Input Validation Testing: A Requirements-Driven, System Level, Early Lifecycle Technique *

Jane Huffman Hayes
Innovative Software Technologies
Science Applications International Corp.
jane.e.hayes@cpmx.saic.com

A. Jefferson Offutt
Information and Software Engineering
George Mason University
ofut@gmu.edu

August 30, 2000

Abstract

This paper addresses the problem of statically analyzing input command syntax as defined in interface and requirements specifications and then generating test cases for input validation testing. The IVT (*Input Validation Testing*) technique has been developed, a proof-of-concept tool (MICASA) has been implemented, and validation has been performed. Empirical validation on actual industrial software (for the Tomahawk Cruise Missile) shows that as compared with senior, experienced testers, MICASA found more requirement specification defects, generated test cases with higher syntactic coverage, and found additional defects. Additionally, the tool performed at significantly less cost.

Jane Huffman Hayes and A. Jefferson Offutt. Input Validation Testing: A Requirements-Driven, System Level, Early Lifecycle Technique. *11th International Conference on Software Engineering & its Applications*, Paris France, December 1998.

1 Introduction

Human users often interface with computers through commands. Commands may be in many forms, including mouse clicks, screen touches, pen touches, voice, and files. A method used extensively by older programs and still used widely today is that of obtaining textual user input through the keyboard. In this paper, a *syntax driven application* accepts inputs from the user, constructed and arranged properly, that control the flow of the application. Programs that accept free-form input, interactive input from the user, free-form numbers, etc. are all examples of syntax driven applications [4].

A *user command* is defined as an input from a user that directs the control flow of a computer program. A *user command language* is a language that has a complete, finite set of actions entered textually through the keyboard, used to control the execution of the software system. *Syntax driven software* is software that has a command language interface. Syntax driven applications must be able to (1) properly handle user commands that may not be constructed and arranged as expected, and (2) properly handle user commands that are constructed and arranged as expected.

*This work is supported in part by the U.S. National Science Foundation under grant CCR-98-04111 and by the Command and Control Systems Program (PMA-281) of the Program Executive Officer Cruise Missiles Project and Joint Unmanned Aerial Vehicles (PEO(CU)), U.S. Navy. Special thanks to Mrs. Theresa Erickson.

The first requirement refers to the need for software to be tolerant of operator errors. *Input-tolerance* is defined as an application's ability to properly process both expected and unexpected input values. Test cases should be developed to ensure that a syntax driven application fulfills both of these requirements. *Input validation testing*, then, is defined as choosing test data that attempt to show the presence or absence of specific faults pertaining to input-tolerance.

1.1 System Testing

Though much research has been done in the area of unit testing, system testing has not garnered as much attention from researchers. This is partly due to the expansive nature of system testing: many unit level testing techniques cannot be practically applied to millions of lines of code. There are well defined testing criterion for unit testing [1, 7, 9, 11] but not so for system testing. Lack of formal research results in a lack of formal, standard criteria, general purpose techniques, and tools.

Much of the research undertaken to date has largely concentrated on testing for performance, security, accountability, configuration sensitivity, start-up, and recovery [1]. These techniques require that source code exist before they can be applied. Such dynamic techniques are referred to as *detective techniques* since they are only able to identify already existing defects. What is more desirable is to discover preventive techniques that can be applied early in the life cycle. *Preventive techniques* help avert the introduction of defects into the software and allow early identification of defects when it is less costly and time consuming to repair them.

1.2 Input Validation

Input validation refers to those functions in software that attempt to validate the syntax of user provided commands/information. It is desirable to have a systematic way to prepare test cases for this software early in the life cycle. By doing this, planned user input commands can be analyzed for completeness and consistency. Currently, no well developed or formalized technique exists for automatically analyzing the syntax and semantics of user commands (if such information is even provided by the developers in requirements or design documents) or for generating test cases for input validation testing. The technique proposed here is preventive in that it will statically analyze the syntax of the user commands early in the life cycle. It is also detective since it generates test cases that can be run on the code.

System level testing techniques that currently address this area are not well developed or formalized. There is a lack of system level testing formal research and accordingly a lack of formal, standard criteria, general purpose techniques, and tools. This paper presents a technique to statically analyze input command syntax and generate test cases for input validation testing early in the life cycle. Validation results show that the IVT method found more specification defects than senior testers, generated test cases with higher syntactic coverage than senior testers, generated test cases that took less time to execute, generated test cases that took less time to identify a defect than senior testers, and found defects that went undetected by senior testers.

We present a new method of analyzing and testing syntax-directed software systems that introduces the concept of generating test cases based on syntactic anomalies statically detected in interface requirements specifications. The overall thesis of this research is that the current practice of analysis and testing of software systems described by natural language, textual tabular interface requirements specifications can be improved through the input validation testing technique. Specifically, the IVT technique statically detects specification defects better and faster than senior testers and generates test cases that identify software faults better and more quickly than senior testers.

To evaluate the thesis, a working prototype system based on the new method was constructed and validated using a three-part experiment. Using this prototype system, we empirically established large improvements over current practice in the number of anomalies statically detected in interface requirements specifications, the duration of time needed to develop and execute test cases, the duration of time needed to identify a defect, and identifying specification defects and software faults.

2 The Input Validation Test Method

Input validation testing (IVT) uses a graph of the syntax of user commands. IVT incorporates formal rules in a test criterion that includes a measurement and stopping rule. This section discusses the four major aspects of the IVT method: (1) how to specify the format of specifications, (2) how to analyze a user command specification, (3) how to generate valid test cases for a specification, and (4) and how to generate error test cases for a specification.

IVT uses a test obligation database, a test case table, and a Microsoft Word file. A *test obligation* is a defect or potential software problem that is detected during static analysis. If a defect is found (such as an overloaded token value), information on the specification table, the data element, and the defect are stored in the test obligation database. Each record represents an obligation to generate a test case to ensure that the static defect detected has not become a fault in the finished software. A test case is generated for each test obligation. The *test case table* is used to record all the test cases that are generated. The *Microsoft Word file* is used to generate test plans and cases in a standard Test Plan format.

2.1 Specifying specification format

The IVT method is specification driven, thus is only useful for systems that have some type of documented interfaces. The IVT method expects a minimum of one data element per user command language specification table (these are also referred to as “Type 1” tables) and expects a minimum of three fields for the data element: (1) data element name, (2) data element size, and (3) expected/allowable values.

2.2 Analyzing user command specifications

A user command language specification defines the requirements that allow the user to interface with the system to be developed. The integrity of a software system is directly tied to the integrity of the system interfaces, both internally and externally [6]. There are three well accepted software quality criterion that apply to interface requirements specifications: completeness, consistency, and correctness [2, 10]. This research only addresses the first two.

Requirements are *complete* if and only if everything that eventual users need is specified [2]. The IVT method assesses the completeness of a user command language specification in two ways. First, the IVT method checks that there are data values present for every column and row of the specification table. Second, the IVT method performs static analysis of the specification tables. The IVT method looks to see if there are hierarchical, recursive, or grammar production relationships between the table elements. For hierarchical and grammar production relationships, the IVT method checks to ensure there are no missing hierarchical levels or intermediate productions. If such defects are detected with the specification table, a test obligation will be generated and stored in the test obligation database. Any recursive relationships detected will be flagged by IVT as confusing to the end user and having the potential to cause the end user to input erroneous data. If recursive relationships are detected with the specification table, a test obligation will be generated and stored in the test obligation database.

Consistency is exhibited "if and only if no subset of individual requirements conflict" [2]. *Internal consistency* refers to conflicts between requirements in the same document. *External inconsistency* refers to conflicts between requirements in related interface documents. In addition to analyzing user command language specification tables, the IVT method also analyzes input/output (or data flow) tables. These tables (also referred to as "Type 3" tables) are found in interface requirements specifications (IRS) and interface design documents (IDD) and are often associated with data flow diagrams. These tables are expected to contain three fields: (1) data element, (2) data element source, and (3) data element destination.

When problems are found, they are used to generate error reports when the requirements are obviously wrong, and test obligations when the requirements have potential problems. Detailed algorithms for these checks are provided in the technical report [5].

The IVT method performs three additional checks on Type 1 tables (user command language specification tables containing syntactic information).

1. Examine data elements that are adjacent to each other. If no delimiters are specified, the IVT method will look to see if two data elements of the same type or with the same expected value are adjacent. If so, a "test obligation" is generated to ensure that the two elements are not concatenated if the user "overtypes" one element and runs into the next element.
2. Check to see if a data element appears as the data type of another data element. If IVT detects such a case, it informs the user that the table elements are ambiguous and a test obligation is generated.
3. Check to see if the expected value is duplicated for different data elements. This is a potential poor interface design because the user might type the wrong value. This situation is similar to when a grammar has overloaded token values. If IVT detects such a case, it informs the user that the table elements are potentially ambiguous and a test obligation is generated.

2.3 Generating valid test cases

The user command language specification is used to generate a covering set of test cases. The syntax graph of the command language is tested by adapting the all-edges testing criterion [3]. Each data element is represented as a node in the syntax graph. Many user command specifications yield loops in the syntax graphs, and the following heuristic is used [1, 8]: execute 0 times through the loop, execute 1 time through the loop, execute X times through the loop, and execute X+1 times through the loop, where X is a reasonably large number. The test cases are generated automatically by traversing the syntax graph.

2.4 Generating error test cases

There are two sources of rules for generating erroneous test cases: the error condition rule base, and the test obligation database. The error condition rule base is based on the Beizer [1] and Marick [8] lists of practical error cases. The test obligation database is built during static analysis. Erroneous test cases are generated from both the error condition rule base and the test obligation database. Four types of error test cases are generated from the error condition rule base:

1. Violation of looping rules when generating covering test cases.
2. Top, intermediate, and field-level syntax errors. A wrong combination is used for 3 different fields, the fields are inverted (left half of string and right half of string are swapped with the first character moved to the middle) and then the three inverted fields are swapped with each other.

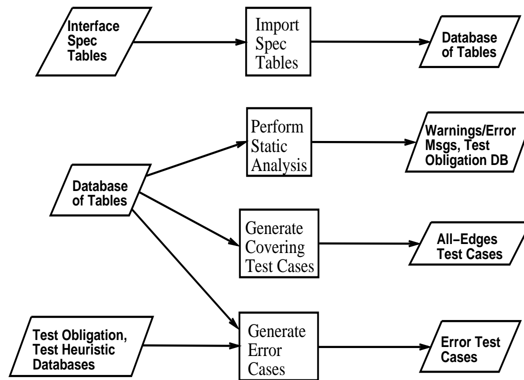

Figure 1: MICASA Architecture

3. Delimiter errors. Two delimiters are inserted into the test case in randomly selected locations.
4. Violation of expected values. Expected numeric values are replaced with alphabetic values, and expected alphabetic values are replaced with numbers.

Two types of error test cases are generated from the test obligation database:

1. Overloaded token static error/ambiguous grammar static error. An overloaded token is inserted into the ambiguous elements of a test case, based on the ambiguous value and the ambiguous character numbers identified during static analysis.
2. Catenation static error. The values that were identified as possibly catenating each other (user accidentally types information into the next field since adjacent fields have the same data type, no expected values, and no delimiters) are duplicated into the adjacent fields.

3 MICASA: A Proof-of-concept System

To demonstrate the effectiveness of IVT, a proof-of-concept system was developed. This tool accepts input specifications, performs the analyses described in Section 2, and automatically generates system-level tests. The tool is called Method for Input Cases and Static Analysis (MICASA), runs under Windows NT, and is written in Visual C++.

A high level architecture is shown in Figure 1. MICASA has four major subsystems, shown in square boxes in the middle. The Import Specification Tables subsystem accepts the interface specifications, and translates them to a standardized, intermediate form. This Database of Tables is then fed to the other three subsystems, which generate messages about the input specifications, test obligations, and test cases. The Test Heuristic Database is encoded directly into the MICASA algorithms.

MICASA accepts flat files and MS Word files that describe the interface specification tables. The files are imported into MS Access tables.

MICASA allows the user to perform a number of static checks on the interface tables, including consistency, completeness, ambiguous grammar, overloaded token, and potential catenation. The input is the interface table information that is created by Import Spec Tables and stored in the MS Access database. The output from static analysis is a set of MS Access database tables containing error records, as well as

printouts of these error reports. An example Ambiguous Grammar Error report is shown in Table 1. An example Overloaded Token Error report is shown in Figure 2. An example Catenation Error report is shown in Figure 3.

Ambiguous Grammar

Tuesday, June 09, 1998

Table Name Values	ID Field	Error Type	Error	Ambiguous Char #	Char #	Description	Class of Values	Value
Table-3-2-4-1-10-ETF-Weaponeering-Dataset	214	General Error	Ambiguous: Duplicate Values	786	781	ELEMENT_WIDTH	No	0
able-3-2-4-1-10-ETF-Weaponeering-Dataset	204	General Error	Ambiguous: Duplicate Values	774	770	PERCENT_BURIAL	No	0
Table-3-2-4-1-10-ETF-Weaponeering-Dataset	205	General Error	Ambiguous: Duplicate Values	808	771	PERCENT_BURIAL	No	
Table-3-2-4-1-10-ETF-Weaponeering-Dataset	206	General Error	Ambiguous: Duplicate Values	807	772	PROB_OF_DAMAGE_GIVEN_A_HIT	No	1
Table-3-2-4-1-10-ETF-Weaponeering-Dataset	207	General Error	Ambiguous: Duplicate Values	804	773	PROB_OF_DAMAGE_GIVEN_A_HIT	No	

Table 1: Example Ambiguous Grammar Report

Overloaded Token

Tuesday, June 09, 1998

Table Name	Character	Description
Table-3-2-4-1-10-ETF-Weaponeering-Dataset	738	NOMINAL_CEP
Table-3-2-4-1-10-ETF-Weaponeering-Dataset	816	PATTERN_TYPE
Table-3-2-4-1-10-ETF-Weaponeering-Dataset	856	TARGET_ALTITUDE
Table-3-2-4-1-10-ETF-Weaponeering-Dataset	863	WEAPON_OR_DISPENSER_TERMINAL_VELCITY
Table-3-2-4-1-10-ETF-Weaponeering-Dataset	868	EJECTION_VELOCITY

Table 2: Example Overloaded Token Error Report

MICASA allows the user to generate all-edges test cases for the Type 1 interface tables stored in MS Access. The input is the interface table information stored in the MS Access database. This function automatically generates test cases to satisfy the all-edges criterion on the syntax graph, and stores them in MS Access database tables. The test cases can be formatted as Test Plans using an MS Word template.

MICASA allows the user to generate error test cases for the Type 1 interface tables stored in MS Access. The input is the interface table information stored in the MS Access database, the test obligation database generated during static analysis, and the Beizer [1] and Marick [8] heuristics for error cases. An error test case is generated for each test obligation in the test obligation database. Next, the Beizer and Marick heuristics are used to generate error cases, as described in Section 2. The user is shown the number of test cases that have already been generated (under Generate Covering Test Cases function), and the user is given the option to generate error cases or to return to the previous function. After error test cases are generated, all duplicate test cases are deleted.

4 Empirical Validation

This section presents empirical results that demonstrate the feasibility, practicality, and effectiveness of the IVT method. Real-world, industry applications were used in a multi-subject experiment to compare the

Possible Catenation Error					
				Wednesday, June 17, 1998	
Table Name	**ID Field**	**Error Type**	**Error**	**Ambiguous Char #**	**Char #**
Table_doc_jecmics	0	Warning	Catenation Warning: Possible Catenation Error.	422	421
Table_doc_jecmics	0	Warning	Catenation Warning: Possible Catenation Error.	400	399
Table_doc_jecmics	0	Warning	Catenation Warning: Possible Catenation Error.	368	367
Table_doc_jecmics	0	Warning	Catenation Warning: Possible Catenation Error.	263	262
Table_doc_jecmics	0	Warning	Catenation Warning: Possible Catenation Error.	220	219

Table 3: Example Catenation Error Report

IVT method with human subjects. The experimental design is described and the experimental subjects are presented, then specific results are presented.

The experienced testers were defined as having at least seven years of software development/information technology experience and at least three years of testing experience. Neither of the authors participated in the experiment.

Existing software subsystems (TPS-DIWS and PTW) of the Tomahawk Cruise missile mission planning system were used. TPS is comprised of roughly 650,000 lines of code running on HP TAC-4s under Unix. TPS is primarily Ada with some FORTRAN and C. A number of Commercial Off-the-Shelf (COTS) products have been integrated into TPS, including Informix, Open Dialogue, and Figaro. TPS was developed by Boeing. DIWS runs under DEC VMS and consists of over 1 million lines of Ada code, with a small amount of Assembler and FORTRAN. Some code runs in multiple microprocessors. DIWS was developed by General Dynamics Electronics Division. PTW is hosted on TAC-4 workstations, runs under Unix, and is written in C. The system is roughly 43,000 lines of code, and was developed by General Dynamics Electronics Division.

4.1 Results and Discussion

Four testers analyzed five requirements specifications documents (one FBI specification, one commercial specification, and three Navy specifications): all four analyzed documents 1 and 2, and two analyzed documents 3, 4, and 5. Two testers generated test cases for the PTW software. The results are shown in Table 4. The specification defects that were found were divided into syntax and semantic defects. Not surprisingly, the automated tool found far more syntactic defects, and the human testers found more semantic defects. The defect detection rate refers to the average number of test cases needed to find a defect, and the minutes per fault found is the mean wall clock time (in minutes) needed to detect each fault.

	MICASA	**Testers**
Syntax Spec. Defects Found	524	21
Total Spec. Defects Found	524	106
Number of test cases	48	7
Software Faults Found	20	27
Defect Detection Rate	7.4	4.6
Minutes Per Fault Found	8.4	72.2

Table 4: Empirical Results

Note that the Testers column includes four testers for the specification analysis, and two for the execution. For the software faults, one tester found 21 faults, and the other found 6. Although one tester found one more fault than MICASA, the cost of using the MICASA tool was much lower. Taking time to develop and execute the tests as a rough approximation of cost, it cost 8.6 times as much for humans to detect faults as for the automated tool. Also, MICASA found specification defects and software faults not found by humans.

An interesting observation has to do with the quality of the specification tables. For part I of the experiment, it was noted that the senior testers did not find a very high percentage of the defects present in the poorest quality specification tables. When specification tables were of particularly poor quality, the participants seemed to make very little effort to identify defects. Instead they seemed to put their effort on the tables that were of higher quality. This phenomenon also showed up in part II of the experiment.

5 Conclusions

Validation results show that the IVT method, as implemented in the MICASA tool, found more specification defects than senior testers, generated test cases with higher syntactic coverage than senior testers, generated test cases that took less time to execute, generated test cases that took less time to identify a defect than senior testers, and found defects that went undetected by senior testers.

The results indicate that static analysis of requirements specifications can detect syntactic defects, and do it early in the lifecycle. More importantly, these syntactic defects can be used as the basis for generating test cases that will identify defects once the software application has been developed, much later in the lifecycle. The requirements specification defects identified by this method were not found by senior testers. Half of the software defects found by this method were not found by senior testers. And this method took on average 8.4 minutes to identify a defect as compared to 72.2 minutes for a senior tester. So the method is efficient enough to be used in practice, and indeed, MICASA is presently being used on the Tomahawk cruise missile project. On the other hand, these results do not indicate that we should "fire the testers". The human testers found several faults that were **not** found by MICASA. These were mostly related to semantic problems that the tool could not focus on. It could be said that in addition to saving large amounts of money, MICASA allows the human testers to focus their energies on the interesting parts of designing test cases for semantic problems.

These results suggest several conclusions for software developers. To testers, it means that they should not overlook syntactic-oriented test cases, and that they should consider introducing syntactic static analysis of specifications into their early life cycle activities. To developers, it means that emphasis must be put on specifying and designing robust interfaces. Developers may also consider introducing syntactic deskchecks of their interface specifications into their software development process. To project managers, it means that interface specifications are a very important target of verification and validation activities. Project managers must allow testers to begin their tasks early in the life cycle. Managers should also require developers to provide as much detail as possible in the interface specifications, facilitating automated analysis as much as possible. Similarly, customers should require interface specifications to include as much information as possible, such as expected data values and whether or not a field is required or optional.

References

[1] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[2] A. M. Davis. *Software Requirements Analysis and Specification.* PTR Prentice Hall, Englewood Cliffs, NJ, 1990.

[3] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software*, 38(3):235–253, 1997.

[4] John Gough. *Syntax Analysis and Software Tools.* Addison-Wesley Publishing Company Inc., New York, New York, 1988.

[5] J. H. Hayes. *Input Validation Testing: A System Level, Early Lifecycle Technique.* PhD thesis, George Mason University, Fairfax VA, 1998. Technical report ISSE-TR-98-02, http://www.ise.gmu.edu/techrep.

[6] J. H. Hayes, J. Weatherbee, and L. Zelinski. A tool for performing software interface analysis. In *Proceedings of the First International Conference on Software Quality*, Dayton, OH, October 1991.

[7] J. R. Horgan and S. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symposium of Quality Software Development Tools*, pages 2–10, New Orleans LA, May 1992.

[8] Brian Marick. *The Craft of Software Testing: Subsystem Testing, Including Object-Based and Object-Oriented Testing.* Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[9] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

[10] N. L. Sizemore. Test techniques for knowledge-based systems. *ITEA Journal*, 11(2), 1990.

[11] L. J. White. Software testing and verification. In Marshall C. Yovits, editor, *Advances in Computers*, volume 26, pages 335–390. Academic Press, Inc, 1987.