Empirical Software Engineering
©2005 Kluwer Academic Publishers, Manufactured in The Netherlands.

Input Validation Analysis and Testing

JANE HUFFMAN HAYES hayes@cs.uky.edu
Computer Science Department, Laboratory for Advanced Networking, University of Kentucky

JEFF OFFUTT ofut@ise.gmu.edu
Information and Software Engineering Department, George Mason University

Abstract. This research addresses the problem of statically analyzing input command syntax as defined in interface and requirements specifications and then generating test cases for dynamic input validation testing. The IVAT (Input Validation Analysis and Testing) technique has been developed, a proof-of-concept tool (MICASA) has been implemented, and a case study validation has been performed. Empirical validation on large-scale industrial software (from the Tomahawk Cruise Missile) shows that as compared with senior, experienced analysts and testers, MICASA found more syntactic requirement specification defects, generated test cases with higher syntactic coverage, and found additional defects. The experienced analysts found more semantic defects than MICASA, and the experienced testers' cases found 7.4 defects per test case as opposed to an average of 4.6 defects found by MICASA test cases. Additionally, the MICASA tool performed at less cost.

Keywords: specification analysis, software testing, input validation, fault-based analysis, fault-based testing, empirical research, syntax-based, interface verification, system testing, static analysis, dynamic testing, case study

1 Introduction

Most software applications rely on structured interfaces, including user interfaces and file interfaces. These interfaces are notorious problem areas, causing problems at all levels of system design, development, testing, and deployment, and especially during actual use. Perry and Evangelist found that at least 66% of these errors arose from interface problems [Perry and Evangelist, 1985]. HP found that 56.9% of all faults were module interface faults [Nakajo et al., 1989; Nakajo and Kume, 1991; Ghosh and Mathur, 1997].

Although not always explicitly defined, a language exists for the interfaces of most software applications. We propose a technique for statically analyzing input command syntax as defined in interface and requirements specifications and then generating test cases for dynamic input validation testing. In approaching testing in this manner, system-level testing can be performed in a systematic, measurable way while ensuring that input validation faults have been targeted. This paper presents the system-level analysis and test data generation technique and empirical results from a proof-of-concept tool used on software from the Tomahawk Cruise Missile system.

The IEEE standard definition of an error is a mistake made by a developer [IEEE, 1999]. An error may lead to one or more faults. Faults are located in the text of the program. A fault is the difference between the incorrect program and some correct program. Note that a fault may be localized in one statement or may be textually dispersed into several locations in the program [Hayes and Offutt, 1999]. The term defect is often used interchangeably with "fault." For example, suppose that a software engineer inadvertently omitted a parameter from a procedure call, even though the interface requirement specification called for it. This error on the engineer's part has introduced a fault or defect (the parameter is missing and the interface does not comply with the specification). This research defines an *interface fault* to be a fault in the external interface of the software; the input data is handled incorrectly or improperly.

Fault-based testing generates test data to demonstrate the absence of a set of pre-specified faults [Morell, 1990]. Similarly, fault-based analysis identifies static techniques (such as traceability analysis [Hayes et al., 2003]) and even specific activities within those techniques (e.g., perform back-tracing to identify unintended functions) that should be performed to ensure that a set of pre-specified

faults do not exist. Note that fault-based analysis is an early lifecycle approach that can be applied prior to implementation [Hayes, 2003]. The Input Validation Analysis and Testing (IVAT) technique is a fault-based analysis and testing technique. It targets input validation faults in requirement specifications (analysis) and in the implemented system (testing).

This research assumes that many post-deployment defects in the software application will be related to the input syntax, and that organizations tend to make the same types of mistakes throughout the development process. That is, faults introduced when describing the interfaces will still exist as latent defects in the developed software, and/or similar faults will be introduced again during design and coding. Our work, reported here, shows this assumption to be accurate: interface faults identified in the interface requirement specification for a Navy software system were still found to exist in the corresponding software release.

1.1. Motivation

Keyboard data entry and automated data generation is error prone, especially if the data must conform to specific format and content guidelines. Successful use of some applications requires a working knowledge of command languages (for example, Structured Query Language (SQL) for relational data base management systems) as well as careful typing. Other applications rely heavily on user-produced or software-generated files to obtain information required for processing. New and emerging technologies such as eXtensible Markup Language (XML) and Simple Object Access Protocol (SOAP) messages are examples of this. As these represent the entry and interface points to applications, they contribute to the aforementioned interface problems. Although these technologies allow for greater flexibility in software inputs, they also bring in additional complexities that make errors more likely. Note that errors may be introduced in a number of ways: (a) a program failing to perform a validation check on data inputs obtained directly from a system user (typed), (b) a user submitting a valid but unintended data input (thinks they typed an M in the gender field for male but were in the field for marital status and the M is now viewed as indicating that the user is Married) (typed), (c) a programmer writing software that generates an erroneous data file that is used by another part of the system (file), and/or (d) a program failing to perform a validation check on data input files (file). If software has not been written in a robust way, erroneous input from the keyboard, the mouse, or a file will result in an application failure. A motivating yet tragic example of this was the loss of the Titan IV-B-32 in April 1999. A manual entry error of a roll rate filter constant, whose value should have been –1.992476 but was entered as –0.1992476, resulted in greatly magnified yaw and pitch in the second burn, that caused uncontrolled tumbling. The loss of the satellite cost $800M and the overall loss including the launcher was $1.2 billion [Hastings, 2003].

1.2. Utility

The IVAT technique has been applied to diverse domains and applications. It provides several benefits to a software product as well as to various parts of a software development organization. It targets interface faults. The analysis portion of the technique helps find potential interface problems very early in the development process that may not have been uncovered by other analysis techniques. This information can be used to improve the requirements and design. The testing portion of the technique generates test cases before code has been developed. Systems engineers can examine the test cases, alerting them to potential problems with how the system may be used (perhaps in ways that they had not envisioned). This also allows the test manager to better estimate the number of testers and hours that will be required for the actual test execution. The automated generation of test cases increases the productivity of the test team during test case development. The test cases will discover latent faults from mistakes made in earlier input validation design and coding work that may not have been uncovered by other testing techniques.

1.3. Objective

This paper presents a new method to analyze and test syntax-directed software systems based on syntactic anomalies in interface requirements specifications. The overall thesis of this research is that the current practice of analyzing and testing software described by structured, textual, tabular, interface requirements specifications developed by hand, can be improved by automating with the input validation analysis and testing (IVAT) technique. As will be explained below, IVAT takes as input textual, tabular, interface requirements with fixed-width data fields. There is no formal grammar and the developers use a semi-natural language. The format used was originally created for the benefit of requirements writers, as opposed to specifically for testing. The data, presented in Section 4, suggests that the IVAT technique statically detects syntactic specification defects better and faster than senior analysts and generates test cases that identify software faults not detected by senior testers.

1.4. Scope

A *user command* is defined as an input from a user that directs the control flow of a computer program. For example, if the user double clicks on a menu item, causing a function to execute, this represents a user command. If a user enters a "P" to indicate the desire to print (per a direction on the screen), the "P" represents a user command. A *user command language* is a language that has a complete, finite set of actions entered textually through the keyboard, and is used to control the execution of the software system. A *syntax-driven application* has a command language interface. SQL is an example of a command language that is used to interact with a syntax-driven application (relational database).

In light of this, two general requirements for syntax-driven applications seem apparent:

Requirement 1. A syntax-driven application shall be able to properly handle user commands that are constructed and arranged as expected.

Requirement 2. A syntax-driven application shall be able to properly handle user commands that may **not** be constructed and arranged as expected.

The first requirement covers the need for software to function properly when given valid inputs. The second requirement refers to the need for software to tolerate operator errors. That is, software should anticipate most classes of input errors and handle them gracefully. Test cases should be developed to ensure that a syntax-driven application fulfills both of these requirements. *Input-tolerance* is defined as an application's ability to properly process both expected and unexpected input values. *Broad power* refers to a technique's ability to find many classes of faults. *Hard power* refers to a technique's ability to find "difficult to detect" classes of faults [Miller et al., 1995]. *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 while exhibiting broad power at the system level.

Morell [1990] provided a theory of fault-based testing. He described the arena as a triple <*P,S,D*> where *P* is the program, *S* is the specification , and *D* is the domain of interest. He further described a fault-based arena as a 5-tuple <*P,S,D,L,A*> where *L* is an n-tuple of locations in *P* and *A* is a set of potential faults. *A* can be further divided into subsets such as A_i where A_i is the set of potential faults that could exist at location l_i in program *P*. So, for example, location 2 (l_2) of a program *P* that has been written to satisfy specification *S* might contain "if x > y." The set A_2 might contain potential faults such as incorrect relational operator (> should be <), incorrect operand (y should be z), or incorrect construct ("if" should be "while"). Potential faults are applied to program locations to generate alternate programs. For example, an alternate program *P'* would be *P* where location 2 has been replaced with "if x <= y." In fault-based testing, we seek test data that differentiates the original program from its alternates, where the alternates are our means of "seeding faults."

We adopt Morell's formalism [1990], but focus on the set of faults A. If we can define the set of all faults A in a program P, then we can identify non-overlapping subsets of the set A. Let A' be a subset of A. A' is the subset of all faults that can be detected using the analysis portion of IVAT (i.e., syntactic interface specification errors). We hypothesize that A' is coupled to another subset of A that we'll call A''. A'' represents a subset of A that can be detected using the testing portion of IVAT (e.g., failure to perform validation of input values or data files). Further, we hypothesize that there is an overlap of A' and A", representing latent defects or repeated errors (engineers make the same errors in a later lifecycle phase that they did in an earlier lifecycle phase). Our work is based on this hypothesis, and the evaluation results indicate that this hypothesis holds (see section 4.4, discussion of $H5_a$). Detailed hypotheses are presented in Section 4.2.

This paper makes several contributions. First, we validate our hypothesis that latent defects (though detectable or even detected early in the life cycle) will remain in the software, and we build our method around this idea. Second, we introduce the input validation analysis portion of our method. This statically examines interface specifications that are available early in the lifecycle and introduces checks for a number of novel interface aspects (such as potential catenation error). Next, we develop a test case generation aspect of our method. To our knowledge, it is the only method that generates test cases based on problems or potential problems found in interface specifications earlier in the lifecycle (based on the static checks that we introduced). In addition, we implemented some existing testing heuristics (after Beizer, 1990, Marick, 1995, and White, 1987) in the test case generation method to ensure its practicality.

The paper is organized as follows: The IVAT technique is presented in Section 2. Section 3 describes the proof-of-concept tool for IVAT and Section 4 discusses the empirical validation of the technique. Related work is presented in Section 5. Section 6 is devoted to conclusions and future work.

2 The Input Validation Analysis and Test (IVAT) Method

Input validation analysis and testing (IVAT) focuses on the specified interfaces of the system and uses a graph of the syntax of user commands. IVAT incorporates formal rules in a test criterion that includes a measurement. This section discusses the four major steps of the process followed by IVAT:

1. specifying the format of user command language specifications
2. analyzing a user command language specification
3. generating valid test cases for a user command language specification
4. generating error test cases for a user command language specification

IVAT uses a test obligation database, a test case table, and a text/word-processing file. A *test obligation* is created when a defect or potential software problem is detected during static analysis; it represents something that must be evaluated during testing. If a defect is found, information about 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 defect detected statically has not become a fault in the finished software. One test case is generated for each test obligation. All the generated test cases are stored in the test case table. The text/word-processing file is used to generate test plans and cases in a standard Test Plan format.

2.1. Specifying the format of user command language specifications

The IVAT method is specification-driven, thus is only useful for systems that have some type of documented interfaces. The IVAT method expects a minimum of one data element per user command language *specification table* and expects a minimum of three fields, in any order, for the data element[1]:

1. data element name
2. data element size
3. data element expected/allowable value

These elements are expected to be regular expressions, as described below. Data element name can be:

Literal character	a
Special character	\b
Iteration	A+

Data element size can be:

Digit	\d
Iteration	A+

Data element expected/allowable value can be:

Empty string	ε
Literal character	a
Special character	\b
Alternation	A	B
Iteration	A+

A future enhancement is to allow data elements to have variable sizes, though this was not observed in the interface documents used for validation. Currently, the prototype tool produced for this research expects the interface specification tables to be defined in a form that satisfies the grammar shown below. Note that Exp_Value refers to a "simple" expected value that is listed in a table, such as "Alphanumeric" or "D." Predefined types such as Alphanumeric are not user definable, but have been enumerated in the prototype tool. The predefined types include: alphanumeric, character (same as alphanumeric), string (same as alphanumeric), and numeric. The addition of other predefined types is addressed in Section 6 as part of future work. Equal_Value refers to an expected value that contains an "=," such as "J=JSIPS." Alt_Value refers to a list of alternative Exp_Value or Equal_Value.

```
Interface_spec_table  —> (Elem_name <TAB> Elem_size <TAB> Elem_value)+
Elem_name             —> (letter | digit | Special)+
Elem_size             —> (digit)+|(digit)+”,”(digit)+
Elem_value            —> Elem_name | type | range|Exp_value | Equal_value|Alt_value
Exp_value             —> (letter | digit | Special)+
Equal_value           —> (letter | digit | Special)+ “=” (letter | digit | Special)+
Alt_value             —> (Elem_value”|”Elem_value)+
Special               —>  | \ | < | > | , | . | ' | |  …
digit                 —> 0 | 1 | 2 | |  …
letter                —> a | b | c | |  …
type                  —> Alphanumeric | Numeric| ….
```

[1] We examined dozens of industrial requirements and interface specifications before deciding on the minimal fields and format. We examined commercial product specifications as well as civil agency and defense agency specifications, for critical and non-critical applications. Though the input command language may seem very straight forward, it is expressive enough to handle the majority of the cases we encountered.

range —> “range [“ (Numeric)+ “,” (Numeric)+ “]”
tab —> ^T

An interface specification table (Interface_spec_table) is one or more entries that consist of an element name (Elem_name) followed by a tab followed by element size (Elem_size) followed by a tab followed by element values (Elem_value). An element name can contain one or more letters, digits, or special characters. An element size can be any number of digits, possibly containing an embedded comma. For example, as can be seen in Figure 1, the Elem_size field is titled “Char #” and the TPS Task Number has the value “3,5” This means that the TPS Task Number spans characters 3, 4, and 5. An element value can be a simple expected value (Exp_value), an expected value that contains an “=” (Equal_value), or a list of alternative expected values (Alt_value), such as “D = DIWS, J = JSIPS.” For example, in Section 2.4, the Startup File table shows that TMM has an expected value of T, but Average_Temp has an expected value of Operating_Temp (and its expected type is also defined in the table). A valid table is “task_type <tab> 1 <tab> D = Digitize|M= Measure.”

From a practical perspective, the benefits of this technique for a program that is not syntax-driven are not known. The research was designed for and validated with syntax-driven applications. However, it may be possible to convert many non-syntax-driven applications to be syntax-driven.

2.2. Analyzing user command language specifications

A user command language specification defines how users will interface with the system (through non-graphical means). The integrity of a software system is directly tied to the integrity of the system interfaces, both internally and externally [Hayes et al., 1991]. Three well accepted software quality criteria apply to interface requirements specifications: completeness, consistency, and correctness [Davis, 1990].

Requirements are *complete* if and only if everything that eventual users need is specified [Davis, 1990]. The IVAT method assesses the completeness of a user command language specification in two ways. First, the IVAT method checks that there are data values present for every column and row of a specification table (as defined in section 2.1). This is to ensure that there is no missing data. Second, the IVAT method performs static analysis of the specification tables. The IVAT method looks to see if there are hierarchical, recursive (circular definition), or grammar production relationships between the table elements. For example, consider the simple example below, in Table 1. The elements Last name and First name occupy positions 1 through 20 and positions 21 through 40 in a file. They both have Name as their element value. Unfortunately, Name has been defined to have an expected value of “Last name.” This recursive relationship, also called a circular definition, represents an issue with the interface table.

Table 1. Sample interface table.

Elem name	Elem size	Elem value
Name	20	Last name
Last name	1,20	Name
First name	21,40	Name

For hierarchical and grammar production relationships, the IVAT method checks to ensure there are no missing hierarchical levels or intermediate productions. An example is provided in Section 2.4. If such defects are detected with the specification table, a test obligation will be generated and stored in the test obligation database. These defects are also reported to the user so that the specification tables can be improved before moving to the next phase of development. Any circular definitions or recursive

relationships detected will be flagged by IVAT as confusing to the end users and having the potential to cause users to input erroneous data. If circular definitions or recursive relationships are detected with the specification table, a test obligation will be generated and stored in the test obligation database. This is done for three reasons. First, we hypothesize that some problems in the interface specifications will go undetected throughout the software development effort and will remain as latent defects when it comes time to test. Second, we hypothesize that developers tend to make the same types of mistakes throughout the development process. That is to say that mistakes similar to those made in the design phase will also be made in the coding phase. Finally, if such test cases generated early in the development process are discussed with the developers, decreased likelihood of those problems remaining in the as-built system should follow due to increased understanding and awareness. The results from testing these hypotheses are discussed in Section 4.2.

Consistency is exhibited "if and only if no subset of individual requirements conflict" [Davis, 1990]. *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 IVAT method also analyzes input/output (or data flow) tables as described in Hayes' dissertation [Hayes, 1998].

Software correctness is defined in IEEE 610.12-1999 [IEEE, 1999] as "The extent to which software meets user expectations.'' Davis defines correctness for requirements as existing "if and only if every requirement stated represents something that is required" [Davis, 1990]. Although on the surface this sounds circular, the intent is that every statement in a set of requirements says something that is necessary to the functionality of the system. The IVAT method does not directly address correctness of requirements. That is, IVAT does not make any judgments of whether or not "every requirement stated represents something that is required" [Davis, 1990]. IVAT assumes that all elements are required and hence should be analyzed.

The IVAT method performs three additional, novel checks on user command language specification tables containing syntactic information.

1. Examine data elements that are adjacent to each other. If no delimiters are specified, the IVAT 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 end user of the developed system "overtypes" one element and runs into the next element. For example, in some input forms it is possible for a user to accidentally type information into the next field since the form moves you to the next field automatically after the user types a specified number of characters. If adjacent fields have the same data type, no Equal_Value, and no delimiters, such a catenation error is possible.
2. Check if a data element appears as the data type of another data element. This indicates the possibility of a circular definition or recursive relationship. An example, without a circular definition, is shown in Section 2.4. If IVAT detects such a case, it informs the user that the table elements are potentially ambiguous and a test obligation is generated. We refer to this as ambiguous grammar (distinct from the general grammar ambiguity problem).
3. Check 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 (token values can be overloaded just as operators in a programming language can be overloaded in the sense that they can occur in more than one element with possibly different meanings or properties). If IVAT detects such a case, it informs the user that the table elements are potentially ambiguous and a test obligation is generated.

The algorithm looks at each pair of elements in the table of token elements, and if they are the same, writes appropriate test cases to the test case table. A simple example of a user command language specification table from our case study on the Tomahawk Cruise Missile is shown in Figure 1. Task ID can have the values "D" or "J" followed by "D" or "M" for Task Type. The next three file positions (3 – 5) are alphanumeric to represent the Tomahawk Planning System (TPS) Task Number. The last three file positions (6 – 8) are alphanumeric to represent Digital Imagery Workstation Suite (DIWS) Task Number. An example of a valid file is "JMT01D02." This indicates that the JSIPS system has requested a measurement by the TPS system as task T01 and as DIWS task D02. Task ID and Task Type have overloaded the expected value "D," thus an overloaded token fault results. TPS Task Number and DIWS Task Number are both alphanumeric making it possible for an inadvertent overtyping to occur, thus a potential catenation fault results. Note that Char # is treated as the Size field.

Program description: The program tasks another subsystem to build digital maps.

Example inputs:

Tasker File

Char #	Data Element	Expected Value
1	Task ID	D = DIWS, J = JSIPS
2	Task Type	D = Digitize, M = Measure
3,5	TPS Task Number	Alphanumeric
6,8	DIWS Task Number	Alphanumeric

Test obligations:

Number	Reason	Char #s	Expected Value
1	Overloaded Token	1,2	D
2	Potential Catenation	5,6	Alphanumeric

Fig. 1. Tomahawk Cruise Missile Mission Planning System Example.

2.3. Generating valid test cases

The user command language specification is used to generate a covering set of test cases. The command language is expressed as a *syntax graph*, where nodes represent data elements from the grammar and edges represent production rules. The syntax graph of the command language is tested by adapting the all-edges testing criterion [White, 1987]. Many user command language specifications yield loops in the syntax graphs Beizer, 1990]. We use the following heuristic to process these loops [Beizer, 1990; Marick, 1995]: execute 0 times through the loop, then 1 time through the loop, then N times through the loop, then N+1 times through the loop, where N is the data element size.

The test cases are generated automatically by traversing the syntax graph. The cover test cases algorithm is shown in Hayes' dissertation [Hayes, 1998]. It walks through each of the specification tables and generates actual values for the test cases. For terminal values, the value is read from the table and written directly to the word-processing file and test case table. For non-terminals (such as "string"), values are selected from the data type and written to the test case table and word-processing file 0, 1, N, and N + 1

times. Beizer also treats these elements as loops in the syntax graph [Beizer, 1990]. The syntax graph in Figure 3 is created from the interface table shown in Figure 2, another actual table from Tomahawk. Originating Segment (subsystem) can be "D" or "N" followed by Task Category of "J," "I," or "P." The third position will be a "D," "S," "C," or "Q" for Task Type followed by five alphanumeric characters to indicate sequence number. A valid Task ID is "DJCT0002."

TASK IDENTIFICATION (TASK ID) – Table 3.2.2-1		
CHARACTER NUMBER	**DESCRIPTION**	**VALUES**
1	ORIGINATING SEGMENT	D=DIWSA N=NIPS
2	TASK CATEGORY	J=JSIPS-N I=TRAINING P=PRODUCTION (ICU ONLY)
3	TASK TYPE	D=DETAILING ONLY S=SCREENING AND DETAILING C=IMAGE CATALOG UPDATE ONLY Q=DATA BASE QUERY
4,8	SEQUENCE NUMBER	ALPHANUMERIC

Fig. 2. Sample User Command Language Specification Table (Table 3.2.2-1 in Tomahawk Cruise Missile).

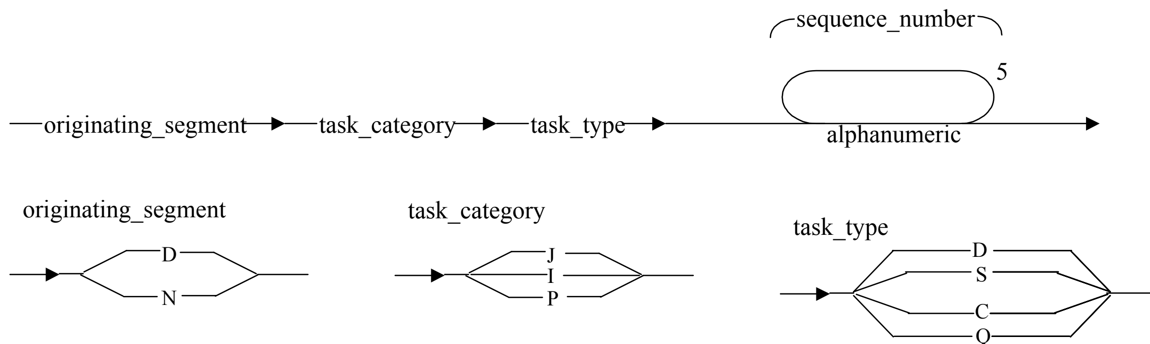

Fig. 3. Syntax Graph for Table 3.2.2-1 (from Figure 2).

For the example above, the following information would be written to the Test Plan file:

Covering Test Cases for Task ID in Figure 2, Tomahawk Table 3.2.2-1

Case 1

Originating Segment = D
Task Category = J
Task Type = D (Loop = Once)
Sequence Number = A Expected_Outcome = Invalid

Case 2

Originating Segment = D
Task Category = J
Task Type = D (Loop = N)
Sequence Number = A Expected_Outcome = Valid
Sequence Number = 1
Sequence Number = R
Sequence Number = 3
Sequence Number = Z

Case 3

Originating Segment = D
Task Category = J
Task Type = D (Loop = N+1)
Sequence Number = T Expected_Outcome = Invalid
Sequence Number = 7
Sequence Number = 8
Sequence Number = P
Sequence Number = Z
Sequence Number = 3

Case 4

Originating Segment = D
Task Category = J (Loop = Zero)
Task Type = D Expected_Outcome = Invalid

The result of this would be written to the Test Case Table:

Table 3.2.2-1

DJDA	Invalid
DJDA1R3Z	Valid
DJDT78PZ3	Invalid
DJD	Invalid

The interface specifications are formalized by looking at the data as a grammar and building appropriate syntax graphs. The test data is said to be *input validation adequate* if it covers these syntax graphs according to all-edges coverage [White, 1987]. Of course, other graph-based coverage criteria could be used.

2.4. Generating error test cases

There are two sources of rules for generating erroneous test cases: The error condition rule base (based on Beizer, 1990 and Marick, 1995 and our work with Miller, 1995) and the test obligation database (purely novel work by the authors). 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. The error condition rule base is based on the Beizer [1990] and Marick [1995] lists of practical error cases and previous work for NRC and EPRI [Miller et al., 1995]. The NRC and EPRI research [Miller et al., 1995] resulted in a detailed fault taxonomy, a detailed verification and validation technique taxonomy, a mapping of the two (what techniques can detect what fault types), and a cost benefit analysis of each technique. Based on this and Beizer [1990] and Marick [1995], four types of error test cases (exhibiting broad power and potentially hard power[2]) were selected and are generated from the error condition rule base[3]:

1. Violation of looping rules when generating covering test cases (as shown in Section 2.3).
2. Top, intermediate, and field-level syntax errors. A wrong combination is used for 3 different fields, the fields are inverted ((1) left half of string and right half of string are swapped, and (2) the first character is moved to the middle)) and then the three inverted fields are swapped with each other (fields a, b, c are now c, a, b or similar).
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.

An example of productions embedded in the interface table (item 2 above) is shown below. Header is a high level data element consisting of System followed by Subsystem followed by Connection followed by Operating Temperature (Operating_Temp) followed by Average temperature (Average_Temp). Subsystem, Connection, and Operating_Temp are terminals, but System is not. System can be either CATS, TMM, or ORC. CATS can have the expected value "C," TMM has an expected value of "T," and ORC has an expected value of "O." Operating temperature has an expected value in the range of 50.0 and 99.9 and occupies four positions in the file. Also, the data element Operating_Temp appears as the data type of the element Average_Temp.

Startup File

Char #	Data Element	Expected Value
1 thru 11	Header	System Subsystem Connection Operating_Temp Average_Temp
1	System	CATS | TMM | ORC
1	CATS	C
1	TMM	T
1	ORC	O
2	Subsystem	A | B | C
3	Connection	L = LAN, W = Standalone
4, 7	Operating_Temp	range [50.0, 99.9]
8, 11	Average_Temp	Operating_Temp

Two types of error test cases are generated from the test obligation database, overloaded token static error/ambiguous grammar static error and catenation static error. For 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. For catenation static error, the values that were identified as possibly concatenating each other are duplicated into the adjacent fields.

[2] All four fault types are detectable by a large percentage of available verification and validation techniques, thus broad power is exhibited (per Miller et al., 1995). It is hypothesized that top, intermediate, and field-level syntax errors are hard to detect, but this has not been validated.

[3] Note that although the types of error cases to generate were inspired by Marick, 1995, Beizer, 1990, and Miller, 1995, the generation of the actual test cases is novel (the algorithms for how to simulate the types of faults). For example, the test case for top, intermediate, and field-level syntax errors is novel and was not reported by Marick, 1995, Beizer, 1990, or Miller, 1995.

3 MICASA: A proof-of-concept system

A proof-of-concept system was developed to demonstrate the effectiveness of IVAT. 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).

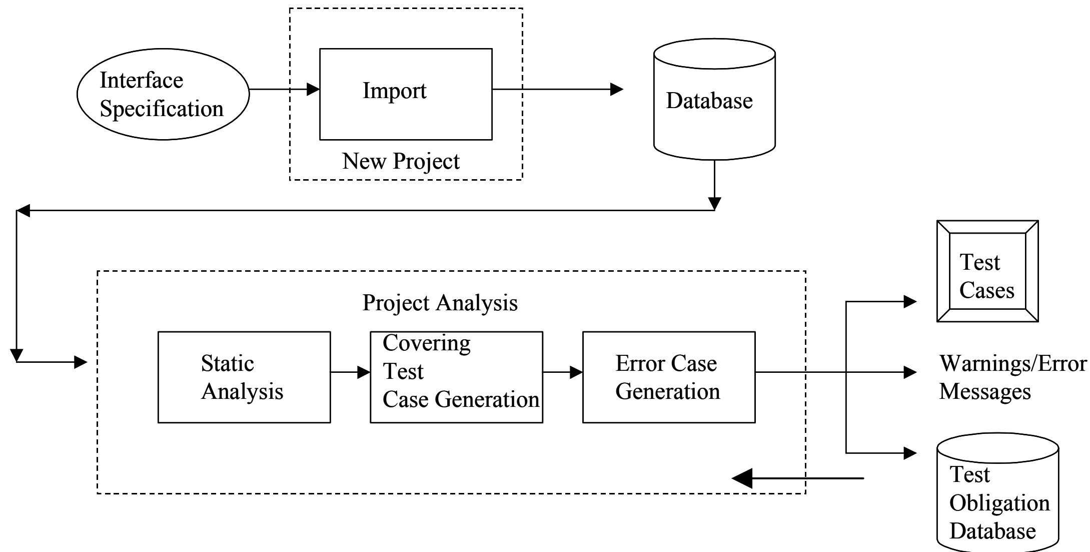

Fig. 4. MICASA Architecture.

MICASA runs under Windows NT, is written in Visual C++, and relies on database management system tables and databases. A high level architecture is shown in Figure 4, with arrows indicating data flow. MICASA has four major subsystems. The Import 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 (Static Analysis, Covering Test Case Generation, and Error Case Generation), that generate messages about the input specifications, test obligations, and test cases. The Error Condition Rule Database is encoded directly into the MICASA algorithms.

MICASA uses a small graphical user interface. Most screens have four buttons and offer the options *Cancel, Back, Next,* and *Finish*. Most interaction by the user is with action buttons. MICASA leads users sequentially through the steps of the IVAT method**.** The user enters the name of a file that contains the input specifications, the data is read, and then static analysis is performed. The user can select specific static analysis tasks (such as catenation check, overloaded token check, or any combination of these) to be performed and then test cases are generated.

3.1. Import

The Import function allows the user to import specification tables from a softcopy interface requirements specification. The Import function performs three major processing steps. First, the introduction screen asks the user if the table is a user command language specification table or a data flow table. If it is a data flow table, the user is asked if consistency checking is to be performed on an entire Computer Software Configuration Item. Second, MICASA gets the name of the file to be imported from the user. Third,

MICASA imports the provided file into database tables. The output from this function is a database of tables for the interface specification.

3.2. Static analysis

The input to this function is the interface table information that is created by Import and stored in the database. The processing steps performed by this function are: (1) when the Database of tables is created, the users can initiate a consistency check on all the specification tables; (2) users can check an individual specification table for overloaded tokens; (3) users can check an individual specification table for ambiguous grammar; and (4) users can check an individual specification table for possible catenation errors.

The output from Static Analysis is a set of database tables containing error records, as well as printouts of these error reports. An example Overloaded Token Error report is shown in Figure 5. This report shows results from analysis of the ETF Weaponeering Dataset table. The first column lists the table name, in this case only one table was analyzed. The second column indicates the character position where the overloaded token was found, position 771 for the first entry. The description column lists the name of the data element where the overloaded token was identified. The Class of Values column indicates whether or not the data element is a complex data type (such as "alphanumeric" or "string."). The Values column indicates the value of the overloaded token. In this case, the listed data elements all have the expected value of 0.

Overloaded Token

Table Name	Character	Description	Class of Values	Values
Table_3_2_4_1_10_ETF_Weaponeering_Dataset	771	PERCENT_BURIAL	No	0
Table_3_2_4_1_10_ETF_Weaponeering_Dataset	816	PATTERN_TYPE	No	0
Table_3_2_4_1_10_ETF_Weaponeering_Dataset	856	TARGET_ALTITUDE	No	0
•				
•				
•				

Fig. 5. Overloaded Token Report.

3.3. Covering test case generation

The Covering Test Case Generation function allows the user to generate all-edges test cases [White, 1987] for the user command language specification tables stored in the database. This function automatically generates test cases to satisfy the all-edges criterion [White, 1987] on the syntax graph. Users enter the document filename and system name (to be printed later on the test cases). The output is a set of database tables containing test cases. These can be displayed using a database management system, or can be formatted as Test Plans using a word processing template.

3.4. Error case generation

The Error Case Generation function allows users to generate error test cases for the user command language specification tables stored in the database. The inputs are the interface table information stored in the database, the test obligation database generated during static analysis, and the Beizer and Marick 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. This function is

automatically initiated after Generate Covering Test Cases (it is not selectable from the menu). Users are shown the number of test cases that have already been generated (under Covering Test Case Generation function), and users are given the option to generate error cases or to return to the previous function. After the Error Case Generation function is complete, all duplicate test cases are deleted. The outputs from this function are additions to the database tables containing test cases. Again, these can be displayed using a database management system or can be formatted as Test Plans using a word processing template.

4 Empirical validation

This section presents empirical results that demonstrate the feasibility, practicality, and effectiveness of the IVAT method. The first author was previously a senior manager at a large software firm and was thus able to access a large, real world, industry application (from the Tomahawk Cruise Missile) that was used in a multi-subject case study to compare the IVAT method as implemented in MICASA with human subjects. The case study context, evaluation hypotheses, plan, design, and finally specific results are presented.

4.1. Case study context

The case study context consists of three parts [Kitchenham et al., 1995]: objectives, baseline, and constraints. The objective of the case study was to examine how well the IVAT method, as implemented in MICASA, performs static analysis and generates test cases. The baseline used for comparison was typical industrial practice, experienced domain expert software testers. There were several external project constraints of note. First, only one software system was available for the execution of test cases. Second, several of the systems used were classified by the Department of Defense. This meant that the number of potential testers was limited. It also means that the full specifications and test cases cannot be published.

4.2. Hypotheses

The goal of the three-part case study (see Section 4.3.2.1) was to test the following evaluation hypotheses (each null hypothesis is followed by an alternative hypothesis):

$H1_0$: There is no difference between the coverage adequacy of test cases generated by IVAT and tests cases generated manually.

$H1_a$: Test cases generated using IVAT and generated manually differ with respect to coverage adequacy, with IVAT providing higher coverage.

$H2_0$: There is no difference in performance between generating test cases with IVAT and generating them manually.

$H2_a$: Test case generation with IVAT and manual test case generation differs with respect to performance, with IVAT requiring less time for generation.

$H3_0$: There is no difference between analyzing interface specifications using IVAT and manual validation with respect to effectiveness.

$H3_a$: Analyzing interface specifications using IVAT and manual validation differs with respect to effectiveness, with IVAT having higher effectiveness.

$H4_0$: There is no difference between IVAT test cases and manual test cases with respect to effectiveness and efficiency.

$H4_a$: IVAT test cases and manual test cases differ with respect to effectiveness and efficiency, with IVAT having higher effectiveness and efficiency.

$H5_0$: No defects identified during analysis of interface specifications will still exist in the software written to conform to those specifications.

$H5_a$: Defects identified during analysis of interface specifications will still exist in the software written to conform to those specifications.

A number of response variables were required for the study and are discussed next. Performance was measured as the time (in minutes) to generate test cases and was the only measure collected during part II of the study. It was expected that MICASA would improve performance by reducing the amount of time it takes to generate test cases. Coverage adequacy was measured using the all-edges coverage criterion [White, 1987] (measured after part II was completed). MICASA was expected to provide higher coverage adequacy than that of the cases generated by testers. Effectiveness was measured as the number of defects found (in a specification or in the software) as well as percentage of effective test cases (percentage of the test cases that detected defects). It was expected that MICASA's test cases would find at least as many defects as those of the testers. Efficiency was defined as the number of test cases needed to detect a fault (average number of test cases needed to find a defect) and minutes expended in order to find a fault. MICASA was expected to require the same or fewer test cases to detect the same or more defects in the same or less time as the test cases of the testers.

The following response (or dependent) variables were used for the case study, with MICASA being the evaluated method and testers being the baseline method:

Response Variable	**Description**	**Case Study Part**
TNSPECDEF	Total number of specification defects detected	I
TNSYNSPECDEF	Total number of syntactic specification defects detected	I
TIME	Total time for the exercise (in minutes)	I, II, III
PSYNCOV	Percentage of syntax coverage	II
PEFFTCASE	Percentage of effective test cases	II
TNDEFDET	Total number of defects detected	III
ANDEFDET	Average number of defects detected per test case	III
ATIME	Average time to identify a defect	I, III

For all parts of the case study, experienced testers were used as the baseline condition. The authors served as facilitators and not as participants. Volunteers were used for the case study. The case study was not part of their work duties. Of the original eleven, three dropped out and one did not complete the case study. Most of them claimed that they did not have time to complete the work due to time constraints from their own jobs and other commitments.

The criteria for being considered an experienced tester was having at least seven years of software development/information technology experience, at least five years of software verification and validation experience, and at least three years of testing

experience. The experienced testers were also very skilled at requirement and interface analysis. Most of these testers had five plus years of hands-on experience with the systems being analyzed and tested. The testers were encouraged to use all their domain expertise when undertaking the case study. In this way, the senior testers were given full advantage over the IVAT method. That is to say that MICASA cannot take advantage of domain-specific or system-specific information, as it is only able to use the syntactic information available in interface specification tables.

4.3. Planning

Planning accomplished for the case study is discussed next.

4.3.1. Subjects and objects

Five specifications were used for the first part of the case study: a Federal Bureau of Investigation (FBI) specification, a commercial product specification, and three U.S. Navy specifications (TPS, DIWS, Precision Targeting Workstation (PTW)). The FBI specification was for a criminal background check software system and the commercial product specification was for a data management information and control software system. The Navy specifications were for three large software subsystems (TPS, DIWS, and PTW) of the Tomahawk Cruise missile mission-planning system. 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 (relational database management system), Open Dialogue (used to develop the TPS GUI inclusive of forms, list boxes, buttons etc.), and Figaro (a software package for graphics such as stick figures of a Tomahawk mission route plan, special shapes, and special 3D representations such as sun shaded terrain). Boeing developed TPS. 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. General Dynamics Electronics Division, now owned by Texas Instruments, developed DIWS. The PTW system 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. For all subject systems, non-graphical interfaces were examined (such as data entry forms and files).

As an indicator of the size of the specifications, the TPS and DIWS specifications together contained over 5400 requirements (with an interface table for approximately every 80 – 100 requirements). The FBI and commercial specifications were smaller, but still had requirements in the thousands. All five specifications were available in ASCII and a junior analyst (not involved in the case study) spent approximately 25 minutes using word processing macros to "mark" the interface specification tables for input into MICASA. This preparation time was factored into the MICASA time.

All defects in the specifications were naturally occurring, not seeded. Two senior analysts, not part of the case study and not authors, performed thorough reviews of the three Navy specification tables to give the researchers a feel for the "quality" of the tables. We did not task the senior analysts with finding all faults in the document. Rather, we charged them with reading over the documents and seeing if the types and number of faults they saw were consistent with what they typically saw on those systems and for those developers. As they had both worked on the systems and with the developers for 10+ years, they were very good judges of whether or not the quality and the faults were "typical." Analysis was not performed on the number or percentage of defects found by MICASA and other testers that were not found by the two analysts, but their reviews gave the researchers a good idea of how many defects existed in each table [Hayes and Offutt, 1999].

4.3.2. Case study design

Eight senior software testers participated in the case study, each following a similar process as the MICASA tool. The case study was divided into three parts: perform static analysis of the specifications (part I), generate test cases for the specifications (part II), and dynamically execute the test cases (part III).

4.3.2.1 Case study parts

The three case study parts are discussed in detail below. The overall case study design is shown in Table 2.

Table 2. Case study design.

	PART I: Analysis of Specification	**PART II: Generation of Test Cases**	**PART III: Test Case Execution**
Manual	2 testers, FBI & commercial 2 testers, FBI, commercial, TPS, DIWS & PTW	6 testers, TPS/DIWS, commercial, PTW	1 tester PTW
Automated	1 tester, FBI, commercial, TPS, DIWS,& PTW	1 tester, TPS/DIWS, commercial, PTW	1 tester PTW

Table 1 shows that this case study is set in an industrial context, examining specifications from various systems. Each system (and acronym) is described in Section 4.3.1. There was no randomization of testers to systems. Participants performed one or more of the three case study parts: (1) specification review; (2) test case generation; (3) test case execution/performance. It should be noted that there was only one system (PTW) examined for Part III.

4.3.2.2 Assignment to case study parts

Four testers analyzed the FBI and commercial documents. A fifth tester used MICASA to automatically analyze the same specifications. The four testers were asked to analyze these documents to allow for examination of possible "skill level" ambiguities of the various testers. By having the testers analyze the same specification, a particularly weak or particularly strong tester could be distinguished. For example, if one tester found a very large number of defects, such as many standard deviations from the mean, in the FBI document, she would be considered an outlier (very adept tester) [Hayes and Offutt, 1999]. In addition, two of the five testers analyzed the TPS, DIWS, and PTW documents. The two testers who reviewed the TPS, DIWS, and PTW documents had at least a passing familiarity with the three systems.

For part II of the case study, six testers manually generated test cases for interface specification tables with one tester using the MICASA tool to automatically generate test cases. Five different tables were used: two specification tables for the TPS to DIWS interface, two specification tables for the commercial software system, and one table for the PTW system. The TPS-DIWS document was overlapping (that is, all testers were asked to use this document) to allow for examination of possible skill level ambiguities of the various testers [Hayes and Offutt, 1999]. The two testers who generated test cases for PTW had experience with PTW. It took the tester a few hours to learn how to use MICASA, and from 7 to 130 minutes for the testers to manually develop each test case.

The design for the third part of the case study consisted of one tester executing the manually generated test cases plus the MICASA generated test cases. The commercial system was not available to the researcher for executing test cases because it is proprietary to the developer. The TPS-DIWS software was still under development during the course of the case study. As a result, one table was used, a specification table for the PTW. A test case was considered to have found a defect if the tester executing the test case observed an unexpected result, examined the result, and determined that a failure had occurred and a fault (or multiple faults) had been detected.

4.3.2.3 Threats to validity

There were several threats to the validity of our study. Internal validity threats deal with the causal relationship between the independent and dependent variables. One potential threat to **internal validity** is that of testers not completing the tasks (often referred to as the mortality threat). We attempted to limit this threat by providing the analysts ample time to perform the specification review tasks. However, there were still analysts who dropped out of the study. We examined the backgrounds of the analysts that dropped out, and did not detect any pattern. The study was conducted over a reasonably long period of time. This represents a **history threat** to internal validity. To minimize and to monitor this threat, the researchers routinely interacted with the volunteers and inquired about events or issues that might arise and impact the results.

There was a potential instrumentation threat to **internal validity** dealing with representativeness of faults. In order to reduce this threat, two analysts manually reviewed the specifications (as described in section 4.3.1) to ensure that the naturally occurring faults were "representative." Threats to external validity deal with whether or not the results can be generalized to apply to industrial practice. A threat to **external validity** (generalization of results) for our study is the representativeness of our subject specifications and software programs. We selected large, industrial software applications across diverse domains and used five different specifications to limit this threat. There is also a threat due to the small sample size. As stated above, we selected five specifications to attempt to limit this threat. There is also a potential threat with respect to the representativeness of the testers used to perform the study. To limit this threat, we used professional testers. We also had the testers use overlapping specifications for two parts of the case study to ensure that skill level ambiguities did not exist. There were no threats to **construct validity**. A possible threat to **conclusion validity** is that of "reliability of treatment implementation" [Wohlin et al., 2000]. Some of the testers did not have enough time to perform the tasks and dropped out. Some testers became frustrated with the poor quality of the documents and stopped before they finished reviewing them fully[3]. Some testers may have spent more time on the task than others (and may have had more free time to apply toward the task). One way to address this would have been to use students. However, we feel that it is very important to use representative participants. We attempted to limit this threat by allowing the testers a long period of time (months) within which to complete the tasks. We believe that our study possesses **experimental reliability**, that is, it can be repeated with the same results [Kitchenham et al., 1995], but this has not yet been validated.

4.4. Results and discussion

Next, we present the results - the means of the response variables. Table 3 presents results for part I of the study. Note that MICASA found a mean 37.4 specification defects (all syntactic) as compared to 7.5 specification defects and 1.5 syntactic specification defects found by the testers. This reflects an almost five-fold improvement in the number of specification defects detected and almost a 25-fold improvement in syntactic specification defects detected. Condition means are given for parts II and III of the study in Table 4. Note that the syntax coverage for MICASA cases was more than three times higher than for the tester's cases. Also, the total time for part III was 22-times longer for the testers than for MICASA. The tester's cases found more defects (7.4 versus 4.6) than the

MICASA cases, though. Results for system and condition means are shown in Table 5. It can be seen in the table that MICASA required far less time to generate test cases than the testers. Table 6 presents the system means. Note that the Commercial system required far less time for the exercise, and that the PTW test cases were more effective than the TPS-DIWS cases.

Table 3. Condition Means for Part I.

Response Variables	Baseline Condition (Testers)	Evaluated Condition (MICASA)
Total number of specification defects detected	7.5	37.4
Total number of syntactic specification defects detected	1.5	37.4

Table 4. Condition Means for Parts II and III.

Response Variables	Baseline Condition (Testers)	Evaluated Condition (MICASA)
Total time for the exercise (Part III)	175.7 minutes	7.9 minutes
Percentage of syntax coverage	31.25%	100%
Average number of defects detected per test case	7.4 defects	4.6 defects
Average time for all defects identified	Given below	Given below
Test Case Execution Time Only	30.9 minutes	2.17 minutes
Test Case Development and Execution Time	72.2 minutes	8.4 minutes

Table 5. System and Condition Means.

Response Variables	TPS-DIWS System		PTW System	
	Testers (Baseline Condition)	MICASA (Evaluated Condition)	Testers (Baseline Condition)	MICASA (Evaluated Condition)
Total time for the exercise - Part II	7.55 minutes	0.06 minutes	135 minutes	0.47 minutes

Table 6. System Means.

Response Variables	TPS-DIWS System	Commercial System	PTW System
Total time for the exercise - Part II	31.8 minutes	12.5 minutes	59.5 minutes
Percentage of effective test cases	62.9%	N/A	92.4%

Table 7. Empirical Results.

		MICASA	Testers
PART I	Syntactic Defects	524 defects	21 defects
	Semantic Defects	0 defects	85 defects
	Total Spec. Defects Found	524 defects	106 defects
PART II	Number of Test Cases	48 test cases	7 test cases
	Coverage	100%	31%
PART III	Software Faults Found	20 faults	21 faults
			6 faults
	Defect Detection Rate	7.4 test cases	4.6 test cases
	Minutes Per Fault Found	8.4 minutes	72.2 minutes

The overall results of the case study are shown in Table 7. The results for part I are shown in the top third of the table (the last entry is "Total Spec. Defects Found"). The tool found more total defects than the testers (524 versus 106), lending support to hypothesis $H3_a$ as opposed to $H3_0$. The specification defects that were found in this part of the study were divided into syntactic and semantic defects. The authors categorized the defects and a non-participating senior analyst subsequently reviewed the categorization. The criteria used were: semantic defects are those whose detection requires knowledge of the system, knowledge of the requirements, or other information not found in the syntactical interface specification tables; and syntactic defects can be detected solely from the information in the interface specification tables. Not surprisingly, the automated tool found far more syntactic defects, and the human testers found more semantic defects. Though there has been some research into the semantic and syntactic nature of defects [Offutt and Hayes, 1996; Woodward and Al-Khanjari, 2000], there is no evidence that one type of defect is "easier to detect" than the other, that one type has greater impact than the other, or that one is more important than the other. Recall that the human testers were given very general instructions on how to review the specifications and were free to search for and document any type of defect they found. In that sense, their specification review was at an advantage over the MICASA tool – it was limited to searching only for syntactic problems.

The second third of the table addresses part II of the study, generation of test cases. MICASA generated 48 test cases as compared to 7 test cases that were developed by the testers. The MICASA test cases achieved 100% statement coverage while the senior tester's cases achieved only 31.25% coverage. This provides evidence for rejecting hypothesis $H1_0$ in favor of $H1_a$, that is, IVAT and manual test cases do differ with respect to coverage adequacy. And it appears that IVAT test cases outperform manual test cases.

The final portion of Table 7 presents the results for part III of the case study. It was hypothesized that the test cases generated using the IVAT method would find more defects than when using manually generated test cases. That was not the case. MICASA

test cases found 20 faults, whereas the senior tester's cases found 27 faults. That is, senior testers found an average of 7.4 defects per test case as opposed to an average of 4.6 defects found by MICASA test cases (the defect detection rate refers to the average number of test cases needed to find a defect). This lends support to the effectiveness portion of hypothesis $H4_0$. The testers rank defects as Priority 1 through 5 as part of standard practice (with Priority 1 being show stoppers with no work-around possible and Priority 5 being inconveniences to the end user). The defects found were all considered to be Priority 4 or 5 (senior tester cases and MICASA cases). So indications are that senior testers did not find more important, higher priority defects than the IVAT method. It should also be noted that the testers in the Tomahawk community tend to assign defects priority levels lower than those assigned as the official priority value by the configuration review board (whereas end users tend to rank defects much higher than the configuration review board). Also, defect rankings do not always reflect the importance of the potential indirect effect. To illustrate, consider the defect that resulted in the Titan IV-B-32 failure. The defect (if detected during testing) would likely have been ranked by the tester as a Priority 4 or 5, as it was "simply an input validation error." But clearly the defect was far more severe than that as it resulted in satellite failure.[4] MICASA found defects similar to the Titan IV-B-32 defect, such as allowing incorrect input values to be accepted for pulses.

The minutes per fault found is the mean wall clock time (in minutes) needed to detect each fault. The typical execution time for a MICASA test case was 7.9 minutes versus 175.7 minutes for the senior testers. One difference was that MICASA generated a text file that could be input to the PTW system versus manual typing of all values. Not all testers generated softcopy test cases with detailed values, though all were given access to word processing tools. One tester's tests documented what a test case was to accomplish, not the detailed values to be entered (though testers were instructed to provide the lowest possible level of detail so that someone other than themselves could run the test cases). This is a fair comparison. The time to generate test cases is considered. The senior analysts could have decided on actual test values during test case generation, typed their test cases, and then also performed a cut and paste to decrease execution time. This also meant that it took MICASA less average execution time to find a defect than for a senior tester (2.17 minutes versus 30.9 minutes). This indicates support for hypothesis $H2_a$ and the efficiency portion of hypothesis $H4_a$, and provides indications in favor of rejecting $H2_0$.

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 (of course, only one tester would be used in practice). 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 a number of specification defects and software faults not found by humans. A few samples of software faults found by MICASA but not by testers are listed below:

1. **DRTOOL001 –** "Error Type '5' was Allowed for Input." This was not an option given in the Interface Requirement Specification.
2. **DRTOOL002** – "CEP Plane was Available when Error Type was '5'." The interface requirement specification says that CEP Plane is only available when error type is '1'.
3. **DRTOOL004 –** "Number of Release Pulses Accepts an Invalid Value"

Examples of software faults found by testers but not by MICASA follow:

[4] The priority scheme for defect reports used by the Department of Defense generally states that a Priority 1 (highest priority) is due to an error that prevents the accomplishment of an operational or mission-essential function, i.e., a "showstopper." However, it has been the experience of the authors that defect reports that are ranked of lower priority (such as a 4 or 5) can also often result in serious consequences.

1. **DRHMNRD2001 – "**Dataset Scrollbar Arrow Acts as Page Down" - Using the arrows on the scrollbar should result in a field by field scrolling, but a page by page scroll occurs instead.
2. **DRHMNRD2002 – "**PTW Software Crashes when Setting Menu Security Level, then ETF Security Level"
3. **DRHMNRD2003 – "**X-Window Interface Problem when Highlighting Text"

An interesting observation has to do with the quality of the specification tables. For part I of the study, 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 (incomplete, inconsistent, difficult to understand), the participants seemed to make very little effort to identify defects. Instead, they seemed to put their effort into the tables that were of higher quality. This phenomenon also showed up in part II of the case study. In other words, human analysts find it tedious and frustrating to analyze poor quality specifications and quickly "give up."[5] Automated tools, such as MICASA, do not have this shortcoming and can be used to analyze the specifications regardless of quality level. There is evidence that human analyst morale increases when using an automated tracing tool to develop requirements traceability matrices (also a tedious task)[6]. It is possible that such an effect may be seen, though not validated in this work, if analysts can use a tool such as MICASA to analyze specifications of poor quality. This is an area for further study.

Note also that we were able to validate our hypothesis that some problems in the interface specifications will go undetected throughout the software development effort and will remain as latent defects in the delivered system, $H5_a$. Of the PTW defects detected by the MICASA test cases, five were generated solely due to test obligations from static analysis of the PTW specification (an additional three were generated based on this as well as due to Marick (1995) and Beizer's (1990) error cases). As the PTW release had been developed before our study began, we were not able to influence its development by informing the developers of the specification defects. However, those defects did turn out to be problematic for the developed software and indicate either latent defects or "similar" mistakes made again in the coding phase. In an ideal development process, the MICASA tool would be run on interface specifications as soon as they are prepared – during the requirements or design phase of the life cycle. Any problems detected would be corrected then in the interface specifications. Also, the test cases could be generated at that point in time and could be given to the developers (so they would know what tests were going to be run after code is developed). The test cases could also be "stored" by the testers for execution after the system has been built.

Below, we present a table that takes a closer look at the test cases that found defects and were generated based on our novel test obligations from static analysis of interface specifications. The rows of the table indicate the type of test case used to detect the fault, e.g., invalid cases (using Beizer and Marick's heuristics), valid cases, overloaded token, etc. The columns present each of the software faults detected during test case execution (e.g., DRTOOL001 is the first defect found by a test case generated by the tool and subsequently documented in a defect report (DR)). An **x** is placed in each row to indicate what type of test case discovered the fault that was documented in the defect report. For example, the defect reported in DRTOOL004 was found by a test case that was generated by the MICASA tool due to a potential overloaded token error (from the interface specification fault found early in the life cycle by the tool). As can be seen in the table below, the test obligations generated due to potential defects identified during the IVAT static analysis (based on the overloaded token, catentation error, and potential circular definition (or recursive relationship) of data element types error) generated test cases that uncovered code defects.

Some other interesting things can be noted from the table. First, it appears that DRTOOL001 through 003 and DRHMNRD2001 through 2003 are semantically large [Offutt and Hayes, 1996], meaning that the number of inputs for which an incorrect output will be

[5] Note that this situation is not unique to case studies. The first author has over 16 years of experience as a manager of independent verification and validation (IV&V) analysts. On occasion, a developer's document would be of such poor quality that the review would be abandoned and a report would be written: "we analyzed the first 15 pages and found these NN problems. Please correct these and the rest of the document before resubmitting."

[6] Some evidence to this effect has been found in the area of automated tracing tools being used by human analysts [Hayes et al., 2005].

generated is large. Almost every test case detected these faults. On the other hand, DRTOOL004 was difficult to detect (i.e., it is semantically small) and was not found by humans. This limited study gives indications that the overloaded token static check leads to test cases that have broad power and hard power [Miller et al., 1995]. DRHMNRD2004 was difficult to detect as only the catenation error static check and the invalid/delimiter error (from the error condition rule base) detected it. This limited study gives indications that the catenation error static check leads to test cases that have broad power and hard power [Miller et al., 1995]. Also, top, intermediate and field-value error test cases (from the error condition rule base) appear to exhibit broad power.

Fault found by:	DRTOOL 001	DRTOOL 002	DRTOOL 003	DRTOOL 004	DRHMNRD 2001	DRHMNRD 2002	DRHMNRD 2003	DRHMNRD 2004
Invalid case (Marick, Beizer)	x	x	x					
Valid	x	x	x					
Overloaded token	x	x	x	x				
Catenation error					X	x	x	x
Data element type is a data element error					X	x	x	
Invalid/delimiter error					X	x	x	x
Invalid/top intermediate error					X	x	x	
Invalid/field-value error					X	x	x	

Section 1.1 discussed a number of ways that errors could be introduced. We now discuss the types of errors used in our work. The first type of error is a program failing to perform validation checking on data entered by the user. All of our tests cases except for the valid case (see column one of the table above for the list) address this type, and for PTW we applied these test cases against interface specifications for direct user data entry. Also, we know from the results of the testing on PTW that there are portions of PTW that fail to perform such validation checking. The second type of error deals with entry of valid but unexpected data input. Our overloaded token test case addresses this error type. We found that PTW also contains this error type. The third error type involves the program not properly generating a file for another part of the system. All of our test cases except for the valid case address this type. For PTW, we did generate invalid files and some did not result in failure. Finally, a program can fail to perform validation checking on data input files. All of our test cases except for the valid case address this type. For PTW, we applied these test cases to file interfaces and we found that some portions of PTW do not properly validate file input.

5 Previous work in input validation testing

To date, the work performed in the area of input validation testing has largely focused on automatically generating programs to test compilers. Techniques have not been developed or automated to assist in static input syntax evaluation and test case generation. Thus there is a lack of formal, standard criteria, general-purpose techniques, and tools. Much of this research is from the early '60s, '70s, and '80s [Bazzichi and Spadafora 1982; Bird and Munoz, 1983; Duncan and Hutchison, 1981; Hanford, 1970; Ince, 1987; Maurer, 1990; Purdom, 1972]. For example, Purdom [1972] describes an algorithm for producing a small set of short sentences so that each production of a grammar is used at least once. Bird and Munoz [1983] developed a test case generator for PL/I that randomly generated executable test cases and predicted their execution. Bauer and Finger [1979] describe a test plan generator for systems that can be specified using an augmented finite state automaton model.

Bazzichi and Spadafora [1982] use a tabular description of a source language to drive a test generator. The test generator produces a set of programs that cover all grammatical constructions of the source language. They applied their technique to a subset of Pascal. The user must build an algorithm to produce parameter values that cannot be determined during the generation process. Hanford [1970] developed a test case generator for a subset of PL/I that used a dynamic grammar to handle context sensitivity. Payne [1978] generates syntax-based test programs to test overload conditions in real time systems. Duncan and Hutchison [1981] use attributed context free grammars to generate test cases for testing specifications or implementations. Maurer [1990] discusses a data-generator generator called DGL that translates context-free grammars into test generators, with the user providing the set of productions that make up the context- free grammar plus constructs for the non-context-free aspects. Ince's survey of test generation techniques [1987] discusses syntax-based testing and the problems of contextual dependencies and of rapid increase in size of the test grammar.

These approaches were not developed to work for more than one domain (e.g., Purdom tested compilers, Bird and Munoz focused on PL/I) and do not generalize. They also do not provide static analysis. The approach presented here does not require source code or formal specifications, and requires minimal input from the user. As Howe et al. point out in [Howe et al., 1997] "…command languages were encoded as grammar productions. This encoding posed difficulties [von Mayrhauser et al., 1994] not the least of which is that for the average system tester these grammars are difficult to write and maintain…" Our approach entirely automates the production of tests and of static analysis.

White and Cohen [1980] discuss domain testing as a test data generation approach. It requires the partitioning of the input domain into subdivisions that are determined by the predicates in the path condition. A related specification-based testing method that we have seen used in industry is category-partition testing [Ostrand 98]. In category-partition testing, the engineer defines categories for the program under test and then partitions these categories into equivalence classes. An input is then selected from each equivalence class, called a choice. Ammann and Offutt [1994] extended this by defining a coverage criterion for category-partition test specifications using formal schema-based functional specifications.

A domain-based testing tool called Sleuth [von Mayrhauser et al., 1994] assists in test case generation for command-based systems (CBS). A command-based system is a computer system that provides a command language user interface. CBS differ from syntax-driven applications in that CBS are based on a command language user interface such as found in SQL whereas syntax-driven applications are broader and may include data files, textual entries in a form, and/or a command language. Sleuth is based on the principle of testing by issuing a sequence of commands and then checking the system for proper behavior. The tester uses Sleuth to perform the following steps: (a) perform command language analysis; (b) perform object analysis (outside the scope of this paper); (c) perform command definition; and d) perform script definition. Command language analysis refers to static analysis of the syntax and semantics of the system being tested (specified graphically by the tester). Command definition is accomplished by using the provided command syntax and semantic rules. Script definition handles the sequencing of commands [von Mayrhauser et al., 1994].

Howe, von Mayrhauser, and Mraz [Howe et al., 1997] used an Artificial Intelligence planning system, combined with Sleuth [von Mayrhauser et al., 1994], to generate black box system-level tests. Their AI planner requires parameters, preconditions (state information that makes a command eligible for execution) and effects (state change) for each command. Lee and Yannakakis [1996] present test generation techniques for Finite State Machines (FSMs). Memon, Soffa, and Pollock [Memon et al., 2001] introduce graphical user interface testing criteria. These coverage criteria use events and event sequences to specify a measure of test adequacy. The interface tables that our technique examines do not offer rich information such as events, states, and/or transitions. Korel [Korel, 1990] discusses an execution-based testing technique. His approach dynamically generates tests by repetitively executing the program and applying criteria to evaluate the quality of the tests. Gupta, Mathur, and Soffa [Gupta et al., 2000] developed a dynamic approach to generation of test input data. The approach exercises a selected branch in a program by initiating test data generation with an arbitrarily chosen input from the input domain of the program. Our approach is static and does not require code.

Beizer [1990] provides a practical discussion on input validation testing (which he calls syntax testing). He proposes that the test engineer prepare a graph to describe each user command, and then generate test cases to cover this graph using coverage techniques such as all-edges [White, 1987]. In addition, he recommends the following simplistic guidelines (Beizer's grammar): "a) do it wrong, b) use a wrong combination, c) don't do enough, d) don't do nothing, and e) do too much." He also suggests building an "anti-parser" to compile BNF and produce "structured garbage" (erroneous test cases).

Marick [1995] also presents a practical approach to syntax testing. He suggests that: (a) test requirements be derived from likely programmer errors and faults, and (b) test requirements should be assumed to be independent unless explicitly shown to be otherwise (assume no subsumption). His list of recommended error cases includes: (a) use of nearby items ("putt" instead of "put"), (b) sequence errors (extra item, last item missing, etc.), (c) alternatives (an illegal alternative, none of the alternatives, etc.), and (d) repetitions (minimum number, 1 repetition, maximum number, etc.).

The MICASA tool implements some of the heuristics of test case generation from Beizer [1990] and Marick [1995]. But our work goes beyond Beizer and Marick in several ways. First, they both presented general, qualitative ideas for generating test cases, but do not provide specific, quantitative descriptions, or an automated tool to implement the ideas. Also, neither Beizer nor Marick address the notion of searching for interface problems early in the lifecycle and then using any problems found to generate test cases for later testing. Also, Marick and Beizer do not address the unique aspects of interface specifications addressed by IVAT such as potential catenation, overloaded token, etc.

Another method that can be used to find faults early in the lifecycle is software inspection. There are many software inspection methods in use. In fact, there are entire families of software inspections. Basili et al. [1996] present a family of reading techniques, including scope-based, perspective-based, and defect-based reading techniques. Code walkthroughs are a very popular form of inspection, but cannot be applied until code exists. Most software inspection methods use a checklist to ensure that all items and issues of interest have been examined. The difference between IVAT and software inspections is that IVAT is automated and targets a specific class of faults.

6 Conclusions and future work

This paper presented a new technique for developing system level tests, an automated tool to support the technique, and case study results from using the tool on a large-scale industrial software system. Tests are generated from the already present structured descriptions of the input syntax[7], thus the term input validation testing. Formal coverage criteria are applied by our algorithms[7], meaning that the technique is a quantifiable process for generating tests from input descriptions.

Validation results to date[8] show that the IVAT method, as implemented in the MICASA tool, found more defects in interface specification tables than senior analysts (syntactic defects), 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 senior testers found more semantic problems in the specifications and generated test cases that had higher defect detection effectiveness than IVAT.

The results indicate that static analysis of requirements specifications can detect syntactic defects, and can do it early in the development process. Finding defects before code has been written, tested, and distributed results in tremendous savings to the

[7] We use the interface specification tables that appear in the requirements specification documents; we do not require an analyst to specify the information in a formal language or use formal methods in any way.
[8] Note that to date we have only compared IVAT to senior testers; we have not compared it to other tools.

developers. These syntactic defects can be used as the basis for generating test cases that will identify latent defects once the software system has been developed, much later in the process. Senior testers did not find any of the requirements specification defects that IVAT found, and half of the software defects (through the test cases) that IVAT found were not found by senior software testers. The IVAT method took on average 8.4 minutes to identify a defect as compared to 72.2 minutes for a senior tester. To a software development, quality assurance, or verification and validation manager, the ability to detect a defect in 8.4 minutes versus 72.2 minutes represents almost a nine-fold savings in time, and hence almost a nine-fold savings in cost to the program. The method is efficient enough to be used in practice, and indeed, MICASA has been used on the Tomahawk cruise missile program and is being considered for adaptation to web services by ACS Software, Inc. The three Tomahawk mission planning subsystems described in Section 4 continue to be used operationally with analysis and testing of new releases used as a major quality enhancing mechanism.

On the other hand, these results do not indicate that we can eliminate testers. The human testers found several faults (via their test cases) that were not found by MICASA. These were mostly related to semantic problems that were out of the scope of the tool. It could be said that in addition to potentially helping to save 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 the software community. 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 desk checks of their interface specifications into their software development process. To program managers, it means that interface specifications are a very important target of verification and validation activities. Program 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 that developers generate interface specifications to include as much information as possible, such as expected data values and whether or not a field is required or optional.

Our work on the Tomahawk system has demonstrated the usefulness of IVAT for critical applications. For critical applications, IVAT could potentially reduce costs for a system and find more defects than human testers. It is recommended that IVAT be used on all subsystems of critical applications during the requirement and testing phases. It should be used during the requirement phase to detect potential interface problems. It should also be used to generate test cases for the testing phase. By detecting interface defects and potential problems early in the life cycle, IVAT can help reduce risk for any system (critical or non-critical). By allowing human testers to concentrate just on the semantic portion of specification reviews and test case generation and by speeding up test case generation (for syntactic defects), IVAT allows managers to better utilize human resources and provides more flexibility in the allocation of those resources. IVAT allows human testers to focus on more interesting semantic problems. It is recommended that IVAT be used on important subsystems of non-critical applications as well as on components that may have many and/or large interface specifications. As for critical applications, it should be used during the requirement phase to detect potential interface problems and in the testing phase to generate test cases.

A question that may come to mind when reading this paper is why not have the developers use the interface specifications to build an input validator rather than rely on the IVAT method? An input validator only tells us that an input is bad, it does not tell us what to do with the input when it is bad. So the application still requires that testing be performed, even if there is an input validator. In fact, the systems tested in our study did have input validation software, yet we still found associated problems.

One area for future research is that of sequencing of commands. The IVAT method examines commands in isolation, not as a sequence. When combining sequences of commands, a combinatorial explosion can quickly occur. Scripting may be the solution to the sequencing challenge as discussed in the Sleuth paper [von Mayrhauser et al., 1994]. Similarly, we do not currently address

context-sensitive semantics, such as a restriction that task type be only S when task category is I or dependence relationships between input variables. Again, we may look to scripting to address this challenge.

The handling of masks (such as DD-MM-YYYY and XXX.XX) is a possible area for future research. MICASA currently treats date masks and real or float masks as numeric data types. Ranges are also not currently checked but are handled the same as masks.

Another area for future research is that of examining table elements that are automatically generated by the subject software as opposed to being entered by the user. The IVAT method automatically generates test case values for every element in a table. The user must ignore those elements when executing the test case. To build an automated solution to this, a field must be added to requirements specifications indicating whether a data element is manually entered or automatically generated.

In examining some of the interface standards that are emerging, it appears that the IVAT method should be applicable. For example, the IVAT method should be applicable to extensible markup language (XML) and simple object access protocol (SOAP). The applicability of IVAT to graphical user interfaces should also be explored. On a field-by-field basis, the application should be very straightforward. But there is potential that the number of test cases generated could grow exponentially. One possible approach for non-text based input is that of visual event grammars (VEG), defined as part of the Graphical Event-Driven Interface Specification And Compilation (GEDISAC) project of Esprit [GEDISAC, 1999]. This is an area for future work.

Note that the study presented did not compare IVAT with other tools or structured techniques. One could speculate that a comparison to structured techniques such as White's would show some similar results (the covering aspect of IVAT and White's are the same), though the individual values selected to achieve the coverage would likely not be the same and hence one cannot say for certain what the defect detection effectiveness would be. Comparing to Beizer and Marick, one would expect similar results for the error test cases in IVAT as the same heuristics are used. However, the test obligations from IVAT are unique and are not in Marick or Beizer, so that would likely mean that IVAT would find more of those types of problems. Yet if Beizer and Marick have tools based on their heuristics, they may also have many other checks in their tools and hence might find other faults not found by IVAT. To know for sure, another study would need to be performed, and that is future work.

Acknowledgements

This research was supported in part by the U.S. National Science Foundation under grant CCR-98-04111 and by the Command and Control Systems Program (PMA281) of the Program Executive Officer Cruise Missiles Project and Joint Unmanned Aerial Vehicles (PEO(CU)), U.S. Navy. Special thanks to Mrs. Theresa Holeski. Many thanks to the analysts who participated in the case study. Thanks to Olga Dekhtyar for helpful discussions on experiments and case studies.

References

Amman, P. and Offutt, J. 1994. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS '94)*, Gaithersburg, MD, June – July 1994, 69 –79.

Basili, V.R., Caldiera G., Lanubile F., and Shull F. 1996. Studies on reading techniques. *In Proceedings of Software Engineering Workshop (SEW 1996)*, Greenbelt, MD, December 1996, 59-65.

Bauer, J. and Finger, A. 1979. Test plan generation using formal grammars. In *Proceedings of the 4th International Conference on Software Engineering*, Munich, Germany, May 1979, 425-432.

Bazzichi, F. and Spadafora, I. 1982. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering 8*, 4 (July), 343-353.

Beizer, B. 1990. *Software Testing Techniques*. Van Nostrand Reinhold, Inc., New York, NY, 2nd edition.

Bird, D. and Munoz, C. 1983. Automatic generation of random self-checking test cases. *IBM Systems Journal 22*, 3, 229-345.

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

Department of Defense. 1988. *DOD-STD-2167A: Defense System Software Development*, Department of Defense, February 1988.

Department of Defense. 1994. *MIL-STD-498: Software Development and Documentation.* Department of Defense, December 1994.

Duncan, A.G. and Hutchison, J.S. 1981. Using attributed grammars to test designs and implementations. In *Proceedings of the 5th International Conference on Software Engineering (ICSE 5)*, San Diego, CA, March 1981,170-177.

Feise, R.J. 2002. Do multiple outcome measures require p-value adjustment? *BMC Medical Research Methodology*, Volume 2:8, June 2002,

Graphical Event-Driven Interface Specification And Compilation (GEDISAC) project of Esprit. 1999. http://www.cordis.lu/esprit/src/35151.htm

Ghosh, S. and Mathur, A.P. 1997. *Testing for Fault Tolerance*. SERC Technical Report TR-175P, August 1997.

Gough, J. 1988. *Syntax Analysis and Software Tools*. Addison-Wesley Publishing Company Inc., New York, New York.

Gupta, N. and Mathur, A.P. and Soffa, M.L. 2000. Generating test data for branch coverage. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, Grenoble, France, September 2000.

Hanford, K.V. 1970. Automatic generation of test cases*. IBM Systems Journal 4*, 242-257.

Hastings, D. 2003. David Hastings' Comsats Page, http://users.ox.ac.uk/~daveh/Space/Military/milspace_comsat.html.

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

Hayes, J.H. 2003. Building a requirement fault taxonomy: Experiences from a NASA verification and validation research project," in Proceedings of the Twelfth International Symposium on Software Reliability Engineering (ISSRE 2003), Denver, CO, November 2003, pp. 49 – 59.

Hayes, J.H. and Burgess, C. 1998. Partially automated in-line documentation (PAID): Design and implementation of a software maintenance tool. In *Proceedings of the 1988 IEEE Conference on Software Maintenance*, Phoenix, AZ, October 1998, 60 - 65.

Hayes, J.H. and Dekhtyar, A. and Osborne, J. 2003. Improving Requirements Tracing via Information Retrieval. In *Proceedings of the International Conference on Requirements Engineering (RE'2003)*, September 2003, pp. 138 - 148.

Hayes, J.H., Dekhtyar, A., and Sundaram, S. 2005. Text Mining for Software Engineering: How Analyst Feedback Impacts Final Results. In *Proceedings of Workshop on Mining of Software Repositories (MSR),* associated with ICSE 2005, St. Louis, MO, May 2005, pp. 58-62.

Hayes, J.H. and Offutt, J. 1999. Increased Software Reliability Through Input Validation Analysis and Testing. In *Proceedings of The Tenth IEEE International Symposium on Software Reliability Engineering*, Boca Raton, Florida, November 1999, 199 - 209.

Hayes, J.H. and Weatherbee, J. and Zelinski, L. 1991. A tool for performing software interface analysis. In *Proceedings of the First International Conference on Software Quality*, Dayton, OH, October 1991, 1-25.

Howe, A.E. and Von Mayrhauser, A. and Mraz, R.T. 1997. Test case generation as an AI planning problem. *Automated Software Engineering 4*, 1(January), 77-106.

IEEE. 1999. IEEE Standard 610.12-1999. *Standard Glossary of Software Engineering Terminology.*

Ince, D.C. 1987. The automatic generation of test data. *The Computer Journal 30*, 1 (February), 63-69.

Institute For Defense Analysis. 1987. *Analysis of software obsolescence in the DoD: Progress report*. IDA Report M-326, February 1987.

Kitchenham, B., Pickard, L., and Pfleeger, S. 1995. Case studies for method and tool evaluation. *IEEE Software 12*, 4(July), 52 – 62.

Korel, B. 1990. Automated software test data generation**.** *IEEE Transactions on Software Engineering 16,* 8(August),870 - 879.

Lee, D. and Yannakakis, M. 1996. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, Berlin, Germany, August 1996, 1090-1123.

Liu, L.M. and Prywes, N.S. 1989. SPECHECK: A specification-based tool for interface checking of large, real-time/distributed systems. In *Proceedings of Information Processing (IFIP)*, San Francisco, 1989. pp. 55 – 60.

Marick, B. 1995. *The Craft of Software Testing: Subsystem Testing, including Object-Based and Object-Oriented Testing.* Prentice-Hall, Englewood Cliffs, New Jersey.

Maurer, P.M. 1990. Generating testing data with enhanced context-free grammars. *IEEE Software 7*, 4(July), 50-55.

Memon, A. and Soffa, M.L. and Pollack, M.E. 2001. Coverage criteria for GUI testing. In *Proceedings of the Ninth ACM International Symposium on the Foundations of Software Engineering (FSE),* Vienna, Austria, November 2001, 256-267.

Miller, L.A. and Hayes J.H. and Mirsky, S. 1995. NUREG/CR-6316, Volume 4, *Guidelines for the Verification and Validation of Expert System Software and Conventional Software: Evaluation of Knowledge Base Certification Methods*. U.S. Nuclear Regulatory Commission, March 1995.

Morell, L.J. 1990. A theory of fault-based testing. *IEEE Transactions on Software Engineering 16*, 8(August), 844-857.

Nakajo, T. and Kume, H. 1991. A case history analysis of software error cause-effect relationship. *IEEE Transactions on Software Engineering 17*, 8 (August), 830-838.

Nakajo, T. and Sasabuchi, K. and Akiyama, T. 1989. Structure approach to software defect analysis. *Hewlett Packard Journal 43*, 23, 50--56.

Offutt, J. and Hayes, J.H. 1996. A semantic model of program faults. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Diego, CA, January 1996, 195-200.

Ostrand, T.J. and Balcer, M.J. 1988. The category-partition method for specifying and generating functional test. *Communications of the* ACM 31,6(June), 676-686.

Parnas, D.L. 1986. Letters to the editors. *American Scientists 74*, (January-February),12-15.

Payne, A.J. 1978. A formalised technique for expressing compiler exercisers. *SIGPLAN Notices 13*, 1(January), 1978, 59 - 69.

Perry, D.E. and Evangelist, W.M. An Empirical Study of Software Interface Faults. In *Proceedings of the International Symposium on New Directions in Computing*, Trondheim, Norway, August 1985, 32-38.

Purdom, P. 1972. A sentence generator for testing parsers. *BIT 12*, (July), 366-375.

Von Mayrhauser, A. and Walls, J. and Mraz, R. 1994. Sleuth: A domain based testing tool. In *Proceedings of the IEEE International Test Conference*, Washington, D.C., October 1994, 840-849.

von Mayrhauser, A. and Walls, J. and Mraz, R. 1994. Testing applications using domain based testing and Sleuth. In *Proceedings of the Fifth International Software Reliability Engineering Conference (ISSRE*), Monterey, CA, November 1994, 206-215.

White, L.J. 1987. Software testing and verification. In *Advances in Computers,* volume 26, M.C. YOVITS, Ed. Academic Press, Inc., 335-390.

White, L.J. and Cohen, E.I. 1980. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering 6*, 247--257, 1980.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., and Wesslen, A. 2000. *Experimentation in Software Engineering*, Kluwer Academic Publishers, London, England.

Wonnacott, R. J. and Wonnacott, T. H. 1982. *Statistics: Discovering Its Power*. John Wiley and Sons, NY, NY, 35, 261, 262.

Woodward, M.R. and Al-Khanjari, Z.A. 2000. Testability, fault size and the domain-to-range ratio: An eternal triangle. In Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '00), Portland, Oregon, 2000, 168 – 172.

Zhu, H. and Hall, P.A.V. and May, J.H.R. 1997. Software test coverage and adequacy. *Communications of the ACM 29*, 4 (December), 366-427.