INPUT VALIDATION TESTING: A SYSTEM LEVEL, EARLY LIFECYCLE TECHNIQUE

by

Jane Huffman Hayes

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
the Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____	A. Jefferson Offutt, Dissertation Director
_____	David Rine, Chairman
_____	Paul Ammann
_____	Elizabeth White
_____	Lance Miller, VP and Director, SAIC
_____	Stephen Nash, Associate Dean for Graduate Studies and Research
_____	Lloyd Griffiths, Dean, School of Information Technology and Engineering

Date: _____

Fall 1998
George Mason University
Fairfax, Virginia

INPUT VALIDATION TESTING: A SYSTEM LEVEL, EARLY LIFECYCLE TECHNIQUE

A dissertation submitted in partial fulfillment of the requirements for the Doctor Of Philosophy degree in Information Technology
at George Mason University

By

Jane Huffman Hayes
Master of Science
University of Southern Mississippi, 1987

Director: A. Jefferson Offutt, Associate Professor
Department of Information and Software Engineering

Fall Semester 1998
George Mason University
Fairfax, Virginia

Dedication

This dissertation is lovingly dedicated to:

My Grandmother, Margaret Ruth Nicholson Huffman, for teaching me to stand up for what I believe in

My Grandfather, John Hubert Gunter, for teaching me that amazing things can be accomplished before breakfast

My Great Grandmother, Jane Ellen Nicholson, for teaching me it's the thought that counts

My Grandmother, Beatrice Gertrude Gunter, for teaching me that love is the most important ingredient in any recipe

My Great Aunt Evie, Evelyn Nicholson, for teaching me that you don't have to be related to be family

My Grandfather, Otho Clarence Huffman, for teaching me that familiarity breeds content

My best friend, Kelly Ann Noss Marcum, for teaching me to listen with the heart

My husband, Gregory Lee Hayes, for his support and dedication, and for showing me that we are in this together

My daughter, Chelsea Anne Hayes, and my son, David Lee Hayes, for hugs, kisses, smiles and lots of patience with a tired Mommy

My father, Dr. George David Huffman, for setting the bar so high and for teaching me not to settle for less

My mother, Judith Ann Gunter Huffman, for teaching and showing me that all things are possible

Acknowledgements

I thank my Lord and personal Savior Jesus Christ for guiding me and watching over me during this endeavor. I thank my parents, Grandparents, and numerous other people who, throughout the course of my life, introduced me to the Word of God and helped me grow in faith in the Lord. "Through Him all things are possible."

I thank my dissertation director, Dr. A. Jefferson Offutt, for his advice, guidance, support, and encouragement throughout my dissertation effort. I thank Dr. David Rine, Dr. Paul Ammann, Dr. Liz White, Dr. Bo Sanden, and Dr. Lance Miller for serving on my committee and providing me guidance.

Thanks to SAIC for supporting my Ph.D.and thanks to PEO(CU) for partially funding this research. I thank the SAIC (and government) team (Theresa Erickson, Brian Mello, Trevor Heller, Paulette Ormsby, Penny Tingler, Brian Cahill) for their support and assistance in developing MICASA and in running the experiment. Many thanks to the numerous volunteers who participated in the three part experiment.

My thanks to my Mom, Dr. Offutt, Charlotte Gauss, Kelly Marcum, and my husband Greg for listening to me bemoan my lack of progress on this dissertation, for putting up with me, and for giving me the occasional swift boot in the rear end.

Table Of Contents

List Of Figures

List Of Tables

Page

ABSTRACT

Input Validation Testing: A System Level, Early Lifecycle Technique

Jane Huffman Hayes, Ph.D.

George Mason University, 1998

Dissertation Director: Dr. A. Jefferson Offutt

In this dissertation, syntax-directed software is defined as an application that accepts inputs from the user, constructed and arranged properly, that control the flow of the application. Input validation testing is defined as techniques that choose test data that attempt to show the presence or absence of specific faults pertaining to input tolerance. A large amount of syntax-directed software currently exists and will continue to be developed that should be subjected to input validation testing. System level testing techniques that currently address this area are not well developed or formalized. There is a lack of system level testing formal research and accordingly a lack of formal, standard criteria, general purpose techniques, and tools. Systems are expansive (size and domain) so unit testing techniques have had limited applicability. Input validation testing techniques have not been developed or automated to assist in static input syntax evaluation and test case generation. This dissertation seeks to address the problem of statically analyzing input command syntax and then generating test cases for input validation testing, early in the life cycle. The IVT technique was developed and a proof-of-concept tool was implemented. Validation results show that the IVT method, as implemented in the MICASA tool, found more specification defects than senior testers, generated test cases with higher syntactic coverage than senior testers, generated test cases that took less time to execute, generated test cases that took less time to identify a defect than senior testers, and found defects that went undetected by senior testers.

Chapter 1

1.0 INTRODUCTION AND OVERVIEW

Human users often interface with computers through commands. Commands may be in many forms: mouse clicks, screen touches, pen touches, voice, files. A method used extensively by older programs (FORTRAN, COBOL) and still used widely today is that of obtaining user input through text entries made via the keyboard. Programs that accept free-form input, interactive input from the user, free-form numbers, etc. are all examples of *syntax driven applications* [12]. In this dissertation, syntax driven application is defined as "an application that accepts inputs from the user, constructed and arranged properly, that control the flow of the application."

1.1 Motivation

Keyboard data entry 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, SQL for relational data base management systems). Other applications rely heavily on user produced files to obtain information required for processing.

Some older applications (such as those implemented in FORTRAN, COBOL, and older generation languages) depend on user input via keyboard entry. There is a large amount of such software, largely undocumented [14], in existence that will need to be maintained for many years to come. According to a survey performed by the Institute for Defense Analyses (IDA), a good deal of first and second generation language software still exists [20]. This study examined a subset of Department of Defense programs (exceeding $15 million) representing 237.6 million source lines of code (SLOC). Of this, 200 million SLOC were for weapon systems and the remaining 37.6 million SLOC were for automated information systems. For weapon systems, 30 million lines of code are first and second generation languages. Over 20 million SLOC (10%) are written in some variant of FORTRAN. Twenty-two million SLOC for automated information systems are written in a variant of COBOL [17]. The importance of these large, older systems has recently been spotlighted by the Year 2000 issue. Much time and money is being invested to modify these systems to handle post-year 2000 dates. It

is believed that many of these older systems have never been subjected to input validation testing. The input validation testing technique suggested here can be used to help build a large testing suite for such systems, as well as to test systems developed using modern programming languages.

Transaction control languages, communications protocols, and user/operator commands (e.g., SQL) are all examples of applications that could benefit from input validation (syntax) testing [5].

1.2 Definitions

The word "syntax" is derived from the Greek *syntassein* meaning "to arrange together." Syntax is defined by Webster as "that part of grammar which treats the proper construction and arrangement of words in a sentence" [41]. In the context of computer programs, a user command is synonymous to a sentence. Webster defines "command" as "to direct authoritatively" [41]. In this dissertation, user command is defined as "an input from a user that directs the control flow of a computer program." User command language is defined as any language having a complete, finite set of actions entered textually through the keyboard, used to control the execution of the software system. Syntax driven software is defined as any software system having a command language interface.

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 may not be constructed and arranged as expected.
>
> Requirement #2: A syntax driven application shall be able to properly handle user commands that are constructed and arranged as expected.

The first requirement refers to the need for software to be tolerant of 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. *Input validation testing*, then, is defined as choosing test data that attempt to show the presence or absence of specific faults pertaining to input-tolerance.

1.3 System Testing

Though much research has been done in the area of unit testing , system testing has not garnered as much attention from researchers. This is partly due to the expansive nature of system testing: many unit level testing techniques cannot be practically applied to millions of lines of code. There are well defined testing criterion for unit testing [42], [18], [20],[31],[5] but not so for system testing. Lack of formal research results in a lack of formal, standard criteria, general purpose techniques, and tools.

Much of the research undertaken to date has largely concentrated on testing for performance, security, accountability, configuration sensitivity, start-up, and recovery [5]. These techniques require that source code exist before they can be applied. Such dynamic techniques are referred to as *detective* techniques since they are only able to identify already existing defects. What is more desirable is to discover *preventive* techniques that can be applied early in the life cycle. Preventive techniques help avert the introduction of defects into the software and allow early identification of defects while it is less costly and time consuming to repair them.

1.4 Input Validation

Input validation refers to those functions in software that attempt to validate the syntax of user provided commands/information. It is desirable to have a systematic way to prepare test cases for this software early in the life cycle. By doing this, planned user input commands can be analyzed for completeness and consistency. It is preferable that planned user commands (user command language specification) be documented in Software Requirement Specifications (SRS), Interface Requirements Specifications (IRS), Software Design Documents (SDD), and Interface Design Documents (IDD). The generated test cases can be used by the developers to guide them toward writing more robust, error-tolerant software. Currently, no well developed or formalized technique exists for automatically analyzing the syntax and semantics of user commands (if such information is even provided by the developers in requirements or design documents) or for generating test cases for input validation testing. The technique proposed here is preventive in that it will statically analyze the syntax of the user commands early in the life cycle. It is also detective since it generates test cases that can be run once the design has been implemented in code.

1.5 Problem Statement

A large amount of syntax-directed software currently exists and will continue to be developed that should be subjected to input validation testing. System level testing techniques that currently address this area are not well developed or formalized. There is a lack of system level testing formal research and accordingly a lack of formal, standard criteria, general purpose techniques, and tools. Systems are expansive (size and domain) so unit testing techniques have had limited applicability. Input validation testing techniques have not been developed or automated to assist in static input syntax evaluation and test case generation. This dissertation addresses the problem of statically analyzing input command syntax and generating test cases for input validation testing early in the life cycle. Validation results show that the IVT method, as implemented in the MICASA tool, found more specification defects than senior testers, generated test cases with higher syntactic coverage than senior testers, generated test cases that took less time to execute, generated test cases that took less time to identify a defect than senior testers, and found defects that went undetected by senior testers.

1.6 Research Thesis

The goal of this research is to improve the quality of English, textual interface requirements specifications and the resulting software implementation of these requirements for syntax-directed software. Specifically, we are interested in formalizing the analysis and testing of interface requirements specifications without introducing the need for testers to learn a new specification language and without increasing the duration of analysis or testing time.

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

To evaluate the thesis, a working prototype system based on the new method was constructed and validated using a three-part experiment. Using this prototype system, we empirically established large improvements over current practice in:

- the number of anomalies statically detected in interface requirements specifications,
- the duration of time needed to develop and execute test cases,
- the duration of time needed to identify a defect, and
- identifying defects.

1.7 Scope of Research

From the software perspective, the IVT technique can be used to test any programs that have interfaces defined in tabular fashion where 3 pieces of information are provided: 1) data element name (description), 2) data element size (size), and 3) value. Theoretically, these elements are expected to be regular expressions, as described below:

Data element name can be:

Literal Character	a
Iteration	A+

Data element size can be:

Empty string	ε
Literal character	a
Iteration	A+

Data element value can be:

Empty string	ε
Literal Character	a
Alternation	A | B
Iteration	A+

Further, the MICASA prototype expects the interface tables to be defined in a form that satisfies the following grammar:

Interface_table	-> (Description tab Size tab Value)+
Description	-> (letter|digit|special)+
Size	-> (digit)+
Value	-> Description | type | Exp_Value | Equal_Value
Exp_Value	-> (letter | digit | special)+
Equal_Value	-> (letter | digit | special)+
Special	-> |\\|<|>|,|.|¶.....
digit	-> 0|1|2|.....
letter	-> a|b|c|.....
type	-> Alphanumeric|Integer|Numeric|.....
tab	-> ^T

From a practical perspective, the benefits of this technique for a program that is not syntax-directed (see above definitions in 1.2) are not known. The research was designed for and validated with syntax-directed applications.

To put the IVT method into a theoretical context, we examine where it falls in the hierarchy of testing techniques. Testing methods can be categorized as being applicable at the unit, integration, or system level. IVT is a system level testing method. System testing methods can be categorized as requirements-based or behavior-based. Requirements-based testing techniques generate cases to ensure that the functional requirements are complete, consistent, accurate, and/or unambiguous. Behavior-based techniques concentrate on the non-functional requirements such as performance, security, and reliability. The IVT research generates system level test cases based on syntactic information found in interface requirements specifications. The result is that syntax-based testing can be added as a system testing category.

The IVT method is both requirements- and syntax-based. IVT generates test data. Automatic test data generation can be static and/or dynamic. The IVT method is static. Requirements-based testing criteria include completeness, consistency, and correctness. The IVT method examines completeness and consistency. Syntax-based testing criteria include typical graph coverage methods such as branch (cover all branches/links of the syntax graph), statement (100 % node coverage), and path (cover 100% of the paths). The IVT method uses branch (covers all links in the syntax graph). Note that prior to the IVT research, syntax-based testing was not considered a system level activity. Therefore the branch criterion was not previously believed to be applicable at the system level. The IVT method has added syntax-based testing as a system level testing technique and has thus added branch criterion as a system level (syntax-based testing) criterion.

1.8 Dissertation Organization

Chapter 2 will review background and related research. This includes mutation testing, system testing techniques, and input validation testing. The dissertation research area will be presented in Chapter 3. Chapter 4 will present the research results as implemented in a prototype system, Method for Interface Cases and Static Analysis (MICASA, pronounced Me-Kaass-Uh). Chapter 5 will discuss the validation of the research, and Chapter 6 discusses future research areas.

Chapter 2

2.0 BACKGROUND AND RELATED WORK

This research uses compiler technology, elements of system testing techniques, and unit testing techniques. Sections 2.1 and 2.2 provides background information on the all-edges testing criterion and mutation testing. Sections 2.3 and 2.4 discusses related work in the areas of system testing and input validation testing.

2.1 All-Edges Testing Criterion

A coverage criterion provides a measurement (usually as a percentage of program items covered) to indicate how well a set of test cases satisfies the criterion. A criterion is a rule or set of rules that define what tests are needed. Since the method for input validation testing presented here is driven by a syntax graph, the all-edges (also called branch coverage) testing criterion is of interest. It demands that every edge in the program's flowgraph be executed by at least one test case [11].

2.2 Mutation Testing

Fault-based testing collects information on whether classes of software faults exist in a program [39, 16, 32]. Mutation testing is an example of fault-based testing [8, 33, 2]. Mutation testing involves the deliberate syntactic modification of individual lines of code in a source program. The resultant programs, called mutants, are then exercised using test cases. Depending on the operators used, $O(N^2)$ mutants are generated, where N is the number of lines of code, so even a simple program can result in hundreds of mutants. Tests that cause a mutant program to fail are said to kill the mutant. The mutant is then considered dead and does not remain in the testing process. This process allows correct programs to be distinguished from those that are close to correct [8]. Mutation testing is based on two basic tenets. One is the coupling effect [8], [31]. This refers to the theory that test cases that can detect simple faults in a program will also detect complex faults in a program. Second, the competent programmer hypothesis postulates that competent programmers write programs that are "close" to correct [8].

2.3 Specification-Based Interface Checking

Large, complex systems are often designed by decomposing them into smaller segments and components that communicate with each other. DOD-STD-2167A [9] and MIL-STD-498 [29] make it clear that mission critical DOD systems follow this model. As a result, a system will be composed of interfaces with the user as well as many interfaces between components. Parnas points out that there are difficult problems of interface design and interface enforcement [35]. Liu and Prywes describe a specification approach called MODEL that uses a dataflow specification, interface specifications (regular expressions), and a module specification to statically analyze the specifications, automatically generate system level and procedural programs from the specifications, and compose and check specifications of connected components [25]. MODEL is intended for real-time, distributed systems and handles deadlock of consumable resources and critical timing constraints. MODEL data specifications are tree structured and are given as file definitions such as:

```
1 ALLOC1 IS FILE (ORG IS MAIL),
  2 MSG(*) IS GROUP,
    3 MSGA (0 : 1) IS RECORD,
      4 PROC_ID IS FIELD (CHAR 1);  [25]
```

Note that the data specifications are regular expressions, thus allowing the specification and later analysis of constraints on the ordering of program events (finite state machine approach). The IVT method also performs consistency checking of interface specifications, but it does not require the user to compose the specifications using regular expressions or file definitions. It uses “informal” interface specifications found in an Interface Requirements Specification document.

2.4 System Testing Techniques

System testing techniques have concentrated on areas such as performance, security, and configuration sensitivity. The goals of system testing are to detect faults that can only be exposed by testing the entire integrated system or some major part of it [5]. Transaction-flow testing [5] and category-partition testing [34] are two system testing methods of interest. *Transaction-flow testing* [5] requires the building of a transaction flowgraph (similar to a flowgraph used for path testing at the unit level). A transaction is defined as a unit of work seen from the user’s point of view (e.g., validating a user’s ATM card, validating a user’s ATM password,

updating a user's account balance, etc.). A transaction flowgraph illustrates the processing steps for a particular type of transaction handled by the system. Conditional and unconditional branches are included. Once the flowgraph has been built and analyzed (using standard inspection and walkthrough techniques), a coverage criterion is selected (statement or branch, for example) and test cases are generated to meet the criterion. Though a coverage criterion can be selected, it is not synonymous with unit testing criterion and its stopping rules. That is because the method of building the transaction flowgraph is informal, is not precisely defined, and is probably not repeatable (i.e., each tester would come up with a different flowgraph and set of covering test cases). Our approach improves upon transaction-flow testing by adding formality, repeatability, and precision.

The *category-partition* testing method is a specification-based method that uses partitioning to generate functional tests for programs. The steps for this technique are: a) analyze the specification; b) partition the categories into choices; c) determine constraints among the choices; and d) write test cases [34]. The result is a set of test cases that should achieve 100% functional requirement coverage, plus give the system a good stress and error condition workout (because of the combinations of category choices). This method was extended to include a minimal coverage criterion by Ammann and Offutt [1]. As with transaction-flow testing, category-partition testing suffers from a lack of formal definition and stopping rules.

2.5 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. 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. For example, Purdom describes an algorithm for producing a small set of short sentences so that each production of a grammar is used at least once [37]. Bird and Munoz developed a test case generator for PL/I that randomly generated executable test cases and predicted their execution [6]. Bauer and Finger describe a test plan generator (TPG) for systems that can be specified using an augmented finite state automaton (fsa) model [3]. Bazzichi and Spadafora 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 results were limited because the generated programs were syntactically correct but not semantically

meaningful [4]. Hanford developed a test case generator for a subset of Pl/I that used a dynamic grammar to handle context sensitivity [13]. Payne generates syntax-based test programs to test overload conditions in real time systems [36]. Duncan and Hutchison use attributed context free grammars to generate test cases for testing specifications or implementations [10]. Maurer 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 [27, 28]. Ince's survey of test generation techniques discusses syntax-based testing and the problems of contextual dependencies and of rapid increase in size of the test grammar [23]. These approaches suffer from a lack of generality (e.g., Purdom tested compilers, Bird and Munoz focused on PL/I) and a lack of static analysis. Our approach improves upon them by not requiring source code or formal specifications, and by requiring minimal input from the user.

More recently, Boris Beizer [5] provides a practical discussion on input validation testing (called syntax testing in his textbook). 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). In addition, he recommends the following simplistic guidelines: a) do it wrong; b) use a wrong combination; c) don't do enough; d) don't do nothing; e) do too much [5]. He also suggests building an "anti-parser" to compile BNF and produce structured garbage (erroneous test cases).

Marick [26] 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.) [26].

A domain-based testing tool called *Sleuth* [40] 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 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 performed by using the provided command syntax and semantic rules. Script definition handles the sequencing of commands [40].

Chapter 3

3.0 THE INPUT VALIDATION TEST METHOD

Input validation testing (IVT) is performed at the system level. Like transaction-flow testing, it focuses on the specified behavior of the system and uses a graph of the syntax of user commands. Like category-partition testing, input validation testing generates specification-based test cases. IVT incorporates formal rules in a test criterion that includes a measurement and stopping rule. The "anti-parser" idea of Beizer inspired the research undertaken in this paper and is the cornerstone of the approach. The practical error cases presented by Beizer and Marick form the basis of the "rule base" used to generate erroneous test cases. Several grammar analysis techniques have been applied as part of the static analysis of the input specification. This Chapter will discuss the four major aspects of the IVT method: how to specify the format of specifications; how to analyze a user command specification; how to generate valid test cases for a specification; and how to generate error test cases for a specification.

The input validation testing approach presented here differs from and extends the research discussed in Chapter 2 in a number of ways. It differs from Purdom, Bazzichi, Beizer, Marick, and von Mayrhauser because it formalizes the coverage criterion and because the approach has been experimentally validated. It extends Purdom, Beizer, and Marick's work by adding syntactic analysis. By generalizing the technique to be multi-domain, and by the virtue of the fact that the technique can be applied during the early life cycle phases, before design or code exist, it enhances/differs from Purdom, Bazichi, and von Mayrhauser. The technique is applied to input specifications and hence differs from Purdom and Bazzichi. The technique does not require the user to learn a new specification language or technique, and the technique requires minimal interaction from the user. In this way it differs from Purdom and von Mayrhauser [37,4,5,26,40]. Further, this technique can be used for functional testing as well. Functional testing selects test cases to assess the implementation of specific required functions [19].

The input validation testing problem has been decomposed into several sub-problems. These are:

1) How to specify the format of information in specifications and/or designs (section 3.1)

2) How to analyze a user command language specification (generally found in an SRS, IRS, SDD, IDD) (section 3.2)

3) How to generate valid test cases for a user command language specification (section 3.3)

4) How to generate erroneous test cases for a user command language specification (section 3.4)

The IVT Method addresses each of these problems, as discussed in the following sections. Some general concepts employed in the IVT Method include the test obligation database, test case table, and Microsoft Word file. The *test obligation database* is used to record defects detected during static analysis. If a defect is found (such as an overloaded token value), information on the specification table, the data element, and the defect are stored in the test obligation database. Each record represents a *test obligation*, the obligation to generate a test case to ensure that the static defect detected has not remained a latent defect in the as-built system. As part of generating erroneous test cases, a test case is generated for each record in the test obligation database. The *test case table* is used to record all the test cases that are generated. Each test case is stored as a single record. The *Microsoft Word file* is used to generate Test Plans/Cases in a standard Test Plan template/format.

3.1 How to Specify the Format of Information in Specifications and/or Designs

Interfaces are specified in as many different ways as there are systems and developers, and in many cases the interfaces are not specified at all. The IVT method is specification driven, thus is only useful for systems that have some type of documented interfaces. Examination of dozens of requirements and design specifications for as many systems showed that there were common elements that could be found in almost every user command language specification (presented in tabular form in every document examined). These elements are data element name or description, data element type, data element size, and expected/allowed values for the data element. Some example tables are shown below. Figure 3.1-1 presents an interface specification from an FBI Interface Design Document. The table is numbered IIIPTR and is named III Pointer, where III stands for Interstate Identification Index. The table has 7 data elements (SID through SEAL) and 7 columns. The columns are Sequence Number (Seq.), the Identifier column (ID), the Name column (has a description of each element), the Min and Max columns (indicates the smallest and largest number of characters or positions for an element),

the format column (where N indicates Numeric, B indicates Binary, A indicates Alphanumeric, and ANS indicates special alphanumeric), and the Code column (contains special semantic information).

IIIPTR III POINTER

Seq.	ID	Name	Min	Max	Frmt	Code
1	SID	STATE IDENTIFICATION NUMBER	3	10	ANS	
2	DPE	DATA POINTER ESTABLISHED	6	6	N	
3	DDE	DATE DECEASED OR EXPUNGED	6	6	N	
4	DECF	III DECEASED FLAG	1	1	B	
5	EXPF	III EXPUNGED FLAG	1	1	B	
6	FIF	FELONY IDENTIFICATION FLAG	1	1	A	FIF DISP CODE TABL
7	SEAL	RECORD SEAL FLAG	1	1	N	

Figure 3.1-1. Sample Table IIIPTR.

Figure 3.1-2 provides a table from a Tomahawk cruise missile Interface Requirement Specification (IRS). The table is numbered 3.2.4.2.1.1.3-5 and is named Product Generation Pointer Record. The table has 10 data elements (Record ID through Product Textual Description File) and has 4 columns. The columns are Item Number (Item No.), Item name (data element name), Description (of the data element and its expected value), and Format (length of data element and data type) where 11 char a/n indicates 11 alphanumeric characters.

Table 3.2.4.2.1.1.3-5 Product Generation Pointer Record

Item No.	Item Name	Description	Format
1	Record ID	Unique identifier for this Product Generation Pointer Record. Task ID (8 chars) Record Type Indicator (1 char indicating the type of product) Record Sequence Number (2 digit integer)	11 char a/n
2	Source Media	Media used to transfer the product generation parameter file to DIWS on-line storage. Value = ENET (for Ethernet)	4 char a/n
3	Number of Source Volumes	Indicates the number of tape volumes. Value is zero.	1 char a/n
4	Source Tape Label	Blank.	13 char a/n
5	Product Generation Parameter File Name	File name of the Product Generation Parameter File. Includes file name and file extension.	30 char a/n
6	Status	Indicates whether or not the Product Generation Parameter File is complete. Y=yes, complete; N=no, incomplete	1 char a/n
7	Source Media	Media used to transfer the product textual description file to DIWS on-line storage. Value = ENET.	4 char a/n
8	Number of Source Volumes	Indicates the number of tape volumes. Value is zero.	1 char a/n
9	Source Tape Label	Label on the Source Tape. This field is filled with blanks.	13 char a/n
10	Product Textual Description File	File name of the Product Textual Description File. Includes file name and file extension.	30 char a/n

Figure 3.1-2. Sample Table 3.2.4.2.1.1.3-5.

Figure 3.1-3 presents a table from a commercial system Program Design Document. The table is named "Attributes not part of doc_jecmics." The table has 9 data elements (JMX_imageStatusCode through JMX_pwdExpireDate) and 5 columns. The columns are Attribute ID (name od the data element), size (width of the field/data element), Type (data type such as Char or Long), Valid Values (for example, data element JMX_maxConnect should have a number representing minutes for a user's session), and Purpose (description of the data element valid values).

Table: Attributes not part of doc_jecmics

Attribute Id	Size	Type	Valid Values	Purpose
JMX_imageStatusCode	1	char		Image status Code
JMX_hitLimit	4	Long	Number	Number of hits from a query Database. default is 1000
JMX_mode	0	Long	JMX_HIGH_REV JMX_ALL_REV JMX_ONE_REV JMX_DWG_BOOK	type of revision. Default is highest revision.
JMX_drawingCount	4	long	Long	Number of drawings matching a criteria
JMX_sheetCount	4	long	Long	Number of Sheets matching a search criteria
JMX_frameCount	4	long	Long	Number of frames matching a search criteria.
JMX_maxConnect	4	Char	Number (min)	Identifies the maximum length of any session for the user, after which the user will be automatically logged out of the system.
JMX_maxIdle	4	Char	Number (min)	Identifies the maximum length of time in minutes that the user is permitted to be idle during any log-on session after which the user is automatically logged of the system
JMX_pwdExpireDate	18	Char	dd-mon-yy	The date a password expires.

Figure 3.1-3. Sample Table Attributes not part of doc_jecmics.

Figure 3.1-4 presents a table from a Navy Interface Design Document. The table is numbered 3.2.4.1.2 and is named Assignment Confirmation Message (ACM) Format. The table has 4 data elements (Message_Type through Confirmation) and 4 columns. The columns are Item No., Item Name (name of the data element), Description (of the data element and its expected value), and Format (length of data element and data type) where 8 char AN indicates 8 alphanumeric characters.

Table 3.2.4.1.2. Assignment Confirmation Message (ACM) Format

Item No.	Item Name	Description	Format
1.0	Message_Type	Identifies the message as an Assignment Confirmation (ASSGNCONF).	9 char A
2.0	Message_Date/Time	Specifies the date/time the message was generated. Format is YYMMDDHHMM. Example: (9001020310).	10 char N
3.0	Task_ID	Mandatory for all messages. The Task ID consists of: Originating Segment (1 char) D - DIWSA N - NIPS Task Category (1 char) J - JSIPS-N I - Training Task Type (1 char) D - Detailing Only S - Screening and Detailing Q - Database Query Sequence Number (5 char) Any unique alphanumeric combination. Valid characters are A-Z and 0-9. Example NJDABCD1	8 char AN
4.0	Confirmation	Indicates whether the task Assignment has been accepted. A - Task Assignment Message and file received; task accepted Q - Queue limit exceeded; task not accepted F - Non-existent file/directory error N - Non-existent node D - Insufficient disk space P - Insufficient privilege on file/directory (file protection violation) I - Internal Error; resend the Task Assignment Message R - Reduced Capability Mode - Task rejected	1 char A

Figure 3.1-4. Sample Table 3.2.4.1.2.

Early on in the research, it was envisioned that this problem of heterogeneous specification (table) formats might be handled by having the user specify the format using yacc. Based on the overarching design goal of minimizing required user interaction, two informal surveys of over 60 software testers (from across industry and government), and trials with experienced programmers and experienced testers, the design was changed to expect documents to conform to a generic format. User guidelines for pre-processing

specification tables of any possible format were developed. The IVT method expects a minimum of one data element per user command language specification table (these are also referred to as "Type 1" tables) and expects a minimum of three fields for the data element:

1. Data Element Name
2. Data Element Size
3. Expected/Allowable Values

3.2 How To Analyze A User Command Language Specification

A user command language specification defines the requirements that allow the user to interface with the system to be developed. The integrity of a software system is directly tied to the integrity of the system interfaces, both internally and externally [15]. There are three well accepted software quality criterion that apply to requirements specifications and interface requirements specifications: completeness, consistency, and correctness [22, 38, 7].

Requirements are *complete* if and only if everything that eventual users or customers need is specified [7]. The goal of minimizing user effort and to use existing textual, English specifications precluded a formal validation of completeness. Instead, the IVT method assesses the completeness of a user command language specification in two ways. First, as a specification table is being imported, the IVT method ensures that there are data values present for every column and row of the table. Second, the IVT method performs static analysis of the imported specification tables. At that point, the IVT method looks to see if there are hierarchical, recursive, or grammar production relationships between the table elements. For hierarchical and grammar production relationships, the IVT method checks to ensure there are no missing hierarchical levels or intermediate productions. If such defects are detected with the specification table, a "test obligation" will be generated and stored in the test obligation database. Any recursive relationships detected will be flagged by IVT as confusing to the end user and having the potential to cause the end user to input erroneous data. If recursive relationships are detected with the specification table, a "test obligation" will be generated and stored in the test obligation database.

Consistency is exhibited "if and only if no subset of individual requirements conflict" [7]. There are two types of consistency: internal and external. *Internal consistency* refers to the lack of conflicts between requirements in the same document. *External inconsistency* refers to the lack of conflicts between requirements in related

interface documents. In addition to analyzing user command language specification tables, the IVT method also analyzes input/output (or data flow) tables. These tables (also referred to as "Type 3" tables) are found in interface requirements specifications (IRS) and interface design documents (IDD) and are often associated with a data flow diagram. These tables are expected to contain the following fields:

- data element
- data element source
- data element destination

Optionally, the table may specify the data type, precision, accuracy, units, etc. for each data element. Figures 3.2-5 and 3.2-6 provide examples of such interface specifications (Type 3 tables) extracted from a Tomahawk cruise missile Software Requirements Specification (Table 3.4.1.40.3-1. Transfer Mission Data to MDDS Outputs) and a NASA System Requirements Specification (Table 6.5.1.2.2-1. Conceptual EOC Data Flows). Figure 3.2-5 presents a table named Transfer Mission Data to MDDS Outputs and numbered 3.4.1.40.3-1. The table has 3 data elements (user displays, status, Mission Ids) and 5 columns. The columns are Item (sequential item number), Description (name of the data element), Units of Measure (such as inch, feet, task), Frequency (how often the data element occurs), and Destination (the function that the data element is passed to).

Table 3.4.1.40.3-1 Transfer Mission Data to MDDS Outputs

Item	Description	Units of Measure	Frequency	Destination(s)
1	User displays	N/A	Variable	User
2	Status	N/A task	Once per Executive	Interactive DBA
3	Mission IDs	N/A	Once per task	Mission Data Transfer to MDDS

Figure 3.2-5. Sample Table 3.4.1.40.3-1.

Figure 3.2-6 presents the Conceptual EOC Data Flows table, numbered 6.5.1.2.2-1. The table has 3 sets of related functions (EOS to IMS, IMS to EOC, and EOC to DADS), 7 data elements (DAR_Platform_Info through

Mission_Historical_Data), and 4 columns. The columns are FROM (name of the function that generates the data item), TO (name of function that accepts the data element), data item, and description of the data item.

Table 6.5.1.2.2-1. Conceptual EOC Data Flows.

FROM	TO	DATA ITEM	DESCRIPTION
EOC	IMS	DAR_Platform_Info	Platform, including orbit information used in DAR generation.
		DAR_Status_Dialog	Information regarding the status of a DAR, including current information on when the observation will take place, why it won't, etc.
		Acq_Plan_Schedule	Instrument operations plans and schedules for user information.
IMS	EOC	DAR	A Data Acquisition Request (DAR), which specifies new data to be acquired by an instrument, constructed at the IMS and forwarded to the EOC for further processing. Also DAR updates.
		DAR_Status_Dialog	Request for current DAR information.
EOC	DADS	Platform_Status_Info	High-level information about the status of aplatform, US or foreign, or the SSF.
		Mission_Historical_Data	Information regarding EOS mission operations, including mission operations history.

Figure 3.2-6. Sample Table 6.5.1.2.2-1.

Tables generally have the form source, name, type, size, precision, and destination. All the tables of a document are examined. For each table, each record is examined. The value of the name column is examined along with its source. Each record in each table is then examined to determine if that same name value exists with the Destination field matching the Source field. If a match is not found, an error message is produced. If a match is found, each field in the matching record is compared. If type, size, or precision do not match, an error message is produced.

For example, if the table for Computer Software Component (CSC) Task Manager states that data element "task id" has a source of Task Manager and a destination of CSC Task Definition, then the table for CSC Task Definition must list data element "task id." In addition, the tables for CSC Task Manager and CSC Task Definition should have the exact same information for data element "task id" concerning data type, size, and precision. This is depicted below.

Table for CSC Task Manager:

Source	Field	Type	Size	Precision	Destination
Task Manager	task id	alpha	15	N/A	Task Definition
Task Manager	error code	integer	N/A	N/A	PC Manager

Table for Task Definition:

Source	Field	Type	Size	Precision	Destination
Task Manager	task id	alpha	15	N/A	Task Definition

If any of these consistency checks fail, an error report is generated. Note that no “test obligation” is generated.

That is because these data flow tables are not the subject of system level testing, but of integration testing.

The algorithm for performing these consistency checks is provided below:

```
algorithm       Type3ConsistencyCheck (A)
output          an error report file for tables in A
declare         found              :      boolean
                record, current    :      integer
                TabMode            :      {INTAB, OUTTAB}
                name               :      string
                CurRec,NextRec     :      tablename, io, description, frequency,
                                          source, destination
begin
 /* Read in all Type 3 tables */

 /* Get table name, input/output, description, source, destination of all tables */

 /* Perform consistency check */
 foreach table CurTab in A do
   foreach record CurRec in CurTab do
     if (CurRec.io = "input") then
          /* Use name in Source field and set Input flag */
          name := CurRec.source
          TabMode := INTAB
     else
          /* Use name in Destination field and set Output flag */
          name := CurRec.destination
          TabMode := OUTTAB
     endif;
          record := current + 1

      /* Loop through all records and look for matches for the Source or Destination of current record*/
       found := false
       foreach record NextRec in CurTab do
          if (CurRec.tablename = name) then
            /* If Input then search through all Output records */
            if (TabMode = INTAB) then
                if (CurRec.tablename = NextRec.destination) then
                    found := true
                    /* If match found, compare Description, Units of Measure, and Frequency of the 2
                            records */
                      if (CurRec.description != NextRec.description OR CurRec.units != NextRec.units OR
                       CurRec.frequency != NextRec.frequency) then
                         /* If anything does not match, write the 2 records and the error message to the file */
                         Write "Inconsistency Found" error message and NextRec and CurRec to report
                            file
                         break   /* out of foreach loop */
                      endif
                endif
```

```
            elseif (TabMode = OUTTAB) then
              begin
               if (CurRec.tablename = NextRec.source) then
                   /* If match found, compare Description, Units of Measure, and Frequency of the 2
                              records */
                    found := true
                    if (CurRec.description != NextRec.description OR CurRec.units != NextRec.units OR
                   CurRec.frequency != NextRec.frequency) then
                        /* If anything does not match, write the 2 records and the error message to the file */
                       Write "Inconsistency Found" error message and NextRec and CurRec to report file
                    endif
                endif
              end
             endif /*Input or output */
           endif  /* Table name found */
         endforeach  /* Each NextRec in CurTab  */
         /* If match is not found, write error message and current record to report file */
         if (not (found)) then
              Write "Source/Destination record not found" message and CurRec to report file
         endif
    endfor /* each record in CurTab */
  endfor /* each table in A */
end Type3ConsistencyCheck (A)
```

To assist with integration testing as well as the auditing of the integration testing effort, the IVT method provides reports of all the "From - To" (Source-Destination) relationships, by CSC name. That is, it provides a list of all the relationships that should be exercised as part of integration testing. A tester could build test cases from this list. An auditor could use this list when examining the integration test folders and perform spot checks to ensure that all relationships are indeed being tested.

Based on the two tables shown in Figures 3.2-7 and 3.2-8, the checklists shown in Figures 3.2-9 and 3.2-10 are generated. The checklist in Figure 3.2-9 indicates that there needs to be a test case for the Route Analysis Executive to Launch Area Analysis interface, the Route Data Access to Launch Area Analysis interface, and the Requested subfunction to Launch Area Analysis interface.

Table – 1. Launch Area Analysis (Inputs)					
Item	**Description**	**Units of Measure**	**Frequency**	**Legality Check**	**Source(s)**
1	Analysis Request	mixed	per request	No	Route Analysis Executive
2	Route data	mixed	as needed	No	Route Data Access
3	Status message	mixed	per request	No	Requested subfunction

Figure 3.2-7. Sample Table - 1.

Table – 2. Route Analysis Executive (Outputs)				
Item	**Description**	**Units of Measure**	**Frequency**	**Destinations**
1	Analysis Request	mixed	as needed	Automatic Vertical Profile Executive, Navigation Accuracy Module, CAPTAIN Launch Footprint Analysis, Flight Simulation, Fast Clobber, Single Mission Attrition Analysis, GPS Jamming Analysis, Performance Analysis, Launch Area Analysis Flexible Targeting Grid Generation
2	Mission ID Request	mixed	as needed	DM
3	Retrieve route request	mixed	as needed	Route Data Retrieval
4	Store route request	mixed	as needed	Route Data Storage
5	DIWS product status request	mixed	per request	PC
6	Set current route request	mixed	as needed	Route Data Editor
7	Route object edit	mixed	per request	Route Data Editor
8	Route data request	mixed	per route	Route Data Access
9	Test error flag request	mixed	as needed	Route Data Error Handler
10	Report containing summary of analysis results	mixed	as needed	user
11	Status message	mixed	per request	Requesting subfunction

Figure 3.2-8. Sample Table - 2.

Required Tests for Launch Area Analysis (Inputs)

Route Analysis Executive to Launch Area Analysis interface
Route Data Access to Launch Area Analysis interface
Requested subfunction to Launch Area Analysis interface

Figure 3.2-9. Required Tests Report for Launch Area Analysis.

Required Tests for Route Analysis Executive (Outputs)

Route Analysis Executive to Automatic Vertical Profile Executive interface
Route Analysis Executive to Navigation Accuracy Module interface
Route Analysis Executive to CAPTAIN Launch Footprint Analysis interface
Route Analysis Executive to Flight Simulation interface
Route Analysis Executive to Fast Clobber interface
Route Analysis Executive to Single Mission Attrition Analysis interface
Route Analysis Executive to GPS Jamming Analysis interface
Route Analysis Executive to Performance Analysis interface
Route Analysis Executive to Launch Area Analysis interface
Route Analysis Executive to Flexible Targeting Grid Generation interface
Route Analysis Executive to DM interface
Route Analysis Executive to Route Data Retrieval interface
Route Analysis Executive to Route Data Storage interface
Route Analysis Executive to PC interface
Route Analysis Executive to Route Data Editor interface
Route Analysis Executive to Route Data Access interface
Route Analysis Executive to Route Data Error Handler interface
Route Analysis Executive to user interface
Route Analysis Executive to Requesting subfunction interface

Figure 3.2-10. Required Tests for Route Analysis Executive.

In addition, for tables such as the one shown in Figure 3.2-9:

Table – 3. Launch Area Analysis (Inputs)

Item	Description	Units of Measure	Frequency	Legality Check
1	Analysis Request	mixed	per request	No
2	Route data	mixed	as needed	No
3	Status message	mixed	per request	No

Figure 3.2-11. Sample Table - 3.

the IVT method will generate a list of every data element that has Legality Check marked “No.” The “No” indicates that the software is not designed to perform a legality check on the data element, to see if it is the

correct data type, has the appropriate units of measure, etc. This list can be used to ensure that input validation testing is performed during integration testing for these data elements.

Software *correctness* is defined in IEEE 729-1983 as "The extent to which software meets user expectations" [21]. Davis defines correctness as existing "if and only if every requirement stated represents something that is required" [7]. Although 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. Note that this is completely divorced from completeness. The IVT method does not address correctness of requirements.

In addition to the three quality criteria of completeness, consistency, and correctness, the IVT method performs three additional checks on Type 1 tables (user command language specification tables containing syntactic information).

1) Examine data elements that are adjacent to each other. If no delimiters are specified (such as ']', '/', ',', ']'), the IVT method will look to see if the same data type or expected value are adjacent. If the elements have identical expected values, or if they have identical data types with no expected values, a "test obligation" is generated. The danger of such interface design is that the two elements might be concatenated if the user "overtypes" one element and runs into the next element, and catenation of the two might be incorrectly processed. This check lends its roots to a grammatical catenation check that ensures that catenation of adjacent pairs of tokens are not incorrectly parsed and/or scanned.

2) Check to see if a data element appears as the data type of another data element. For example, suppose data element 1 is named All_Alpha and has these properties: data type: not specified, size: 5, expected value: AAAAA. Data element 5 is named Alpha_Too and has these properties: data type: All_Alpha, size: not specified, expected value: not specified. If IVT detects such a case, it informs the user that the table elements are potentially ambiguous and a test obligation is generated. The algorithm for performing the ambiguous grammar check is provided below:

```
foreach record in CurTab do
  if (record.name = NextRec.name) then
    Write error message and write entry to test obligation database
  endif
  if (record.class_values = Yes) then
```

```
    if (record.class = NextRec.class) then
      Write error message and entry to test obligation database
    endif
  endif
endforeach
```

3) Check to see if the expected value is duplicated for different data elements. For example, the expected values "D" and "J" are repeated for elements Originating Segment and Task Type in the following table:

Character Number	Description	Value	Value Description
1	ORIGINATING SEGMENT	D	DIWSA
1	ORIGINATING SEGMENT	N	NIPS
2	TASK CATEGORY	J	JSIPS-N
2	TASK CATAGORY	I	Training
2	TASK CATEGORY	P	Production
3	TASK TYPE	D	Detailing Only
3	TASK TYPE	J	Screening and Detailing
3	TASK TYPE	C	Image Catalog Update Only
3	TASK TYPE	Q	Database Query

This is a potential poor interface design because the user might type a "D" in the Task Type slot, but meant to indicate "DIWSA" for originating Segment. Or the user might type a "J" in the Originating Segment position, but really meant "Screening and Detailing" for Task Type. This is synonymous to a grammar having overloaded token values. If IVT detects such a case, it informs the user that the table elements are potentially ambiguous and a test obligation is generated.

The data structure that stores the table information is defined below:

define
Record: element #, elementname, position, class_of_values, size, class, value #, value
CurRec: Record
CurTab: Table of Records
AllTabs: Table of CurTab
enddefine

For the table shown in Figure 3.2-12 above, a graphical representation might be depicted as shown in Figure 3.2-13.

Table 3.2.2-1

TASK IDENTIFICATION (TASK ID)

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 THRU 8	SEQUENCE NUMBER	ALPHANUMERIC

Figure 3.2-12. Sample Type 1 Table 3.2.2-1.

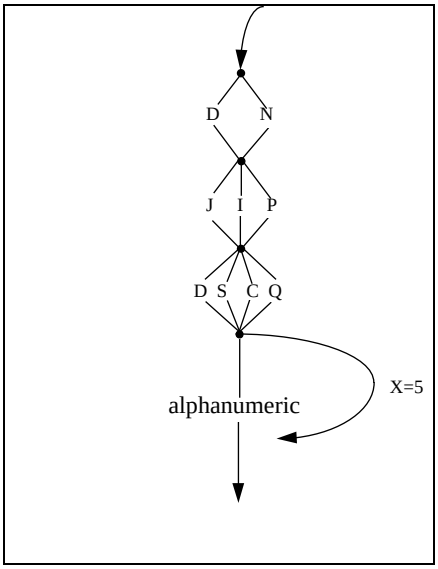


Figure 3.2-13. A Graphical Representation of Type 1 Table 3.2.2-1.

The above mentioned static checks of the Type 1 Tables (overloaded token value, ambiguous grammar) are performed using the following two algorithms, Overloaded Token Value and Ambiguous Grammar:

```
algorithm OverloadedTokenValue
define
        Element:  Record of element #, elementname, position, class_of_values, size, class,
                        value #, value
        CurTab:  Table of elements
        i, j, c:    integer
        v:          value
        A:          array of values

begin    /* overloaded token */

read in CurTab
c = 1
foreach Element in CurTab do
  V = GetValue (Element)
  A[C] = V
  C = C + 1
endforeach

for I = 1 to Size (A) do
  for J = 2 to Size (A) do
    if (A[I] = A[J]) then
      Write error message and record to test obligation database
endfor /* J */
endfor /* I */

end OverloadedTokenValue /* overloaded token */
```

```
algorithm AmbiguousGrammar
define
        class_of_values = {true, false}
        Element: Record of element #, elementname, position, class_of_values, size, class,
                        value #, value
        CurTab:  Table of elements
        i,j,c:   integer
        v:       value
        A:       array of v
        X:       class_of_values
        Y:       class
        B:       array of X
        D:       array of Y


begin   /* ambiguous grammar */

read in CurTab

c = 1
foreach Element in CurTab do
  V = GetName (Element)
  A[C] = V
  X = GetClassofValue (Element)
  Y = GetClass (Element)
  B[C] = X
  D[C] = Y
  C = C + 1
endforeach

for I = 1 to Size(A) do
  for J = 2 to Size(A) do
    if A[I] = A[J] then
      Write error message and record to test obligation database
    endif
    if (B[I] = Yes) then
      if (D[I] = A[J]) then
        Write error message and record to test obligation database
      endif
    endif
endfor /* J */
endfor /* I */

end AmbiguousGrammar /* ambiguous grammar */
```

3.3 How To Generate Valid Test Cases for a User Command Language Specification

The user command language specification is used to generate a covering set of test cases. One can think of the syntax graph as being similar to a program control flowgraph [5,11,12,18]. The all-edges testing criterion [11] is adapted to generate test cases that cover the syntax graph. The processing involves traversing the syntax graph and generating test cases to satisfy the all-edges testing criterion. Each data element is considered a node in the syntax graph. Valid, covering test cases are annotated as such. Many user command specifications contain loops. To ensure the IVT method is as general as possible, it has been designed to handle loops. To handle loops, the following heuristic will be used [5, 26]:

0 times through the loop
1 time through the loop
X times through the loop
X+1 times through the loop

After the user command specification (table information) has been read in and statically analyzed, the following algorithm is used to generate covering test cases and expected results:

define

Record:	TableName, Element_Num, ElementName, Position, Class_of_Values, Size, Class,Value_Num, Value
CurRec:	Record
CurTab:	Table of Record
AllTabs:	Table of CurTab
V:	TableName
W, CurValue:	CurRec.Value
I:	Integer
Loop_Handler:	[Once, X, X_Plus_One, Zero]
Expected_Outcome:	[Valid, Invalid]

```
begin  /* covering test cases */
  foreach Table CurTab in AllTabs do
    V = Get(CurTab.TableName)
    write V to MS Word file and test case table
    foreach Record CurRec in CurTab do
      while (CurRec.ElementName != PrevRec.ElementName) do
        write CurRec.ElementName to MS Word file and test case table
        I = 1

        /* If not Class of Values (e.g., expected values given instead of class like char, alpha, integer), write
              the current expected value (CurRec.Value[I] to MS Word file */
```

```
        if (CurRec.Class_of_Values = No) then
          write CurRec.Value[I] to MS Word file and test case table
          CurValue = CurRec.Value[I]
           I = I + 1
        else   /* it is Class of Values  */
          for Loop_handler = Once to Zero do

           /* Handle the loop 0, 1, X, and X + 1 times test cases  */
            CASE Loop_handler of
              Once: /* 1 time through loop */
                 begin
                      /* Select_Value selects a value from the class of values (an integer or alpha, e.g.) */
                    W = Select_Value(CurRec.Value)
                     write W to MS Word file and test case table
                     /* Size_Check returns VALID if I is =< CurRec.Size and INVALID otherwise  */
                       Expected_Outcome = Size_Check(I)
                        I = I + 1
                end;  /* Once  */
              X:     /* X times through the loop  */
                  begin
                    while (CurRec.ElementName != PrevRec.ElementName) do
                       W = Select_Value(CurRec.Value)
                        write W to MS Word file and test case table
                        I = I + 1
                        CurRec = GetNext(CurTab)
                      endwhile
                      Expected_Outcome = Valid
                    end  /* X */
              X_Plus_One:   /* X + 1 times through the loop  */
                    begin
                       while (CurRec.ElementName != PrevRec.ElementName) do
                          W = Select_Value(CurRec.Value)
                          write W to MS Word file and test case table
                          I = I + 1
                            CurRec = GetNext(CurTab)
                        endwhile
                       W = Select_Value(CurRec.Value)
                        I = I + 1
                        Expected_Outcome = Invalid
                     end  /* X_Plus_One  */
              Zero:    Expected_Outcome = Invalid
            endcase
          endfor  /* Loop_Handler */
        endif /* Class of Values = No */
      endwhile  /* ElementNames not equal  */
    endforeach   /* Record CurRec  */
    write Expected_Outcome to MS Word file and test case table
  endforeach  /* Table CurTable */
end  /* covering test cases */
```

Example

Assume a specification table with four data elements (see figure 3.2-12): Originating Segment, Task Category, Task Type, and Sequence Number. Sequence Number is represented by a class of values (loop), it is ALPHANUMERIC and has a size of 5. Recall that static analysis must be run prior to test case generation. During static analysis, checks are made to ensure that there are no blank fields in the table and checks are made to determine whether or not a data element is represented by a class of values.

The procedure starts by reading in the table and writing the table name to the MS Word file and to the MS Access test case table. The procedure then processes each data element in the table.

First Three Data Elements. The first element name (Originating Segment) is written to the MS Word file and test case table. Class of Values is NO for Originating Segment so the Ith actual value for Originating Segment (I is 1, so the value from the table is D) is written to the MS Word file and test case table. The procedure moves to the next data element, Task Category. The element name (Task Category) is written to the MS Word file and test case table. Class of Values is NO for Task Category so the Ith actual value for Task Category (I is 1, so the value from the table is J) is written to the MS Word file and test case table. The procedure moves to the next data element, Task Type. The element name (Task Type) is written to the MS Word file and test case table. Class of Values is NO for Task Type so the Ith actual value for Task Type (I is 1, so the value from the table is D) is written to the MS Word file and test case table. The procedure moves to the next data element, Sequence Number. The element name (Sequence Number) is written to the MS Word file and test case table. Class of Values is YES for Sequence Number.

For Loop_Handler. At this point, the For loop commences. The CASE statement is executed the first time with Loop_handler equal to Once. Sequence Number is type ALPHANUMERIC. Routine Select_Value is called and randomly selects an ALPHANUMERIC value, A in this case. A is written to the MS Word file and test case table. Expected_Outcome is set to Invalid because Size_Check sees that I is not equal to the current record size of 5. “Invalid” is written to the MS Word file and test case table. The processing described above under First Three Data Elements is performed at the bottom of the loop, and then the For loop is executed again

for Loop_handler equal to X. This time, the while loop is executed, with Select_Value being called 5 times (the master table that was built during static analysis contains five records numbered 4 through 8 with Element Name of Sequence Number). Expected_Outcome is set to Valid and is written to the MS Word file and test case table.

The processing described above under First Three Data Elements is performed at the bottom of the loop, and then the For loop is executed again for Loop_handler equal to X_Plus_One. This time, the while loop is executed, with Select_Value being called 5 times (the master table that was built during static analysis contains five records numbered 4 through 8 with Element Name of Sequence Number). After the while loop, Select_Value is called one more time and a 6^{th} value is written to the MS Word file and test case table. The Expected_Outcome is set to Invalid and is written to the MS Word file and test case table. The processing described above under First Three Data Elements is performed at the bottom of the loop, and then the For loop is executed a final time for Loop_handler equal to Zero. This time, no value for Sequence Number is written to the MS Word file or test case table. The Expected_Outcome is set to Invalid and is written to the MS Word file and test case table.

For the example above, the following information would be written to the MS Word file:

Covering Test Cases for 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 = X)
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 = X+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

In addition, the following would be written to the Test Case Table:

Table 3.2.2-1

DJDA	Invalid
DJDA1R3Z	Valid
DJDT78PZ3	Invalid
DJD	Invalid

Consider the following table with no loops in the syntax graph (expected values such as “D”, “N”, “J” are specified as opposed to alpha, char, real).

Record Number	Character Number	Description	Values	Class of Values
1	1	ORIGINATING SEGMENT	D	No
2	1	ORIGINATING SEGMENT	N	No
3	1	ORIGINATING SEGMENT	J	No
4	2	TASK CATEGORY	I	No
5	2	TASK CATEGORY	P	No
6	3	TASK TYPE	S	No
7	3	TASK TYPE	C	No
8	4	TASK NUMBER	1	No
9	4	TASK NUMBER	2	No

For this table, the following information would be written to the MS Word file:

Covering Test Cases for Table 3.2.2-1

Originating Segment = D
Task Category = I (No Loop)
Task Type = S Expected_Outcome = Valid
Task Number= 2

In addition, the following would be written to the Test Case table:

Table 3.2.2-1

DIS2 Valid

3.4 How To Generate Erroneous Test Cases for a User Command Language Specification

There are two sources of rules for generating erroneous test cases: the error condition rule base; and the test obligation database. The error condition rule base is based on the Beizer [5] and Marick [26] lists of practical error cases discussed in Section 2.5. The test obligation database is built during static analysis as discussed in section 3.2. Error case generation is discussed below.

1) *Error Condition Rule Base*. Four types of error test cases are generated based on the error condition rule base:

- Violation of looping rules when generating covering test cases. The rules for modifying graphs with loops (field length) were provided in Section 3.1 in the Covering Test Cases algorithm
- Top, intermediate, and field-level syntax errors. A wrong combination is used for 3 different fields, the fields are inverted (left half of string and right half of string are swapped with the first character moved to the middle) and then the three inverted fields are swapped with each other (reference 5 suggests using a wrong combination and reference 26 suggests a sequence order, our method extends these with the inversion and swap) – the algorithm is presented below
- Delimiter errors. Two delimiters are inserted into the test case in randomly selected locations (reference 5 suggests violating delimiter rules) - the algorithm is presented below
- Violation of expected values. An expected numeric value will be replaced with an alphabetic value, and an expected alphabetic value will be replaced with a number (reference 26 suggests violating allowable alternatives) – the algorithm is given below.

algorithm TopIntermediateFieldSyntaxErrors
define

Record:	TableName, Element_Num, ElementName, Position, Class_of_Values, Size, Class,Value_Num, Value
CurRec:	Record
I:	Integer
Lefthalfstring, Righthalfstring:	String
First_character:	Char
First_several_characters:	String
TmpStr:	String
TestCaseSet:	Array of String

```
TestCaseSet = 0
begin  /* top, intermediate, and field-level syntax errors  */
  store old test
  for I = 1 to 2 do
     randomly select Element_Num     /* an element to be manipulated   */
       /* random number generated that indexes into CurRec*/
     store the random number
      /* to ensure same value not selected second time through loop  */
```

```
    if (CurRec.Class_of_Values = Yes and CurRec.Class = alphanumeric) then
      make copy of CurRec.Value
       /* get right half of string */
      RightHalfStr = right half of CurRec.Value
      /* get the first character of string */
      First_character = first character of CurRec.Value
      /* get the left half of string minus the first character  */
      Lefthalfstring = left half of string minus the first character
       /* reassemble string */
      TmpStr = lefthalfstring + first_character + righthalfstring
    else
      make copy of CurRec.Value
      /* get right half of string */
      RightHalfStr = right half of CurRec.Value
      /* get the first character of string */
      First_character = first character of CurRec.Value
      /* get first several characters of string */
      First_several_characters = left half of string minus the first several characters
      /* reassemble string */
      TmpStr = lefthalfstring + first_ several_characters + righthalfstring
    endif
    /* compare new test case to old test case to ensure not duplicate */
    if (TmpStr != TestCaseSet) then
      /* write test case to test case table */
      TestCaseSet = TestCaseSet U {TmpStr}
    endif
  endloop
end TopIntermediateFieldSyntaxErrors /* top, intermediate, field-level syntax errors  */
```

```
algorithm          DelimiterErrors
define
  RandNum1, RandNum2:          Integer
  Firstthird, secondthird:     String
  RandSet:                     Array of integer
  Record:                      TableName, Element_Num, ElementName, Position, Class_of_Values,
                               Size, Class,Value_Num, Value
  CurRec:                      Record
  FirstThirdStr, SecondThirdStr:  String
  RestStr, TmpStr:          String
  TestCaseSet:                 Array of String


RandSet = 0
TestCaseSet = 0
begin  /* delimiter errors  */
    / *randomly select 1 number, these character numbers will be manipulated   */
    RandSet = RandSet U {GetRandNum()}
     /* to ensure same values not selected second time through loop  */
```

```
      if (RandNum1 > RandNum2) then
         switch the values
         /* want RandNum1 to be less than RandNum2  */
      endif
       /* get first third of test  */
      FirstThirdStr = First third of CurRec.Value   /* (from character 1 to RandNum1) */
      /* get second third of test */
      SecondThirdStr = Second third of CurRec.Value   /* (from RandNum1 to RandNum2)*/
      /* get rest of the test */
      RestStr = Remainder of CurRec.Value     /* (from RandNum2 to end)*/
      TmpStr = FirstThirdStr + randDel() + SecondThirdStr + randDel() + RestStr
        /* RandDel() returns a randomly selected delimiter  */
      /* write test case to test case table*/
      TestCaseSet = TestCaseSet U {TmpStr}
end DelimiterErrors /* delimiter error  */


algorithm          ViolateAllowableValues
define
  Record:                          TableName, Element_Num, ElementName, Position, Class_of_Values,
                                   Size, Class,Value_Num, Value
  CurRec:                          Record
  CurTab:                          Table of Record
  RandNum1, RandNum2, I:           Integer
  Firstthird, secondthird:         String
  X:                               Char

begin  /* violate allowable values  */
foreach CurRec in CurTab do
     randomly select Element_Num     /* an element to be manipulated   */
      /* random number generated that indexes into CurRec*/
      if CurRec.Values = (integer, number, numeric, real, float,decimal, binary) then
         /* get random alphabetic value, X */
          X = RandAlpha()
      else
         /* get random numeric value, X */
         X = RandNum()
      /* replace randomly selected position of CurRec.Value with X */
      I = GetRand(1,size(CurRec())
      CurRec[I] = X
     write test case to test case table
  endforeach
end ViolateAllowableValues /* violate allowable rules */
```

2) *Test Obligation Database*. Two types of error test cases are generated based on the test obligation database:

- 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 (section 3.2). The algorithm for generating these test cases is given below

- Catenation static error. The values that were identified as possibly catenating each other (user accidentally "types" information into the next field since adjacent fields have the same data type, no expected values, and no delimiters) are duplicated into the adjacent fields – the algorithm is given below.

```
define
  Record:                  TableName, Element_Num, ElementName, Position, Class_of_Values,
                           Size, Class,Value_Num, Value
  CurRec:                  Record
  CurTab:                  Table of Record
  TestObRecord:            ErrorCode, AmbCharNum, AmbigValue, CharNum
  TestObRec:               TestObRecord
  TestObTab:               Table of TestObRec
  RandNum1, RandNum2:      Integer
  Firstthird, secondthird: String
  tempvalue:               Char

begin  /* static error test cases  */
foreach TestObRec in TestObTab do
  load current test case for CurRec corresponding to TestObRec
  if TestObRec.ErrorCode = 1 then
    current test case(TestObRec.AmbCharNum) = TestObRec.AmbigValue
    current test case(TestObRec.CharNum) = TestObRec.AmbigValue
    write new test case and "Valid/Overloaded Token Static Error" to test case table
  elseif TestObRec.ErrorCode = 2 then
    tempvalue = current test case(TestObRec.CharNum)
    current test case(TestObRec.AmbigCharNum) = tempvalue
    write new test case and "Invalid/Catenation Static Error" to test case table
  endif
write test case to test case table
endforeach
end  /* static error test cases */
```

Appendix A presents the MS Word file as well as the test case report (includes the covering test cases and the error test cases) for a sample interface specification table (also in Appendix A).

In summary, Chapter 3 has presented the general concepts of the IVT method: test obligation database, MS Word file, and the test case table. In addition, algorithms and examples were presented for how the IVT method handles specifying the format of information in specifications and/or designs, analyzing a user command language specification, generating valid test cases for a user command language specification, and generating erroneous test cases for a user command language specification.

Chapter 4

4.0 MICASA: A PROTOTYPE SYSTEM

In order to demonstrate the effectiveness of the IVT method, a prototype system was developed. This enabled us to evaluate the method and examine its usefulness for real-life interface specifications. This chapter describes the design and implementation of the prototype system, Method for Input Cases and Static Analysis (MICASA, pronounced Me-Kass-Uh) based on the IVT method.

System Description

MICASA runs under Windows NT. It is written in Visual C++ and relies heavily on MS Access tables and databases. The system was developed to meet two major objectives:

1) To implement and demonstrate the feasibility and effectiveness of the IVT method; and

2) To minimize the inputs needed from the user.

A high level architecture was defined and is shown in Figure 4.0-1.

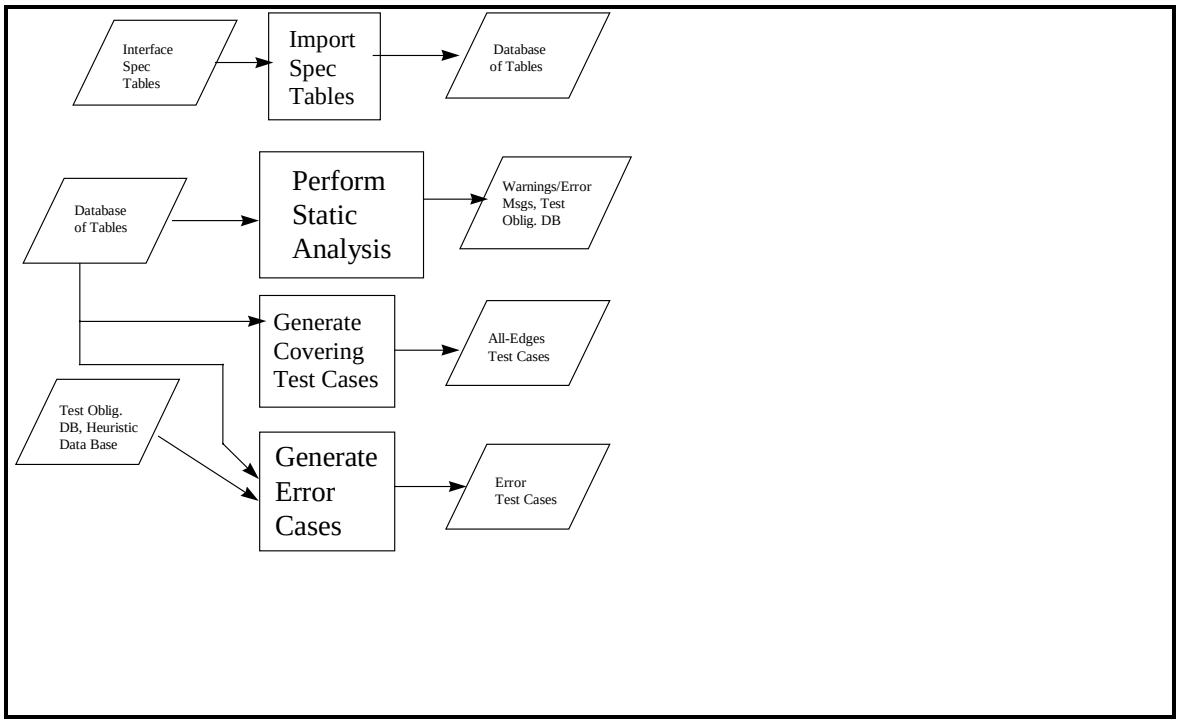


Figure 4.0-1. MICASA Architecture.

To achieve the second goal, a small graphical user interface was designed for MICASA. The typical screen has four clickable buttons and options of <Cancel>, <Back>, <Next>, and <Finish> available to the user. Very little input is required from the user, most of the interaction is with radio buttons. MICASA leads users sequentially through the steps of the IVT method. The flow of the screens of IVT is shown in Figure 4.0-2.

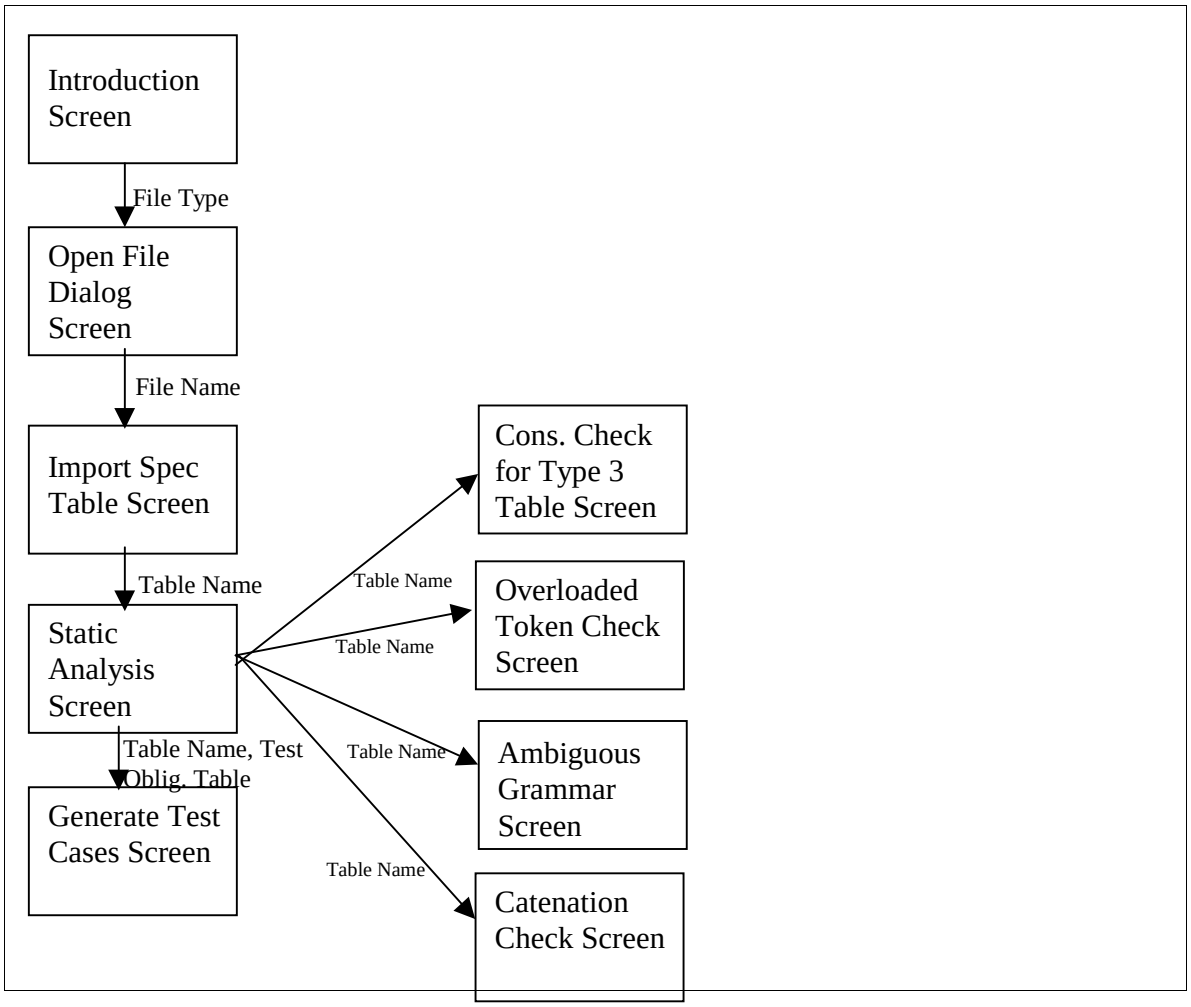


Figure 4.0-2. MICASA Screen Flow.

Import Specification Tables

As described in Chapter 3, the Import Spec Tables function allows the user to import information from a digital interface specification. MICASA accepts flat files and MS Word files of the interface specification tables. The processing steps performed by this function are:

a) In the introduction screen (figure 4.0-3), the user is asked if the table is type 1 or 3 (defined in section 3.1 and 3.2). If the user indicates it is a type 3 table, MICASA asks if consistency checking is to be performed on an entire Computer Software Configuration Item (CSCI).

b) The user specifies the name of the file(s) to be imported.

c) MICASA imports the provided file(s) into MS Access tables. The output from this function is an MS Access database of tables for the interface specification. Figure 4.0-4 depicts the MICASA screen for the Import Spec Tables function.

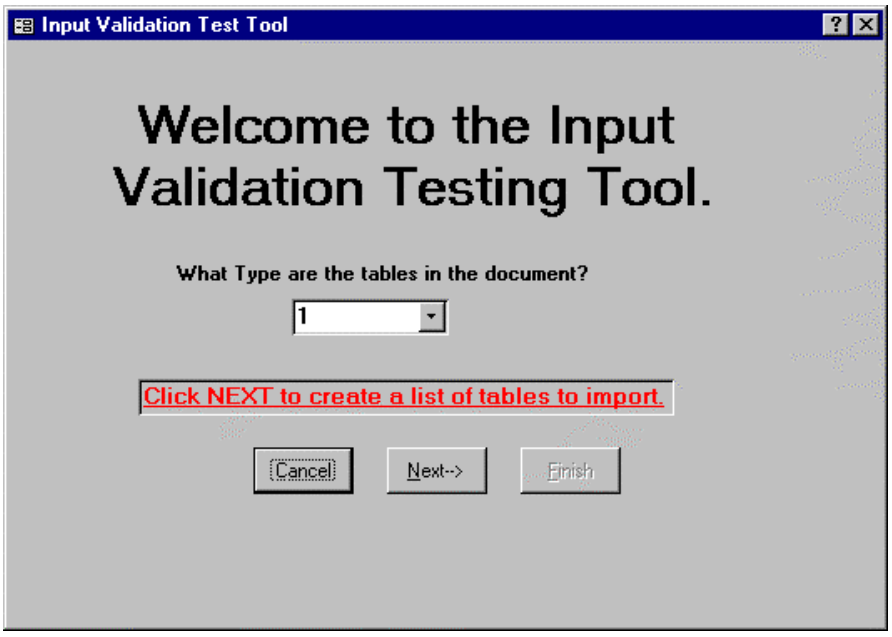


Figure 4.0-3. Introduction Screen.

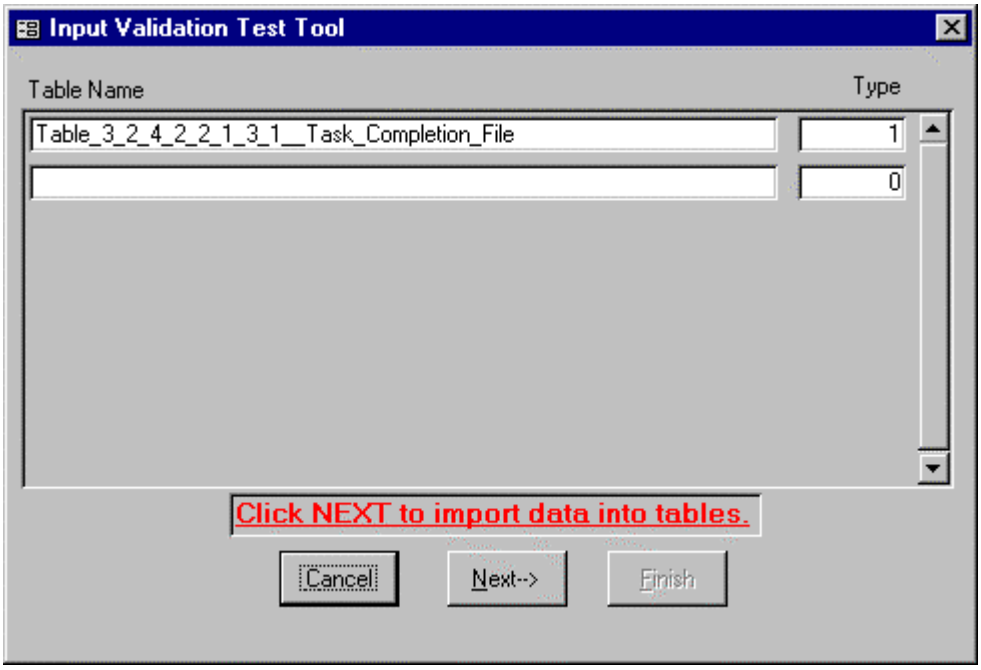


Figure 4.0-4. Import Data into Tables.

Perform Static Analysis

As described in Chapter 3, the Perform Static Analysis function allows the user to perform a number of static checks on the interface tables: consistency, completeness, ambiguous grammar, overloaded token, and potential catenation. The input is the interface table information that is created by Import Spec Tables and is stored in the MS Access database. The processing steps performed by this function are:

a) The user can perform static analysis as shown in Figure 4.0-5.

b) When the main table is created, the user can initiate a consistency check on the table, as shown in Figure 4.0-6.

c) The user can check for overloaded tokens, as shown in Figure 4.0-7.

d) The user can check for ambiguous grammar, as shown in Figure 4.0-8.

e) The user can check for possible catenation errors, as shown in Figure 4.0-9.

The output from this function is a set of MS Access database tables containing error records, as well as printouts of these error reports. The Ambiguous Grammar Error report format is shown in Figure 4.0-10. The Overloaded Token Error report format is shown in Figure 4.0-11. The Catenation Error report format is shown in Figure 4.0-12.

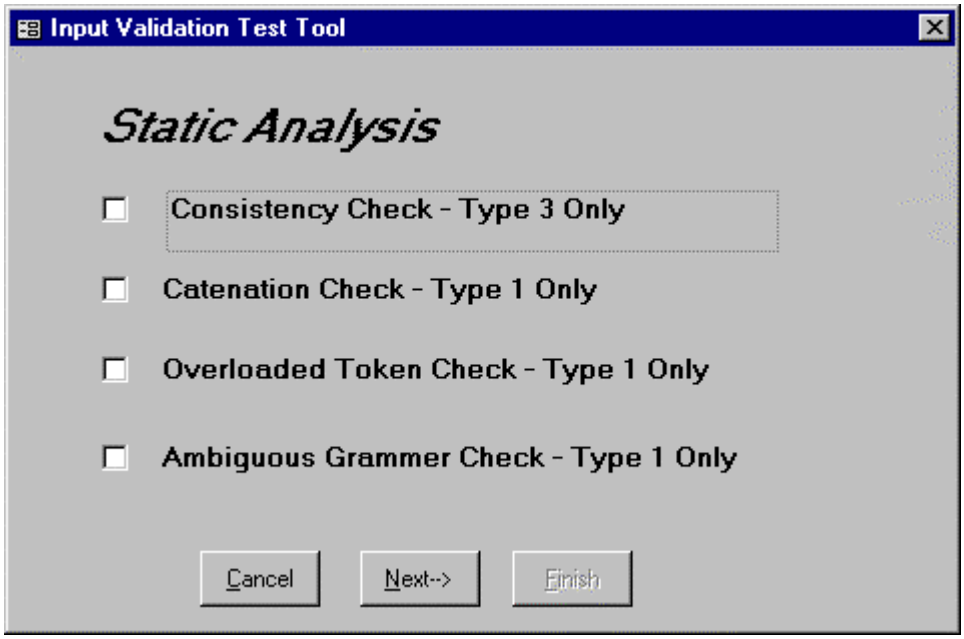


Figure 4.0-5. Static Analysis.

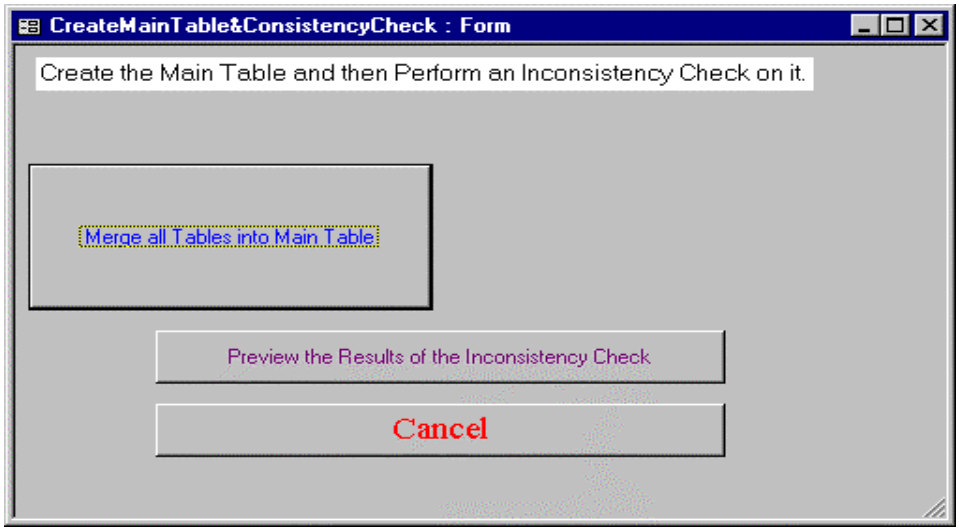


Figure 4.0-6. Consistency Check.

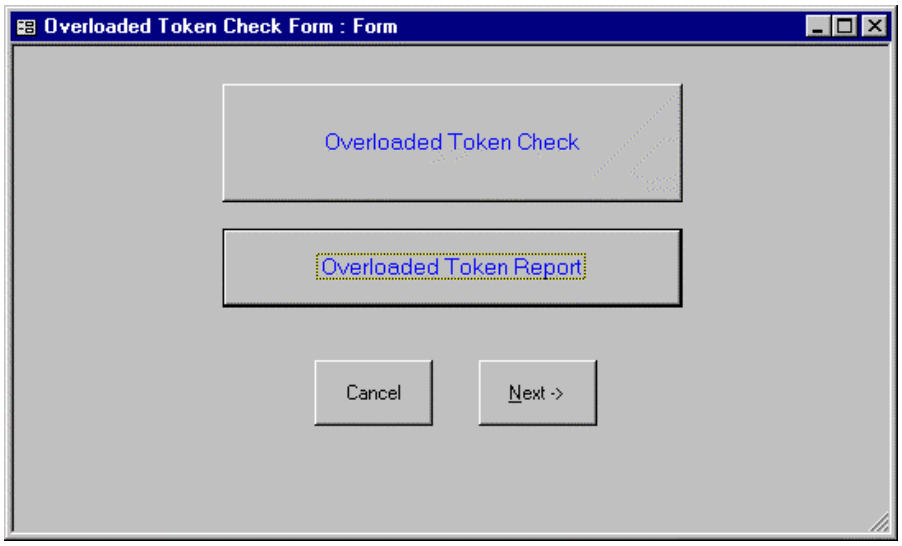


Figure 4.0-7. Overloaded Token Check.

Figure 4.0-8. Ambiguous Grammar Check.

Figure 4.0-9. Check for Possible Catenation Errors.

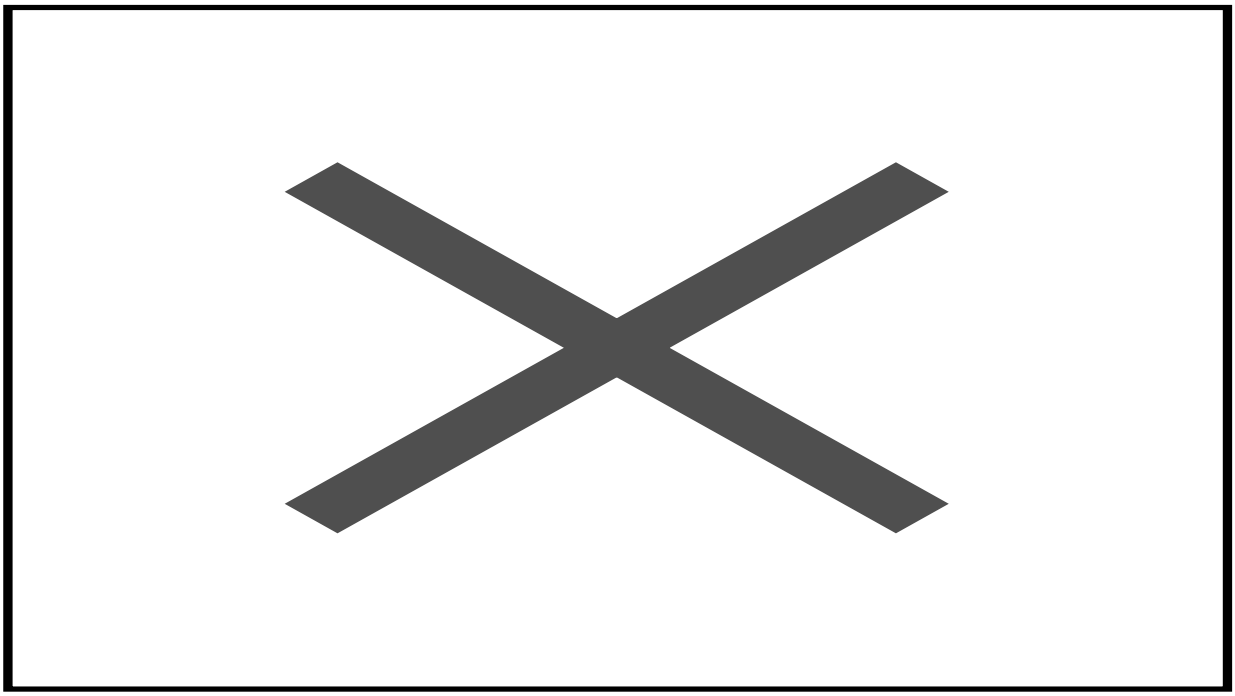


Figure 4.0-10. Ambiguous Grammar Report.

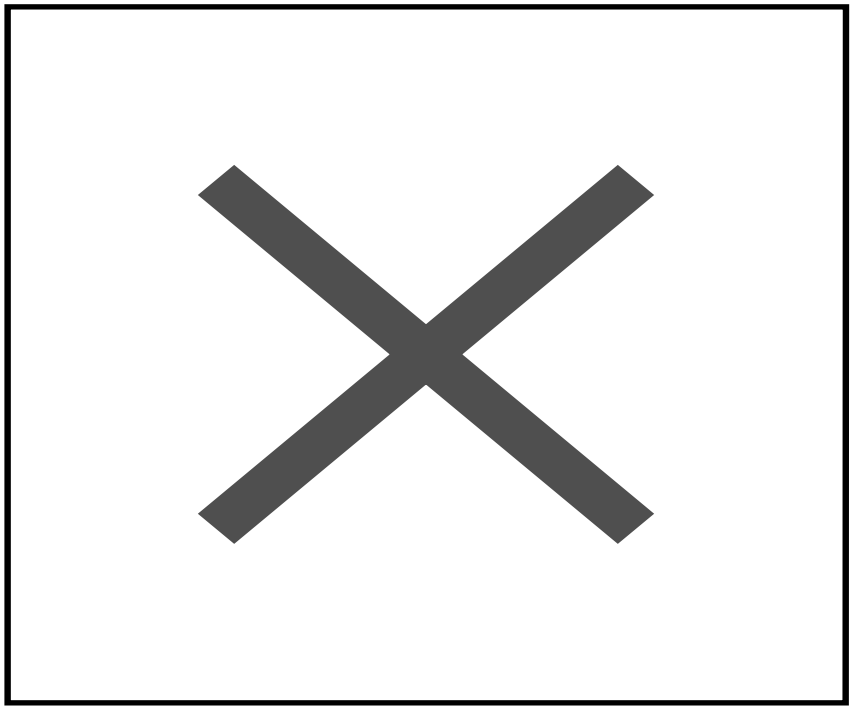


Figure 4.0-11. Overloaded Token Error Report.

Possible Catenation Error

Wednesday, June 17, 1998

Table Name	Id Field	Error Type	Error	Ambiguous Char #	Char #
Table___doc_jecmics	0	Warning	Catenation Warning: Possible Catenation Error.	422	421
Table___doc_jecmics	0	Warning	Catenation Warning: Possible Catenation Error.	400	399
Table___doc_jecmics	0	Warning	Catenation Warning: Possible Catenation Error.	368	367
Table___doc_jecmics	0	Warning	Catenation Warning: Possible Catenation Error.	263	262
Table___doc_jecmics	0	Warning	Catenation Warning: Possible Catenation Error.	220	219

Figure 4.0-12. Catenation Error Report.

Generate Covering Test Cases

As described in Chapter 3, the Generate Covering Test Cases function allows the user to generate all-edges test cases for the Type 1 interface tables stored in the MS Access tables. The input is the interface table information as stored in the MS Access database. The processing performed by this function is that the all-edges testing criterion is applied to generate covering test cases. If there is a loop in the syntax graph (field length (X) is greater than 1 and only a data type is specified) for one of the tables, a heuristic is used. The loop is executed 0 times, 1 time, X times, and X+1 times.

The user will be asked to enter the document and system name, as shown in Figure 4.0-13. The output from this function is a set of MS Access database tables containing test cases. These can be displayed in MS Access, or can be formatted as Test Plans using a MS Word template. A sample Test Plan is shown in the Appendix.

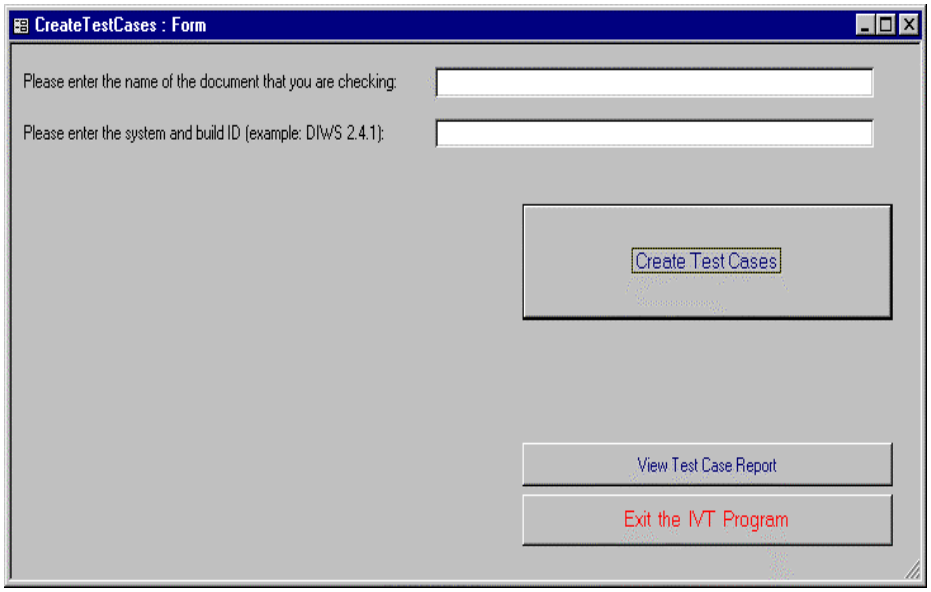


Figure 4.0-13. Generate Test Cases.

Generate Error Test Cases

As described in Chapter 3, the Generate Error Test Cases function allows the user to generate error test cases for the Type 1 interface tables stored in the MS Access tables. The input is the interface table information as stored in the MS Access database, the test obligation database generated during static analysis, and the Beizer [5] and Marick [26] heuristics for error cases. The processing performed by this function is that an error test case is generated for each test obligation in the test obligation database. Next, the Beizer, Marick heuristics are used to generate error cases. Chapter 3 describes the error categories that are applied. This function is automatically performed after Generate Covering Test Cases. The user will be shown the number of test cases that have already been generated (under Generate Covering Test Cases function), giving the user the option to generate error cases or to return to the previous function. After the Generate Error Test Cases function is complete, all duplicate test cases are deleted. The output from this function are additions to the MS Access database tables containing test cases. These can be displayed in MS Access or can be formatted as Test Plans using a MS Word template. A sample Test Plan is shown in the Appendix.

Chapter 5

5.0 VALIDATION

One common criticism of academic research is that it is often not validated, or is validated using unrealistic, simplistic conditions or tests. With this in mind, we considered a number of validation methods. Formal mathematical validation was not appropriate because the specifications used (as input) are textual and informal. To validate the IVT method in real-world, industry applications, experimentation was selected as the validation method. Experimentation allows the environment and confounding factors to be controlled to an extent while carefully studying the effectiveness and efficiency of the IVT method. The research was validated using a multi-subject experiment. This Chapter will present the experimental design used to validate the IVT method, and an overview of the experimental findings. Validation results show that the IVT method, as implemented in the MICASA tool, found more specification defects than senior testers, generated test cases with higher syntactic coverage than senior testers, generated test cases that took less time to execute, generated test cases that took less time to identify a defect than senior testers, and found defects that went undetected by senior testers.

Experimental Design

The goal of the experiment was to examine how well the IVT method performs static analysis and generates test cases as compared to senior software testers. To accomplish this goal, the experiment was designed to use ten senior testers, each performing the same activities as the MICASA tool. The experiment was divided into 3 parts:

- Part I – performing analysis of the specifications
- Part II – generating test cases for the specifications
- Part III – executing the test cases

Volunteers were used for the experiment, and many dropped out or did not complete the experiment. The three experiment parts are discussed below. The overall experimental design is shown in Table 5.0-1.

Table 5.0-1. Planned Analysis-of-Variance Model for Overall Experiment.

SOURCES OF VARIANCE	SYMBOL	NUMBER OF LEVELS	DEGREES OF FREEDOM
Between-Subjects Sources			
Systems	E	3	2
Condition (Exp'l vs. Control)	C	2	1
Interaction between systems and conditions	E*C	6	1
Subjects nested within systems and conditions (Between-Subjects error)	S(EC)	n.a.	36

In examining Table 5.0-1, there are a number of terms that must be defined. A **factor**, **variable**, or **treatment** refers to one of the primary experimental manipulations, of which there were two:

1) assigning participants to *systems*; System A, B, and/or C, and
2) assigning participants to one of the two *conditions*, Experimental (MICASA) or Control (manual method).

Participants performed one or more of the three experiment Parts: one having to do with specification review; one relating to test case generation; and one relating to test case execution/performance.

Interaction refers to the possibility that performance for a particular factor may depend on another factor. For example, it may be the case that specification review with MICASA was better than using the manual approach, but the difference between these two conditions was greater for System A than for System B or System C. Were this the case, it could be said that "there was an interaction between the condition and system factors."

Significance refers to whether or not an observed difference in performance can be attributed to the manipulations of the experiment, or is something that could be expected to occur on a chance basis due to the random sampling effects or other effects of random variables. Most of the statistical analyses performed were concerned with the question of significance of a particular factor or interaction and computed an estimate of the probability that an observed difference occurred by chance. By convention, if the computed probability of observed effects was less than 0.05 of having occurred by chance, expressed as $p < 0.05$, the performance difference is said to be significant, meaning that it is to be interpreted as arising not from random events but from the experimental manipulation [30].

Symbol refers to the identifier for the source of variance (for example, condition is C). **Number of levels** indicates how many different items there were for that source of variance, for example there were two conditions (experimental and control). **Degrees of freedom** refers to pieces of information or observations [43].

The expected results of the validation activity were:

a) the all-edges coverage percentage of test cases generated (by automated input validation test generation) will be at least as high as for the cases generated manually

b) the time to generate test cases (with automated input validation test generation) will be less than the time required to generate them manually;

c) the number of specification faults identified (with automated input validation) will be greater than those found manually

d) the number of defects/test case (with automated input validation test generation) will be greater than those found by manually generated cases

To determine whether these results were met, the dependent variables shown in Table 5.0-2 were used for the experiment.

Table 5.0-2. Description of the Planned Experiment Dependent Variables.

No.	Dependent Variable	Description	Method (Part)
1	TNSPECDEF	Total number of specification defects detected	I
2	TNSYNSPECDEF	Total number of syntactic specification defects detected	I
3	TIME	Total time for the exercise	I, II, III
4	PSYNCOV	Percentage of syntax coverage	II
5	PEFFTCASE	Percentage of effective test cases	II
6	TNDEFDET	Total number of defects detected	III
7	ANDEFDET	Average number of defects detected per test case	III
8	ATIME	Average time to identify a defect	I, III

For all parts of the experiment, experienced testers (most had at least 7 years of software development/information technology experience and 3 years of testing experience) were used as the control condition. The researcher served as the experiment facilitator and did not serve as an experiment participant. The experiment participant's key qualifications are shown in Table 5.0-3.

Table 5.0-3. Experiment Participant Key Qualifications.

ID	ORGANIZATION	EXPERIMENT PART(S) PERFORMED	HIGHEST DEGREE AND FIELD	YEARS EXPERIENCE (SW ENG/TEST)	COMPLETION	TESTING EXPERIENCE
B1	SAIC	1	M.S.; Comp. Sci.	20/3	Completed	IV&V, SW development, QA
B2	Perot Systems	1,2	M.B.A; Mktg and Finance	12/3	Completed Part 1, Partially completed Part 2	System testing, SW development, thread and integration testing
B3	SAIC	1,2	M.S.; Eng. Mgmt.	12/10	Dropped out	IV&V, independent testing, SW development, QA
B4	SAIC	1,2	B.A.; English	15/7	Completed	System testing, integration testing, SW development
B5	SAIC	1,2	B.S.; CIS	7/5	Dropped out	IV&V, independent testing
B6	Independent Consultant	2	M.S.; Ops Research	40/6	Completed	IV&V, independent testing
B7	George Mason University	1,2	M.S.	4/1.5	Completed	Software design, programming, and testing
B8	SAIC	2,3	B.S.	3/3	Completed	IV&V, system testing, digital imagery systems, Navy intelligence systems
B9	SAIC	1	B.S.	12/12	Completed	IV&V, independent testing, SW development
B10	SAIC	1,2 with tool	B.S.; CIS	7/5	Completed	IV&V, independent testing, SW development
B11	SAIC	2,3	B.A.	9/3	Completed	Testing, digital imagery systems, Navy intelligence systems

Part I: Static Analysis of Specifications

The design for Part I of the experiment consisted of 4 testers manually analyzing interface specification tables with 1 tester using the MICASA prototype to automatically analyze the same specifications. Five different tables were analyzed: two specification tables (3.2.4.2.1.1.3-5, 3.2.4.2.2.1.3-1) for the Tomahawk Planning System (TPS) to Digital Imagery Workstation Suite (DIWS) interface (Tomahawk document) (System A); two specification tables (Attributes Part of Doc_Jecmics, Attributes Not Part of Doc_Jecmics) for the commercial version of the Joint Engineering Data Management Information and Control System (JEDMICS) (System B); and a specification table (3.2.4.2-1) for the Precision Targeting Workstation (PTW) from the 3900/113 document (U.S. Navy) (System C).

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

The TPS-DIWS specification was “overlapping” (that is, all 5 testers were asked to analyze this document) to allow for examination of possible “skill level” ambiguities of the various testers. By having all 5 testers analyze the same specification, a particularly weak or particularly strong tester could be distinguished. For example, if one tester found five times as many defects as the other testers in the TPS-DIWS document, she would be considered an outlier (very adept tester). One tester was not able to complete analysis of this document, however.

All defects were naturally occurring, not seeded. Testers B1 and B9 performed thorough reviews of all three subject system tables to give the researcher a feel for the “quality” of the tables. Analysis was not performed on the number or percentage of defects found by MICASA and other testers that were not found by Testers B1 and B9, but their reviews gave the researcher a good idea of how many defects existed in each table.

The Part I design is depicted below. The first table describes the specifications that were analyzed by the testers, and whether or not the analysis was manual (by hand) or automated. The table views Part I of the experiment based on Subject (Tester) and System. For example, Tester B1 analyzed the TPS-DIWS tables (of System A) by hand, but Tester B10 used the MICASA tool to examine them.

	TPS-DIWS [SYSTEM A]	JEDMICS [SYSTEM B]	PTW [SYSTEM C]
Tester B1	By hand (control condition)	By hand	N/A
Tester B2	By hand	By hand	N/A
Tester B4	By hand	N/A	By hand
Tester B7	By hand	N/A	N/A
Tester B9	N/A	N/A	By hand
Tester B10	MICASA (experimental condition)	MICASA	MICASA

The next table summarizes the same information, but looks at it from a Condition and System perspective.

Condition	*System* A (TPS-DIWS)	B (JEDMICS)	C (PTW)
Experimental (MICASA)	Tester B10	Tester B10	Tester B10
Control (By Hand)	Testers B1, B2, B4, B7	Testers B1, B2	Testers B4, B9

Part I: Static Analysis

The repeated measures for Part I of the experiment are: number of specification defects detected; number of syntactic specification defects detected; and time it took to analyze the specification. To minimize confounding effects, each tester was provided the following set of instructions for performing Part I of the experiment. All communication with the testers was identical. If one tester asked a question, the question and reply were forwarded to all the testers.

Instructions to Testers: "As you are developing test cases for the specification, please note any deficiencies or problems that you see. Examples of deficiencies include missing information and ambiguous information. For this specification table:

Table 5.2.3.4 Task Recognizer

Item	*Description*	*Size*	*Expected Value*
1	*Task ID*	*10*	*A/N/AN*
2	*Name*	*20*	*Blank*
3	*Task Priority*	*1*	
4	*Task Complete*	*1*	*Y or N*

problems or deficiencies might include:

Table 5.2.3.4 Problems

1. *For Task ID, in column "Expected Value", the "A/N/AN" may not be clear.*
2. *For Name, in column "Expected Value", does "Blank" mean blank fill?*
3. *For Task Priority, there is no value in column "Expected Value."*

For each specification table that you develop test cases for, please keep a list of the deficiencies/problems you find. As a minimum, please provide the information shown above (table number and name; a list of deficiencies found for that table). Please keep track of how long it takes you to develop the test cases."

Part II: Generate Test Cases

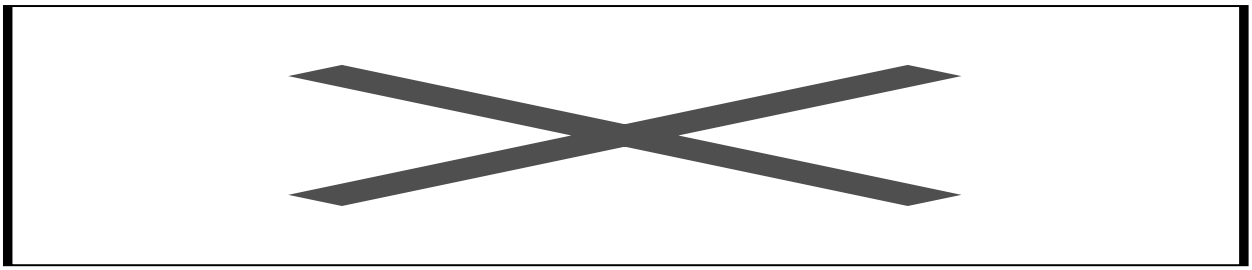
The design for Part II of the experiment consisted of 6 testers manually generating test cases for interface specification tables with 1 tester using the MICASA prototype to automatically generate cases. Five different tables were used: two specification tables (Table 1: 3.2.4.2.1.1.3-5, Table 2: 3.2.4.2.2.1.3-1) for the Tomahawk Planning System to Digital Imagery Workstation Suite interface (Tomahawk document); two specification tables (Attributes Part of Doc_Jecmics, Attributes Not Part of Doc_Jecmics) for the commercial version of the Joint Engineering Data Management Information and Control System (JEDMICS); and a specification table (3.2.4.2-1) for the Precision Targeting Workstation (PTW) from the 3900/113 document (U.S. Navy). The TPS-DIWS document was overlapping (that is, all 7 testers were asked to use this document) to allow for examination of possible skill level ambiguities of the various testers. By having all 7 testers use the same specification, a particularly weak or particularly strong tester could be distinguished. For example, if one tester generated five times fewer test cases than the other testers for the TPS-DIWS document, she would be considered an outlier (very weak tester). One tester was not able to complete test cases for this document, however.

The Part II design is depicted below. The first table describes the tables that were analyzed by the testers, the number of cases generated, and whether or not the test cases were generated manually (by hand) or automatically. The table views Part II of the experiment based on Subject (Tester) and System. For example, Tester B2 generated test cases for the two

TPS-DIWS tables (of System A) by hand, but Tester B10 used the MICASA tool. Note that none of the testers finished test cases for the JEDMICS specification, so only System A and C were analyzed.

	TPS-DIWS [SYSTEM A], # CASES TABLE 1, # CASES TABLE 2	PTW [SYSTEM C], # CASES
Tester B2	By hand, N/A, 3 cases	N/A
Tester B4	By hand, 3 cases, 4 cases	N/A
Tester B6	By hand, N/A, 59 cases	N/A
Tester B7	By hand, 25 cases, 11 cases	N/A
Tester B8	N/A	By hand, 4 cases
Tester B10	MICASA (experimental condition), 33 cases, 16 cases	MICASA, 53 cases
Tester B11	N/A	By hand, 3 cases

The next table summarizes the same information, but looks at it from a Condition and System perspective.



The repeated measures were: % syntax coverage; % effective test cases; and time to generate test cases. To minimize confounding effects, each tester was provided the following set of instructions for performing Part II of the experiment. All communication with the testers was identical. If one tester asked a question, the question and reply were forwarded to all the testers.

Instructions to Testers: *"System-level interfaces are described in interface requirement specification tables. Suppose that you are performing system-level testing. Develop test cases that you feel will adequately test each interface (each interface is described in a specification table). Following the instructions listed above for Part I, note any problems you find with the specification as you are developing test cases. Even if you find problems or deficiencies with items in the specification table, such as missing or ambiguous information, still generate a test case to address that item. For this specification table:*

Table 5.2.3.4 Task Recognizer

Item	*Description*	*Size*	*Expected Value*
1	*Task ID*	*10*	*A/N/AN*
2	*Name*	*20*	*Blank*
3	*Task Priority*	*1*	
4	*Task Complete*	*1*	*Y or N*

test cases might include:

Test Cases for Table 5.2.3.4 Task Recognizer

Case 1:	*Task ID:*	*"Task000001" (10)*
	Name:	*" " (20 blanks)*
	Task Priority:	*" " (1 blank)*
	Task Complete:	*"Y" (1)*
Case 2:	*Task ID:*	*"Task2" (5)*
	Name:	*" " (1)*
	Task Priority:	*"" (0)*
	Task Complete:	*"N" (1)*

For each specification table that you develop test cases for, please keep a list of the test cases. As a minimum, please provide the information shown above (table number and name; a list of test cases for that table). Please keep track of how long it takes you to develop the test cases."

Part III: Execute Test Cases

The design for Part III of the experiment consisted of 1 tester executing the manually generated test cases plus the MICASA generated test cases. The manually generated test cases were formatted to be identical to the MICASA generated cases. Only the researcher knew which cases came from MICASA and which cases were manually generated. The JEDMICS system was not available to the researcher for executing test cases because this commercial product is viewed as proprietary by the developer. The TPS-DIWS software was still under development during the course of the experiment. As a result, one table was used, a specification table for the Precision Targeting Workstation (PTW).
The Part III design is depicted below. The first table describes the number of cases executed for the PTW system (whether test cases were generated manually or by MICASA). The table views Part III of the experiment based on Subject (Tester) and System. Tester B11 executed all her own test cases (3) for the PTW table (of System C), as well as 43 of the MICASA-generated cases. Tester B8 executed all of his own test cases (4) for System C as well as 10 of the MICASA-generated cases.

	PTW [SYSTEM C], # CASES EXECUTED

B11	By hand test cases, 3 cases
B11	MICASA test cases, 43 cases
B8	By hand test cases, 4 cases
B8	MICASA test cases, 10 cases

The next table summarizes the same information, but looks at it from a Condition and System perspective.

Condition	*System*
	C (PTW)
Experimental (MICASA Cases)	Tester B11, B8
Control (Manually Generated Cases)	Tester B11, B8

Part III: Execute Test Cases

The repeated measures were number of defects detected per test case, and time to execute test cases. The testers were provided the following set of instructions for performing Part III of the experiment.

<u>Instructions to Testers:</u> "Run the test cases and note any failures encountered. Document each failure on a Software Trouble Report (STR). For example, suppose that when executing the following Table 5.2.3.4 test case:

Case 1: Task ID: "Task000001" (10)
Name: " " (20 blanks)
Task Priority: " " (1 blank)
Task Complete: "Y" (1)

a failure occurred. The system hung up and had to be rebooted. The STR form might be filled out as follows:

(TYPE OR PRINT)

SOFTWARE TROUBLE REPORT [X] ***SOFTWARE CHANGE PROPOSAL [ ]***	*1. DATE:* *23 January 1998*	*2. STR/SCP NUMBER* *JEH0001*

*3.*TITLE (BRIEFLY DESCRIBE IN 80 CHARACTERS OR LESS)*

Crash During Task Recognizer Test

4. ORIGINATOR* *J. Hayes*	*5. ORIGINATOR XREF*	*6.* PHONE*	*7.* SITE*

8. HOW DETECTED (CHECK ONE) *[X] TESTING [ ] OPERATIONS [ ] OTHER*	*9.* ORIG RECOMMENDED PRIORITY* *1 2 3 4 5*

10. FREQUENCY (CHECK ONE)
[X] ONE TIME [ ] SELDOM [ ] FREQUENTLY [ ] CONTINUALLY

11. ORIG RECOMMENDED CATEGORY *[X] SW [ ] DOC [ ] DESIGN [ ] RQMT*	12. *REFERENCE (STR/SCP OR DOCUMENT)* *Table 5.2.3.4*

13. **PRODUCT & RELEASE:**_____*
*COMPONENT ITEM AFFECTED:*_____
*[ ] S/W: FUNCTION:*_____
*[ ] DOC: TITLE/DOC #:*_____ *Rev:*_____

14. TROUBLE DESCRIPTION (360 CHARACTERS OR LESS)*

While running Case 1 for Table 5.2.3.4 Task Recognizer, a failure occurred. The software system locked up and had to be rebooted in order to continue.

[ ] Continued Next Page

15. PROPOSED SOLUTION:

Attachments: *[ ] Continued Next Page*

For each failure encountered, please complete a software trouble report. Please keep track of how long it takes you to run the test cases."

Discussion/Observations

For part I of the experiment, the researcher noted that the senior testers did not find a very high percentage of the defects present in the poorest quality specification tables. The researcher observed that when specification tables were of particularly poor quality, the participants seemed to make very little effort to identify defects. Instead they seemed to assert their effort on the tables that were of higher quality. This phenomenon also showed up in part II of the experiment. The researcher noted that participants did not even attempt to develop test cases for the poor quality specification tables. For future research, it may be desirable to formally categorize tables as being low, moderate, or high quality (perhaps by identifying all present defects and/or seeding additional defects) prior to asking participants to analyze them. This would permit formal analysis of any possible interaction between participant's performance and specification table quality.

For part II of the experiment, a dependent variable was added. *Percentage of effective test cases* was not originally part of the experimental design. During part II analysis, the researcher noticed that many test cases were duplicated (hand generated and MICASA-generated). Though the test cases were not identical to each other syntactically, the cases were semantically repetitive. For example, one test case modified an alphanumeric field from "TASK" to "TASP" while a second test case modified the same field to "PASK." To determine the percentage of effective test cases, the researcher built a matrix of every data element in the specification table. Each test case was analyzed to determine the data elements that had been modified and how. Each time a new combination of changes was used, a test case was counted as an effective test case. Test cases that syntactically modified the same fields were counted as duplicates.

Percentage of effective test cases was calculated as:

Percentage of Effective Test Cases = (Number of Effective Test Cases/Total Number of Test Cases) * 100

Coverage of test cases was measured as described in Section 3.3, with a minimum of four test cases expected for each specification table containing at least one loop. The four test cases expected were 0 times through the loop, one time through the loop, X times through the loop (where X is the length of the data element), and X+1 times through the loop. A set of test cases covering all four of these conditions would receive a 100% coverage measure (for 4 out of 4 possible conditions).

For all three parts of the experiment, there was interesting synergism between the defects found by the tool and those found by senior testers. For part I, roughly 5% or less of the defects found by the IVT method (MICASA) were also found by senior testers (and vice versa). These were obviously syntactic defects. The senior testers concentrated almost exclusively on semantic defects. Interestingly, the test cases generated by the senior testers were more syntactically oriented. The researcher noted that the IVT method test cases found 13 of the 21 defects found by tester B8 and one of the six defects found by tester B11. Also, the MICASA test cases found five defects that were not detected by the senior tester cases. The senior testers emphasized the semantic aspects of the system, not the syntactic aspects of the requirements specification.

Overview of Findings

As discussed above, the experiment performed is concerned with two primary factors: condition and system. The data collected was analyzed by applying the analysis-of-variance (ANOVA) model to the dependent variables described in Table 5.0-2. The factors were looked at individually, as well as in combination with each other. Of interest are the significant factors and interactions [30].

Table 5.0-4 presents the dependent variables (rows), the factors (columns), and whether or not the combinations have statistical significance (cells). For example, the probability of the differences in the total number of specification defects detected for condition being due to chance is .0042, or 4.2 in 1,000. In other words, this

difference is very statistically significant and is interpreted as being attributable to real differences among the conditions. On the other hand, the differences in total time for exercise for condition could be attributable to random events and are not significant. A value is not considered to be statistically significant if the probability of the result being due to chance is greater than 0.05; a dash (--) is entered in the cell whenever the probability is greater than 0.05.

Because many testers dropped out or did not complete all tasks assigned to them, there were more data values available for some specification tables and Systems than for others. For example, in part I of the experiment, only two testers analyzed the JEDMICS specification table, but four testers analyzed the two TPS-DIWS specifications. For Condition and System, all data values were analyzed using ANOVA Single Factor analysis (of MS Excel). For SYS*COND, an attempt was made to use all data values with ANOVA Two-Factor Without Replication. With unequal numbers of values per System, MS Excel would not process this data. Therefore, a reduced set of values were analyzed using ANOVA Two-Factor with Replication. In order to reduce the data values (so that there were the same number for each System), the worst values for senior testers were removed from the set (to improve the results of the senior testers). The best values for the senior testers were kept, so for TPS-DIWS there were four data values thrown out (where senior testers found 0,0, 1, and 1 defects during part I, these were the lowest number of defects detected by the senior testers). Similarly, the worst times to complete tasks were thrown out for senior testers (used only the fastest times for the senior testers). This method of data reduction clearly skews the data in favor of the control condition and not in favor of the experimental condition.

Appendix B presents all the experimental means (averages) in tabular format as well as data used to obtain the experimental means. Appendix C presents the defects found during part I of the experiment by MICASA and senior testers for one JEDMICS (System B) specification table. Appendix D presents the defects detected by executing MICASA-generated and senior tester-generated test cases on PTW (System C). The remainder of this section will be dedicated to explaining the findings summarized in the tables.

Table 5.0-4. Significance of Findings for the Dependent Variables

DEPENDENT VARIABLES	FACTORS: COND	SYS	SYS* COND
Total number of specification defects detected	.0042	--	--
Total number of syntactic specification defects detected	.0007	--	--
Total time for the exercise	Given below	Given below	Given below
Part I	--	.015	--
Part II	--	.0096	1.44E-05
Part III	6.22E-15	N/A	N/A
Percentage of syntax coverage	1.12E-10	--	--
Percentage of effective test cases	--	.039	--
Total number of defects detected	--	N/A	N/A
Average number of defects detected per test case	.0305	N/A	N/A
Average time for all defects identified	Given below	Given below	Given below
Test Case Execution Time Only	2.02E-10	N/A	N/A
Test Case Development and Execution Time	4.15E-08	N/A	N/A

-- Means no statistical significance

Condition Effects

A concern for this experiment was the effect of condition. How did the MICASA tool perform as compared to the participants without a tool? Figures 5.0-1 through 5.0-7 present a graphical depiction of representative experimental results for condition, discussed below.

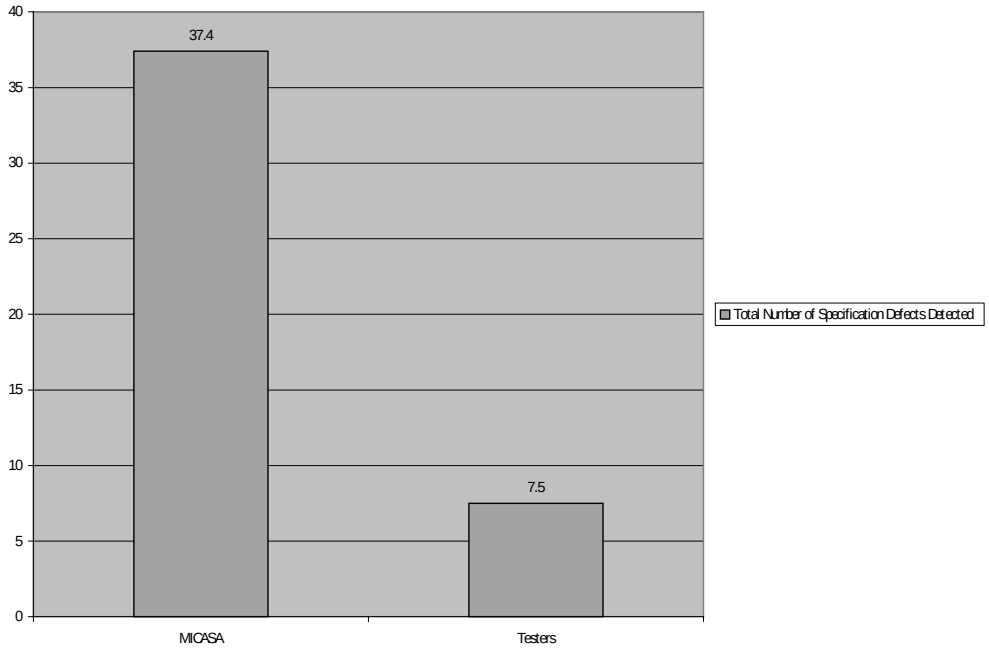


Figure 5.0-1. Total Number of Specification Defects Detected for Condition.

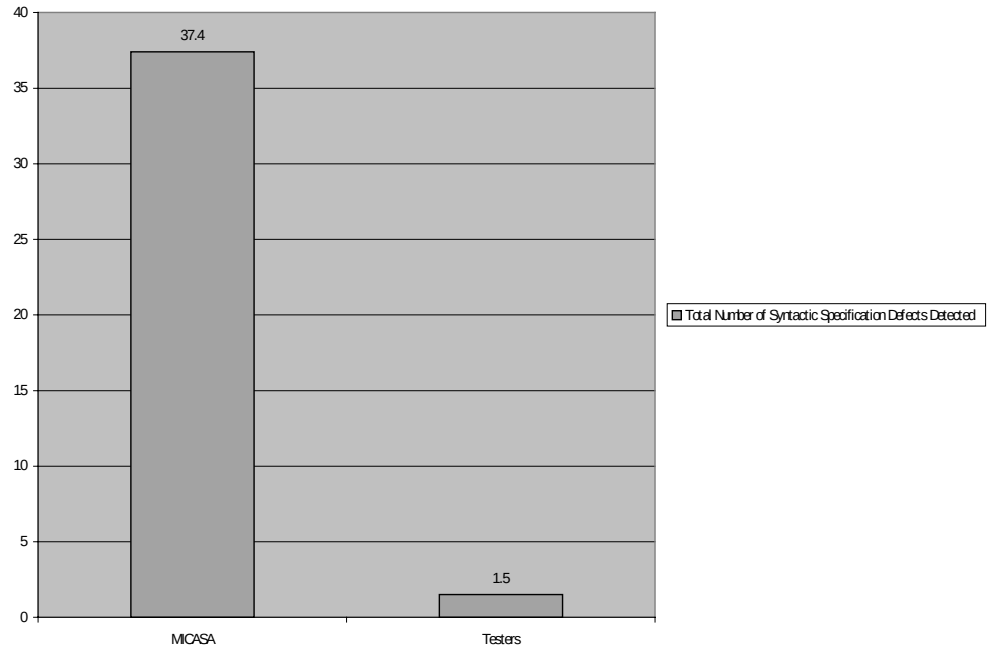


Figure 5.0-2. Total Number of Syntactic Defects Detected for Condition.

Specification Analysis (Part I) Findings

IVT Method (MICASA tool) Found More Specification Defects than Control Condition (Senior Testers). It was hypothesized that the experimental condition (MICASA tool) would find more defects than the control condition (manual) and that it would take less time to do so. The MICASA tool, for all systems, found significantly more total defects than the senior testers with no tool (37.4 as compared to 7.5). The MICASA tool found significantly more syntactic defects than the senior testers (37.4 as compared to 1.5).

Test Case Generation (Part II) Findings

IVT Method (MICASA tool) Achieved Higher Test Case Coverage than Control Condition (Senior Testers). It was hypothesized that the coverage percentage of the test cases generated by the experimental condition (MICASA tool) would be at least as high as those generated by the control condition (manual) and that it would take less time to do so. The MICASA tool, for all systems, achieved a higher coverage percentage than the senior testers with no tool (100% as compared to 31.25%).

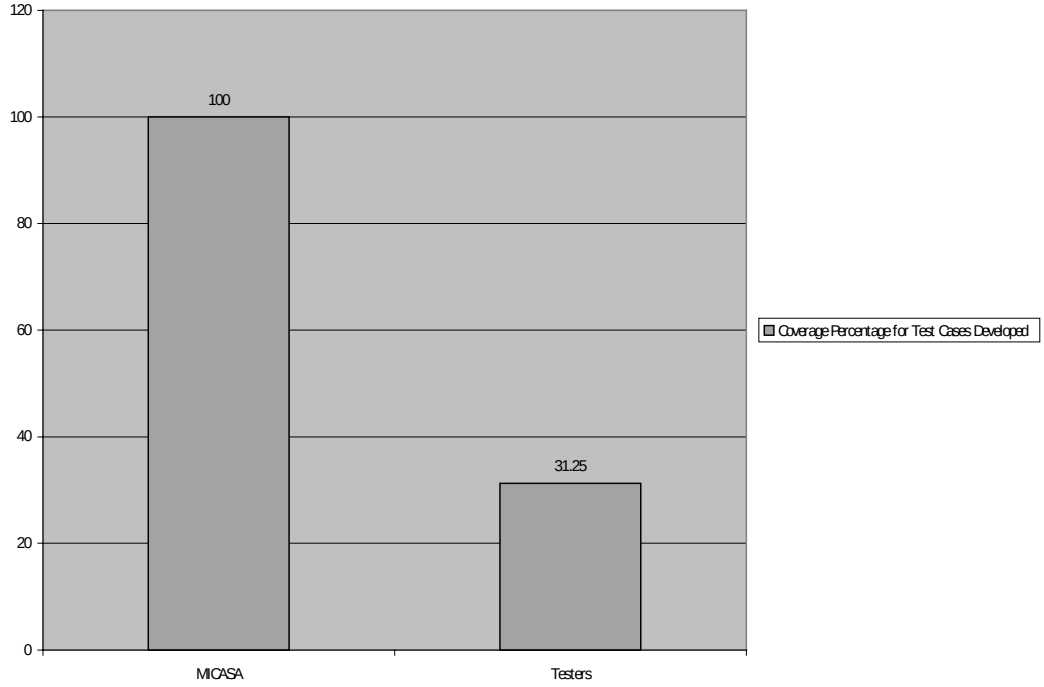


Figure 5.0-3. Coverage Percentage of Test Cases Developed for Condition.

Test Case Execution (Part III) Findings

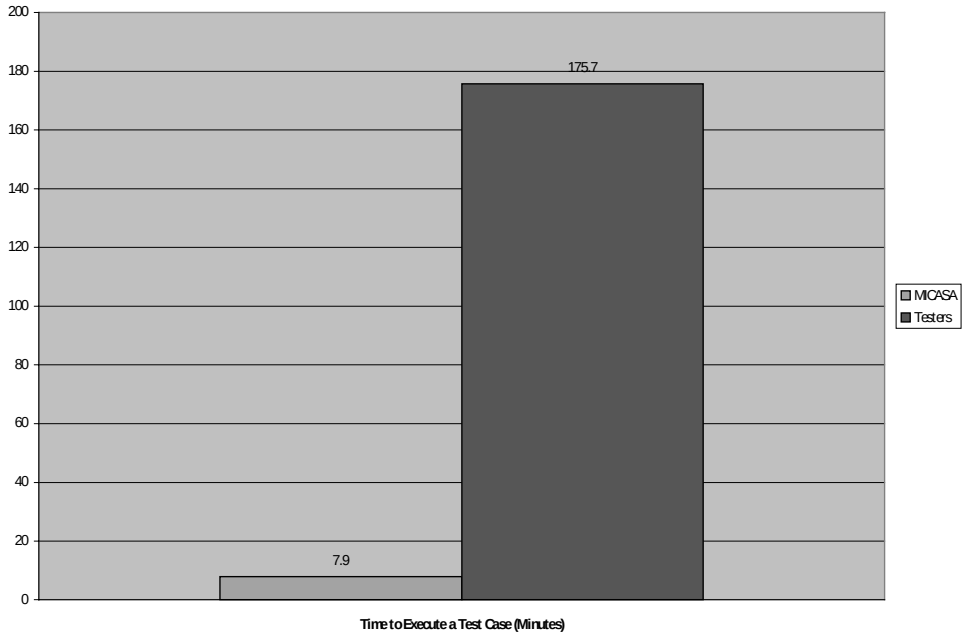


Figure 5.0-4. Average Time to Execute Test Cases for Condition.

IVT Method (MICASA tool) Test Cases Faster to Execute Than Those Developed by Control Condition (Senior Testers). It was hypothesized that the average time required to execute an IVT test case would be less than for those generated by the control condition (manual). The MICASA test cases were executed in an average of 7.9 minutes as compared to 175.7 minutes for manually generated test cases.

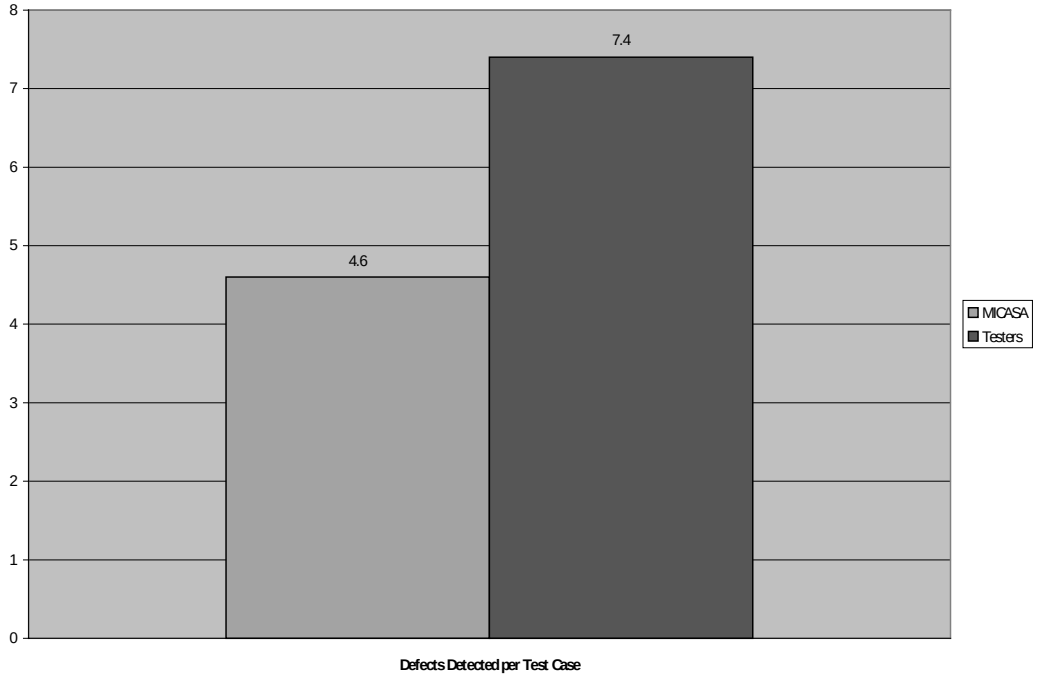


Figure 5.0-5. Defects Detected Per Test Case for Condition.

Control Condition (Senior Testers) Found More Defects Per Test Case Than IVT Method (IVT Tool). It was hypothesized that the test cases generated using the IVT method would find a greater number of defects than when using manually generated test cases. That was not the case as 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. It should be noted that the senior 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 it was not the case that senior testers found more important, higher priority defects than the IVT method.

IVT Method (MICASA tool) Test Cases Require Less Average Execution Time to Identify a Defect Than Those Developed by Control Condition (Senior Testers). Considering only execution time, the IVT method cases took only 2.17 minutes to find a defect, as opposed to 30.9 minutes for manually generated test cases.

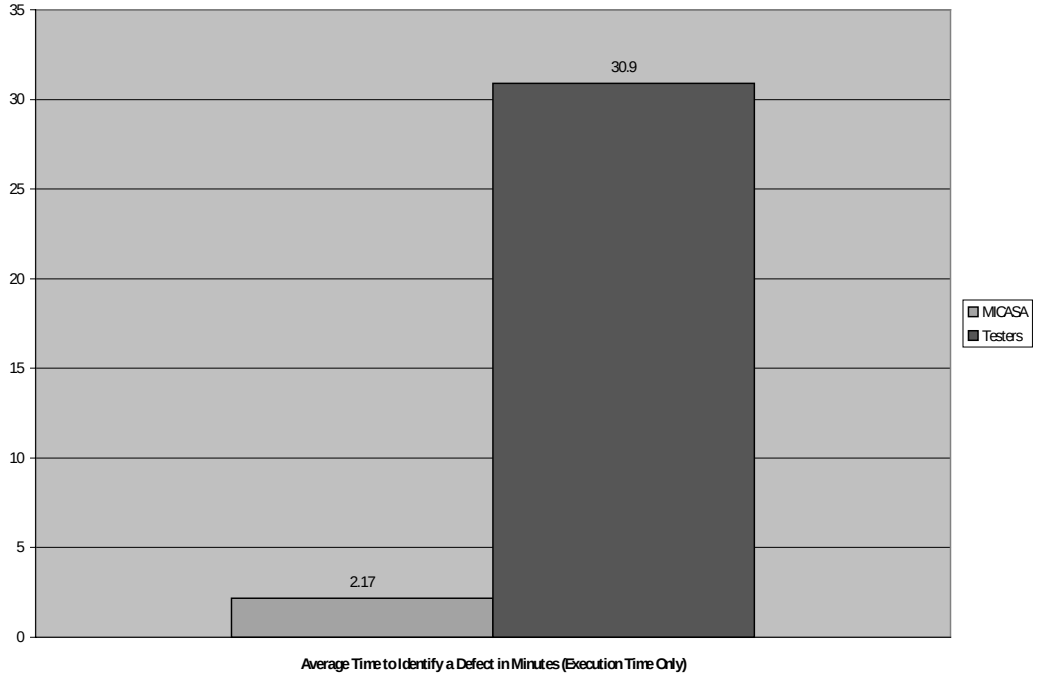


Figure 5.0-6. Average Time to Identify a Defect in Minutes (Execution Time Only).

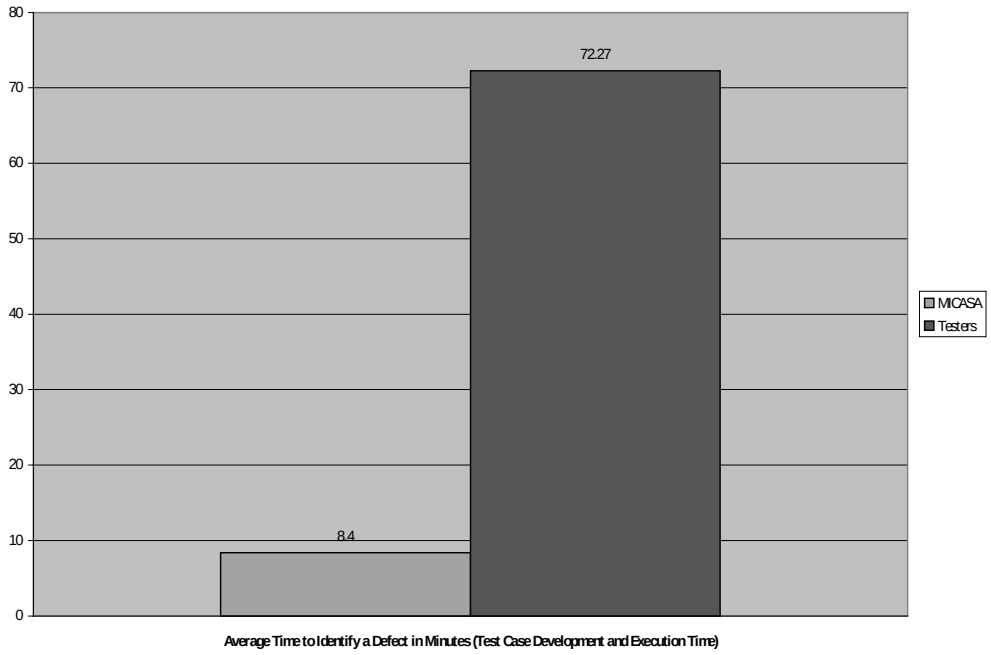


Figure 5.0-7. Average Time to Identify a Defect (Test Case Development and Execution Time).

IVT Method (MICASA tool) Test Cases Require Less Average Total Time to Identify a Defect Than Those Developed by Control Condition (Senior Testers). Considering test case development and execution time, the IVT method cases took only 8.4 minutes to find a defect, as opposed to 72.2 minutes for manually generated test cases.

System Effects

The specification tables for three different systems (TPS-DIWS, JEDM, and PTW) were examined, to ensure "generalness" of the IVT method and to ensure that experimental results were not biased by only using one system's specifications. How did the systems effect the experimental results? Figures 5.0-8 through 5.0-10 present a graphical depiction of representative experimental results for system, discussed below.

Specification Analysis (Part I) Findings

System C More Difficult to Analyze than Systems A and B. System C (PTW) took significantly more time for senior testers and MICASA to analyze than the other two systems. It took an average of 59.5 minutes to analyze System A, while it only took an average of 31.8 minutes to analyze System A and 12.5 minutes to analyze System B.

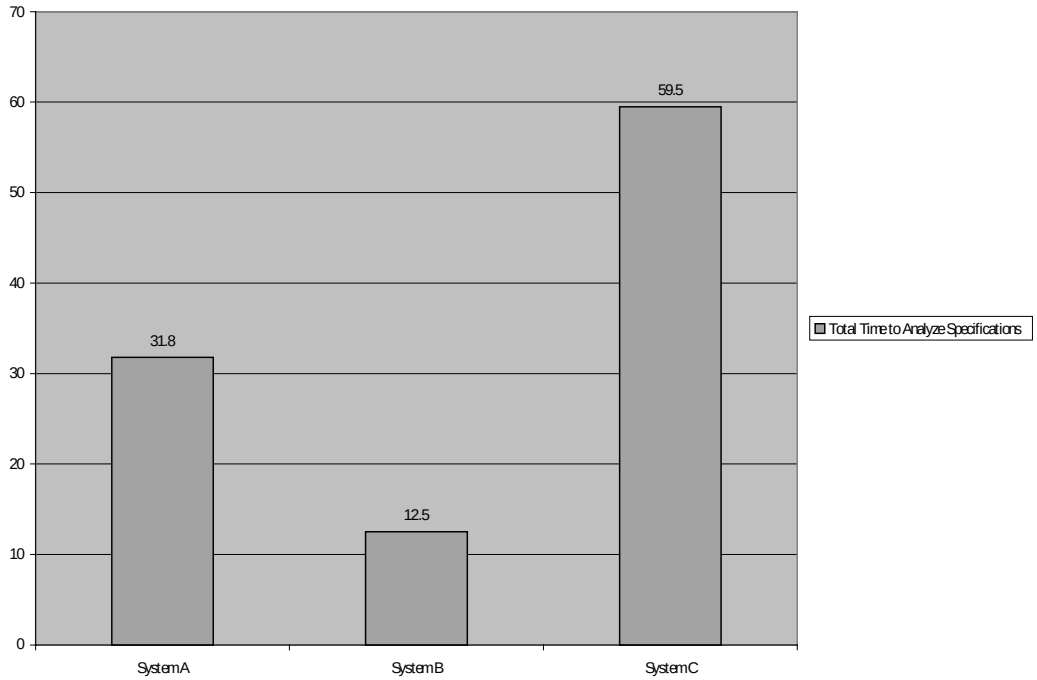


Figure 5.0-8. Total Time to Analyze Specifications for System.

Test Case Generation (Part II) Findings

Higher Percentage of Effective Test Cases for System C than for System A. The test cases generated for System C (PTW) (both by senior testers and MICASA) were more effective than those developed for System A. The System C test cases were 92.4% effective, while the System A test cases on average were 62.9% effective.

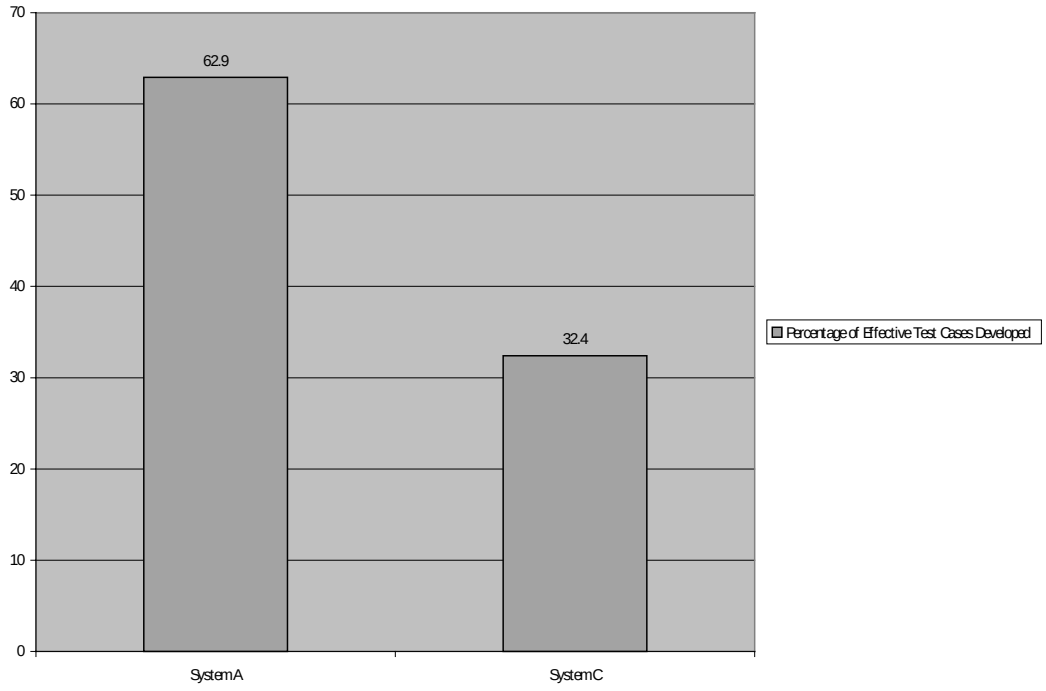


Figure 5.0-9. Percentage Effective Test Cases Developed by System.

More Minutes Required to Develop a Test Case for System C than for System A. The test cases generated for System C (PTW) (both by senior testers and MICASA) took more than 12 times as long to develop as those for System A. The average time to develop a test case for System A was 5 minutes, while it took, on average, 67.7 minutes to develop a test case for System C.

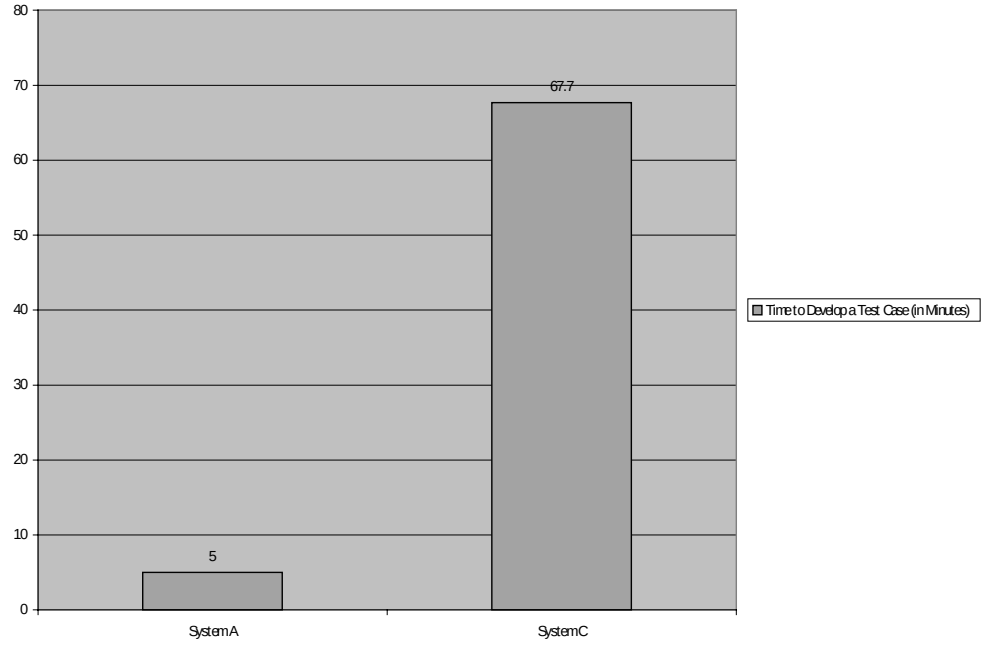


Figure 5.0-10. Average Time to Develop a Test Case by System.

Test Case Execution (Part III) Findings

As only System C was used for Part III of the experiment, there are no System results.

System and Condition Effects

Next we combined the factors of system (systems A, B, and C) and condition (experimental and control). How did the MICASA tool perform as compared to the participants without a tool, from both a system and condition perspective? Figure 5.0-11 presents a graphical depiction of representative experimental results for system, discussed below.

Specification Analysis (Part I) Findings

None of the part I findings for System and Condition were statistically significant.

Test Case Generation (Part II) Findings

Time to Develop a Test Case Higher for System C and for Senior Testers. The System*Condition interaction for Time to Develop a Test Case was statistically significant. System C required 135 minutes for senior testers to develop test cases and 0.47 minutes for MICASA to generate a test case. System A required 7.55 minutes for a senior tester to develop a test case and 0.06 minutes for MICASA to develop a test case.

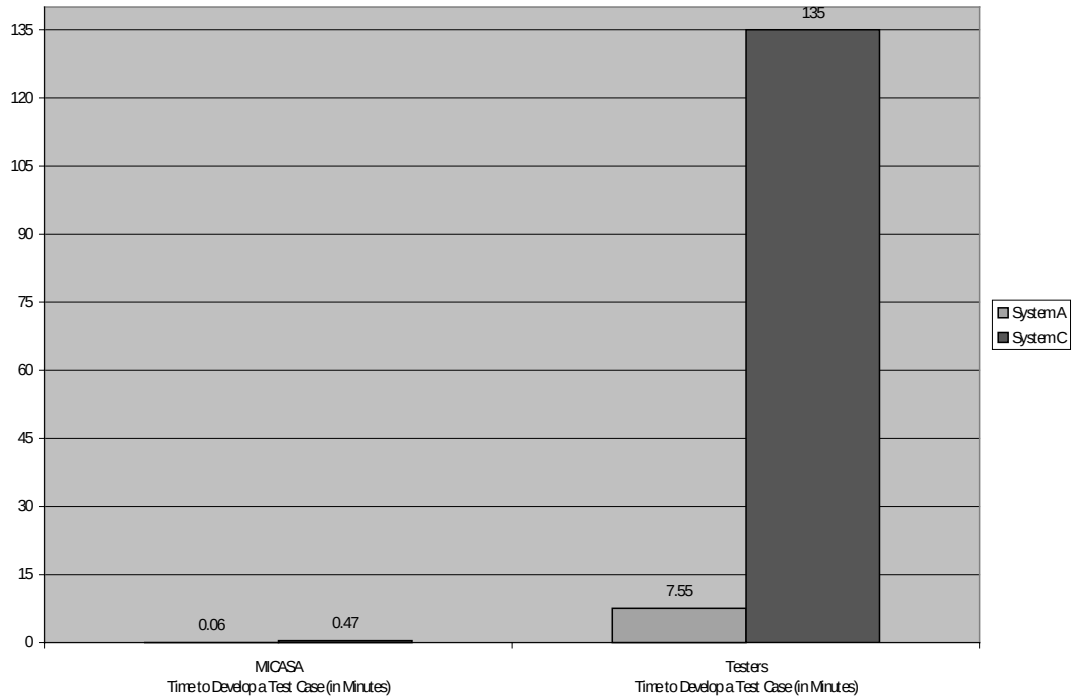


Figure 5.0-11. Average Time to Develop a Test Case by System and Condition.

Test Case Execution (Part III) Findings

As only System C was used for Part III of the experiment, there are no System and Condition results.

Chapter 6

6.0 CONCLUSIONS AND FUTURE RESEARCH

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

The results indicate that static analysis of requirements specifications can detect syntactic defects, and do it early in the lifecycle. More importantly, these syntactic defects can be used as the basis for generating test cases that will identify defects once the software application has been developed, much later in the lifecycle. The requirements specification defects identified by this method were not found by senior testers. Half of the software defects found by this method were not found by senior testers. And this method took on average 8.4 minutes to identify a defect as compared to 72.27 minutes for a senior tester. So the method is efficient enough to be used in addition to testing activities already in place.

There were several inefficiencies in the implementation and algorithms used to detect syntactic defects in the requirements specifications (particularly between overloaded token and ambiguous grammar). This resulted in a number of extremely similar entries in the Test Obligation Database. As a result, the test cases generated for these entries were not effective. The researcher is confident that changes to this portion of MICASA will result in a smaller number of test cases being generated with a higher percentage of the test cases being effective.

What do the research results mean to us? To testers, it means that they should not overlook syntactic-oriented test cases, and that they should consider introducing syntactic static analysis of specifications into their early life cycle activities. To developers, it means that emphasis must be put on specifying and designing robust interfaces. Developers may also consider introducing syntactic deskchecks of their interface specifications into

their software development process. To project managers, it means that interface specifications are a very important target of verification and validation activities. Project managers must allow testers to begin their tasks early in the life cycle. Managers should also require developers to provide as much detail as possible in the interface specifications, facilitating automated analysis as much as possible. Similarly, customers should require interface specifications to include as much information as possible, such as expected data values and whether or not a field is required or optional.

One area for future research is that of sequencing of commands. The IVT method views commands in isolation, not as a sequence. This is particularly important for highly automated software systems (like the Apple Macintosh operating system) that rely heavily on user selections (mouse clicks). 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 [40].

The handling of masks (such as DD-MM-YYYY and XXX.XX) is a possible area for future research. Date masks are an important area due to the emphasis on ensuring that software systems are Year 2000 compliant. MICASA currently treats date masks and real or float masks as numeric data types. In solving this challenge, user interaction may be required to ensure that masks are correctly parsed.

Another area for future research is that of table elements that are automatically generated by the subject software as opposed to being entered by the user. The IVT 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.

The IVT method may adapt well to the analysis and testing of graphical user interfaces (GUIs). The GUI specifications could be analyzed and then used as the basis for test case generation, with mouse clicks and menu picks replacing keyboard keystrokes.

Careful analysis of each of the static anomalies (overloaded token, ambiguous grammar, etc.) and the effectiveness of the test cases generated based on that test obligation is another area for future research. It may be the case that some test obligation types generate test cases that find many defects, while others may not be as effective. Similarly, the semantic-oriented static defects detected by senior testers could be examined to determine if additional semantic-type checks can be added to the IVT method.

Another possible future research area involves data element dependencies. Some specification table data elements depend on the values of other elements (this relationship is generally not stated in the specification table). To help solve this, a dependency field should be added to requirements specification tables. This field (semantic rules) could then be used to drive test case generation.

Appendix A

Table 3.2.4.1-10. ETF Weaponeering Dataset

Description	Size	Values
TABLE_ID	12	varchar
FORMAT_VERSION_ID	4	int
DAMAGE_CRITERION	15	varchar
DAMAGE_CRITERION_SEQUENCE_NO	2	varchar
SECURITY_CATEGORY (Weaponeering Dataset)	2	varchar
CLASSIFICATION_LEVEL (Weaponeering Dataset))	1	char
NUMBER_OF_SEC_HANDLING_INSTR	1	tinyint
SEC_CONTROL_HANDLING_INSTR (Weaponeering Dataset)	40	null
NUMBER_OF_LINES (Aimpoint Strike Objective)	3	varchar
AIMPOINT_OBJECTIVE	128	(U)
WEAPON_TO_TARGET_FLAG	1	null
PREDICTED_PROB_OF_DAMAGE (SSPD or SSFD)	3	null
NUMBER_OF_LINES (Expected Results)	3	varchar
EXPECTED_RESULTS	128	(U)
NUMBER_OF_LINES (Related Aimpoints)	3	varchar
RELATED_AIMPOINTS	132	null
NUMBER_OF_LINES (BDA Visible Damage)	3	varchar
BDA_VISIBLE_DAMAGE	128	(U)
NUMBER_OF_LINES (BDA Sensor Requirements)	3	varchar
BDA_SENSOR_REQUIREMENTS	128	(U)
NUMBER_OF_LINES (BDA EEI)	3	varchar
BDA_ESSENTIAL_ELEMENTS_INFO	128	(U)
ERROR_TYPE	1	null
CEP_PLANE	1	char
NOMINAL_CEP	4	CEP_MAXIMUM
NOMINAL_REP	4	null
NOMINAL_DEP	4	null
NOMINAL_IMPACT_ANGLE	3	null
EFFECTIVENESS_INDEX_TYPE	1	null
MAE_VALUE_OR_VAN	4	null
EFFECTIVE_MISS_DISTANCE (EMD)	2	0

PERCENT_BURIAL	1	0
PROB_OF_DAMAGE_GIVEN_A_HIT	3	1.00
ELEMENT_LENGTH_OR_RADIUS	4	OBJECT_LENGTH_OR_RADIUS
ELEMENT_WIDTH	4	OBJECT_WIDTH
ELEMENT_HEIGHT	2	OBJECT_HEIGHT
ELEMENT_AZIMUTH	2	OBJECT_AZIMUTH
ELEMENT_AREA_LENGTH_RADIUS	4	OBJECT_AREA_LENGTH_ OR_RADIUS
ELEMENT_AREA_WIDTH	4	OBJET_AREA_WIDTH
WEAPON_RELIABILITY	3	0.99
RANGE_OFFSET_FROM_DMPI	4	0 1
DEFLECTION_OFFSET_FROM_DMPI	4	0
PATTERN_TYPE	1	0 3
PROB_ DAMAGE_OR_AVAIL_PASSES	3	null
NUMBER_OF_PASSES_REQUIRED	3	null
EXPECTED_PD_FROM_NUM_PASSES	3	null
WEAPON_TYPE	1	6
MISSILE_VELOCITY	2	430
NUMBER_OF_GS_PULLED	2	0.0
RELEASE_ALT_OF_FIRST_WEAPON	4	null
DIVE_ANGLE	2	null
NUMBER_OF_RELEASE_PULSES	2	null
WEAPONS_RELEASED_PER_PULSE	2	2
INTERVALOMETER_SETTING	2	null
DISTANCE_BTWN_STATIONS	2	smallint
MER_TER_RELEASE	1	N
TARGET_ALTITUDE	4	0
WEAPON_OR_DISPENSER_TERMINAL_VELOCITY	4	250
SECOND_LEG_TERMINAL_VELOCITY	2	135
EJECTION_VELOCITY	3	0
FUNCTIONING_TIME	4	0.50
SLANT_RANGE	4	null
JMEM_IMPACT_ANGLE	3	null

STICK_LENGTH	2	null
STICK_WIDTH	2	null
BALLISTIC_ERROR	1	tinyint
SUBMUNITION_RELIABILITY	3	numeric
NUMBER_OF_SUBMUNIT_DISPENSER	2	7
PATTERN_LENGTH	2	smallint
PATTERN_WIDTH	2	smallint
PROBABILITY_OF_NEAR_MISS	3	1.00
PROBABILITY_OF_DIRECT_HIT	3	0.00
RANGE_BIAS	3	0.00
PROBABILITY_OF_RELEASE	3	1.00
INTERPOLATION_FLAG	1	N
INTERPOLATION_POINT	3	null
INTERP_LOWER_BOUND	3	null
MAE_AT_LOWER_BOUND	4	null
INTERP_UPPER_BOUND	3	null
MAE_AT_UPPER_BOUND	4	null
NUMBER_OF_LINES (Weaponeering Notes)	3	varchar
WEAPONEERING_NOTES_1	255	(U)
WEAPONEERING_NOTES_2	145	null

Input Validation for Table_3_2_4_1_10___ETF_Weaponeering_Dataset

Test ID: Tx.xxx

Criticality:

System & Build: TEST 6.9.98

DR Number:

Requirements/Documentation

EFT

Purpose

The purpose of this test is to verify correct operation of the input validation capabilities.

Procedure

The test cases for table Table_3_2_4_1_10___ETF_Weaponeering_Dataset are defined as follows:

Case 1064 : Invalid

Case 1065 : Invalid

Case 1066 : Valid

Case 1067 : Invalid

Case 1064 : Invalid

The purpose of this test case is to verify that the input validation capability recognizes incorrect input.

Test Case~ (U)
(U) (U)
(U) (U)
CEP_MA0 0 1.00OBJECOBJECOBJEOBJOBJECOBJET0.990 1 0 064300.02 Y0 250 1350 0.50 7 1.000.000.00 1.00N(U)

Test Case Breakdown:

AIMPOINT_OBJECTIVE ~(U)

EXPECTED_RESULTS ~(U)

BDA_VISIBLE_DAMAGE ~(U)

BDA_SENSOR_REQUIREMENTS ~(U)

BDA_ESSENTIAL_ELEMENTS_INFO ~(U)
NOMINAL_CEP ~CEP_MA
EFFECTIVE_MISS_DISTANCE (EMD) ~0
PERCENT_BURIAL ~0
PROB_OF_DAMAGE_GIVEN_A_HIT ~1.00
ELEMENT_LENGTH_OR_RADIUS ~OBJEC
ELEMENT_WIDTH ~OBJEC
ELEMENT_HEIGHT ~OBJE
ELEMENT_AZIMUTH ~OBJ
ELEMENT_AREA_LENGTH_RADIUS ~OBJEC
ELEMENT_AREA_WIDTH ~OBJET
WEAPON_RELIABILITY ~0.99
RANGE_OFFSET_FROM_DMPI ~0 1
DEFLECTION_OFFSET_FROM_DMPI ~0
PATTERN_TYPE ~0
WEAPON_TYPE ~6
MISSILE_VELOCITY ~430
NUMBER_OF_GS_PULLED ~0.0
WEAPONS_RELEASED_PER_PULSE ~2
MER_TER_RELEASE ~Y
TARGET_ALTITUDE ~0
WEAPON_OR_DISPENSER_TERMINAL_VELOCITY ~250
SECOND_LEG_TERMINAL_VELOCITY ~135
EJECTION_VELOCITY ~0
FUNCTIONING_TIME ~0.50
NUMBER_OF_SUBMUNIT_DISPENSER ~7
PROBABILITY_OF_NEAR_MISS ~1.00
PROBABILITY_OF_DIRECT_HIT ~0.00
RANGE_BIAS ~0.00
PROBABILITY_OF_RELEASE ~1.00
INTERPOLATION_FLAG ~N
WEAPONEERING_NOTES_1 ~(U)

Case 1065 : Invalid

The purpose of this test case is to verify that the input validation capability recognizes incorrect input.

Test Case~ Z23VKX2KQ(U)
R4T(U)
V47(U)
N(U) I(U)
5VCEP_MA228660 0 1.00OBJECOBJECOBJEOBJOBJECOBJET0.990 1 0
096464300.02492 56Y0 250 1350 0.50 2850297 171.000.000.00 1.00N21841T(U)
A

Test Case Breakdown:

TABLE_ID ~Z

FORMAT_VERSION_ID ~2

DAMAGE_CRITERION ~3

DAMAGE_CRITERION_SEQUENCE_NO ~V

SECURITY_CATEGORY (Weaponeering Dataset) ~K

CLASSIFICATION_LEVEL (Weaponeering Dataset)) ~X

NUMBER_OF_SEC_HANDLING_INSTR ~2

SEC_CONTROL_HANDLING_INSTR (Weaponeering Dataset) ~K

NUMBER_OF_LINES (Aimpoint Strike Objective) ~Q

AIMPOINT_OBJECTIVE ~(U)

WEAPON_TO_TARGET_FLAG ~R

PREDICTED_PROB_OF_DAMAGE (SSPD or SSFD) ~4

NUMBER_OF_LINES (Expected Results) ~T

EXPECTED_RESULTS ~(U)

NUMBER_OF_LINES (Related Aimpoints) ~V

RELATED_AIMPOINTS ~4

NUMBER_OF_LINES (BDA Visible Damage) ~7

BDA_VISIBLE_DAMAGE ~(U)

NUMBER_OF_LINES (BDA Sensor Requirements) ~N

BDA_SENSOR_REQUIREMENTS ~(U)

NUMBER_OF_LINES (BDA EEI) ~I

BDA_ESSENTIAL_ELEMENTS_INFO ~(U)

ERROR_TYPE ~5

CEP_PLANE ~V

NOMINAL_CEP ~CEP_MA

NOMINAL_REP ~2
NOMINAL_DEP ~2
NOMINAL_IMPACT_ANGLE ~8
EFFECTIVENESS_INDEX_TYPE ~6
MAE_VALUE_OR_VAN ~6
EFFECTIVE_MISS_DISTANCE (EMD) ~0
PERCENT_BURIAL ~0
PROB_OF_DAMAGE_GIVEN_A_HIT ~1.00
ELEMENT_LENGTH_OR_RADIUS ~OBJEC
ELEMENT_WIDTH ~OBJEC
ELEMENT_HEIGHT ~OBJE
ELEMENT_AZIMUTH ~OBJ
ELEMENT_AREA_LENGTH_RADIUS ~OBJEC
ELEMENT_AREA_WIDTH ~OBJET
WEAPON_RELIABILITY ~0.99
RANGE_OFFSET_FROM_DMPI ~0 1
DEFLECTION_OFFSET_FROM_DMPI ~0
PATTERN_TYPE ~0
PROB_ DAMAGE_OR_AVAIL_PASSES ~9
NUMBER_OF_PASSES_REQUIRED ~6
EXPECTED_PD_FROM_NUM_PASSES ~4
WEAPON_TYPE ~6
MISSILE_VELOCITY ~430
NUMBER_OF_GS_PULLED ~0.0
RELEASE_ALT_OF_FIRST_WEAPON ~2
DIVE_ANGLE ~4
NUMBER_OF_RELEASE_PULSES ~9
WEAPONS_RELEASED_PER_PULSE ~2
INTERVALOMETER_SETTING ~5
DISTANCE_BTWN_STATIONS ~6
MER_TER_RELEASE ~Y
TARGET_ALTITUDE ~0
WEAPON_OR_DISPENSER_TERMINAL_VELOCITY ~250
SECOND_LEG_TERMINAL_VELOCITY ~135

EJECTION_VELOCITY ~0

FUNCTIONING_TIME ~0.50

SLANT_RANGE ~2

JMEM_IMPACT_ANGLE ~8

STICK_LENGTH ~5

STICK_WIDTH ~0

BALLISTIC_ERROR ~2

SUBMUNITION_RELIABILITY ~9

NUMBER_OF_SUBMUNIT_DISPENSER ~7

PATTERN_LENGTH ~1

PATTERN_WIDTH ~7

PROBABILITY_OF_NEAR_MISS ~1.00

PROBABILITY_OF_DIRECT_HIT ~0.00

RANGE_BIAS ~0.00

PROBABILITY_OF_RELEASE ~1.00

INTERPOLATION_FLAG ~N

INTERPOLATION_POINT ~2

INTERP_LOWER_BOUND ~1

MAE_AT_LOWER_BOUND ~8

INTERP_UPPER_BOUND ~4

MAE_AT_UPPER_BOUND ~1

NUMBER_OF_LINES (Weaponeering Notes) ~T

WEAPONEERING_NOTES_1 ~(U)

WEAPONEERING_NOTES_2 ~A

Case 1066 : Valid

The purpose of this test case is to verify that the input validation capability recognizes correct input.

Test Case~
ZTU237073ZBO52N85C8NS1BVQKWX2K33V96IZ9ITD9YAUDD2KBKNK89OKF
FXOOZLWHGVCQ6J(U)
R4614T3T(U)
VNO4V7SL(U)
NO4(U)
IMT(U)
5VCEP_MA288530216932815666803820 0

1.00OBJECOBJECOBJEOBJOBJECOBJET0.990 1 0
091561677404564300.02440449212 5320635Y0 250 1350 0.50
2643281465713079229143 7 180772941.000.000.00
1.00N28019889266940812635 9T81(U)
AX8ENR6G3XIB6RYCA8H6QGK14JM54OVG7JALVNX0ZGCIH9GX6JJ11OFUEJ4
YQUIAJ8B4CZJ6BVR5Q33CUS2P9FYY0R94GXJ50ODWDYLDWTU47HFZ3L42N
VNSRV1LJ5JFFIGS8OKYIYLB8JY9N

Test Case Breakdown:

TABLE_ID ~ZTU

FORMAT_VERSION_ID ~23707

DAMAGE_CRITERION ~3ZBO52N85C8NS1B

DAMAGE_CRITERION_SEQUENCE_NO ~VQ

SECURITY_CATEGORY (Weaponeering Dataset) ~KW

CLASSIFICATION_LEVEL (Weaponeering Dataset)) ~X

NUMBER_OF_SEC_HANDLING_INSTR ~2

SEC_CONTROL_HANDLING_INSTR (Weaponeering Dataset)
~K33V96IZ9ITD9YAUDD2KBKNK89OKFFXOOZLWHGVC

NUMBER_OF_LINES (Aimpoint Strike Objective) ~Q6J

AIMPOINT_OBJECTIVE ~(U)

WEAPON_TO_TARGET_FLAG ~R

PREDICTED_PROB_OF_DAMAGE (SSPD or SSFD) ~4614

NUMBER_OF_LINES (Expected Results) ~T3T

EXPECTED_RESULTS ~(U)

NUMBER_OF_LINES (Related Aimpoints) ~VNO

RELATED_AIMPOINTS ~4V

NUMBER_OF_LINES (BDA Visible Damage) ~7SL

BDA_VISIBLE_DAMAGE ~(U)

NUMBER_OF_LINES (BDA Sensor Requirements) ~NO4

BDA_SENSOR_REQUIREMENTS ~(U)

NUMBER_OF_LINES (BDA EEI) ~IMT

BDA_ESSENTIAL_ELEMENTS_INFO ~(U)

ERROR_TYPE ~5

CEP_PLANE ~V

NOMINAL_CEP ~CEP_MA

NOMINAL_REP ~288530

NOMINAL_DEP ~216932

NOMINAL_IMPACT_ANGLE ~8156
EFFECTIVENESS_INDEX_TYPE ~6
MAE_VALUE_OR_VAN ~680382
EFFECTIVE_MISS_DISTANCE (EMD) ~0
PERCENT_BURIAL ~0
PROB_OF_DAMAGE_GIVEN_A_HIT ~1.00
ELEMENT_LENGTH_OR_RADIUS ~OBJEC
ELEMENT_WIDTH ~OBJEC
ELEMENT_HEIGHT ~OBJE
ELEMENT_AZIMUTH ~OBJ
ELEMENT_AREA_LENGTH_RADIUS ~OBJEC
ELEMENT_AREA_WIDTH ~OBJET
WEAPON_RELIABILITY ~0.99
RANGE_OFFSET_FROM_DMPI ~0 1
DEFLECTION_OFFSET_FROM_DMPI ~0
PATTERN_TYPE ~0
PROB_ DAMAGE_OR_AVAIL_PASSES ~915
NUMBER_OF_PASSES_REQUIRED ~61677
EXPECTED_PD_FROM_NUM_PASSES ~4045
WEAPON_TYPE ~6
MISSILE_VELOCITY ~430
NUMBER_OF_GS_PULLED ~0.0
RELEASE_ALT_OF_FIRST_WEAPON ~2440
DIVE_ANGLE ~44
NUMBER_OF_RELEASE_PULSES ~921
WEAPONS_RELEASED_PER_PULSE ~2
INTERVALOMETER_SETTING ~5320
DISTANCE_BTWN_STATIONS ~635
MER_TER_RELEASE ~Y
TARGET_ALTITUDE ~0
WEAPON_OR_DISPENSER_TERMINAL_VELOCITY ~250
SECOND_LEG_TERMINAL_VELOCITY ~135
EJECTION_VELOCITY ~0
FUNCTIONING_TIME ~0.50

SLANT_RANGE ~26432

JMEM_IMPACT_ANGLE ~8146

STICK_LENGTH ~5713

STICK_WIDTH ~079

BALLISTIC_ERROR ~22

SUBMUNITION_RELIABILITY ~9143

NUMBER_OF_SUBMUNIT_DISPENSER ~7

PATTERN_LENGTH ~1807

PATTERN_WIDTH ~7294

PROBABILITY_OF_NEAR_MISS ~1.00

PROBABILITY_OF_DIRECT_HIT ~0.00

RANGE_BIAS ~0.00

PROBABILITY_OF_RELEASE ~1.00

INTERPOLATION_FLAG ~N

INTERPOLATION_POINT ~280

INTERP_LOWER_BOUND ~198

MAE_AT_LOWER_BOUND ~892669

INTERP_UPPER_BOUND ~408

MAE_AT_UPPER_BOUND ~126359

NUMBER_OF_LINES (Weaponeering Notes) ~T81

WEAPONEERING_NOTES_1 ~(U)

WEAPONEERING_NOTES_2
~AX8ENR6G3XIB6RYCA8H6QGK14JM54OVG7JALVNX0ZGCIH9GX6JJ11OFUEJ
4YQUIAJ8B4CZJ6BVR5Q33CUS2P9FYY0R94GXJ50ODWDYLDWTU47HFZ3L42
NVNSRV1LJ5JFFIGS8OKYIYLB8JY9N

Case 1067 : Invalid

The purpose of this test case is to verify that the input validation capability recognizes incorrect input.

Test Case~
ZTUN2370743ZBO52N85C8NS1BZVQJKW5X224K33V96IZ9ITD9YAUDD2KBKN
K89OKFFXOOZLWHGVCEQ6JH(U)
RI46149T3TA(U)
VNOL4VZ7SLK(U)
NO4L(U)
IMTF(U)
53VRCEP_MA288530921693208156760680382 60 0

1.00OBJECOBJECOBJEOBJOBJECOBJET0.990 1 0
0915661677740454643 00.02440744192192 532036350Y0 250 1350 0.50
26432281466571310798221914347 18079729441.000.000.00
1.00N28091984892669740871263592T81F(U)
AX8ENR6G3XIB6RYCA8H6QGK14JM54OVG7JALVNX0ZGCIH9GX6JJ11OFUEJ4
YQUIAJ8B4CZJ6BVR5Q33CUS2P9FYY0R94GXJ50ODWDYLDWTU47HFZ3L42N
VNSRV1LJ5JFFIGS8OKYIYLB8JY9NX

Test Case Breakdown:

TABLE_ID ~ZTUN

FORMAT_VERSION_ID ~237074

DAMAGE_CRITERION ~3ZBO52N85C8NS1BZ

DAMAGE_CRITERION_SEQUENCE_NO ~VQJ

SECURITY_CATEGORY (Weaponeering Dataset) ~KW5

CLASSIFICATION_LEVEL (Weaponeering Dataset)) ~X2

NUMBER_OF_SEC_HANDLING_INSTR ~24

SEC_CONTROL_HANDLING_INSTR (Weaponeering Dataset)
~K33V96IZ9ITD9YAUDD2KBKNK89OKFFXOOZLWHGVCE

NUMBER_OF_LINES (Aimpoint Strike Objective) ~Q6JH

AIMPOINT_OBJECTIVE ~(U)

WEAPON_TO_TARGET_FLAG ~RI

PREDICTED_PROB_OF_DAMAGE (SSPD or SSFD) ~46149

NUMBER_OF_LINES (Expected Results) ~T3TA

EXPECTED_RESULTS ~(U)

NUMBER_OF_LINES (Related Aimpoints) ~VNOL

RELATED_AIMPOINTS ~4VZ

NUMBER_OF_LINES (BDA Visible Damage) ~7SLK

BDA_VISIBLE_DAMAGE ~(U)

NUMBER_OF_LINES (BDA Sensor Requirements) ~NO4L

BDA_SENSOR_REQUIREMENTS ~(U)

NUMBER_OF_LINES (BDA EEI) ~IMTF

BDA_ESSENTIAL_ELEMENTS_INFO ~(U)

ERROR_TYPE ~53

CEP_PLANE ~VR

NOMINAL_CEP ~CEP_MA

NOMINAL_REP ~2885309

NOMINAL_DEP ~2169320

NOMINAL_IMPACT_ANGLE ~81567
EFFECTIVENESS_INDEX_TYPE ~60
MAE_VALUE_OR_VAN ~6803826
EFFECTIVE_MISS_DISTANCE (EMD) ~0
PERCENT_BURIAL ~0
PROB_OF_DAMAGE_GIVEN_A_HIT ~1.00
ELEMENT_LENGTH_OR_RADIUS ~OBJEC
ELEMENT_WIDTH ~OBJEC
ELEMENT_HEIGHT ~OBJE
ELEMENT_AZIMUTH ~OBJ
ELEMENT_AREA_LENGTH_RADIUS ~OBJEC
ELEMENT_AREA_WIDTH ~OBJET
WEAPON_RELIABILITY ~0.99
RANGE_OFFSET_FROM_DMPI ~0 1
DEFLECTION_OFFSET_FROM_DMPI ~0
PATTERN_TYPE ~0
PROB_ DAMAGE_OR_AVAIL_PASSES ~9156
NUMBER_OF_PASSES_REQUIRED ~616777
EXPECTED_PD_FROM_NUM_PASSES ~40454
WEAPON_TYPE ~6
MISSILE_VELOCITY ~430
NUMBER_OF_GS_PULLED ~0.0
RELEASE_ALT_OF_FIRST_WEAPON ~24407
DIVE_ANGLE ~441
NUMBER_OF_RELEASE_PULSES ~9219
WEAPONS_RELEASED_PER_PULSE ~2
INTERVALOMETER_SETTING ~53203
DISTANCE_BTWN_STATIONS ~6350
MER_TER_RELEASE ~Y
TARGET_ALTITUDE ~0
WEAPON_OR_DISPENSER_TERMINAL_VELOCITY ~250
SECOND_LEG_TERMINAL_VELOCITY ~135
EJECTION_VELOCITY ~0
FUNCTIONING_TIME ~0.50

SLANT_RANGE ~264322

JMEM_IMPACT_ANGLE ~81466

STICK_LENGTH ~57131

STICK_WIDTH ~0798

BALLISTIC_ERROR ~221

SUBMUNITION_RELIABILITY ~91434

NUMBER_OF_SUBMUNIT_DISPENSER ~7

PATTERN_LENGTH ~18079

PATTERN_WIDTH ~72944

PROBABILITY_OF_NEAR_MISS ~1.00

PROBABILITY_OF_DIRECT_HIT ~0.00

RANGE_BIAS ~0.00

PROBABILITY_OF_RELEASE ~1.00

INTERPOLATION_FLAG ~N

INTERPOLATION_POINT ~2809

INTERP_LOWER_BOUND ~1984

MAE_AT_LOWER_BOUND ~8926697

INTERP_UPPER_BOUND ~4087

MAE_AT_UPPER_BOUND ~1263592

NUMBER_OF_LINES (Weaponeering Notes) ~T81F

WEAPONEERING_NOTES_1 ~(U)

WEAPONEERING_NOTES_2
~AX8ENR6G3XIB6RYCA8H6QGK14JM54OVG7JALVNX0ZGCIH9GX6JJ11OFUEJ
4YQUIAJ8B4CZJ6BVR5Q33CUS2P9FYY0R94GXJ50ODWDYLDWTU47HFZ3L42
NVNSRV1LJ5JFFIGS8OKYIYLB8JY9NX

Test Case Report

Date: Tuesday, June 09, 1998

NOTE: ~~ is used to separate out the test case

Id	*Table Name*	*Type of Test Case*	*Test Case*
1064	Table_3_2_4_1_10___ETF_Weaponeering_Dataset	Invalid	(U) (U) (U) (U) (U) CEP_MA0 0 1.00OBJECOBJECOBJEOBJOBJECOBJET0.990 1 0 064300.02 Y0 250 1350 0.50 7 1.000.000.00 1.00N(U)
1065	Table_3_2_4_1_10___ETF_Weaponeering_Dataset	Invalid	Z23VKX2KQ(U) R4T(U) V47(U) N(U) I(U) 5VCEP_MA228660 0 1.00OBJECOBJECOBJEOBJOBJECOBJET0.990 1 0 096464300.02492 56Y0 250 1350 0.50 2850297 171.000.000.00 1.00N21841T(U) A

NOTE: ~~ is used to separate out the test case

Id	*Table Name*	*Type of Test Case*	*Test Case*
1066	Table_3_2_4_1_10___ETF_Weaponeering_Dataset	Valid	ZTU237073ZBO52N85C8NS1BVQKWX2K33V96IZ9ITD9YAUDD2KBKNK89OKFFXOOZLWHGVCQ6J(U) R4614T3T(U) VNO4V7SL(U) NO4(U) IMT(U) 5VCEP_MA28853021693281566680382 0 1.00OBJECOBJECOBJEOBJOBJECOBJET0.990 1 0 091561677404564300.02440449212 5320635Y0 250 1350 0.50 2643281465713079229143 7 180772941.000.000.00 1.00N2801988926694081263 59T81(U) AX8ENR6G3XIB6RYCA8H6QGK14JM54OVG7JALVNX0ZGCIH9GX6JJ11OFUEJ4YQUIAJ8B4CZJ6BVR5Q33CUS2P9FYY0R94GXJ50ODWDYLDWTU47HFZ3L42NVNSRV1LJ5JFFIGS8OKYIYLB8JY9N

1067Table_3_2_4_1_10___ETF_Weaponeering_Dataset	Invalid	ZTUN2370743ZBO52N85C8NS1BZVQJKW5X224K33V96IZ9ITD9YAUDD2KBKNK89OKFFXOOZLWHGVCEQ6JH(U) RI46149T3TA(U) VNOL4VZ7SLK(U) NO4L(U) IMTF(U) 53VRCEP_MA288530921693208156760680382600 1.00OBJECOBJECOBJEOBJOBJECOBJET0.990 1 0 0915661677740454643000.02440744192192 532036350Y0 250 1350 0.50 264322814665713107982219143471807972944 1.000.000.00 1.00N28091984892669740871263592T81F(U) AX8ENR6G3XIB6RYCA8H6QGK14JM54OVG7JALVNX0ZGCIH9GX6JJ11OFUEJ4YQUIAJ8B4CZJ6BVR5Q33CUS2P9FYY0R94GXJ50ODWDYLDWTU47HFZ3L42NVNSRV1LJ5JFFIGS8OKYIYLB8JY9NX

NOTE: ~~ is used to separate out the test case

Id	*Table Name*	*Type of Test Case*	*Test Case*
1084	Table_3_2_4_1_10___ETF_Weaponeering_Dataset	Valid/Overloaded Token Static Error	ZTU~~23707~~3ZBO52N85C8NS1B~~VQ~~KW~~X~~2~~K33V96IZ9ITD9Y AUDD2KBKNK89OKFFXOOZLWHGVC~~Q6J~~(U) ~~R~~4614~~T3T~~(U) ~~VNO~~4V~~7SL~~(U) ~~NO4~~(U) ~~IMT~~(U) ~~5~~V~~CEP_MA~~288530~~216932~~8156~~6~~680382~~0 ~~0 ~~1.00~~OBJEC~~OBJEC~~OBJE~~OBJ~~OBJEC~~OBJET~~0.99~~0 ~~1 ~~0 ~~0~~915~~61677~~4045~~6~~430~~0.0~~2440~~44~~921~~2 ~~5320~~635~~Y~~0 ~~250 ~~135~~0 ~~0.50 ~~26432~~8146~~5713~~079~~22~~9143~~7 ~~1807~~7294~~1.00~~0.00~~0.00 ~~1.00~~N~~280~~198~~892669~~408~~12635(~~T81~~(U) ~~AX8ENR6G3XIB6RYCA8H6QGK14JM54OVG7JALVNX0ZGCIH9GX6JJ11 OFUEJ4YQUIAJ8B4CZJ6BVR5Q33CUS2P9FYY0R94GXJ50ODWDYLDWTU 47HFZ3L42NVNSRV1LJ5JFFIGS8OKYIYLB8JY9N~~

NOTE: ~~ is used to separate out the test case

Id	*Table Name*	*Type of Test Case*	*Test Case*
1085	Table_3_2_4_1_10___ETF_Weaponeering_Dataset	Valid/Overloaded Token Static Error	ZTU~~23707~~3ZBO52N85C8NS1B~~VQ~~KW~~X~~2~~K33V96IZ9ITD9Y AUDD2KBKNK89OKFFXOOZLWHGVC~~Q6J~~(U) ~~R~~4614~~T3T~~(U) ~~VNO~~4V~~7SL~~(U) ~~NO4~~(U) ~~IMT~~(U) ~~5~~V~~CEP_MA~~288530~~216932~~8156~~6~~680382~~0 ~~0 ~~1.00~~OBJEC~~OBJEC~~OBJE~~OBJ~~OBJEC~~OBJET~~0.99~~0 ~~1 ~~0 ~~0~~915~~61677~~4045~~6~~430~~0.0~~2440~~44~~921~~2 ~~5320~~635~~Y~~0 ~~250 ~~135~~0 ~~0.50 ~~26432~~8146~~5713~~079~~22~~9143~~7 ~~1807~~7294~~1.00~~0.00~~0.00 ~~1.00~~N~~280~~198~~892669~~408~~126359~~U81~~(U) ~~AX8ENR6G3XIB6RYCA8H6QGK14JM54OVG7JALVNX0ZGCIH9GX6JJ11 OFUEJ4YQUIAJ8B4CZJ6BVR5Q33CUS2P9FYY0R94GXJ50ODWDYLDWTU 47HFZ3L42NVNSRV1LJ5JFFIGS8OKYIYLB8JY9N~~

Appendix B

Table B1. Condition Means (if Condition is significant)

NUMBER	DEPENDENT VARIABLE	CONTROL CONDITION (MANUAL)	EXPERIMENTAL CONDITION (MICASA)
1	Total number of specification defects detected	7.5	37.4
2	Total number of syntactic specification defects detected	1.5	37.4
3	Total time for the exercise	Given below	Given below
	Part I	--	--
	Part II	--	--
	Part III	175.7	7.9
4	Percentage of syntax coverage	31.25	100
5	Percentage of effective test cases	--	--
6	Total number of defects detected	--	--
7	Average number of defects detected per test case	7.4	4.6
8	Average time for all defects identified	Given below	Given below
	Test Case Execution Time Only	30.9	2.17
	Test Case Development and Execution Time	72.2	8.4

Table B2. System and Condition Means (if significant)

NUMBER	DEPENDENT VARIABLE	SYSTEM A		SYSTEM B		SYSTEM C	
		Manual	MICASA	Manual	MICASA	Manual	MICASA
1	Total number of specification defects detected	--	--	--	--	--	--
2	Total number of syntactic specification defects detected	--	--	--	--	--	--
3	Total time for the exercise	Given below	Given below	Given below	Given below	Given below	Given below
	Part I	--	--	--	--	--	--
	Part II	7.55	0.06	N/A	N/A	135	0.47
	Part III	N/A	N/A	N/A	N/A	N/A	N/A
4	Percentage of syntax coverage	--	--	--	--	--	--
5	Percentage of effective test cases	--	--	--	--	--	--
6	Total number of defects detected	N/A	N/A	N/A	N/A	N/A	N/A
7	Average number of defects detected per test case	N/A	N/A	N/A	N/A	N/A	N/A
8	Average time for all defects identified	N/A	N/A	N/A	N/A	N/A	N/A

Table B3. System Means (if significant)

NUMBER	DEPENDENT VARIABLE	SYSTEM A	SYSTEM B	SYSTEM C
1	Total number of specification defects detected	--	--	--
2	Total number of syntactic specification defects detected	--	--	--
3	Total time for the exercise	Given below	Given below	Given below
	Part I	--	--	--
	Part II	31.8	12.5	59.5
	Part III	N/A	N/A	N/A
4	Percentage of syntax coverage	--	--	--
5	Percentage of effective test cases	62.9	N/A	92.4
6	Total number of defects detected	N/A	N/A	N/A
7	Average number of defects detected per test case	N/A	N/A	N/A
8	Average time for all defects identified	N/A	N/A	N/A

Part III of Experiment
Time to Execute Test Cases (in minutes)

Tester's Case (for Control)	Minutes for Control (Senior Tester)	Minutes for Experimental (MICASA)
B11	77	29
B11	46	38
B8	70	31
B8	286	45
B8	201	5
B8	258	25
B8	292	7
		4
		3
		2
		2
		2
		4
		2
		3
		2
		2
		2
		2
		2
		2
		2
		2
		1
		2
		2
		1
		2
		1
		2
		2
		2
		2
		1
		2
		1
		2
		1
		2
		1
		1
		1
		57
		23
		29
		13
		9

Part III of Experiment
Number of Defects Detected Per Case

Tester's Case (for Control)	# Defects for Control Case (Senior Tester)	# Defects for Experimental Case (MICASA)
B11	2	0
B11	3	3
B8	1	4
B8	15	1
B8	2	4
B8	0	5
B8	4	4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		4
		11
		11
		12
		14
		11

Part III of Experiment
Average Time (minutes) to Identify a Defect (Execution Only) - Condition

Tester's Case (for Control)	# Min. per Defect for Control Case (Senior Tester)	# Min. per Defect for Experimental Case (MICASA)
B11	38.5	12.66666
B11	15.33333	7.75
B11	70	45
B8	19.06666	1.25
B8	22.33333	5
B8	28.66666	1.75
B8	22.46153	1
		0.75
		0.5
		0.5
		0.5
		1
		0.5
		0.75
		0.5
		0.5
		0.5
		0.5
		0.5
		0.5
		0.5
		0.5
		0.25
		0.5
		0.5
		0.25
		0.5
		0.25
		0.5
		0.5
		0.5
		0.5
		0.25
		0.5
		0.25
		0.5
		0.25
		0.5
		0.25
		0.25
		0.25
		5.181818
		2.090909
		2.416666
		0.928571
		0.818181

Part II of Experiment
Test Case % Syntax Coverage – System and Condition

Table Name	% Coverage for Control Case (Senior Tester)	% Coverage for Experimental Case (MICASA)
ETF	25	100
ETF	25	100
P Gen	31.25	100
P Gen	31.25	100
T Compl	25	100
T Compl	50	100

Part II of Experiment
Test Case % Syntax Coverage – Condition

% Coverage for Control Case (Senior Tester)	% Coverage for Experimental Case (MICASA)
25	100
25	100
31.25	100
31.25	100
25	100
50	100

Part II of Experiment
Time (minutes) to Develop a Test Case – System and Condition

Minutes for Control Case (Senior Tester)	Minutes for Experimental Case (MICASA)
130	0.47
140	0.47
7.9	0.06
7.2	0.06

Part II of Experiment
% Effective Test Cases – System

% Effective for System A	% Effective for System C
75	100
100	100
13.5	84.9
72.72	84.9
100	
84	
56.25	
56.25	
56.25	
56.25	
42.4	
42.4	

Part I of Experiment
Total # Syntactic Spec. Defects – Condition

# Defects Found for Control Case (Senior Tester)	# Defects Found for Experimental Case (MICASA)
0	6
0	6
1	6
1	6
1	34
0	34
0	34
0	34
5	89
3	89
1	7
1	7
0	86
8	86

Part I of Experiment
Total #Time – System

System A	System B	System C
15	10	15
1.5	0.75	75
15	27	74
60	27	74
34	5	
34	1	
34	15	
34	15	
15		
0.4		
15		
120		
33		
33		
33		
33		

Appendix C

Defects Detected for JEDMICS (System C) Specification Table During Static Analysis

Table Analyzed:

Table: Attributes not part of doc_jecmics

Attribute Id	Size	Type	Valid Values	Purpose
JMX_imageStatusCode	1	char		Image status Code
JMX_hitLimit	4	Long	Number	Number of hits from a query Database. Default is 1000
JMX_mode	0	Long	JMX_HIGH_REV JMX_ALL_REV JMX_ONE_REV JMX_DWG_BOOK	type of revision. Default is highest revision.
JMX_drawingCount	4	long	Long	Number of drawings matching a criteria
JMX_sheetCount	4	long	Long	Number of Sheets matching a search criteria
JMX_frameCount	4	long	Long	Number of frames matching a search criteria.
JMX_maxConnect	4	Char	Number (min)	Identifies the maximum length of any session for the user, after which the user will be automatically logged out of the system.
JMX_maxIdle	4	Char	Number (min)	Identifies the maximum length of time in minutes that the user is permitted to be idle during any log-on session after which the user is automatically logged of the system
JMX_pwdExpireDate	18	Char	dd-mon-yy	The date a password expires.

Defects Detected:

Type of Defect	Who Found the Defect?	Defect Description
Syntactic	Testers B1, B2	JMX_Mode has a size of 0 but valid values and Long given
Semantic	Tester B1	JMX_drawingCount type is Long, valid value Long, but purpose is a number
Semantic	Tester B1	JMX_sheetCount type is Long, valid value Long, but purpose is a number
Semantic	Tester B1	JMX_frameCount type is Long, valid value Long, but purpose is a number
Semantic	Tester B1	JMX_pwdExpireDate is not Y2K compliant
Syntactic	MICASA	Item JMX_maxIdle could run together with next field (JMX_pwdExpireDate) since both types expect digits, no delimiters specified, and no actual expected values given – Possible catenation error
Syntactic	MICASA	Item JMX_maxConnect could run together with next field (JMX_maxIdle) since same data type, no delimiters specified, and no actual expected values given – Possible catenation error
Syntactic	MICASA	Item JMX_frameCount could run together with next field (JMX_maxConnect) since both types expect digits, no delimiters specified, and no actual expected values given – Possible catenation error
Syntactic	MICASA	Item JMX_sheetCount could run together with next field (JMX_frameCount) since same data type, no delimiters specified, and no actual expected values given – Possible catenation error
Syntactic	MICASA	Item JMX_drawingCount could run together with next field (JMX_sheetCount) since same data type, no delimiters specified, and no actual expected values given – Possible catenation error
Syntactic	MICASA	JMX_imageStatusCode actual value of blank

Appendix D

Table of Comparisons between MICASA and Senior Tester Procedures

Procedure Name	Comp. Time Hour:Min	No. of NEW DRs (total DRs)	DR Number(s) *NEW* and Other
MANUAL TEST CASES			
01.IRS-1 Generate a Baseline WDU Weaponeering Data	1:17	2 (2)	*DRHUMAN001* *DRHUMAN002*
01.IRS-2 Generate a Baseline BLU Weaponeering Dataset	0:46	3 (3)	*DRHUMAN003* *DRHUMAN004* *DRHUMAN005*
01.IRS-3 Generate a Weaponeering Dataset Excluding all Optional Fields	1:10	1 (1)	DRHUMAN006
Totals for this set:	**3:13**	**6 (6)**	
(Time required to develop these three test cases: 7 hours)			
Case 1: WDU-36B VDM	6:46	15 (15)	DRHMNRD2001 DRHMNRD2002 DRHMNRD2003 DRHMNRD2004 DRHMNRD2005 DRHMNRD2006 DRHMNRD2007 DRHMNRD2008 DRHMNRD2009 DRHMNRD2010 DRHMNRD2011 DRHMNRD2012 DRHMNRD2014 DRHMNRD2016 DRHMNRD2017
Case 2: WDU-36B HAM	3:21	2 (9)	DRHMNRD2001 DRHMNRD2002 DRHMNRD2003 DRHMNRD2004 DRHMNRD2005 DRHMNRD2011 DRHMNRD2013 DRHMNRD2014 DRHMNRD2015
Case 3: WDU-36B PWD	4:18	0 (9)	DRHMNRD2001 DRHMNRD2002 DRHMNRD2003 DRHMNRD2004 DRHMNRD2005 DRHMNRD2011 DRHMNRD2013 DRHMNRD2014 DRHMNRD2015

Case 4: BLU-97	4:52	4 (13)	DRHMNRD2001 DRHMNRD2002 DRHMNRD2003 DRHMNRD2004 DRHMNRD2005 DRHMNRD2011 DRHMNRD2013 DRHMNRD2014 DRHMNRD2015 DRHMNRD2018 DRHMNRD2019 DRHMNRD2020 DRHMNRD2021
Totals for this set:	19:17	21 (46)	
(Time required to develop these four test cases: 6.5 hours)			
MICASA TEST CASES			
Case 2067: Invalid	0:29	0 (0)	
Case 2068: Invalid	0:38	3 (3)	*DRTOOL001* *DRTOOL002* *DRTOOL003*
Case 2069: Valid	0:31	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 2070: Invalid	0:45	0 (1)	DRHUMAN002
Case 778: Valid/Overloaded Token Static Error	0:05	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 779: Valid/Overloaded Token Static Error	0:25	1 (5)	DRTOOL001 DRTOOL002 DRTOOL003 DRTOOL004 DRHUMAN002
Case 780: Valid/Overloaded Token Static Error	0:07	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 788: Valid/Overloaded Token Static Error	0:04	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 813: Valid/Overloaded Token Static Error	0:03	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 815: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 817: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 818: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002

Case 820: Valid/Overloaded Token Static Error	0:04	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 821: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 822: Valid/Overloaded Token Static Error	0:03	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 823: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 824: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 825: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 826: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 827: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 828: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 829: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 830: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 831: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 832: Valid/Overloaded Token Static Error	0:01	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 833: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 834: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002

Case 835: Valid/Overloaded Token Static Error	0:01	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 836: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 837: Valid/Overloaded Token Static Error	0:01	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 838: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 839: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 840: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 841: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 842: Valid/Overloaded Token Static Error	0:01	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 843: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 844: Valid/Overloaded Token Static Error	0:01	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 845: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 847: Valid/Overloaded Token Static Error	0:01	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 848: Valid/Overloaded Token Static Error	0:02	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 849: Valid/Overloaded Token Static Error	0:01	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 850: Valid/Overloaded Token Static Error	0:01	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002

Case 851: Valid/Overloaded Token Static Error	0:01	0 (4)	DRTOOL001 DRTOOL002 DRTOOL003 DRHUMAN002
Case 852: Invalid/Top Intermediate	0:57	3 (11)	DRHMNRD2001 DRHMNRD2002 DRHMNRD2003 DRHMNRD2004 DRHMNRD2005 DRHMNRD2014 DRHMNRD2015 DRHMNRD2016 DRTOOL2001 DRTOOL2002 DRTOOL2003
Case 853: Invalid/Top Intermediate	0:23	0 (11)	DRHMNRD2001 DRHMNRD2002 DRHMNRD2003 DRHMNRD2004 DRHMNRD2005 DRHMNRD2014 DRHMNRD2015 DRHMNRD2016 DRTOOL2001 DRTOOL2002 DRTOOL2003
Case 854: Invalid/Delimiter Error	0:29	1 (12)	DRHMNRD2001 DRHMNRD2002 DRHMNRD2003 DRHMNRD2004 DRHMNRD2005 DRHMNRD2014 DRHMNRD2015 DRHMNRD2016 DRTOOL2001 DRTOOL2002 DRTOOL2003 DRTOOL2004
Case 855: Invalid/Delimiter Error	0:13	0 (14)	DRHMNRD2001 DRHMNRD2002 DRHMNRD2003 DRHMNRD2004 DRHMNRD2005 DRHMNRD2008 DRHMNRD2009 DRHMNRD2010 DRHMNRD2014 DRHMNRD2015 DRHMNRD2016 DRTOOL2001 DRTOOL2002 DRTOOL2003

Case 856: Invalid/Field-Value Error	0:09	0 (11)	DRHMNRD2001 DRHMNRD2002 DRHMNRD2003 DRHMNRD2004 DRHMNRD2005 DRHMNRD2014 DRHMNRD2015 DRHMNRD2016 DRTOOL2001 DRTOOL2002 DRTOOL2003
Case 1161: Invalid/Delimiter Error	Test Cases not run, Tester could see that only the Free Text fields had been changed and knew that PTW does not check these fields (by design)		
Case 1162: Invalid/Top Intermediate			
Case 1163: Invalid/Top Intermediate			
Case 1164: Invalid/Delimiter Error			
Case 1167: Invalid/Field-Value Error			
Totals for this set:	**6:20**	**8 (224)**	
(Time required to develop these test cases: 25 min.)			

List of Discrepancy Reports (DRs) found during Manual Testing

1. **DRHUMAN001** – Expected PD from Number of Passes (Priority 5)
   This is a Conditional field, and is only applicable when PDNA is greater than 1. However, PDNA value was input as .5, and yet this field was available and allowed a value to be entered and saved.

2. **DRHUMAN002** – The Weaponeering Notes field was not accessible from the WDU Dataset (Priority 4)

3. **DRHUMAN003** – Nominal CEP was available when Error Type was set to 3 (Priority 5)

4. **DRHUMAN004** – IRS/M says Probability of Release is only TLAM-C, yet available on BLU (Priority 5)

5. **DRHUMAN005** – Interpolation Flag missing 'D' for Distance Measurements (Priority 4)

6. **DRHUMAN006** – Found Several Fields Required for MTF Even Though JMEM Data Not Used (Priority 4)

   Found 12 fields that are marked as 'Conditional – Required, if JMEM data is provided' for creating an MTF. In this particular test case, no JMEM was used, and these fields should not have been required for an MTF.

7. **DRHMNRD2001 –** Dataset scrollbar arrow acts as Page Down (Priority 5) - Using the arrows on the scrollbar should result in a field by field scrolling, but a page by page scroll occurs instead.

8. **DRHMNRD2002 –** PTW software crashes when Setting Menu Security Level, then ETF Security Level (Priority 4)

9. **DRHMNRD2003 –** X-Window Interface problem when highlighting text (Priority 4)

10. **DRHMNRD2004 –** Error Type Field accepts invalid data (Priority 4)

11. **DRHMNRD2005** – "Bridge" in EI Type Selection Window spelled "Bridget" (Priority 5)

12. **DRHMNRD2006** – Pattern Types in Integrated JMEM should match PCJMEM (Priority 4)

13. **DRHMNRD2007 –** System should not allow "D" for Interpolation Flag Field for VDM Mission (Priority 5)

14. **DRHMNRD2008** – Interpolation Point allows 5 char. input (Priority 5)

15. **DRHMNRD2009** – JMEM Clear Output puts garbage char. in Interpolation Flag Field (Priority 4)

16. **DRHMNRD2010** – JMEM Guided yields "-?.00" in SSPD field (Priority 4)

17. **DRHMNRD2011 -** Additional Related Aimpoints Required to Satisfy Objective Field will not accept data (Priority 4)

18. **DRHMNRD2012 –** Range for Nominal Impact Angle is too large for VDM (Priority 5)

19. **DRHMNRD2013** – Value for Nominal Impact Angle should not be modifiable for HAM or PWD (Priority 5)

20. **DRHMNRD2014** – Probability of Near Miss, Probability of Direct Hit, and Range Bias should be grayed out, unless Error Type is "1" (Priority 4)

21. **DRHMNRD2015 –** Interpolation fields should be grayed out when Interpolation Flag is set to "N" (Priority 4)

22. **DRHMNRD2016** – Software will not accept Interpolation Bounds until MAE Bounds are saved (Priority 4)

23. **DRHMNRD2017 –** IRS/M Range for Dive Angle should be –90-90 (Priority 4)

24. **DRHMNRD2018** – "Release Altitude of First Weapon Displayed" Range doesn't match actual range (Priority 5)

25. **DRHMNRD2019** – Cannot save Weaponeering Dataset with PDNA > 9.99 (Priority 4)

26. **DRHMNRD2020** – Weapons Released Per Pulse accepts values out of range (Priority 5)

27. **DRHMNRD2021** – Pattern Length Accepts "0" as input (Priority 5)

List of DRs found during MICASA Testing

1. **DRTOOL001** - Error Type '5' was Allowed for Input (Priority 5)
   This was not an option given in the IRS/M.

2. **DRTOOL002** - CEP Plane was Available when Error Type was '5' (Priority 5)
   IRS/M says CEP Plane only available when error type is '1'.
3. **DRTOOL003** - Nominal DEP and Nominal REP Available when Error Type was '5' (Priority 5)
   IRS/M says CEP Plane only available when error type is '3'.

4. **DRTOOL004** – Number of Release Pulses Accepts an Invalid Value (Priority 5)

5. **DRTOOL2001 -** Invalid Error Message when attempting to create Invalid Weaponeering Dataset (Priority 5)

6. **DRTOOL2002** – CEP Plane is not grayed out when Error Type is set to “5” (Priority 4) (002)

7. **DRTOOL2003 –** Nominal REP and Nominal DEP are not grayed out when Error Type is set to “5” (Priority 4) (003)

8. **DRTOOL2004** – System allows invalid entry for Weapons Released Per Pulse (Priority 4)

References

REFERENCES

[1] Ammann, P. and Offutt, A. Jefferson. Using formal methods to derive test frames in category-partition testing. *Proceedings of COMPASS '94*, Gaithersburg, MD, June -July 1994, pp. 69 - 79.

[2] DeMillo, R. A., Guindi, D. S., King, K. N., McCracken, W. M., and A. Jefferson Offutt. An extended overview of the {M}othra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Alberta, July 1988, pp. 142 – 151, IEEE Computer Society Press.

[3] Bauer, J. and Finger, A. Test plan generation using formal grammars. *ICSE 4: Proceedings of the 4th International Conference on Software Engineering*, Munich, 1979, pp.425-432.

[4] Bazzichi, F. and Spadafora, I. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, July 1982, pp. 343-353.

[5] Beizer, B. Software Testing Techniques. Van Nostrand Reinhold, New York, New York, 1990.

[6] Bird, D. L. and Munoz, C. U. Automatic generation of random self-checking test cases. *IBM Systems Journal*, Volume 22, No. 3, 1983, pp. 229-245.

[7] Davis, A. Software Requirements Analysis and Specification. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[8] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. Hints on test data selection: Help for the practicing programmer. *IEEE Computer 11*,4 (Apr. 1978), 34-41.

[9] DOD-STD-2167A. Defense System Software Development. Department of Defense, February 1988.

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

[11] Frankl, P. G., and Weiss, S. N. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification* (1991), Victoria, BC, IEEE Computer Society Press, pp. 154-164.

[12] Gough, K. John. Syntax Analysis and Software Tools. Addison-Wesley Publishing Company, New York, New York, 1988.

[13] Hanford, K. V. Automatic generation of test cases. *IBM Systems Journal*, Volume 9, No. 4, 1970, pp. 242 - 257.

[14] (Huffman) Hayes, J. And C. Burgess. "Partially Automated In-Line Documentation (PAID): Design and Implementation of a Software Maintenance Tool," In *The Proceedings of the 1988 IEEE Conference on Software Maintenance*, Phoenix, AZ, October 1988.

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

[16] Hayes, J. Huffman. Testing object-oriented systems: A fault-based approach," In *The Proceedings of the International Symposium on Object-Oriented Methodologies and Systems (ISOOMS*), Springer-Verlag Lecure Notes on Computer Science series, Number 858, September 1994, Palermo, Italy.

[17] Hook, A. A survey of computer programming languages currently used in the Department of Defense: An executive summary, *Crosstalk*, October 1995, p. 4 - 5.

[18] Horgan, J. R., and London, S. A data flow coverage testing tool for C. In *Proceedings of the Symposium of Quality Software Development Tools*, New Orleans, Louisiana, May 1992, pp. 2 - 10.

[19] Howden, W.E. Functional Program Testing. IEEE Transactions on Software Engineering. SE-6(2), pp. 162-169, March 1980.

[20] IDA Report M-326. Analysis of Software Obsolescence in the DoD: Progress report. Institute for Defense Analyses, June 1987.

[21] IEEE Standard 729-1983. Standard Glossary of Software Engineering Terminology. IEEE, 13 February 1983.

[22] IEEE Standard 830-1984. Software Requirements Specifications. IEEE, 10 February 1984.

[23] Ince, D. C. The automatic generation of test data. *The Computer Journal*, Volume 30, No. 1, 1987, pp. 63 - 69.

[24] King, K. N. and Offutt, A.Jefferson. A FORTRAN language system for mutation-based software testing. *Software-Practice and Experience*, 21(7):685-718, July 1991.

[25] Liu, L. M. and Prywes, N. S. SPCHECK: A Specification-Based tool for interface checking of large, real-time/distributed systems. In *Proceedings of Information Processing (IFIP)*, San Francisco, 1989.

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

[27] Maurer, P. M. Reference manual for a data generation language based on probabilistic context free grammars. Technical Report CSE-87-00006, University of Florida, Tampa, 1987.

[28] Maurer, P. M. Generating test data with enhanced context-free grammars. *IEEE Software*, July 1990, pp. 50 - 55.

[29] MIL-STD-498. Software Development and Documentation. Department of Defense, December 1994.

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

[31] Offutt, A. Jefferson. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):3-18, January 1992.

[32] Offutt, A. Jefferson. and Hayes, J. Huffman. "A semantic model of program faults," In *The Proceedings of ISSTA*, ACM, January 1996.

[33] Offutt, A.J., Lee, A., Rothermel, G., Untch, R. and Zapf, C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99-118, April 1996.

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

[35] Parnas, D. L. Letters to the editors, *American Scientists*, Volume 74, January-February 1986, pp. 12 - 15.

[36] Payne, A. J. A formalised technique for expressing compiler exercisers. *SIGPLAN Notices*, Volume 13, No. 1, January 1978, pp. 59 - 69.

[37] Purdom, P. A sentence generator for testing parsers. *BIT*, Volume 12, 1972, pp. 366 - 375.

[38] Sizemore, N. L. Test techniques for knowledge-based systems. *ITEA Journal*, Vol. 11, No. 2, 1990.

[39] Voas, J. M. PIE: A dynamic failure-based technique, *IEEE Transactions on Software* Engineering, Volume 18, No. 8, August 1992.

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

[41] Webster, N. Webster's New Collegiate Dictionary. G&C Merriam Company, Springfield, Massachusetts, 1977.

[42] White, L. J. "Software Testing and Verification." In Advances in Computers. Edited by Marshall Yovits, Volume 26, 1987, Academic Press, Inc., pp. 335 – 390.

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

CURRICULUM VITAE

Jane Huffman Hayes was born on July 31, 1962 in Xenia, Ohio, and is an American citizen. She graduated from Zionsville High School, Zionsville, Indiana, in 1980. She received her Bachelor of Arts from Hanover College in 1983. She was employed by Defense Intelligence Agency for one year and then by Science Applications International Corporation for one and a half years. She received her Master of Science in Computer Science from the University of Southern Mississippi in 1987. She is in her fourteenth year of employment with Science Applications International Corporation.