {
    if (Matches(“/*”))
        {
        P();
        while (CommentCharacter())
            ;
        return TRUE;
        }
    return FALSE;
    }

int NumberorCharacter()
    {
    if (EndFile)
        return FALSE;
    if (Comment())
        {
        if (!Matches(“*/”))
            error(“*/”);
        P();
        return TRUE;
        }
    if (Number())
        {
        if (!SaveorReplaceNumber())
            error(“SaveorReplaceNumber”);
        return TRUE;
        }
    if (AnyCharacter())
        {
        P();
        return TRUE;
        }
    return FALSE;
    }

int Program()
    {
    if (NumberorCharacter())
        {
        while (NumberorCharacter())
            ;
        return TRUE;
```

```
        }
    return FALSE;
    }

int File()
    {
    ReadLine();
    if (Spaces())
        ;
    if (Program())
        {
        if (!EndofFile())
            error("EndofFile");
        return TRUE;
        }
    return FALSE;
    }
```

And the general purpose USRP Header file:

```
/*       This program was reformatted by prettyc     */

/*  Useful Self Replicating Program C Language Version
*/


/*  Begin Useful Self Replicating Program C Language
Boilerplate */


/* written by:
  Alan A. Jorgensen

Tue Nov 21 10:58:03 EST 1995
*/

#include <stdio.h>
#include <string.h>
#define FALSE 0
#define TRUE 1
#define MaxLineLength 255
char Line[MaxLineLength];
int cp = 0;
int Length;
```

```
int EndFile = FALSE;
char Character;
char MatchStr[MaxLineLength] = “”;
int LineNo = 0;
int ReadLine()
    {
    int i = 0;
    int c;
    for (i = 0; i < 254
    &&(c = getchar()) != EOF
    && c != '\n'; i++)
        Line[i] = c;
    if (c == EOF)
        {
        EndFile = TRUE;
        Line[0] = 0;
        cp = 0;
        Length = 0;
        return 0;
        }
    LineNo++;
    if (c == '\n')
        {
        Line[i] = '\n';
        Line[i + 1] = 0;
        Length = i + 1;
        }
    else
        {
        Line[i] = 0;
        Length = i;
        }
    cp = 0;
    if (c == EOF)
        {
        EndFile = TRUE;
        return EOF;
        }
    return 0;
    }

int error(msg)
char * msg;
    {
    int i;
```

```
    printf("\nLine %d:\n", LineNo);
    printf("%s", Line);
    for (i = 0; i < cp; i++)
        if (Line[i] == '\t')
            printf("\t");
    else
        printf(" ");
    printf("^\n");
    printf("%s expected.\n", msg);
    exit(1);
    return TRUE;
    }

char lower(letter)
char letter;
    {
    if (letter >= 'A'
    && letter <= 'Z')
        return(letter + 'a' - 'A');
    else
        return(letter);
    }

void LowerCase(Str)
char *Str;
    {
    while (*Str != 0)
        {
        *Str = lower(*Str);
        Str++;
        }
    }

Skip(i)
int i;
    {
    cp += i;
    if (cp >= Length)
        ReadLine();
    }

int Matches(s)
char * s;
    {
    int i;
```

```
    char *st = s;
    if (EndFile)
        return FALSE;
    for (i = 0;(*s != 0)
    && *s == *(Line +cp +(i++)); s++)
        ;
    if (*s)
        return FALSE;
    else
        {
        if (i == 1)
            {
            Character = Line[cp];
            MatchStr[0] = Character;
            MatchStr[1] = 0;
            }
        else
            strcpy(MatchStr, st);
        Skip(i);
        return TRUE;
        }
    }

int Alphabetic = FALSE;
int KeyMatches(s)
char * s;
    {/* Matches AND the next character is not a label
character  */
    int i;
    char *st = s;
    if (EndFile)
        return FALSE;
    for (i = 0;(*s != 0)
    && *s == *(Line +cp +(i++)); s++)
        ;
    if (*s)
        return FALSE;
    else
        {
        if (((Line[cp +i] >= '0')
        &&(Line[cp +i] <= '9'))
        ||((Line[cp +i] >= 'a')
        &&(Line[cp +i] <= 'z'))
        ||((Line[cp +i] >= 'A')
        &&(Line[cp +i] <= 'Z'))
```

```
        ||(Line[cp +i] == '_'))
            return FALSE;
        if (i == 1)
            {
            Character = Line[cp];
            MatchStr[0] = Character;
            MatchStr[1] = 0;
            }
        else
            strcpy(MatchStr, st);
        Skip(i);
        return TRUE;
        }
    }

int UpperCaseLetter()
    {
    if ((Line[cp] >= 'A')
    &&(Line[cp] <= 'Z'))
        {
        Character = Line[cp];
        MatchStr[0] = Character;
        MatchStr[1] = 0;
        Skip(1);
        return TRUE;
        }
    else
        return FALSE;
    }

int LowerCaseLetter()
    {
    if ((Line[cp] >= 'a')
    &&(Line[cp] <= 'z'))
        {
        Character = Line[cp];
        MatchStr[0] = Character;
        MatchStr[1] = 0;
        Skip(1);
        return TRUE;
        }
    else
        return FALSE;
    }
```

```
int Digit()
    {
    if ((Line[cp] >= '0')
    &&(Line[cp] <= '9'))
        {
        Character = Line[cp];
        MatchStr[0] = Character;
        MatchStr[1] = 0;
        Skip(1);
        return TRUE;
        }
    else
        return FALSE;
    }

int OctalDigit()
    {
    if ((Line[cp] >= '0')
    &&(Line[cp] <= '7'))
        {
        Character = Line[cp];
        MatchStr[0] = Character;
        MatchStr[1] = 0;
        Skip(1);
        return TRUE;
        }
    else
        return FALSE;
    }

int HexDigit()
    {
    if ((Line[cp] >= '0')
    &&(Line[cp] <= '9')
    ||(Line[cp] >= 'a')
    &&(Line[cp] <= 'f')
    ||(Line[cp] >= 'A')
    &&(Line[cp] <= 'F'))
        {
        Character = Line[cp];
        MatchStr[0] = Character;
        MatchStr[1] = 0;
        Skip(1);
        return TRUE;
        }
```

```
    else
        return FALSE;
    }

int Letter()
    {
    if (UpperCaseLetter())
        return TRUE;
    else
        return LowerCaseLetter();
    }

int Space()
    {
    return Matches(“ “);
    }

int Tab()
    {
    return Matches(“\t”);
    }

int LineBreak()
    {
    if (Matches(“\n”))
        {
        return TRUE;
        }
    return FALSE;
    }

int EndofFile()
    {
    return EndFile;
    }

int AnyCharacter()
    {
    if (EndFile == TRUE)
        return FALSE;
    MatchStr[0] = Line[cp];
    MatchStr[1] = 0;
    Skip(1);
    return TRUE;
    }
```

```
/*  End Useful Self Replicating Program C Languange
Boilerplate */
```

Actual validation is accomplished with a Korn Shell script that produces a list of numberic string locations and then generates, compiles, and executes each permutation. The script produces a log indicating the specific permutation, the permuted line, before and after, and the results of running the test against that permutation. When the compile was unsuccessful, this is noted in the log and the test is not run on that permutation. This was generally the case when the parser located the "0" or the "7" in 0x7FFFFFFF. The parser could be enhanced to recognize more complex token types.

```
# TestTest -- This test evaluates the capability of Test
#   to locate bugs
#   in RunAvg.  fltinj is used to inject faults into
#   RunAvg.c, compiles them
#   and runs them.

function TestCase
    {
    let Case+=1
    echo >>Log
    echo "_____________________________________________"
>>Log
    echo "Case $Case $(date)" >>Log
    echo >>Log
    fltinj $L $C $V <RunAvg.c >TC.c

    diff RunAvg.c TC.c >>Log

    gcc -g -o RunAvg TC.c 2>>Log
    if [[ $? != 0 ]]
    then echo "Compile Failed" >>Log
       let CompileFails+=1
```

```
    else
       Validate | wc -l | read ErrorLines
       if (( ErrorLines != 4 ))
       then echo “Test Failed” >>Log
          let TestFails+=1
       else echo “Test Passed!” >>Log
          let TestPasses+=1
          head -1 TC.c >>Log
       fi
    fi
    }

function RunCase
    {
    read Line
    while read x L x C x Va
    do
       let V=Va-1
       TestCase
       let V=Va+1
       TestCase
       if (( $Case >= $Cases ))
       then break
       fi
    done
    }

if [[ $1 = “” ]]
then Cases=50000
else Cases=$1
fi
rm -f Log

fltinj <RunAvg.c >Cases


Case=0
CompileFails=0
TestFails=0
TestPasses=0
cat Cases | RunCase
echo >>Log
let Total=CompileFails+TestFails+TestPasses
echo “Out of $Total test cases” >>Log
echo “$CompileFails cases failed to compile.” >>Log
```

```
echo “$TestFails failed the test.” >>Log
echo “$TestPasses passed the test.” >>Log
```

And the resulting Log file:

```
_____________________________________________________
Case 1 Sat Oct 23 13:18:00 EDT 1999

0a1
> /* Line 17 Character Position 18 Replaced with 1999.
*/
17c18
< #define MaxValues 2000
---
> #define MaxValues 1999
Test Failed

_____________________________________________________
Case 2 Sat Oct 23 13:18:01 EDT 1999

0a1
> /* Line 17 Character Position 18 Replaced with 2001.
*/
17c18
< #define MaxValues 2000
---
> #define MaxValues 2001
Test Failed

_____________________________________________________
Case 3 Sat Oct 23 13:18:02 EDT 1999

0a1
> /* Line 18 Character Position 17 Replaced with 999999.
*/
18c19
< #define MaxEntry 1000000
---
> #define MaxEntry 999999
Test Failed

_____________________________________________________
```

```
Case 4 Sat Oct 23 13:18:04 EDT 1999

0a1
> /* Line 18 Character Position 17 Replaced with
1000001. */
18c19
< #define MaxEntry 1000000
---
> #define MaxEntry 1000001
Test Failed

_____________________________________________________
Case 5 Sat Oct 23 13:18:05 EDT 1999

0a1
> /* Line 19 Character Position 15 Replaced with -1. */
19c20
< #define MaxInt 0x7FFFFFFF
---
> #define MaxInt -1x7FFFFFFF
TC.c:59: missing white space after number `1'
Compile Failed

_____________________________________________________
Case 6 Sat Oct 23 13:18:05 EDT 1999

0a1
> /* Line 19 Character Position 15 Replaced with 1. */
19c20
< #define MaxInt 0x7FFFFFFF
---
> #define MaxInt 1x7FFFFFFF
TC.c:59: missing white space after number `1'
Compile Failed

_____________________________________________________
Case 7 Sat Oct 23 13:18:05 EDT 1999

0a1
> /* Line 19 Character Position 17 Replaced with 6. */
19c20
< #define MaxInt 0x7FFFFFFF
---
> #define MaxInt 0x6FFFFFFF
TC.c:60: parse error before `CONFIGURATION_ERROR'
```

```
TC.c:79: warning: data definition has no type or storage
class
TC.c:80: parse error before `}'
Compile Failed

_____________________________________________________
Case 8 Sat Oct 23 13:18:05 EDT 1999

0a1
> /* Line 19 Character Position 17 Replaced with 8. */
19c20
< #define MaxInt 0x7FFFFFFF
---
> #define MaxInt 0x8FFFFFFF
Test Failed

_____________________________________________________
Case 9 Sat Oct 23 13:18:06 EDT 1999

0a1
> /* Line 20 Character Position 19 Replaced with 79. */
20c21
< #define LineLength 80
---
> #define LineLength 79
Test Failed

_____________________________________________________
Case 10 Sat Oct 23 13:18:08 EDT 1999

0a1
> /* Line 20 Character Position 19 Replaced with 81. */
20c21
< #define LineLength 80
---
> #define LineLength 81
Test Failed

_____________________________________________________
Case 11 Sat Oct 23 13:18:09 EDT 1999

0a1
> /* Line 49 Character Position 22 Replaced with 34. */
49c50
< char LineLengthFormat[35];
```

```
---
> char LineLengthFormat[34];
Test Passed!
/* Line 49 Character Position 22 Replaced with 34. */

_____________________________________________________
Case 12 Sat Oct 23 13:18:10 EDT 1999

0a1
> /* Line 49 Character Position 22 Replaced with 36. */
49c50
< char LineLengthFormat[35];
---
> char LineLengthFormat[36];
Test Passed!
/* Line 49 Character Position 22 Replaced with 36. */

_____________________________________________________
Case 13 Sat Oct 23 13:18:12 EDT 1999

0a1
> /* Line 52 Character Position 53 Replaced with 0. */
52c53
<   sprintf(LineLengthFormat, “%%.%ds\n”, LineLength -
1);
---
>   sprintf(LineLengthFormat, “%%.%ds\n”, LineLength -
0);
Test Failed

_____________________________________________________
Case 14 Sat Oct 23 13:18:13 EDT 1999

0a1
> /* Line 52 Character Position 53 Replaced with 2. */
52c53
<   sprintf(LineLengthFormat, “%%.%ds\n”, LineLength -
1);
---
>   sprintf(LineLengthFormat, “%%.%ds\n”, LineLength -
2);
Test Failed

_____________________________________________________
Case 15 Sat Oct 23 13:18:14 EDT 1999
```

```
0a1
> /* Line 58 Character Position 29 Replaced with 1. */
58c59
< #if (MaxEntry + (MaxValues / 2)) > ( MaxInt /
MaxValues )
---
> #if (MaxEntry + (MaxValues / 1)) > ( MaxInt /
MaxValues )
Test Passed!
/* Line 58 Character Position 29 Replaced with 1. */

_____________________________________________________
Case 16 Sat Oct 23 13:18:15 EDT 1999

0a1
> /* Line 58 Character Position 29 Replaced with 3. */
58c59
< #if (MaxEntry + (MaxValues / 2)) > ( MaxInt /
MaxValues )
---
> #if (MaxEntry + (MaxValues / 3)) > ( MaxInt /
MaxValues )
Test Passed!
/* Line 58 Character Position 29 Replaced with 3. */

_____________________________________________________
Case 17 Sat Oct 23 13:18:16 EDT 1999

0a1
> /* Line 59 Character Position 59 Replaced with 1. */
59c60
< X CONFIGURATION_ERROR - MaxValues * MaxEntry +
MaxValues / 2 must be <
---
> X CONFIGURATION_ERROR - MaxValues * MaxEntry +
MaxValues / 1 must be <
Test Passed!
/* Line 59 Character Position 59 Replaced with 1. */

_____________________________________________________
Case 18 Sat Oct 23 13:18:17 EDT 1999

0a1
> /* Line 59 Character Position 59 Replaced with 3. */
```

```
59c60
< X CONFIGURATION_ERROR - MaxValues * MaxEntry +
MaxValues / 2 must be <
---
> X CONFIGURATION_ERROR - MaxValues * MaxEntry +
MaxValues / 3 must be <
Test Passed!
/* Line 59 Character Position 59 Replaced with 3. */

_____________________________________________________
Case 19 Sat Oct 23 13:18:19 EDT 1999

0a1
> /* Line 75 Character Position 50 Replaced with 0. */
75c76
<   if (strnlen(Message, LineLength) > LineLength - 1)
---
>   if (strnlen(Message, LineLength) > LineLength - 0)
Test Failed

_____________________________________________________
Case 20 Sat Oct 23 13:18:20 EDT 1999

0a1
> /* Line 75 Character Position 50 Replaced with 2. */
75c76
<   if (strnlen(Message, LineLength) > LineLength - 1)
---
>   if (strnlen(Message, LineLength) > LineLength - 2)
Test Passed!
/* Line 75 Character Position 50 Replaced with 2. */

_____________________________________________________
Case 21 Sat Oct 23 13:18:21 EDT 1999

0a1
> /* Line 94 Character Position 11 Replaced with -1. */
94c95
<   for (c = 0; c < l; c++)
---
>   for (c = -1; c < l; c++)
Test Failed

_____________________________________________________
Case 22 Sat Oct 23 13:18:22 EDT 1999
```

```
0a1
> /* Line 94 Character Position 11 Replaced with 1. */
94c95
<   for (c = 0; c < l; c++)
---
>   for (c = 1; c < l; c++)
Test Passed!
/* Line 94 Character Position 11 Replaced with 1. */

_____________________________________________________
Case 23 Sat Oct 23 13:18:23 EDT 1999

0a1
> /* Line 95 Character Position 16 Replaced with -1. */
95c96
<     if (s[c] == 0)
---
>     if (s[c] == -1)
TC.c: In function `strnlen':
TC.c:96: warning: comparison is always 0 due to limited
range of data type
Test Failed

_____________________________________________________
Case 24 Sat Oct 23 13:18:24 EDT 1999

0a1
> /* Line 95 Character Position 16 Replaced with 1. */
95c96
<     if (s[c] == 0)
---
>     if (s[c] == 1)
Test Failed

_____________________________________________________
Case 25 Sat Oct 23 13:18:25 EDT 1999

0a1
> /* Line 114 Character Position 7 Replaced with 0. */
114c115
<   exit(1);
---
>   exit(0);
Test Failed
```

```
_____________________________________________________
Case 26 Sat Oct 23 13:18:26 EDT 1999

0a1
> /* Line 114 Character Position 7 Replaced with 2. */
114c115
<   exit(1);
---
>   exit(2);
Test Failed


_____________________________________________________
Case 27 Sat Oct 23 13:18:28 EDT 1999

0a1
> /* Line 137 Character Position 13 Replaced with -1. */
137c138
< int Values = 0;  /* Number of valid values entered. */
---
> int Values = -1;  /* Number of valid values entered.
*/
Test Failed


_____________________________________________________
Case 28 Sat Oct 23 13:18:29 EDT 1999

0a1
> /* Line 137 Character Position 13 Replaced with 1. */
137c138
< int Values = 0;  /* Number of valid values entered. */
---
> int Values = 1;  /* Number of valid values entered. */
Test Failed


_____________________________________________________
Case 29 Sat Oct 23 13:18:30 EDT 1999

0a1
> /* Line 138 Character Position 10 Replaced with -1. */
138c139
< int Sum = 0;  /* Sum of the valid values entered. */
---
> int Sum = -1;  /* Sum of the valid values entered. */
Test Failed
```

```
_____________________________________________________
Case 30 Sat Oct 23 13:18:31 EDT 1999

0a1
> /* Line 138 Character Position 10 Replaced with 1. */
138c139
< int Sum = 0;  /* Sum of the valid values entered. */
---
> int Sum = 1;  /* Sum of the valid values entered. */
Test Failed

_____________________________________________________
Case 31 Sat Oct 23 13:18:32 EDT 1999

0a1
> /* Line 145 Character Position 13 Replaced with -1. */
145c146
<   if (Sum != 0)
---
>   if (Sum != -1)
Test Failed

_____________________________________________________
Case 32 Sat Oct 23 13:18:33 EDT 1999

0a1
> /* Line 145 Character Position 13 Replaced with 1. */
145c146
<   if (Sum != 0)
---
>   if (Sum != 1)
Test Failed

_____________________________________________________
Case 33 Sat Oct 23 13:18:34 EDT 1999

0a1
> /* Line 147 Character Position 16 Replaced with -1. */
147c148
<   if (Values != 0)
---
>   if (Values != -1)
Test Failed
```

```
_____________________________________________________
Case 34 Sat Oct 23 13:18:36 EDT 1999

0a1
> /* Line 147 Character Position 16 Replaced with 1. */
147c148
<   if (Values != 0)
---
>   if (Values != 1)
Test Failed

_____________________________________________________
Case 35 Sat Oct 23 13:18:37 EDT 1999

0a1
> /* Line 160 Character Position 13 Replaced with -1. */
160c161
<   SetHistory(0, 0);
---
>   SetHistory(-1, 0);
Test Failed

_____________________________________________________
Case 36 Sat Oct 23 13:18:38 EDT 1999

0a1
> /* Line 160 Character Position 13 Replaced with 1. */
160c161
<   SetHistory(0, 0);
---
>   SetHistory(1, 0);
Test Failed

_____________________________________________________
Case 37 Sat Oct 23 13:18:39 EDT 1999

0a1
> /* Line 160 Character Position 16 Replaced with -1. */
160c161
<   SetHistory(0, 0);
---
>   SetHistory(0, -1);
Test Failed

_____________________________________________________
```

```
Case 38 Sat Oct 23 13:18:40 EDT 1999

0a1
> /* Line 160 Character Position 16 Replaced with 1. */
160c161
<   SetHistory(0, 0);
---
>   SetHistory(0, 1);
Test Failed


_____________________________________________________
Case 39 Sat Oct 23 13:18:41 EDT 1999

0a1
> /* Line 168 Character Position 15 Replaced with 0. */
168c169
< #define Exit (-1)
---
> #define Exit (-0)
Test Failed


_____________________________________________________
Case 40 Sat Oct 23 13:18:43 EDT 1999

0a1
> /* Line 168 Character Position 15 Replaced with 2. */
168c169
< #define Exit (-1)
---
> #define Exit (-2)
Test Passed!
/* Line 168 Character Position 15 Replaced with 2. */


_____________________________________________________
Case 41 Sat Oct 23 13:18:44 EDT 1999

0a1
> /* Line 176 Character Position 15 Replaced with -1. */
176c177
<   int Number = 0;
---
>   int Number = -1;
Test Failed


_____________________________________________________
```

```
Case 42 Sat Oct 23 13:18:45 EDT 1999

0a1
> /* Line 176 Character Position 15 Replaced with 1. */
176c177
<   int Number = 0;
---
>   int Number = 1;
Test Failed


_____________________________________________________
Case 43 Sat Oct 23 13:18:46 EDT 1999

0a1
> /* Line 186 Character Position 18 Replaced with -1. */
186c187
<     if ((Digit < '0')
---
>     if ((Digit < '-1')
TC.c: In function `Input':
TC.c:187: warning: multi-character character constant
Test Failed


_____________________________________________________
Case 44 Sat Oct 23 13:18:47 EDT 1999

0a1
> /* Line 186 Character Position 18 Replaced with 1. */
186c187
<     if ((Digit < '0')
---
>     if ((Digit < '1')
Test Failed


_____________________________________________________
Case 45 Sat Oct 23 13:18:48 EDT 1999

0a1
> /* Line 187 Character Position 16 Replaced with 8. */
187c188
<     ||(Digit > '9'))
---
>     ||(Digit > '8'))
Test Failed
```

```
_____________________________________________________
Case 46 Sat Oct 23 13:18:49 EDT 1999

0a1
> /* Line 187 Character Position 16 Replaced with 10. */
187c188
<     ||(Digit > '9'))
---
>     ||(Digit > '10'))
TC.c: In function `Input':
TC.c:188: warning: multi-character character constant
Test Failed


_____________________________________________________
Case 47 Sat Oct 23 13:18:50 EDT 1999

0a1
> /* Line 192 Character Position 17 Replaced with -1. */
192c193
<         Number = 0;
---
>         Number = -1;
Test Failed


_____________________________________________________
Case 48 Sat Oct 23 13:18:52 EDT 1999

0a1
> /* Line 192 Character Position 17 Replaced with 1. */
192c193
<         Number = 0;
---
>         Number = 1;
Test Passed!
/* Line 192 Character Position 17 Replaced with 1. */


_____________________________________________________
Case 49 Sat Oct 23 13:18:53 EDT 1999

0a1
> /* Line 197 Character Position 30 Replaced with 9. */
197c198
<       if (Number > MaxEntry / 10)
---
>       if (Number > MaxEntry / 9)
```

```
Test Passed!
/* Line 197 Character Position 30 Replaced with 9. */

_____________________________________________________
Case 50 Sat Oct 23 13:18:54 EDT 1999

0a1
> /* Line 197 Character Position 30 Replaced with 11. */
197c198
<       if (Number > MaxEntry / 10)
---
>       if (Number > MaxEntry / 11)
Test Failed

_____________________________________________________
Case 51 Sat Oct 23 13:18:55 EDT 1999

0a1
> /* Line 206 Character Position 24 Replaced with -1. */
206c207
<       else if (Digit - '0' > MaxEntry - 10 *Number)
---
>       else if (Digit - '-1' > MaxEntry - 10 *Number)
TC.c: In function `Input':
TC.c:207: warning: multi-character character constant
Test Failed

_____________________________________________________
Case 52 Sat Oct 23 13:18:56 EDT 1999

0a1
> /* Line 206 Character Position 24 Replaced with 1. */
206c207
<       else if (Digit - '0' > MaxEntry - 10 *Number)
---
>       else if (Digit - '1' > MaxEntry - 10 *Number)
Test Failed

_____________________________________________________
Case 53 Sat Oct 23 13:18:57 EDT 1999

0a1
> /* Line 206 Character Position 40 Replaced with 9. */
206c207
<       else if (Digit - '0' > MaxEntry - 10 *Number)
```

```
---
>       else if (Digit - '0' > MaxEntry - 9 *Number)
Test Failed


_____________________________________________________
Case 54 Sat Oct 23 13:18:58 EDT 1999

0a1
> /* Line 206 Character Position 40 Replaced with 11. */
206c207
<       else if (Digit - '0' > MaxEntry - 10 *Number)
---
>       else if (Digit - '0' > MaxEntry - 11 *Number)
Test Failed


_____________________________________________________
Case 55 Sat Oct 23 13:19:00 EDT 1999

0a1
> /* Line 219 Character Position 17 Replaced with 9. */
219c220
<         Number = 10 *Number - '0' + Digit;
---
>         Number = 9 *Number - '0' + Digit;
Test Failed


_____________________________________________________
Case 56 Sat Oct 23 13:19:01 EDT 1999

0a1
> /* Line 219 Character Position 17 Replaced with 11. */
219c220
<         Number = 10 *Number - '0' + Digit;
---
>         Number = 11 *Number - '0' + Digit;
Test Failed


_____________________________________________________
Case 57 Sat Oct 23 13:19:02 EDT 1999

0a1
> /* Line 219 Character Position 31 Replaced with -1. */
219c220
<         Number = 10 *Number - '0' + Digit;
---
```

```
>         Number = 10 *Number - '-1' + Digit;
TC.c: In function `Input':
TC.c:220: warning: multi-character character constant
Test Failed


_____________________________________________________
Case 58 Sat Oct 23 13:19:03 EDT 1999

0a1
> /* Line 219 Character Position 31 Replaced with 1. */
219c220
<         Number = 10 *Number - '0' + Digit;
---
>         Number = 10 *Number - '1' + Digit;
Test Failed


_____________________________________________________
Case 59 Sat Oct 23 13:19:04 EDT 1999

0a1
> /* Line 237 Character Position 15 Replaced with -1. */
237c238
<   int Number = 0;
---
>   int Number = -1;
Test Failed


_____________________________________________________
Case 60 Sat Oct 23 13:19:05 EDT 1999

0a1
> /* Line 237 Character Position 15 Replaced with 1. */
237c238
<   int Number = 0;
---
>   int Number = 1;
Test Passed!
/* Line 237 Character Position 15 Replaced with 1. */


_____________________________________________________
Case 61 Sat Oct 23 13:19:06 EDT 1999

0a1
> /* Line 250 Character Position 19 Replaced with -1. */
250c251
```

```
<       if (Number < 0)
---
>       if (Number < -1)
Test Passed!
/* Line 250 Character Position 19 Replaced with -1. */

_____________________________________________________
Case 62 Sat Oct 23 13:19:08 EDT 1999

0a1
> /* Line 250 Character Position 19 Replaced with 1. */
250c251
<       if (Number < 0)
---
>       if (Number < 1)
Test Failed

_____________________________________________________
Case 63 Sat Oct 23 13:19:09 EDT 1999

0a1
> /* Line 257 Character Position 36 Replaced with 0. */
257c258
<       else if (Values > MaxValues - 1)
---
>       else if (Values > MaxValues - 0)
Test Failed

_____________________________________________________
Case 64 Sat Oct 23 13:19:10 EDT 1999

0a1
> /* Line 257 Character Position 36 Replaced with 2. */
257c258
<       else if (Values > MaxValues - 1)
---
>       else if (Values > MaxValues - 2)
Test Failed

_____________________________________________________
Case 65 Sat Oct 23 13:19:12 EDT 1999

0a1
> /* Line 264 Character Position 26 Replaced with 0. */
264c265
```

```
<         Values = Values + 1;
---
>         Values = Values + 0;
Test Failed


_____________________________________________________
Case 66 Sat Oct 23 13:19:13 EDT 1999

0a1
> /* Line 264 Character Position 26 Replaced with 2. */
264c265
<         Values = Values + 1;
---
>         Values = Values + 2;
Test Failed


_____________________________________________________
Case 67 Sat Oct 23 13:19:14 EDT 1999

0a1
> /* Line 275 Character Position 36 Replaced with 79. */
275c276
<           char RunningAverageReport[80];
---
>           char RunningAverageReport[79];
Test Passed!
/* Line 275 Character Position 36 Replaced with 79. */


_____________________________________________________
Case 68 Sat Oct 23 13:19:15 EDT 1999

0a1
> /* Line 275 Character Position 36 Replaced with 81. */
275c276
<           char RunningAverageReport[80];
---
>           char RunningAverageReport[81];
Test Passed!
/* Line 275 Character Position 36 Replaced with 81. */


_____________________________________________________
Case 69 Sat Oct 23 13:19:16 EDT 1999

0a1
> /* Line 277 Character Position 24 Replaced with 1. */
```

```
277c278
<               (Values / 2)) / Values));
---
>               (Values / 1)) / Values));
Test Failed


_____________________________________________________
Case 70 Sat Oct 23 13:19:17 EDT 1999

0a1
> /* Line 277 Character Position 24 Replaced with 3. */
277c278
<               (Values / 2)) / Values));
---
>               (Values / 3)) / Values));
Test Failed


_____________________________________________________
Case 71 Sat Oct 23 13:19:18 EDT 1999

0a1
> /* Line 304 Character Position 16 Replaced with 0. */
304c305
<   if (Params != 1)
---
>   if (Params != 0)
Test Failed


_____________________________________________________
Case 72 Sat Oct 23 13:19:20 EDT 1999

0a1
> /* Line 304 Character Position 16 Replaced with 2. */
304c305
<   if (Params != 1)
---
>   if (Params != 2)
Test Failed


_____________________________________________________
Case 73 Sat Oct 23 13:19:21 EDT 1999

0a1
> /* Line 306 Character Position 26 Replaced with 0. */
306c307
```

```
<     if (!strcmp(Parameter[1], “test”))
---
>     if (!strcmp(Parameter[0], “test”))
Test Failed


_____________________________________________________
Case 74 Sat Oct 23 13:19:22 EDT 1999

0a1
> /* Line 306 Character Position 26 Replaced with 2. */
306c307
<     if (!strcmp(Parameter[1], “test”))
---
>     if (!strcmp(Parameter[2], “test”))
Test Failed


_____________________________________________________
Case 75 Sat Oct 23 13:19:23 EDT 1999

0a1
> /* Line 309 Character Position 23 Replaced with 1. */
309c310
<       sscanf(Parameter[2], “%d”, &InitialCount);
---
>       sscanf(Parameter[1], “%d”, &InitialCount);
Test Failed


_____________________________________________________
Case 76 Sat Oct 23 13:19:24 EDT 1999

0a1
> /* Line 309 Character Position 23 Replaced with 3. */
309c310
<       sscanf(Parameter[2], “%d”, &InitialCount);
---
>       sscanf(Parameter[3], “%d”, &InitialCount);
Test Failed


_____________________________________________________
Case 77 Sat Oct 23 13:19:25 EDT 1999

0a1
> /* Line 310 Character Position 23 Replaced with 2. */
310c311
<       sscanf(Parameter[3], “%d”, &InitialSum);
```

```
---
>       sscanf(Parameter[2], “%d”, &InitialSum);
Test Failed


_____________________________________________________
Case 78 Sat Oct 23 13:19:26 EDT 1999

0a1
> /* Line 310 Character Position 23 Replaced with 4. */
310c311
<       sscanf(Parameter[3], “%d”, &InitialSum);
---
>       sscanf(Parameter[4], “%d”, &InitialSum);
Test Failed


_____________________________________________________
Case 79 Sat Oct 23 13:19:28 EDT 1999

0a1
> /* Line 313 Character Position 31 Replaced with 0. */
313c314
<     else if (!strcmp(Parameter[1], “trial”))
---
>     else if (!strcmp(Parameter[0], “trial”))
Test Failed


_____________________________________________________
Case 80 Sat Oct 23 13:19:29 EDT 1999

0a1
> /* Line 313 Character Position 31 Replaced with 2. */
313c314
<     else if (!strcmp(Parameter[1], “trial”))
---
>     else if (!strcmp(Parameter[2], “trial”))
Test Failed


_____________________________________________________
Case 81 Sat Oct 23 13:19:30 EDT 1999

0a1
> /* Line 315 Character Position 23 Replaced with 1. */
315c316
<       sscanf(Parameter[2], “%d”, &InitialCount);
---
```

```
>       sscanf(Parameter[1], “%d”, &InitialCount);
Test Failed

_____________________________________________________
Case 82 Sat Oct 23 13:19:31 EDT 1999

0a1
> /* Line 315 Character Position 23 Replaced with 3. */
315c316
<       sscanf(Parameter[2], “%d”, &InitialCount);
---
>       sscanf(Parameter[3], “%d”, &InitialCount);
Test Failed

_____________________________________________________
Case 83 Sat Oct 23 13:19:32 EDT 1999

0a1
> /* Line 316 Character Position 23 Replaced with 2. */
316c317
<       sscanf(Parameter[3], “%d”, &InitialSum);
---
>       sscanf(Parameter[2], “%d”, &InitialSum);
Test Failed

_____________________________________________________
Case 84 Sat Oct 23 13:19:33 EDT 1999

0a1
> /* Line 316 Character Position 23 Replaced with 4. */
316c317
<       sscanf(Parameter[3], “%d”, &InitialSum);
---
>       sscanf(Parameter[4], “%d”, &InitialSum);
Test Failed

_____________________________________________________
Case 85 Sat Oct 23 13:19:34 EDT 1999

0a1
> /* Line 320 Character Position 31 Replaced with 0. */
320c321
<     else if (!strcmp(Parameter[1], “long”))
---
>     else if (!strcmp(Parameter[0], “long”))
```

```
Test Failed

_____________________________________________________
Case 86 Sat Oct 23 13:19:36 EDT 1999

0a1
> /* Line 320 Character Position 31 Replaced with 2. */
320c321
<     else if (!strcmp(Parameter[1], “long”))
---
>     else if (!strcmp(Parameter[2], “long”))
Test Failed

_____________________________________________________
Case 87 Sat Oct 23 13:19:37 EDT 1999

0a1
> /* Line 323 Character Position 11 Replaced with -
834729263. */
323c324
<
“123456789012345678901234567890123456789012345678901234567890\
---
>           “-834729263\
Test Failed

_____________________________________________________
Case 88 Sat Oct 23 13:19:38 EDT 1999

0a1
> /* Line 323 Character Position 11 Replaced with -
834729261. */
323c324
<
“123456789012345678901234567890123456789012345678901234567890\
---
>           “-834729261\
Test Failed

_____________________________________________________
Case 89 Sat Oct 23 13:19:39 EDT 1999

0a1
> /* Line 324 Character Position 0 Replaced with -
834729263. */
```

```
324c325
< 12345678901234567890123456789012345678901234567890”
---
> -834729263”
Test Failed

_____________________________________________________
Case 90 Sat Oct 23 13:19:40 EDT 1999

0a1
> /* Line 324 Character Position 0 Replaced with -
834729261. */
324c325
< 12345678901234567890123456789012345678901234567890”
---
> -834729261”
Test Failed

_____________________________________________________
Case 91 Sat Oct 23 13:19:41 EDT 1999

0a1
> /* Line 341 Character Position 7 Replaced with -1. */
341c342
<   exit(0);
---
>   exit(-1);
Test Failed

_____________________________________________________
Case 92 Sat Oct 23 13:19:43 EDT 1999

0a1
> /* Line 341 Character Position 7 Replaced with 1. */
341c342
<   exit(0);
---
>   exit(1);
Test Failed

Out of 92 test cases
3 cases failed to compile.
74 failed the test.
15 passed the test.
```

Appendix H – Markov Chain Case Study Code

The following file is the parameter file, in general, defining the constraint values obtained from the data dictionary.

```
/* Parameters.h -- compilable configuration file */

/* White Space */

#define Tab '\t'
#define Space ' '
#define NewLine '\n'

#define WhiteSpace(c) ((c == Tab) || (c == Space) || (c == NewLine))

#define EndFile(c) (c == EOF)

/* Parameters.h -- compilable configuration file */

typedef int TextIndex;  /* 1 .. MaxText */
#define EndofWords -1

/* Smallest Printable Character */

#define SmallestPrintableCharacter '!'
```

```
/* Largest Printable Character */

#define LargestPrintableCharacter '~'

#define ValidASCII(c) ((c >= SmallestPrintableCharacter) \
                    && (c <= LargestPrintableCharacter))

/* Maximum Word Length */

#define MaximumWordLength 20

/* Maximum Number of Words */

#define MaximumNumberofWords 50000

/* Maximum Number of Unique Words */

#define MaximumNumberofUniqueWords 20000

/* Maximum Valid Successors */

#define MaximumValidSuccessors MaximumNumberofWords

/* Maximum Words Output */

#define MaximumWordsOutput 10000

/* Maximum available text space including null characters. */
```

```
/* This is the Maximum number of unique words times the maximum
    word length, plus a null character for each word, plus one
    additional null at the beginning so that each word will have
    a null preceding and a null following.
*/

#define MxUniqWords MaximumNumberofUniqueWords

#define MaxText (MxUniqWords * MaximumWordLength + MxUniqWords + 1)
```

The specified error messages are included in a separate file:

```
/* ErrorMessages.h -- Defines all system error messages.

    Collecting them all in one place makes translation to other languages
    easier.

    Also, each error is uniquely identified by the message name.

*/

char InvalidCharacterMessage[] = "Invalid Character. Not a valid text file.";
char InvalidWordLengthMessage[] = "Word too long.";
char TooManyWordsErrorMessage[] = "Too many words encountered.";
char TooManyNewWordsErrorMessage[] = "Too many new words encountered.";
char TooManySuccessorsErrorMessage[] = "System Error.  Too many successors.";
char InvalidErrorMessageMessage[] = "System Error.  Error message truncated.";
char InvalidWordMessage[] = "System Error.  Word storage corrupted.";
```

```
char FatalErrorMessage[] = "Fatal Error.  Program Terminates.";
char ErrorMessageLengthErrorMessage[] = "Error Message Length Overrun.";
char   BinaryTreeErrorMessage[] = "System Error in Binary Search.  \
Binary Tree Corrupt.";
char TooManyCharactersMessage[] = "Input text too long.";
char InvalidNextRequestMessage[] = "Invalid request for Next Word.";
char InvalidTextIndexRequestMessage[] = "Invalid request for Text Index.";
char InvalidSuccessorRequestMessage[] = "Invalid request for Successor.";
```

The main program simply calls on the relevant functions to perform input, compute the valid successors and output the text using only valid successors:

```
/* TexGen -- Reads in a text file and permutes it by a random walk based
    on successors to matching word pairs.

    This program is a re-implementation of the “Markov Algorithm” from
    Kernighan and Pike, 1999.  The original implementation was designed in
    a manner that did not allow testing of constraints.  This implementation
    provides for bounded conditions and ensures that the constraints are
    enforced.

    Alan A. Jorgensen
    August, 1999.

*/
```

```
#define DE BUG

#include “Parameters.h”
```

```
main ()
    {
    int w;

    InputWords();

#ifdef DEBUG
    printf("A sorted list of the input words.\n");
    PrintTree();
#endif

    ListValidSuccessors();


#ifdef DEBUG

    printf("Indexed list of all words, link to next identical word,\n");
    printf("and link to first occurance of the pair.\n");
    for (w = 1;w <= Words; w++)
       printf("%d %s %d %d\n",w,&Text[WordList[w].TextIndex], Next(w),
          First(w));

    printf("A list of all word pairs in sequence with choice counts.\n");
    for (w = 1; w < Words; w ++)
       {
       int s;
       printf("%s %s %d ", &Text(w), &Text(w+1), Successors(w+1));
       for (s = 0; s < Successors(w+1); s++)
```

```
          printf("%d ",SuccessorList[Successor(w+1)+s]);
       printf("\n");
       }
    printf("This is the permuted output list.\n");
#endif

    OutputPermutedText();
    }

/* OutputPermutedText -- outputs text by a random walk through the source text.
    A choice of path occurs when the are duplicate word pairs. */

OutputPermutedText()
    {
    int Word = 1;
    int Words = NumberofWords();
    int WordsOutput = 0;
    while ((Word <= Words)
    &&(WordsOutput < MaximumWordsOutput))
        {
#ifdef DEBUG
        printf("%d ", Word);
#endif
        PrintWord(Word);
        WordsOutput++;
        Word = ValidSuccessor(Word);
        }
    }
```

The first component is the Text Component:

```
/* TextComponent.c -- Component that processes and controls access to the
    Words and Word Text. */

#define DE BUG

#include <stdio.h>
#include “Parameters.h”
#include “Errors.h”
#include “ErrorMessages.h”

int StoredCharacters;                    /* Number of Text characters used */
char Text[MaxText];

typedef struct
    {
    int TextIndex;  /* Index to Text */
    int Lesser;       /* Index to Word List of Lesser Word */
    int Greater;      /* Index to Word List of Greater Word */
    int Next;    /* Link to next Identical Word */
    } WordType;
```

```
/* Access functions for the Word List structure */

#define TextIndex(N) (WordList[N].TextIndex)
#define Lesser(N) (WordList[N].Lesser)
#define Greater(N) (WordList[N].Greater)
#define Text(N) (Text[TextIndex(N)])
#define Next(N) (WordList[N].Next)

int Words;                  /* Number of Words Used */
WordType WordList[MaximumNumberofWords];          /* Word structure */

int NumberofWords()
    {
    return Words;
    }

int NextIdenticalWord(Word)
    {
    if (Word <= Words) return Next(Word);
    else
       SetError(InvalidNextRequestMessage);
       ErrorExit;
    }
```

```
int TextPointer(Word)
    {
    if (Word <= Words) return TextIndex(Word);
    else
       SetError(InvalidTextIndexRequestMessage);
       ErrorExit;

    }

PrintWord(Word)
    {
    int CharacterPosition;  /* Position of current character in Text */
    int WordLength; /* Current Length of the current word. */

    if ((Word < 1) || (Word > Words))
       {
       SetError(InvalidWordMessage);
       ErrorExit;
       }
    for (CharacterPosition = TextIndex(Word), WordLength=0;
       (Text[CharacterPosition] != 0) && (WordLength < MaximumWordLength);
       WordLength++, CharacterPosition++ )
       printf("%c",Text[CharacterPosition]);
    printf("\n");
    }
```

```
InputWords()
    {
    int CC = 0;           /* Current character; may be end of file, but
                        not to start with.  */

    int DupWord;        /* The word list index of the same word as the
                        current word. */

    int WordSize;       /* Number of characters in the current word. */

    int NewWords;       /* The number of Unique Words encountered. */

    Words = 0;
    NewWords = 0;
    StoredCharacters = 0;

    /* Start the Text table with a null so that all words start with and
       end with a null */
    Text[StoredCharacters++] = 0;

    Text[StoredCharacters] = 0;    /* Start with a null word */

    CC = getchar();

    /* Skip White Space */

    while (WhiteSpace(CC)) CC = getchar();
```

```
/* Read in the next word */

while (!EndFile(CC))
   { /* start a new word */

   if (Words >= MaximumNumberofWords)
      {
      SetError (TooManyWordsErrorMessage);
      ErrorExit ;
      }
   if (NewWords >= MaximumNumberofUniqueWords)
      {
      SetError (TooManyNewWordsErrorMessage);
      ErrorExit ;
      }

   /* Start the Word list with word 1 so that we can use word zero as
      the null pointer.  In addition Word[0] is null. */

   Words += 1;
   NewWords += 1;
   WordSize = 0;

   Lesser(Words) = 0;
   Greater(Words) = 0;
   Next(Words) = Words;
   TextIndex(Words) = StoredCharacters;
```

```
      /* TextIndex(Words) is the index of the current word text. */

      while (!EndFile(CC) && !WhiteSpace(CC) && (StoredCharacters < MaxText))
         {   /* we have a new character that is not white space */

         if (! ValidASCII(CC))
            {

#ifdef DEBUG
            if (Words > 2)
                printf("%d after %s %s %s\n",CC, &Text(Words-2),
                   &Text(Words-1), &Text(Words));
#endif
            SetError(InvalidCharacterMessage);
            ErrorExit;
            }

         /* Exit if the current word is too long. */

         if (WordSize >= MaximumWordLength)
            {
            SetError(InvalidWordLengthMessage);
            ErrorExit;
            }

         /* Exit if we have exceeded the character storage limitation. */

         if (StoredCharacters >= MaxText)
```

```
      {
      SetError(TooManyCharactersMessage);
      ErrorExit;
      }

   /* A valid character has been read. */

   WordSize++;

   /* Store the character in the text buffer.  If it is a
      duplicate word, we will remove it later. */

   Text[StoredCharacters++] = CC;
   CC = getchar() ;
   }

/* The word is now stored */

/* Terminate the word string if there is sufficient memory. */

if (StoredCharacters >= MaxText)
   {
   SetError(TooManyCharactersMessage);
   ErrorExit;
   }
else Text[StoredCharacters++] = 0;

/* Delete the word and re-index if it is a duplicate */
```

```
   DupWord = BinarySearch(Words);

   if (DupWord != Words)
      { /* Word is Duplicate */
         /* Delete the Text */
         StoredCharacters = TextIndex(Words);
         TextIndex(Words) = TextIndex(DupWord);
         Next(Words) = Next(DupWord);
         Next(DupWord) = Words;
         NewWords--;
      }

   /* Skip additional white space characters */

   while (WhiteSpace(CC)) CC = getchar();

   }
}
```

```
/* BinarySearch.c  -- searches for duplicate word entries and returns the
    word list index of the duplicate.  If the word is not already present
    the word is inserted and the word list index to it is returned. */

int BinarySearch()
    {
    int Compare;    /* Results of string compare, -1,0,1 */
    int Node = 1;       /* Start with the first word of the word list */

    /* Next(Node) >= 1 tells us that the pointer is correct
       TextIndex(Node) then points to the word in Text, The current word
       text is at TextIndex(Words).  */

    while (Node != 0)
       { /* The node exists. */

       /* The strncmp routine will compare the prefixes of strings.  The
       system is designed to prohibit strings greater than the Maximum
       Word Length but if the system memory becomes corrupt, strncmp
       will deliver an incorrect result and not detect the memory
       corruption. */

       Compare = strncmp( &Text(Words), &Text(Node), MaximumWordLength);

       if (Compare == 0) return Node;  /* The current node matches. */

       if (Compare < 0)
          if (Lesser(Node) == 0)
```

```
            {
            Lesser(Node)  = Words;
            return (Words);
            }
         else Node = Lesser(Node);
       else if (Greater(Node)== 0)
            {
            Greater(Node) = Words;
            return (Words);
            }
         else Node = Greater(Node);
       }
    SetError(BinaryTreeErrorMessage);
    ErrorExit;
    }

#ifdef DEBUG
PrintSubtree (int N)
    {
    if (N == 0) return;
    PrintSubtree(Lesser(N));
    printf("%s\n",&Text(N));
    PrintSubtree(Greater(N));
    }
```

```
PrintTree ()
    {
    PrintSubtree(1);
    }
#endif
```

and the second is the Successors Component:

```
/* SuccessorComponent.c -- Computes and manages the successor list. */

#define DE BUG
/*

    Each Word except the first and last starts a two word prefix.  We must
    count the total number of prefixes that match that word (and its
    successor) and create pointers to the addition duplicate prefixes in the
    Successor list.  No pointers are placed in the Successor list for words
    that do not start a duplicated prefix.  */

#include “Parameters.h”
#include “Errors.h”
#include “ErrorMessages.h”

int SuccessorList[MaximumNumberofWords];

struct
    { int Successor;
      int ValidSuccessorWords;
      int First;    /* Index to the first word of the first occurrence of a
                     matching pair */
    } Successors[MaximumNumberofWords];
```

```
/* Valid Successor Access Functions */

#define Successor(Word) (Successors[Word].Successor)
#define ValidSuccessorWords(Word) (Successors[Word].ValidSuccessorWords)
#define First(Word) (Successors[Word].First)

#ifdef DEBUG
PSL(s,n)
    {
    int i;
    for (i=0;i<n;i++)
    printf("%d ",SuccessorList[s+i]);
    printf("\n");
    }
#endif
```

```
ListValidSuccessors()
    {
    int successors = 0;  /* The total number of successors assigned in the
              Successor List. */
    int Word;         /* The current word number. */
    int Words = NumberofWords();  /* The total number of words input
*/
    /* Reset all successor counts */

    for (Word = 1; Word < Words; Word++)
       ValidSuccessorWords(Word) = 0;
```

```
/* List the successors for all words.
   ( A null word has been defined for Word 0). */

for (Word = 1; Word < NumberofWords() ; Word++)
   {
   if (ValidSuccessorWords(Word) == 0)
      { /* We have not encountered this word pair before so search
         for matching pairs */
      int NextWord = NextIdenticalWord(Word);
      /* Next Word that equals this word */
      First(Word) = 0;
      while (NextWord != Word)
         {
         if (ValidSuccessorWords(NextWord) == 0)
             { /* We haven't already counted this one */
             /* The words are the same if the Text Indices are
                Identical. */
             if (TextPointer(NextWord-1) == TextPointer(Word-1))
                { /* The prior word is the same.
                     There are now at least two matching pairs of
                     words. */
                First(Word) = Word;
                if (ValidSuccessorWords(Word) == 0)
                     { /* Start a Successor list */
                     Successor(Word) = successors;
                     SuccessorList[successors++] = Word+1;
                     SuccessorList[successors++] = NextWord+1;
```

```
                    ValidSuccessorWords(Word) += 2;
                    }
                else
                    { /* The Successor List is already started. */
                    SuccessorList[successors++] = NextWord+1;
                    ValidSuccessorWords(Word) += 1;
                    }
                /* Mark next word as counted */
                ValidSuccessorWords(NextWord) = 1;
                /* Refer each pair to the first occurrence of the
                    pair */
                First(NextWord) = Word;
                }
            }
         NextWord = NextIdenticalWord(NextWord);
         }
      }
   if (ValidSuccessorWords(Word) == 0)
      { /* There were no matching pairs */
      ValidSuccessorWords(Word) = 1;  /* Mark it as counted */
      }
   }
}
```

```
int ValidSuccessor(int Word)
    {
    if (Word > NumberofWords())
       {
       SetError(InvalidSuccessorRequestMessage);
       ErrorExit;
       }

    if (First(Word) != 0)
        {   /* This is not the first place that this matching pair is
           ** found.  So locate the first occurrence of this matching pair
           ** where the count is kept. */
        Word = First(Word);

        /*
        ** There is more than one choice, so pick one
        */
        Word = SuccessorList[Successor(Word)
             + rand() % ValidSuccessorWords(Word)];
        }
    else
        Word++;
    return Word;
    }
```

The remaining detail is the error handler functions with the appropriate header definition file:

```
/* Errors.h --  Error system include file */

/* Definition of path for error messages */

#define ErrorDevice stdout
#define DisplayDevice stdout

/* Macro to set error message code */

#define SetError(Message) seterror(Message)

/* Macro to display the error message and exit process with error code. */

#define ErrorExit errorexit()

/* Macro to return an error message to caller */

#define Return(ErrorMessage) {ErrorCode = ErrorMessage; return;}

/* Macro to test for existence of error */

#define Error (ErrorCode != NULL)

/* Macro to display error message and reset error code. */
```

```
#define DisplayError displayerror()


/* Errors.c -- Error handling package.  Used in conjunction with the error
    Macros in Errors.h */

#include <stdio.h>
#include “Errors.h”
#include “Parameters.h”
#include “ErrorMessages.h”

/* LineLength, though 80 characters, can only accommodate 79 characters
without line wrap.

To constrain output to no more that 79 characters, for configurable line
length we need a dynamic format string as follows.
*/

char LineLengthFormat[35] = “%.79s\n” ;

#ifndef LineLenth
#define LineLength 80
#endif
```

```
/* A function is provided to redefine the maximum output message length. */

SetFormat()
    {
    sprintf(LineLengthFormat, “%%.%ds\n”, LineLength - 1);
#ifdef DEBUG
    printf(“%s”, LineLengthFormat);
#endif
    }
```

```
/* Error Output

This component displays all messages to the operator and constrains
output to the required display area.  */

Output(FILE *Destination, char Message[])
    {   /* Output message but truncated to allowed length */
    fprintf(Destination, LineLengthFormat, Message);

    /*
    ** Fatal error if Message is too long
    */
    if (strnlen(Message, LineLength) > LineLength - 1)
        {
        SetError(ErrorMessageLengthErrorMessage);
        ErrorExit;
        }
    }
```

```
/* Global Error System */

char * ErrorCode = NULL;

/* Since C does not provide a strnlen function and strlen can address out
of bounds with an improperly terminated string, we provide a strnlen
function that allows bounding the string length search. */

int strnlen(char s[], int l)
    {
    int c;
    for (c = 0; c < l; c++)
        if (s[c] == 0)
            return c;
    return l;
    }
```

```
/* Procedure to display the error message and exit process with error
** code. */

errorexit()
    {
    DisplayError;

    /*
    ** We do not use “Output” here because Output calls error exit on an error.
    */
    fprintf(ErrorDevice, “%s\n”, FatalErrorMessage);
    exit(1);
    }

/* Procedure to set error message code */

seterror(char * Message)
    {
    ErrorCode = Message;
    }

/* Procedure to display error message and reset error code. */

displayerror()
    {
    Output(ErrorDevice, ErrorCode);
    ErrorCode = NULL;
    }
```