Increased Software Reliability Through Input Validation Analysis and Testing *

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

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

Jane Huffman Hayes and A. Jefferson Offutt. Increased Software Reliability Through Input Validation Analysis and Testing. *The Tenth IEEE International Symposium on Software Reliability Engineering (ISSRE '99)*, pages 199--209, Boca Raton, FL, November 1999.

Abstract

The Input Validation Testing (IVT) technique has been developed to address the problem of statically analyzing input command syntax as defined in English textual interface and requirements specifications and then generating test cases for input validation testing. The technique does not require design or code, so it can be applied early in the lifecycle. A proof-of-concept tool has been implemented and validation has been performed. Empirical validation on industrial software shows that the IVT method found more requirement specification defects than senior testers, generated test cases with higher syntactic coverage than senior testers, and found defects that were not found by the test cases of senior testers. Additionally, the tool performed at a much-reduced cost.

1 Introduction

Many computer programs interact with users through various sorts of commands and many software faults are a result of mishandling commands. To be considered reliable, software must process valid commands correctly and respond to invalid commands with reasonable tolerance. Commands may be in many forms, including mouse clicks, screen touches, pen touches, voice, and files. Programs that accept free-form input, interactive input from users, and free-form numbers are all examples of syntax driven applications [7]. In this paper, a *syntax driven application* accepts inputs from the users, constructed and arranged properly, that direct the processing of the software application. Some GUI-based systems and most mode-based reactive systems such as traffic signal controllers are not syntax driven. Also, some system can be mixed, with parts of the their interfaces being syntax driven, and some not. This paper presents a new technique for testing syntax driven applications. The technique has been automated and validated, and can be applied to a variety of software applications.

There is a large amount of software that depends on user input via keyboard entry, largely undocumented [10], that will need to be maintained for many years to come. According to a survey performed by the U.S. Institute for Defense Analyses (IDA), a good deal of first and second generation language software still exists [6]. 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 [2]. Beizer presents one fault taxonomy that suggests that about 8.1% of faults come from problems in the requirements. As voice-controlled software becomes more widespread, the problem of testing syntax driven applications will become applicable to a much broader set of applications.

A *user command* is a user input that directs the control flow of a computer program. A *user command language* has a complete, finite set of commands that are entered textually through the keyboard. *Syntax driven software* has a command language interface. Syntax driven applications must be able to properly handle user commands that are not constructed and arranged as expected, and must be able to properly handle user commands that are constructed and ar-

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

ranged 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 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.

1.1 System Testing

Though much research has been done in the area of unit testing, system testing has not garnered as much attention from researchers. This is partly due to the expansive nature of system testing: many unit level testing techniques cannot be practically applied to millions of lines of code. There are many well defined testing criterion for unit testing [2, 21] but not for system testing. Lack of formal research results in a lack of formal, standard criteria, general purpose techniques, and tools.

Much of the system-level research undertaken to date has largely concentrated on testing for performance, security, accountability, configuration sensitivity, start-up, and recovery [2]. These techniques require that source code already exist. Such dynamic techniques are referred to as *detective techniques* since they are only able to identify already existing defects. Techniques that can be applied early in the life cycle are more desirable. *Preventive techniques* help avert the introduction of defects into the software and allow early identification of defects when it is less costly and time consuming to repair them.

1.2 Input Validation

Input validation refers to those functions in software that attempt to validate the syntax of user-provided commands/information. It is desirable to have a systematic way to prepare test cases for this software early in the life cycle. By doing this, user input commands can be analyzed for completeness and consistency. It is preferable that user commands be documented in Software Requirement Specifications (SRS), Interface Requirements Specifications (IRS), Software Design Documents (SDD), and Interface Design Documents (IDD). The 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 on the code.

The goal of this research is to improve the quality of natural language, textual interface requirements specifications and the resulting software. This is done by formalizing the analysis and testing of interface requirements specifications without forcing testers to learn a new specification language and without increasing the cost of testing. This paper presents a new method for analyzing and testing syntax-directed software that generates test cases from syntactic anomalies in specifications. A proof-of-concept system was constructed and validated. We empirically established large improvements over current practice in the number of anomalies statically detected in specifications, the time needed for testing, and the number of specification defects and software faults found.

2 Background

A goal of system testing is to detect faults that can only be exposed by testing the entire integrated system or some major part of it [2]. The research in this paper is related transaction-flow testing [2] and specification-based interface checking. A *transaction* is a unit of work from a 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 (TFG)* combines transactions to represent processing steps for one complete type of transaction. Conditional and unconditional branches can be used. Once the flowgraph has been built and analyzed (using standard inspection and walkthrough techniques), test cases are generated. Our approach improves upon transaction-flow testing by adding formality, repeatability, and precision.

2.1 Specification-Based Interface Checking

Large, complex systems are often designed by decomposing them into smaller segments and components that communicate with each other. Two standards by the U.S. Department of Defense (DoD), DoD-STD-2167A [16] and MIL-STD-498 [17], make it clear that mission critical DoD systems follow this model. As a result, a system will be composed of interfaces

with users as well as many interfaces between components. Parnas points out that there are difficult problems of interface design and interface enforcement [18]. Liu and Prywes describe a specification approach that uses a dataflow specification, interface specifications defined using 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 [13]. The IVT method also performs consistency checking of interface specifications, but does not require users to compose specifications using regular expressions or file definitions. It uses "informal" interface specifications found in an Interface Requirements Specification document.

2.2 Previous Work in Input Validation Testing

To date, the work performed in the area of input validation testing has largely focused on automatically generating programs to test compilers. Techniques have not been developed or automated to assist in static input syntax evaluation and test case generation. Thus there is a lack of formal, standard criteria, general purpose techniques, and tools. Previous research [1, 3, 5, 8, 12, 15, 19] test specific systems, are code-based for specific languages, or require extensive training by testers. The IVT approach does not require source code or formal specifications, and requires minimal input from users. It also entirely automates the production of tests and of static analysis.

More recently, Beizer [2] provides a practical discussion on input validation testing (called syntax testing). He proposes that the test engineer prepare a graph to describe each user command, and then generate test cases to cover this graph using coverage techniques such as all-edges. Marick [14] also presents a practical approach to syntax testing, suggesting a number of informal guidelines. A domain-based testing tool called Sleuth [20] assists in test case generation for command-based systems (CBS). 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.

3 The Input Validation Test Method

Input validation testing (IVT) focuses on the specified behavior of the system and uses a graph of the syntax of user commands. IVT incorporates formal rules in a test criterion that includes a measurement and stopping rule. Several grammar analysis techniques have been applied as part of the static analysis of the input specification. This section discusses the four major aspects of the IVT method: (1) how to specify the format of specifications, (2) how to analyze a user command specification, (3) how to generate valid test cases for a specification, and (4) and how to generate error test cases for a specification.

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

3.1 Specifying specification format

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

3.2 Analyzing user command specifications

A user command language specification defines the requirements that allow users to interact 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 [11]. There are three well accepted software quality criteria that apply to interface requirements specifications: completeness, consistency, and correctness [4]. Unfortunately, there have been no studies to determine what percent of problems can be attributed to each criterion. This research only addresses the first two.

Requirements are *complete* if and only if everything that eventual users need is specified [4]. The IVT method assesses the completeness of a user command language specification in two ways. First, the IVT

method checks that there are data values present for every column and row of the specification table. Second, the IVT method performs static analysis of the specification tables. The IVT method looks to see if there are hierarchical, recursive, or grammar production relationships between the table elements. For hierarchical and grammar production relationships, the IVT method checks to ensure that 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 users and having the potential to cause the end users 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" [4]. *Internal inconsistency* refers to conflicts between requirements in the same document. *External inconsistency* refers to conflicts between requirements in related interface documents. In addition to analyzing user command language specification tables, the IVT method also analyzes input/output (or data flow) tables. These tables (referred to as "Type 3" tables[1]) are found in interface requirements specifications and interface design documents and are often associated with data flow diagrams. These tables are expected to contain three fields: (1) data element, (2) data element source, and (3) data element destination.

Completeness and consistency are automatically checked by the IVT method. When problems are found, they are used to generate error reports when the requirements are obviously wrong, and test obligations when the requirements have potential problems. Detailed algorithms for these checks are provided in the technical report [9].

Davis defines correctness for requirements as existing "if and only if every requirement stated represents something that is required" [4]. 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. The IVT method does not address correctness of requirements.

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

1. Examine data elements that are adjacent to each other. If no delimiters are specified, the IVT method will look to see if two data elements of the same type or with the same expected value are adjacent. If so, a *test obligation* is generated to ensure that the two elements are not concatenated if a user "overtypes" one element and runs into the next element. The algorithm is shown in Figure 1. It looks at each record in the table of test objects and writes appropriate test cases to the test case table.

2. Check to see if a data element appears as the data type of another data element. If IVT detects such a case, it informs the user that the table elements are ambiguous and a test obligation is generated.

3. Check to see if the expected value is duplicated for different data elements. This is a potential poor interface design because users might type the wrong value. This situation is similar to when a grammar has overloaded token values. If IVT detects such a case, it informs the user that the table elements are potentially ambiguous and a test obligation is generated. The algorithm is shown in Figure 2. It looks at each pair of elements in the table of token elements, and if they are the same, writes appropriate test cases to the test case table.

[1] Although a "space" was left for Type 2 tables, it turned out that none were actually found in any requirements examined.

```
algorithm:  OverloadedTokenValue (CurTab)
input:      A table that contains token elements.
output:     Test obligations.
declare:    Element -- Record of tokens:  (element_num,
            element_name, position,
            class_of_values, size, class, value_num, value)
            i, j, c -- integer
            v -- value
            A -- array of values

OverloadedTokenValue (CurTab)
BEGIN -- Algorithm OverloadedTokenValue
   READ in CurTab
   c = 1
   FOREACH Element in CurTab DO
      v = GetValue (Element)
      A [c] = v
      c = c + 1
   END FOREACH

   FOR i = 1 TO Size (A)-1 DO
      FOR j = i+1 TO Size (A) DO
         IF (A[i] == A[j]) THEN
            Write error message and record
            to test oblication database
         ENDIF
      ENDFOR
   ENDFOR
END Algorithm OverloadedTokenValue
```

Figure 2: **The OverloadedTokenValue Algorithm**

```
algorithm:  CatenateStaticError (CurTab)
input:      A table of test object records.
output:     Test cases for catenation.
declare:    CurRec -- Record being processed.
            CurTab -- Table being processed.
            TestObRecord -- Record of test objects (ErrorCord, AmbCharNum, AmbigValue, CharNum).
            TestObTab -- Table of TestObRecord.

CatenateStaticError (CurTab)
BEGIN -- Algorithm CatenateStaticError
   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 cases and "Valid/Overloaded Token Static Error" to test case table
      ELSE IF (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
   END FOREACH
END Algorithm CatenateStaticError
```

Figure 1: **The CatenateStaticError Algorithm**

3.3 Generating valid test cases

The user command language specification is used to generate a covering set of test cases. The syntax graph of the command language is tested by adapting the all-edges testing criterion [21]. Each data element is represented as a node in the syntax graph. Many user command specifications yield loops in the syntax graphs, and the following heuristic is used [2, 14]: execute 0 times through the loop, execute 1 time through the loop, execute N times through the loop, and execute N+1 times through the loop, where N is a reasonably large number. If the loops is determinant (i.e., a `for` loop), then N can be determined from the maximum number of iterations, if not, then a reasonably large number can be chosen. (The current tool uses 10 as a default, but that can be easily changed.) The test cases are generated automatically by traversing the syntax graph. Figure 3 shows the **CoverTest-Cases** algorithm. It walks through each of the previously generated tables and generates actual values for the test cases.

3.4 Generating error test cases

There are two sources of rules for generating erroneous test cases: the error condition rule base, and the test obligation database. The error condition rule base is based on the Beizer [2] and Marick [14] lists of practical error cases. Unfortunately, space does not allow these tables to be included in this paper. The test obligation database is built during static analysis. Erroneous test cases are generated from both the error condition rule base and the test obligation database. Four types of error test cases are generated from the error condition rule base:

1. Violation of "looping" rules when generating covering test cases. For example, if the syntax of the interface states that the input must have 1 alphabetic character followed by 6 numeric characters, the implied loops are violated by creating tests with 0 and 7 numeric characters.
2. Top, intermediate, and field-level syntax errors. If the grammar for the input syntax has several levels of hierarchy, then terminal and non-terminal symbols are interchanged at the intermediate levels in the grammar hierarchy.
3. Delimiter errors. Error test cases are generated by inserting two delimiters into valid test cases in randomly selected locations.
4. Violation of expected values. Expected numeric values are replaced with alphabetic values, and expected alphabetic values are replaced with numbers.

Two types of error test cases are generated from the test obligation database. The first is for an 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. Second is for a 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

```
algorithm:   CoverTestCases (AllTables)
input:       All CurTab tables.
output:      Valid and invalid test cases to cover the input grammar.
declare:     Record -- Record of table tokens:  (table_name, element_num, element_name,
             position, class_of_values, size, class, value_num, value)
             CurTab -- Current table being processed
             CurRec -- Current record being examined
             V -- Name of a table
             W, CurValue -- Current record values
             i -- integer
             LoopHandler -- {Once, N, N_Plus_one, Zero}
             Expected_Outcome -- {Valid, Invalid}

CoverTestCases (AllTables)
BEGIN -- Algorithm CoverTestCases
  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, N, and N + 1 times test cases
                CASE Loop_Handler OF
                Once:  -- 1 time through loop
                    -- Select_Value selects a value from the class of values
                    W = Select_Value (CurRec.Value)
                    Write W to MS Word file and test case table
                    -- Size_Check returns Valid if i =< CurRec.Size, Invalid otherwise
                    Expected_Outcome = Size_Check(I)
                    i = i + 1
                N: -- N times through the loop
                    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
                N_Plus_One:  -- N + 1 times through the loop
                    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
```

Figure 3: **The CoverTestCases Algorithm**

```
                    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 Algorithm CoverTestCases
```

Figure 3: **The CoverTestCases Algorithm – continued**

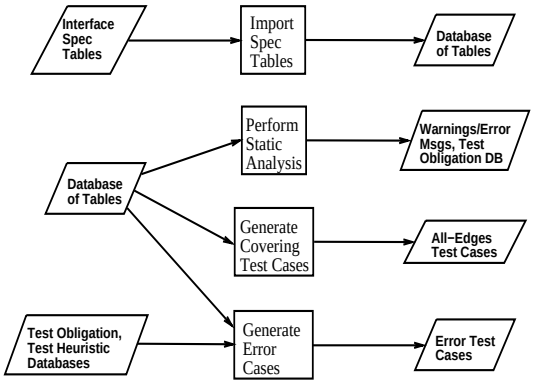

Figure 4: MICASA Architecture

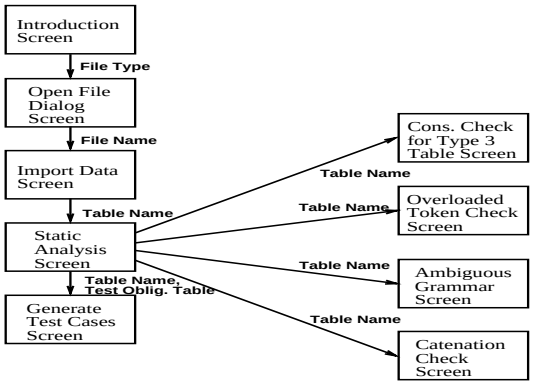

Figure 5: MICASA Screen Flow

data type, no expected values, and no delimiters) are duplicated into the adjacent fields.

4 MICASA: A Proof-of-concept System

To demonstrate the effectiveness of IVT, a proof-of-concept system was developed. This tool accepts input specifications, performs the analyses described in Section 3, and automatically generates system-level tests. The tool is called Method for Input Cases and Static Analysis (MICASA).

MICASA runs under Windows NT. It is written in about 6500 lines of Visual C++ and relies on MS Access tables and databases. A high level architecture is shown in Figure 4. MICASA has four major subsystems, shown in square boxes in the middle. The Import Specification Tables subsystem accepts the interface specifications, and translates them to a standardized, intermediate form. This Database of Tables is then fed to the other three subsystems, which generate messages about the input specifications, test obligations, and test cases. The Test Heuristic Database is encoded directly into the MICASA algorithms. During use, MICASA took between 5 and 30 seconds per test case to execute.

MICASA uses a small graphical user interface. Most screens have four buttons and offer the options `Cancel`, `Back`, `Next`, and `Finish`. Most interaction by users is with radio buttons. MICASA leads users sequentially through the steps of the IVT method. This sequence of screens is illustrated in Figure 5. The steps are primarily sequential, a user enters the name of a file that contains the input specifications, the data is read, and then static analysis is performed. Users can select which types of static analysis to perform, then test cases are generated.

4.1 Import Specification Tables

The Import Spec Tables function allows users to import information from digital interface specifications. MICASA accepts flat files and MS Word files that describe the interface specification tables. This function performs three major processing steps. First, the introduction screen asks the users if the table is type 1 or 3 (defined in Section 3). If a type 3 table, MICASA asks if consistency checking is to be performed on an entire Computer Software Configuration Item. The second step is to get the name of the file(s) to be imported from the user. Third, 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.

4.2 Perform Static Analysis

The Perform Static Analysis function allows users 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 stored in the MS Access database. There are five processing steps performed by this function: (1) users can perform static analysis, (2) when the main table is created, the users can initiate a consistency check on the table, (3) users can check for overloaded tokens, (4) users can check for ambiguous grammar, and (5) users can check for possible catenation errors. The output from static analysis is a set of MS Access database tables containing error records, as well as printouts of these error reports.

4.3 Generate Covering Test Cases

Users can generate all-edges test cases for the Type 1 interface tables stored in MS Access. The input is the interface table information stored in the MS Access database. This function automatically generates test cases to satisfy the all-edges criterion on the syntax graph. Users enter the document and system name. The output 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 an MS Word template.

4.4 Generate Error Test Cases

Users can generate error test cases for the Type 1 interface tables stored in MS Access. The input is the interface table information stored in the MS Access database, the test obligation database generated during static analysis, and the test case heuristics. An error test case is generated for each test obligation in the test obligation database. Next, the Beizer and Marick heuristics are used to generate error cases, as described in Section 3. This function is automatically performed after Generate Covering Test Cases. Users are shown the number of test cases that have already been generated (under Generate Covering Test Cases function), and users are given 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 outputs from this function are additions to the MS Access database tables containing test cases. Again, these can be displayed in MS Access or can be formatted as Test Plans using an MS Word template.

5 Empirical Validation

This section presents empirical results that demonstrate the feasibility, practicality, and effectiveness of the IVT method. Real-world, industry applications were used in a multi-subject experiment to compare the IVT method with human subjects. The experimental design and the experimental subjects are described, then specific results are presented.

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. Eight senior testers were used, each performing the same activities as the MICASA tool. The experiment was divided into three steps: (1) perform analysis of the specifications, (2) generate test cases for the specifications, and (3) execute the test cases.

Volunteers were used for the experiment, and many dropped out or did not complete the experiment. All but one tester worked with the first author, but the experiment was **not** part of their work duties. Most of them already had at least a passing familiarity with the software that was used. The experienced testers were defined as having at least seven years of software development/information technology experience and at least three years of testing experience. Neither of the authors participated in the experiment.

Existing software subsystems of the Tomahawk Cruise missile mission planning system were used. Tomahawk Planning System (TPS) is comprised of roughly 650,000 lines of code running on HP TAC-4s under Unix. TPS was developed by Boeing and primarily written in 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. Digital Imagery Workstation Suite (DIWS) runs under DEC VMS and consists of over 1 million lines of Ada code, with a small amount of Assembler and FORTRAN. Some code runs in multiple microprocessors. DIWS was developed by General Dynamics Electronics Division. Precision Targeting Workstation (PTW) is hosted on TAC-4 workstations, runs under Unix, and is written in C. The system is roughly 43,000 lines of code, and was developed by General Dynamics Electronics Division.

The design for Part I of the experiment consisted of four testers manually analyzing interface specification tables with one tester using the MICASA pro-

totype 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 TPS to 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 PTW from the 3900/113 document (U.S. Navy) (System C).

The TPS-DIWS specification was "overlapping", (that is, all five testers were asked to analyze this document), to allow for examination of possible "skill level" ambiguities of the various testers. By having all five 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 B8 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 B8, but their reviews gave the researcher a good idea of how many defects existed in each table.

The design for Part II of the experiment consisted of six testers manually generating test cases for interface specification tables with one 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; and a specification table (3.2.4.2-1) for the Precision Targeting Workstation from the 3900/113 document (U.S. Navy). The TPS-DIWS document was overlapping (that is, all seven testers were asked to use this document) to allow for examination of possible skill level ambiguities of the various testers. One tester was not able to complete test cases for this document, however. It took the testers a few hours to learn to use MICASA, and from 7 to 130 minutes to develop each test case.

The design for Part III of the experiment consisted of one 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.

5.1 Results and Discussion

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

Table 1: **Empirical Results**

	MICASA	**Testers**
Syntax spec. defects found	524	21
Total spec. defects found	524	106
Number of test cases	48	7
Software faults found	20	27
Defect detection rate	7.4	4.6
Minutes per fault found	8.4	72.2

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

An interesting observation has to do with the quality of the specification tables. For part I of the ex-

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

6 Conclusions

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

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

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

References

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

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

[3] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–345, 1983.

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

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

[6] Institute for Defense Analysis. Analysis of software obsolescence in the DoD: Progress report. IDA Report M-326, Institute for Defense Analyses, February 1987.

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

[8] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, (4):242–257, 1970.

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

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

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

[12] D. C. Ince. The automatic generation of test data. *The Computer Journal*, 30(1):63–69, February 1987.

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

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

[15] Peter M. Maurer. Generating testing data with enhanced context-free grammars. *IEEE Software*, 7(4), July 1990.

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

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

[18] D. L. Parnas. Letters to the editors. *American Scientists*, 74:12–15, January-February 1986.

[19] P. Purdom. A sentence generator for testing parsers. *BIT*, 12:366–375, July 1972.

[20] A. von Mayrhauser, J. Walls, and R. Mraz. Sleuth: A domain based testing tool. In *Proceedings of the IEEE International Test Conference*, pages 840–849, 1994.

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