An API Testing Method

Alan A. Jorgensen and James A. Whittaker

Advanced Engineering Technology, Incl
150 West University BoulevardMelbourne,
Florida 32901aaj@aet-usa.com

Florida Tech—Computer Science
150 West University Boulevard
Melbourne, Florida 32901
jw@cs.fit.edu

This paper presents a technique to test APIs that combines aspects of two published software testing methods, namely Markov modeling and category partitioning. Markov modeling provides a basis for model based testing. Markov modeling establishes the context for generating API calls and call sequences within a single test case. For modeling purposes, each combination of parameter values for each function call is a unique "input." Category partitioning helps select parameter values and effective combinations of multiple parameters for individual API calls. Small examples demonstrate these techniques and two case study summaries illustrate its effectiveness. One case under laboratory conditions established proof-of-concept and the other applicability to a large commercial API. Some aspects of these techniques are manually intensive and suggest a need for automation.

Introduction

An API (Application Programming Interface) is a collection of software functions and procedures, called *API calls*, that can be executed by other software applications. Application developers code that links to existing APIs to make use of their functionality. This link is seamless and end-users of the application are generally unaware of using a separately developed API.

During testing, a test harness—an application that links the API and methodically exercises its functionality—is constructed to simulate the use of the API by end-user applications. The interesting problems for testers are:

This work was sponsored in part by a grant from Microsoft Research and in part by a grant from the Army Research Lab.

1. Ensuring that the test harness varies parameters of the API calls in ways that verify functionality and expose failures. This includes assigning common parameter values as well as exploring boundary conditions.

2. Generating interesting parameter value combinations for calls with two or more parameters.

3. Determining the content under which an API call is made. This might include setting external environment conditions (files, peripheral devices, and so forth) and also internal stored data that affect the API.

4. Sequencing API calls to vary the order in which the functionality is exercised and to make the API produce useful results from successive calls.

This paper addresses the construction of this test harness by combining two published testing techniques that, as an ensemble, address the major problems posed by API testing. Specifically, we use the category-partition technique to directly address parameter selection and variation (item 1) and parameter combination (item 2), and Markov modeling to address the analysis and enumeration of software and environment states (item 3) and generating input sequences (item 4).

In the next section, we review the essential points of each of these techniques. We refer to the original work of Ostrand and Balcer [1] and Whittaker [2] to keep this review concise. Next, we proceed by describing the hybrid technique and illustrating its application. Finally, we present results from two case studies to provide proof-of-concept. The first case study is a collection of platform-independent, public domain Ada routines, called *AdaGraph*, which consist of about 30 API calls that implement basic drawing and graphing functions. The second is a more modern, and much larger, C++ API on the Windows platform, Microsoft's Collaborative Data Objects (CDO) API that has 120 calls to handle messaging for Windows applications.

A Hybrid Technique for API Testing

Refer to [1] and [2], respectively, for complete details about these techniques.

Category partitioning

The category-partition technique is a general-purpose functional testing method that goes through a series of decompositions based on characteristics of the input domain. For our purposes, there are three basic steps to category-partitioning: 1) create a set of *categories* that describe properties of inputs; 2) partition the categories into *choices* that enumerate specific values or value-ranges that inputs can assume; and 3) determine constraints among the choices that describe how the choices interact. The following paragraphs describe each of these steps in detail.

Suppose a routine takes as input a character string. Step 1 of the category-partition method requires thinking through attributes of this input that have an effect on the behavior of the routine. For example, pertinent categories might be the *length* of the string and the *content* of the string because both of these attributes will have to be varied in order to properly test the routine. In general, categories are not detailed, instead they represent high-level attributes of the inputs.

Step 2 adds more detail to the categories by further partitioning them into specific values or value-ranges that can be assigned to the input. For example, the *length* category could be assigned a value of 32 or a range of values, say, 1-16. We use the following guidelines in choosing values and value-ranges:

- Create a choice for each value or value-range that forces a default setting to occur. For example, if the default is to ignore spaces in the string, then we'd want choices that specify that spaces be present in the input so that this behavior is tested.
- Create a choice for each value that causes the output to change in some substantive way. Output could be a return value for a function call or visible displayed results. For example, if the length of the string is set to 0, it is illegal and an error message will be displayed. In this case, zero should be a choice for the length category so that this behavior get tested.
- Create a choice for each value on or around distinguishable "boundaries." Obviously, 0 is again a choice for the length category and so is 1 since it is adjacent to the actual boundary.

Step 3 consists of writing constraints that describe how choices for one category affect choices for another category. For example, a constraint that specifies that an input choice cannot have embedded spaces *and* cannot be of length 0 is required to keep us from considering impossible choice combinations. Conversely, if we want to force certain choice-combinations to occur, then a constraint can also be written to achieve this.

When category partitioning is complete, a single input, like the string example above, might be partitioned into any number of actual values that are necessary for testing. When performing category-partitioning on an API call with multiple parameters, this number can grow quite large, for example, in their original paper, Ostrand and Balcer [1] partitioned a simple find command with two parameters that resulted in 40 individual calls, each with a different parameter variation.

Category Partitioning Example

The "Beep" function of MS Windows provides us a simple illustrative example. From the Visual Studio documentation we can obtain the specification for "Beep:"

"The **Beep** function generates simple tones on the speaker. The function is synchronous; it does not return control to its caller until the sound finishes.

```
BOOL Beep(
  DWORD dwFreq,        // sound frequency, in hertz
  DWORD dwDuration  // sound duration, in milliseconds
);"
```

The parameter *dwFreq* specifies the frequency of the sound in hertz and must be in the range 37..32,767. The parameter *dwDuration* specifies the sound duration in milliseconds.

When successful the function returns a nonzero integer. Otherwise it returns zero and establishes an additional error code accessable by **GetLastError**.

For each of these parameters we can establish categories based on the specification:

Categories(dwFreq) = {Too Low, Nominal, Too High}

Category(dwFreq) $\in$ Categories(dwFreq)

Category(dwFreq) = Too Low iff DWORDMin <= dwFreq < 37

Category(dwFreq) = Nominal iff 37 <= dwFreq <= 32767

Category(dwFreq) = To High iff 32767 < dwFreq <= DWORDMax

Where DWORDMin is the minimum value represented within the type DWORD and DWORDMax is the maximum value within this type.

Again, from the specification, two categories are implied since the idea of negative time is still the stuff of science fiction.

Categories(dwDuration) = {Ridiculous, Milliseconds}

Where

Category(dwDuration) $\in$ Categories(dwDuration)

Category(dwDuration) = Ridiculous iff dwDuration < 0

Category(dwDuration) = Milliseconds iff 0 <= dwDuration <= DWORDMax

What category partitioning says is that the *way* the system responds depends on only the category. The system will respond in the same manner for dwFreq = 37 as it does for dwFreq = 32767 and all values in between (given that dwDuration does not change).

These categories are based on the specification. We will see that there are other categories, unspecified. Someone must decide whether these additional categories are bugs or undocumented features.

Assuming for the moment that we have captured the categories of Beep, and to ensure that the partition values of each category are correct, we need two input conditions for each category. This is true for categories that are ranges, as is the case in this example. Further, when more than one variable can be presented, then the input conditions are the cross product of the category partition values as shown in Table 1.

• Table 1 Beep Category Partitioning

Category (dwFreq)	Partition (dwFreq)	Category (dwDuration)	Partition (dwDuration)
Too Small	DWORDMin	Ridiculous	DWORDMin
			-1
		Milliseconds	0
			DWORDMax
	36	Ridiculous	DWORDMin
			-1
		Milliseconds	0
			DWORDMax
Nominal	37	Ridiculous	DWORDMin
			-1
		Milliseconds	0
			DWORDMax
	32767	Ridiculous	DWORDMin
			-1
		Milliseconds	0
			DWORDMax
Too Large	32768	Ridiculous	DWORDMin
			-1
		Milliseconds	0
			DWORDMax
	DWORDMax	Ridiculous	DWORDMin
			-1
		Milliseconds	0
			DWORDMax

We can specify the nature of the BOOL function as follows:

```
code
#include <windows.h>
end

Spec BOOL Worked Beep(DWORD dwFreq, DWORD dwDuration);
```

From this we can develop code using a tool currently under development to generate a test wrapper to call this function from the command line.

This reduces to the following test cases (and results)

```
Executing: DWORDMin = -40000

Executing: DWORDMax = 40000

Executing: Beep DWORDMin
DWORDMin
Worked: FALSE

Executing: Beep DWORDMin -1
Worked: FALSE

Executing: Beep DWORDMin 0
Worked: FALSE

Executing: Beep DWORDMin
DWORDMax
Worked: FALSE

Executing: Beep 36 DWORDMin
Worked: FALSE

Executing: Beep 36 -1
Worked: FALSE

Executing: Beep 36 0
Worked: FALSE

Executing: Beep 36 DWORDMax
Worked: FALSE

Executing: Beep 37 DWORDMin
Worked: TRUE

Executing: Beep 37 -1
Worked: TRUE

Executing: Beep 37 0
Worked: TRUE

Executing: Beep 37 DWORDMax
Worked: TRUE

Executing: Beep 32767 DWORDMin
Worked: TRUE

Executing: Beep 32767 -1
Worked: TRUE

Executing: Beep 32767 0
Worked: TRUE


Executing: Beep 32767 DWORDMax
Worked: TRUE

Executing: Beep 32768 DWORDMin
Worked: FALSE

Executing: Beep 32768 -1
Worked: FALSE

Executing: Beep 32768 0
Worked: FALSE

Executing: Beep 32768 DWORDMax
Worked: FALSE

Executing: Beep DWORDMax
DWORDMin
Worked: FALSE

Executing: Beep DWORDMax -1
Worked: FALSE

Executing: Beep DWORDMax 0
Worked: FALSE

Executing: Beep DWORDMax
DWORDMax
Worked: FALSE
```

This is not, however, the full story. The assumption of category partitioning is that all values within the partition produce the same result. That is to say, the functional results of Beep 0 0 should be the same as the results for Beep 36 0. since 0 and 36 are in the same category of dwFreq,Too Low.

```
Executing: Beep 36 0

Worked: FALSE
```

```
Executing: Beep 0 0

Worked: TRUE
```

This is the discovery of a new partition of dwFreq expanding the definition to:

Categories(dwFreq) = {Too Low, Zero, Nominal, Too High}

Category(dwFreq) $\in$ Categories(dwFreq)

Category(dwFreq) = Too Low iff DWORDMin <= dwFreq < 37 AND dwFreq $\neq$ 0.

Category(dwFreq) = Zero iff dwFreq = 0

Category(dwFreq) = Nominal iff 37 <= dwFreq <= 32767

Category(dwFreq) = To High iff 32767 < dwFreq <= DWORDMax

The result "Executing: Beep 32767 –1, Worked: TRUE" is unexpected. The function returns true (and produces a tone) for negative durations.

Markov Modeling

A Markov model for software testing is a couple (S, δ) where S is the set of all *operational states* of a software system and δ:$S \times I \times [0\ldots1] \rightarrow S$ is the non-deterministic transition function, where I is the set of externally generated inputs of the software. Operational states describe internal or external objects that influence the behavior of the software under test. In general, we are interested in objects that affect the way the software reacts to external stimuli. We say that software is in state j when a collection of objects has certain values and in state k when they have different values. State j is characterized by the allowable external inputs and disabled (or, at least reacted to in a different manner) by the software. State k will have a different set of allowable inputs that mark it as distinct from j.

The transition function δ describes how the application of external input causes state changes within the software. The probability distribution associated with each state represents the operational profile, i.e., the probability that the corresponding inputs will be applied during typical use, for that state. As long as the probabilities sum to one for each $s \in S$, the model is a finite state, discrete parameter Markov chain.

Knowledge of the state space is important to testers for two reasons. First, it allows syntactically valid test cases—allowable sequences of inputs directly applicable to the software—to be machine generated without human intervention. Without knowledge of the state space, many such input sequences otherwise appear invalid, requiring testers to manually edit them so that they will execute. Second, knowing the states makes it easier to predict expected outputs. The output that software produces depends on the input applied and the current state of the software. Since testers track the software's state as testing progresses, it is more straightforward to predict the expected output (which is needed to check for correct behavior, i.e., to develop an oracle).

The *operational modes* of the software's support the determination of the states and transition function, S and δ. An operational mode is a variable that abstracts objects that govern the way a software system responds (with output as opposed to computation) to system input. For example the variable "phone status = ringing *or* not ringing" is an operational mode for a phone switch because it governs whether the output for the "take the phone off the hook" input is "connected to caller" or "dial tone," respectively. Assigning a specific value to each operational mode of the system identifies a state, $s \in S$.

Testers begin by enumerating the application's inputs[1] and then systematically investigating possible operational modes. An application generally has a number of operational modes that characterize its behavior. At any given point in the execution of the software, each operational mode will have a specific value from the set of possible values for that mode. As inputs are applied to the software, a change in the value of one or more operational modes can occur, signifying that the application has changed its internal state. We model these states of the software as the state set of a Markov chain. The transitions between states are labeled with external inputs that cause the corresponding state change and an occurrence probability that represents the proportion of time we want that particular input to be generated from that particular state during testing.

Generating test cases from the Markov chain starts by establishing an initial state and then randomly "walking" through state transitions according to the transition probabilities. This Monte-Carlo style simulation effectively generates paths through the chain from a specified start state until some specified termination state occurs. Applying the sequence of inputs directly to the software and monitoring the output constitutes execution of the test case.

The Markov modeling technique models the possible ways to sequence multiple API calls based on external environment considerations and internal, persistent data. The API call-instances determine operational modes that are the basis for determining the states. API call-instances resulting from category-partitioning label the state transitions.

Markov Modeling Example

To illustrate Markov modeling we chose a simplistic subset of the C library file IO commands, fopen, fclose, fprintf, and fgets. Though we should apply category partitioning to each of these functions to provide a thorough test, we consider only a simple subset of the parameter values for each of the commands:

```
Open Read: Handle = fopen("testfile","r");
Open Write: Handle = fopen("testfile","w");
Close: FailCode = fclose(Handle);
Write: Characters = fprintf(Handle,"%s\n","Test Record");
Read: FailCode = fgets(OutputString,500,Handle);
```

We further limit the complexity of our example by only considering three operational modes:

File Status
Records in File
Position

Where
Domain(File Status) = {Does not exist, Closed, Open for Read, Open for Write},
Domain(Records in File) = {0, 1, 2}
Domain(Position) = {0,1,2}

The number of records in our test file is arbitrarily limited to two and this will require that we not allow the "Write" input to occur after writing two records to the file. The Position determine the number of the

[1] Inputs are loosely defined here as "external events generated by users." In the case of most UI software, inputs are keystrokes and mouse clicks. For an API, an input is a function call with a specified set of parameters.

record last read or written and therefore, when a file is opened the position is 0 until a read or write occurs. When “Position” is equal to “Records in File,” the actual file position is at the end of file.

There are many other operational modes. For instance, we could consider concurrent operation of more than one file. There are other operational mode values. For instance, a file can exist, be closed, but be locked by another application. We will keep our model “simple.”

The number of theoretical states is the cross product of the operational modes and, in theory, 4 * 3 * 3 = 27 states. In our example, however, certain combination of operational modes cannot exist. For instance, if the file does not exist, there can never be records in it or the file position can never exceed the number of records.

We identify the state labels with a triple corresponding to the values of the operational mode where:

D = Does not exist,
C = Closed,
R = Open for Read and
W = Open for Write.

The 13 allowed states for our model are (“-“ denotes “don’t care”):

D---	File does not exist; There are no records and no position.
C0--	File exists but has no records.
C1--	File exists with one record.
C2--	File exists with two records.
W00	Open for writing with no records in the file. Positioned at end of file.
W11	Open for writing and one record in the file. Positioned at end of file.
W22	Open for writing and two records in the file. Positioned at end of file.
R00	Open for reading with no records in the file. Positioned at end of file.
R10	Open for reading with one record but positioned at the beginning of the file.
R11	Open for reading with one record in the file but positioned at the end of file.
R20	Open for reading with two records in the file but positioned at the beginning of the file.
R21	Open for reading with two records and the first record has been read.
R22	Open for reading with two records and positioned at the end of file.

State transitions are the cross product of the states and the inputs. Table 2 is the state transition table for the stdio subsystem. Even for this simple state model, the diagram that illustrates the state model is too complex for representation here. This table makes certain assumptions about how this system of functions should operate. For instance, the table assumes that an attempt to open a file that is already open will not affect the state. Similarly, if a file is open for reading (or writing) and attempt to write (or read) will be rejected and therefore not cause a change of state. Alas, as we shall, see, this does not model the reality of these functions.

• Table 2 - Desired State Transitions, stdio Subsystem

Input/State	**D0-**	**C0-**	**C1-**	**C2-**	**W00**	**W11**	**W22**	**R00**	**R10**	**R11**	**R20**	**R21**	**R22**
Open Read	(1)	R00	R10	R20	R00	R00	R00	R00	R10	R10	R20	R20	R20
Open Write	W00	W00	W00	W00	W00	W00	W00	W00	W00	W00	W00	W00	W00
Close	D0-	C0-	C1-	C2-	C0-	C1-	C2-	C0-	C1-	C1-	C2-	C2-	C2-
Write	D0-	C0-	C1-	C2-	W11	W22	(2)	R00	R10	R11	R20	R21	R22
Read	D0-	C0-	C1-	C2-	W00	W11	W22	R00	R11	R11	R21	R22	R22

Notes:

(1) Creates a NULL file handle
(2) An output is not allowed in this state because it would create more than two records.

The actual case is somewhat different. In reality, the assumption that the *stdio* subsystem rejects calls in conflict with the current state is false. For example, when we open a file for writing that is already open for writing and has two records, it will be destroyed. Table 3 represents the state transition table as actually measured. By examining each possible transition and monitoring the correctness of the state achieved by that transition we discover very interesting properties of the actual system. Some of those properties should be considered "bugs." (The behavior of this system is not actually specified; e.g., there is no specification for the behavior of the system when a file is opened for read and a file write occurs.)

Table 3 - Actual State Transitions, stdio Subsystem

Input/State	**D0-**	**C0-**	**C1-**	**C2-**	**W00**	**W11**	**W22**	**R00**	**R10**	**R11**	**R20**	**R21**	**R22**
Open Read	(1)	R00	R10	R20	(5)	(5)	(5)	R00	R10	R11	R21	R22	R22
Open Write	W00	W00	W00	W00	W00	W11	W22	(6)	(6)	(6)	(6)	(6)	(6)
Close	D0-	C0-	C1-	C2-	C0-	C1-	C2-	C0-	C1-	C1-	C2-	C2-	C2-
Write	D0-	C0-	C1-	C2-	W11	W22	(2)	R00	R10	R11	R20	(4)	R22
Read	D0-	C0-	C1-	C2-	W00	(3)	(3)	R00	R11	R11	R21	R22	R22

Notes:

(1) Creates a NULL file handle
(2) An output is not allowed in this state because it would create more than two records.
(3) An undefined record is written to the output file.
(4) The input file pointer is advanced by the write command. Subsequent read is incorrect.
(5) State transitions to some that appears to be **R00**
(6) State transitions to some that appears to be **W00**

Thus we have seen that by analyzing just the inputs of a system we can locate category portioning bugs and by analysizing state, we can identify state transition bugs. We extend these capabilities but combining these two techniques.

The Hybrid Technique

We define a hybrid technique that uses both category-partitioning and Markov modeling to test an API. Category-partitioning is applied first to each individual API call to analyze and partition the parameters of the call into interesting choices and value-combinations. A single API call with a few parameters will be transformed into a potentially large number of specific instances of the call where each instance has a different set of parameter choices and values. This set of API call instances determines the state set of the Markov model. We perform category-partitioning separately on each parameter of each call within an API. The selected parameter value sets for each call are values selected from the cross products of the category partitions of each parameter set. Ideal values are category range limiting values.

The Hybrid Example

We can apply category partitioning to each of the parameters of each of the calls to extend the Markov modeling technique to the hybrid technique. For the purposes of example only, we shall apply category partitioning only for the file name parameter of the fopen for write command. So far we have only considered only “testfile” for the name. The file name parameter is a string and therefore can have multiple categories over the string properties length, alphabet, syntax and semantics. For instance, what can happen with a file name of zero length? Of length 10,000? File name specifies a path name which has certain syntax and semantic requirements. What happens when that is violated? Consider

```
Handle = fopen (“..\..\..\..\..\..\..\..\..\..\..\..\testfile”,”w”);
```

(One would predict that requesting the creation of a file “beneath the root” would be invalid; in fact, however, this opens a file in the root directory.)

We can consider the category of valid names over the printable ASCII alphabet that are a single character long. We can group these into “valid” and “invalid” names. We expect that the behavior of the system for valid names is consistent with the model already defined, but by adding the new category of file names, we would expect different behavior. The question is, which names are valid and which names are not? Some experimentation may be required when the specification of validity is not available.

For our simple model, we have chosen calls that pass strings. The discourse above about the properties of names applies in general to strings and we can consider strings to have at least two categories, valid and invalid. When we specify multiple parameters, we must allow for the cross product of the input categories of all parameters. The fopen input used in the model so far considered only valid strings for filename and mode and so for our hybrid model, in addition to the input specified we must also consider input of the following categories:

fopen valid invalid

fopen invalid valid

fopen invalid invalid

Table 4 expands the transition table to include the category partitioning of the parameters for each call. When we discover additional categories, we must expand the transition table to include the new input categories. Similarly, these new input categories might create new states, and for each new state, transitions must be determined for each input.

Table 4 - Competed Hybrid stdio State Transition Table

Input/State	D0-	C0-	C1-	C2-	W00	W11	W22	R00	R10	R11	R20	R21	R22
Open Valid Read	(1)	R00	R10	R20	(5)	(5)	(5)	R00	R10	R11	R21	R22	R22
Open Invalid Read	(1)	C0-	C1-	C2-	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)
Open Valid Write	W00	W00	W00	W00	W00	W11	W22	(6)	(6)	(6)	(6)	(6)	(6)
Open Invalid Write	D0-	C0-	C1-	C2-	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)
Open Invalid Invalid	D0-	C0-	C1-	C2-	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)
Close	D0-	C0-	C1-	C2-	C0-	C1-	C2-	C0-	C1-	C1-	C2-	C2-	C2-
Write Valid	D0-	C0-	C1-	C2-	W11	W22	(2)	R00	R10	R11	R20	(4)	R22
Write Invalid	D0-	C0-	C1-	C2-	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)
Read	D0-	C0-	C1-	C2-	W00	(3)	(3)	R00	R11	R11	R21	R22	R22

Notes:

(7) The next state has not been determined nor estimated.

Actual testing practice should test differing valid and invalid strings. Where possible, testing should include exhaustive patterns; but when this is not possible random selection from the possible choices is better than repeating values that have previously been tested. Where discrete boundaries are defined, such as the length of a string, the lengths precisely at and precisely immediately above (or below) those discrete boundaries should be tested.

Differing values of "valid" strings causing different system behavior from the same state belong to different partitions and should be modeled separately.

Results from a Laboratory Experiment

Students at FIT successfully applied this technique to AdaGraph. The AdaGraph API is a set of public domain Ada routines providing basic drawing capabilities for Ada applications.

Twenty-one API routines were included in the test. The test team chose six states for AdaGraph with 13 possible transitions. The average number of parameters per API call was 2.54. Category partitioning identified 215 different inputs.

random selection of the arcs of the state model generated the test cases for this project. The test case generation completion criterion was the coverage of all transitions (and all states).

Results from an Industrial Experiment

The Collaborative Data Objects (CDO) API is organized as a collection of twenty software objects, implemented in C++, that handle messaging tasks for Microsoft Windows applications. There is a variety of applications of CDO, including electronic mail.

The initial model contained 34 states with 65 distinct inputs and a total of 1061 transitions.

The student test team developed five additional models:

• Table 5 - CDO Models, Parameters

States	Transactions	Inputs
103	1337	33
44	658	34
52	2218	82
17	367	36
36	739	49

Test Results

Each product we tested had already been tested by their respective vendors and released. Therefore, we did not anticipate that we would find a huge number of failures. We performed the laboratory experiment without the benefit of access to the AdaGraph developers (thus, the problems we encountered cannot be verified as actual product failures), however, we received full cooperation from Microsoft developers for CDO. Faculty and students at Florida Tech conducted both projects.

A single student modeled and tested AdaGraph, with minimal assistance and as a first-time testing effort and over a four-month period as a part-time assignment. A team of six part-time students modeled and tested CDO, also as a first time testing effort, over a 9-month period.

The AdaGraph project discovered two potential failures during the execution of the test.

Destroy_Graph_Window:

When an attempting to create a new window after calling Destroy_Graph_Window to delete a window, the new window is not, in fact, created. The program terminates without an error message.

Goto_XY:

Goto_XY accepts coordinate parameters specifying a point outside of the selected window without indicating an error.

The first simple model of CDO uncovered 3 serious defects reported to and corrected by Microsoft. Microsoft received a confidential report from FIT on the defects uncovered by the additional models.

Acknowledgments

The authors thank the Florida Tech students who participated in this project, including S. Al-Maati, J. Chipi, M. Douglas, I. El-Far, J. Kim, N. Roberts, J. Varghese and G. Wout, and particularly Jeonseok Kim and Alvin Flanegan who were the student directors for the AdaGraph and CDO experiments respectively. We also acknowledge the many discussions with B. Jensen, H. Jensen and J. Tierney of Microsoft Corp. who offered advice and criticism during this project. A special thanks to Joanne McAlpine for her proof reading skills, though the remaining mistakes are ours alone.

References

1. T. Ostrand and A. Balcer, The category-partition technique for specifying and generating functional tests, *Communications of the ACM*, *20*, 10 (1988) 676-686.

2. J. A. Whittaker, Stochastic software testing. *Annals of Software Engineering 4*, (1997) 115-131.