1999

Test suite development using a structured framework

Nelson Eugene Hastings
Iowa State University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Test suite development using a structured framework

by

Nelson Eugene Hastings

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Major Professors: James A. Davis and Doug W. Jacobson

Iowa State University

Ames, Iowa

1999

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Nelson Eugene Hastings

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Committee Member

Signature was redacted for privacy.

Committee Member

Signature was redacted for privacy.

Committee Member

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

For the Major Program

Signature was redacted for privacy.

For the Graduate College

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND NOMENCLATURE

API Application Programming Interface

ASN.1 Abstract Syntax Notation One

B(x) Behavior Of x

BER Basic Encoding Rules

C(x) Set Of Abstract Components Defined By Specification x

$C_I(z, x)$ Set Of Components Defined By Specification x Instantiated By z

CA Certificate Authority

CE Cause–Effect

CH Certificate Holder

COM Common Object Model

CRL Certificate Revocation List

DER Distinguished Encoding Rules

DLL Dynamically Linked Library

DNF Disjunctive Normal Form

DTR Derived Test Requirements

dom(x) Domain Of Function x

DSS Digital Signature Standard

ECDSA Elliptic Curve Digital Signature Algorithm

EQ Equilateral Triangle

F_x Set Of Formats For x

FIPS Federal Information Processing Standard

GUI Graphical User Interface

I(x) Set Of Abstract Inputs Defined By Specification x

i An Abstract Input

$I_I(x)$ Set Of Instantiated Inputs For Implementation x

i' An Instantiated Input

IF_x Set Of Invalid Formats For x

IS_x General Input Space Defined By Specification x

ISO Isosceles Triangle

ISS_x Invalid Input Space Defined By Specification x

IUT Implementation Under Test

LDAP Lightweight Directory Access Protocol

m An Abstract Message

m' An Instantiation Of A Message

$M_G(m)$ Set Of Abstract Components That Can Generate Abstract Message m

$M_P(m)$ Set Of Abstract Components That Can Process Abstract Message m

$M_{I_G}(m)$ Set Of Instantiated Components That Can Generate An Instance Of Message m

$M_{I_P}(m)$ Set Of Instantiated Components That Can Process An Instance Of Message m

MAC Message Authentication Code

MFC Microsoft Foundation Classes

MIME Multipurpose Internet Mail Exchange

MISPC Minimum Interoperability Specification For PKI Components

N Set Of Natural Numbers

O(x) Set Of Abstract Outputs Defined By Specification x

o An Abstract Output

$O_I(x)$ Set of Instantiated Outputs For Implementation x

o' An Instantiated Output

OLE Object Linking And Embedding

OOB Out Of Band

ORA Organizational Registration Authority

P(x) Power Set Of The Set x

$P_{Pair}(y, x)$ Set Of Abstract Input-Output Pairs For Transaction y With Respect To Component x

$P_{I_{Pair}}(y, x)$ Set Of Instantiated Input-Output Pairs For Transaction y With Respect To Component x

PKCS Public Key Cryptographic Standard

PKI Public Key Infrastructure

PVE Path Validation Engine

PVIS Power Set Of Valid Input Space

REP Repository

S(x) Abstract Test Suite With Respect To Specification x

$S_R(x)$ Required Abstract Test Suite With Respect To Specification x

$S_O(x)$ Optional Abstract Test Suite With Respect To Specification x

T(x) Set Of Abstract Transactions Defined By Specification x

T(x,c) Set Of Abstract Transactions Defined By Specification x For Abstract Component c

$T_R(x, c)$ Set Of Required Abstract Transactions Defined By Specification x For Abstract Component c

$T_O(x, c)$ Set Of Optional Abstract Transactions Defined By Specification x For Abstract Component c

$T_R(x)$ Set Of Required Abstract Transactions Defined By Specification x

$T_O(x)$ Set Of Optional Abstract Transactions Defined By Specification x

$T_M(x)$ Set Of Abstract Messages Of Transaction x

$T_{I_M}(x)$ Set Of Instantiated Messages Of Transaction x

SCA Scalene Triangle

SHA Secure Hash Algorithm

TCP Transmission Control Protocol

TS Set Of Test Strategies

TT Set Of Testing Templates

TTF Test Template Framework

VDM Vienna Development Method

VF_x Set Of Valid Formats For x

VIS Valid Input Space

VIS_x Valid Input Space Defined By Specification x

ACKNOWLEDGEMENTS

I would like to thank the many people in at Iowa State University and the National Institute of Standards and Technology. I am deeply indebted to Dr. James A. Davis and Dr. Doug W. Jacobson for being my thesis committee co-chairman and major advisors for my stay at Iowa State University. A special thanks to Dr. Jonnie Wong, Dr. Albert Baker, and Dr. Charles Wright for serving as thesis committee members. I would like to recognize the entire faculty, graduate students, and staff of the Electrical and Computer Engineering Department for making my graduate studies at ISU challenging and interesting.

I am very grateful to Stuart Katzke, Miles Smid, and Donna Dodson for giving me the opportunity to complete my research while working at NIST. A special thanks to the staff of the Computer Security Division for keeping me focused on the prize. I would like to specifically recognize Tim Polk, Dr. David Cooper, Kathy Lyons-Burke, Noel Nazario, William Burr, and San Vo. A very special thanks to Anabelle Lee for reviewing my dissertation and providing valuable guidance and comments.

I would like to thank all of the members of my family. A special thanks to my mother whose support and love have always been there for me throughout my life. A highly emotional thanks to my spouse, Amy Ackerberg-Hastings, for her understanding, support, and love during these last three years. You are next!!! Finally, I dedicate this work to the memory of my step-father, Dr. James R. Vogt, who taught me more than I ever realized.

ABSTRACT

The rationale for conducting any type of testing is ultimately to draw reasonable conclusions about the subject based on the results of the test or tests performed. Given a new piece of software, a software engineer may wish to determine the software's ability to handle inputs outside its specified domain, the software's performance on a given computing platform, or the software's ability to interact with other similar pieces of software. The testing performed by software engineers and analysis of the test results is the general area known as software testing.

In this thesis, a newly-proposed structured framework for test suite development is introduced to capture the interaction between the applications being tested, to investigate the use of an application's invalid input space for the generation of test cases, and to explore the notation that test suites can be expressed on two levels, abstractly and concretely via instantiation. In addition, the proposed structured framework is applied to the Minimum Interoperability Specification for Public Key Infrastructure (PKI) Components (MISPC) standard for the development of an abstract test suite. A part of the abstract test suite was instantiated, or implemented, and executed against a reference implementation of an MISPC specified Certificate Authority (CA) to explore the proposed structured frameworks capabilities. The result of this thesis demonstrate the limitation of structured test suite development frameworks that do not utilize an application's invalid input space for test case generation and the benefits of being able to express test suites at both abstract and concrete levels. In addition, the instantiated test suite revealed the MISPC CA reference implementation could not process a few valid MISPC messages and generated some invalid MISPC messages of its own.

1 INTRODUCTION

An Overview of Software Testing

The rationale for conducting any type of testing is ultimately to draw reasonable conclusions about the subject based on the results of the test(s) performed. The context of the test(s) will determine the scope, quality, and validity of the conclusions that can be drawn from the test result(s). For example, if the subject to be tested is a newly discovered material, the materials engineer will test the material to determine its physical properties of interest: corrosiveness, hardness, brittleness, conductivity, etc. Given a sample of the new material, the materials engineer may want to know its tensile strength. The tensile strength of a material determines its ability to withstand stress without failure (breaking). The materials engineer may perform a series of tensile strength tests and conclude that the new material's tensile strength is X, but what does this tensile strength value really mean? Some context needs to be given for the tensile strength value of X in order to determine its scope, quality and validity. Were the test samples drawn from the same lot of the new material? If so, this value may be good only for that specific lot of the new material and the way that lot of material was produced. If not, the tensile strength value of X may be good for the new material in general. Another element that may give context to the tensile strength value are the climatic conditions (temperature, pressure, humidity) under which the tests were performed. If the climatic conditions were the same for all tests, then the value of X would only be a good value under those specific climatic conditions. If the tests were performed under different climatic conditions, then the value of X would hold true over a range of climatic conditions. Based on analysis of the test results and their context, the material engineer may draw the conclusion that the material's tensile strength of X would be a useful material for constructing an airframe's superstructures. Just as new materials can be the subject of test(s), so may new computer applications or software. The computer or software engineer tests a piece of software to determine its execution and logical properties of interest: utilization of resources, tolerance to errors, interoperability with other applications, functionality provided by the application, etc. Given a new piece of software, the software engineer may wish to determine the application's ability to handle erroneous input. After

a series of tests, the software engineer may determine that the software handles about half the errors that are presented. What is the meaning of this result? What types of erroneous inputs were created? How were these inputs presented to the software? What was the configuration of the host system that the software was executed on? Once these questions are answered, the result will have some context in which it can be viewed. The input may have been a string of random bits or ASCII bytes. The input may have been read in from a file or keyed in from the keyboard. The host system may have had an 8088 processor with 16KB of RAM running DOS or it could have had a Pentium Pro processor with 128M of RAM running Windows 98. The software engineer may draw the conclusion that the application may be acceptable for a hobbyist that uses it for recreation, but not for large corporation that would use it to conduct multi-million dollar transactions. The testing performed and analysis of results by the software engineer is the general subject known as software testing.

The fundamental limits and theoretical foundations of software testing need to be understood before any type of software test can be designed, implemented, and executed. Ideally, all possible inputs and the execution of every statement of the source code would be used to exhaustively test whether or not an application performs correctly. The main limitation to the exhaustive testing approach is the time required to perform such tests. For example, if an application accepts a string of 10 characters and using a very simple character set consisting of just lower case letters, the number of different possible inputs is 26^{10} or about 1.4116709×10^{14}. If one input can be tested every 1 nanosecond, that means it would take 4.4763792×10^{15} years to perform an exhaustive test of the inputs. To put this figure into perspective, the estimated age of the universe is about 10^{11} years [36]. However even if the time limitation of exhaustive tests could be overcome, that still would not guarantee that all errors of the application would be revealed [18]. The pseudo-code shown in Figure 1.1 demonstrates that all the statements could be exercised without revealing the error. In the example, there exists a conceptual error that adding three numbers together and dividing the result by three will guarantee that the original numbers were equal. The example demonstrates one of the most fundamental ideas in software testing which is that tests only show the presence of errors but not the absence of them [13]. Because exhaustive testing is impractical and practical tests only show the presence of errors, a framework to quantify the relative power of tests and testing strategies to each other becomes an important topic for software testing. In [18], a framework was developed for determining the relative power of tests and test strategies after challenging the idea of the use of correctness proofs. The practicality of deriving a valid correctness proof is shown to be very difficult, if not impossible, because of the many assumptions required. Some of the assumptions required for a valid correctness proof include: the existence of a

```
IF ((X + Y + Z) / 3 = X)
    THEN PRINT ("X, Y, and Z are Equal")
    ELSE PRINT ("X, Y, and Z are Not Equal")
```

Figure 1.1 Pseudo-Code Example.

correct specification, incorporation of the complete runtime environment incorporated into the proof, and the actual proof derivation must be error free. The proposed framework is based on the definitions of a test being complete, reliable, and valid. A test is said to be complete, if by successfully passing a specific test, one can imply that the tested software contains no errors sought by the test. The ability of a test to reveal errors is defined as its reliability while the validity of a test is its ability to produce meaningful results. The proposed framework uses these definitions to evaluate a test's capability to establish a program's correctness. However, in [24], this proposed framework was proven to be infeasible because no computable procedure exists for establishing a program's correctness and a new framework is proposed based on determining a test strategy's (particularly the path testing strategy) reliability. The idea of correctness proofs is revisited in [17] which tries to simplify the proof by incorporating the results of tests, but this framework turns out to be informal and becomes difficult to apply in a more general framework. [19] takes a different approach to determining the relative power of tests and test strategies by exploring a framework that uses a specification as the basis for evaluating tests. A formal proof is provided that shows the set of programs reliably tested using a specification independent test is empty. This limitation of specification independent testing demonstrates the need for a specification when evaluating the relative power of tests.

In general, a test suite is used to perform a test or collection of tests on a computer application or a piece of software. The test suite's testing strategy is the method used to select and possibly generate the tests and test data to be used to investigate the piece of software [4, 5, 28]. The following three testing strategies are generally used in software testing: black-box testing, white-box testing, and hybrid testing. The black-box testing strategies are used to derive tests based on the specification of the software. The white-box testing strategies are used to derive tests based on the structure of the software or the way it is implemented. A hybrid testing strategy uses a combination of both black-box and white-box testing strategies. These testing strategies can be combined and applied to develop a test suite for a given piece of software. However, the way strategies are utilized and their

effectiveness depends on several factors about the software being tested as well as the objectives of the test designer. What is the structure of the software? Is it simply a single stand-alone function or a complex collection of modules that interact with each other? What interfaces to the software does the tester have access to during the testing? Is it the software's application program interface (API) or a graphic user interface (GUI)? What type of code does the tester have access to when designing the tests? Is it the software's source or executable code? What does the test designer want to demonstrate about the software? Does the software interoperate with other software? How does the software perform under erroneous conditions? These are a few of the factors that determine the testing strategies chosen for the development of an effective test suite.

The first step in the development of an effective test suite is to determine the amount of information the test suite designer will have about the software being tested. At a minimum, the test designer must have information about the desired functional behavior of the software to be tested. However, the test designer may have more detailed information such as how the software should be structured. In either case, an informal, formal, or hybrid[1] specification is used to convey this information to the test designer. An informal specification uses a natural language, such as English, to describe the desired properties of the software while a formal specification uses a more rigorous notation, such as mathematical symbols or a specification language (such as Z). Each type of specification has its advantages and disadvantages. An informal specification is generally easier to understand but is generally more ambiguous than a formal specification. A formal specification is generally less ambiguous than an informal specification but is more difficult to understand. A formal specification may allow for properties of the software to be derived by a mathematical proof which an informal specification cannot achieve. Whatever form a specification takes, it is required in the development of a test suite to provide a basis for comparison of the software being tested. Without a specification, a specific test suite would have no context and any results obtained would be meaningless. A formal proof of the need for a specification in software testing is presented by Gourlay in [20].

The development of a test suite continues by determining what can be tested and how the test can be implemented given a specific piece of software. Determining the amount and type of access the test suite is given to the software to be tested is one important factor. This will determine how much of the specification given to the test designer can be verified. If the test designer is not given access to the internal structure or source code of the software, then no matter what the specification says about the structure of the software, it cannot be verified. Not allowing the test designer access to the internal

[1] A hybrid specification consists of both informal and formal specifications.

structure or source code of the software forces the designer to develop a test suite based on functional or black-box testing strategies [4, 5, 28]. However, if permission is given to the test designer to access the internal structure or source code of the software, then the test designer can use the extra information to develop a test suite based on structural or white-box testing strategies. This additional access allows white-box tests to determine more precisely how much of the software's source code is exercised as compared to black-box tests. After a level of access has been determined, the test designer can then evaluate how testable the software is based on what can be observed and controlled by a test suite [16]. The software must provide at least one mechanism for the test suite to observe or gather information about its behavior and output, otherwise no data can be collected for analysis. Likewise, the software must provide the test suite the ability to control or stimulate it to produce an observable behavior. The observe-ability and control-ability of the software will affect what the test suite will be able to test, how the test is performed, and the effectiveness and quality of the test.

Once a specification and the testability of the software have been evaluated, the test designer can begin to develop objectives that a specific test suite will be able to realize which are in line with his own or his employer's motives, goals, and perspectives. Test suites can be classified into three non-mutually exclusive groups based on the objectives and goals of the test designer. Conformance or requirements testing determines a software application's ability to provide specific functionalities or behaviors that are defined by a specification. In addition, the specification may also dictate how the software should implement the specified functionalities or behaviors. A test designer who works for a company that certifies that vendor applications follow a given specification is motivated to design effective conformance tests, so that the company's reputation of quality evaluations is maintained. Assurance or reliability testing determines the confidence a user can have about the operation of the application software under normal and adverse conditions. A test designer who works for an application vendor is motivated to design assurance tests, so the company's reputation of delivering quality software applications is maintained. An assurance test suite should, at a minimum, ensure that the application provides all the functionality or behaviors claimed, operates without errors or failure when executed by a user under both normal and abnormal operating conditions, and interfaces with other software as required. Finally, interoperability testing evaluates an applications ability to interact with other software to produce meaningful behaviors. For interoperability testing to be useful, there must exist some relationship between the applications that interact with each other. A test designer who works for a company that produces an e-mail server is motivated to design interoperability tests that demonstrate its server's ability to interact with e-mail clients developed by many different vendors, so the company's

reputation of providing software compatibility is strengthened.

One of the final aspects the test designer must consider is how the test suite will be implemented. The implementation of a test suite will ultimately determine the scope of what can be tested and the value of the test results. One question to be answered is how the test suite will exercise an application to be tested. In general, static or dynamic testing techniques are used to exercise the software application. Static testing techniques are applied to an applications source code without actually executing it. Static testing is an effective technique when the level of access to an application allows for structural or white–box test suite development. However, static testing is not effective when the level of access to an application only provides for the functional or black–box test suite development. Dynamic testing techniques allow a test suite to actually execute the application while it is being tested. Dynamic testing techniques are the way to test an application when the test suite is developed from black–box testing strategies. If the appropriate level of access to an application is granted, then both static and dynamic testing techniques can be used when implementing a test suite. Another important implementation question to be answered is how the test suite and application to be tested will be configured relative to each other. In Figure 1.2, three ways of configuring an application to be tested, or the Implementation Under Test (IUT), and a test suite are shown. Figure 1.2(a) shows a test suite designed to interact only with the high level (user) interfaces of the implementation under test (IUT) and a reference implementation. This configuration simulates users of the IUT as well as the users of the reference implementation and assumes that both the component connectivity and reference implementation are properly implemented. If the IUT is implemented correctly, it should be able to interact with the reference implementation which demonstrates the configuration's effectiveness for interoperability test suites [22]. The configuration shown in Figure 1.2(b) eliminates the presence of a reference implementation. In this configuration, the test suite once again interacts with the high level (user) interfaces of the IUT, but directly interacts with the component connectivity. This configuration simulates the users of the IUT as well as a complete reference implementation and assumes that only the component connectivity is properly implemented. In the final configuration shown in Figure 1.2(c), all extra implementation assumptions are removed by eliminating reliance on both a reference implementation and component connectivity. In this configuration, the test suite again interacts with the high level (user) interfaces of the IUT, but also its low level (system) interfaces. This last configuration may not be good for interoperability test suites but may be adequate for conformance test suites. After the configuration of the test suite and IUT is determined, the test designer has some knowledge about how the test suite and IUT will interact with each other. Now, the test designer must determine how the test suite will

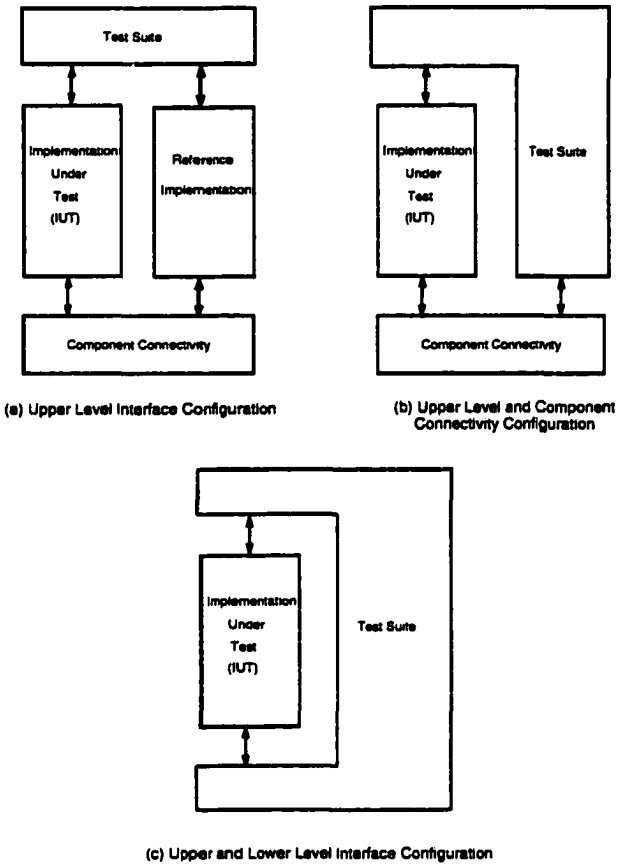

Figure 1.2 Test Suite Configurations.

generate test cases (data) and evaluate the results of the IUT's behavior. Different techniques with varying degrees of effectiveness are used in the generation of test cases to stimulate an application to produce an expected behavior. The test cases along with their mapping to the expected behavior of the application can be housed in an oracle that can be used by the test suite to compare an application's actual versus expected behavior. The topic of test case generation will be explored more fully throughout the rest of this thesis.

The Structured Development of Test Suites

The structured development of a test suite simply put is the development of a test or set of tests using various testing strategies and techniques in an organized or structured fashion. The way the various testing strategies and techniques are organized provides a structured framework for the development of

a test suite. In addition to organization, a structured framework gives the test designer a perspective on how the different testing strategies and techniques relate to each other and the effect they have on the development of the test suite. In general, every test suite developed requires some level of organization and structure, so that its results will have meaning. However, the level of organization and structure varies between different test suites in the absence of a consistent structured framework. Also, the level of organization and structure varies due to the lack of a framework that unifies different testing strategies and the effectiveness of different test design techniques. A structured framework provides a way to consistently unify the different testing strategies under a single design technique.

Structured test suite development research has mainly been in the form of frameworks used in the derivation of test data for test suites. The research has focused on specification based frameworks because of the results in [19, 20] which validate the use of specifications in test suite development. Specification based frameworks use the specification of an application when deriving test data for the test suite [3]. Specification based frameworks have the advantage of deriving test data for a test suite independent of the way an application is implemented which is useful when a specific implementation of an application is not available. However, when a specific implementation of an application is available, the specification based framework has the drawback of not easily integrating test data derived from the specific implementation into the framework. For example, a specification may state that variable X can have any integer value, but a specific implementation may have a maximum limit on the value for variable X. As a result, a structured framework should integrate the use of both specification based frameworks as well as non-specification based frameworks when possible. Non-specification based frameworks derive test data based on the way an application is implemented. Investigation of specification based frameworks is useful because many times test suites for an application as well as the application itself are developed in parallel and a test suite designer does not have access to specifics about an application's final implementation.

Specification based structured framework research has focused on the use of formal specifications which can be divided into either algebraic or model based specifications. Algebraic based specifications describe applications in terms of an application's operations while model based specifications use states and state transitions to describe an application. The research using algebraic specification has been driven by protocol testing, specifically, trace analysis and has been extensively investigated in contrast to the research using model based specifications [39]. Model based specification research has focused on test strategies called partition testing which divide the domain (or input space) of an application into equivalent domains. The set of inputs that can stimulate an application to produce the same

reaction or result creates an equivalent domain. In partition testing, if one element of an equivalent domain results in the application passing a specific test, it is expected (perhaps incorrectly) that all elements within the equivalent domain will allow the application to pass the test. The partition testing research using model based specifications can be grouped based on the technique used to define the specification, such as Z or Vienna Development Method (VDM). In 1988, Hall used the predicates of Z specification operations to create simple partitions of the input space [21]. In another study that used a VDM specification, Dick and Faivre reduced an input expression into a disjunctive normal form (DNF) to partition an input space by the disjunctions [14]. These and other important works explore how to apply partition testing techniques in a structured framework, but did not result in a unifying framework for expressing the relationship of the partitioning sub–strategies. To address the lack of a unifying framework, Stocks proposed the Test Template Framework in [39] to fill this void.

The Test Template Framework: An Example of a Specification–Based Framework for Structured Test Suite Development

The Test Template Framework (TTF) was introduced to provide a structured framework for test suite development using model based specifications [39, 38]. The application of TTF is independent of the model based specification and of the testing strategies applied. TTF requires that a valid input space (VIS) be specified which is usually obtained directly from the specification. After a VIS has been defined, testing strategies (TS) are systematically applied to partition the VIS. A Test Template Hierarchy (TTH) is created using a function that maps the VIS and a given testing strategy to a set of test templates (TT). The TTH generation function can be expressed as $TTH : PVIS \times TS \rightarrow TT$. The PVIS is the power set of the VIS and is used because the TTH function can operate on any subset of the VIS to allow for recursive application of the operation. TS is the set of possible testing strategies. Because the TTH function further partitions the PVIS, TT is the power set of PVIS (TT=P(PVIS)) which allows the expression to be rewritten as $TTH : PVIS \times TS \rightarrow P(PVIS)$.

For example, TTH(VIS, abc) will take the VIS and partition it into a set of test templates based on the *abc* testing strategy. Let's say the *abc* testing strategy results in partitioning the VIS into VIS_P1, VIS_P2, and VIS_P3. The TTH function can be applied to one of the sub–partitions using a different testing strategy to create further partitions. For example, TTH(VIS_P2, xyz) results in VIS_P2 being partitioned into VIS_P2_S1 and VIS_P2_S2 using the *xyz* testing strategy. A graphical representation of the resulting TTH is shown in Figure 1.3. The resulting abstract test templates are the leaves of the

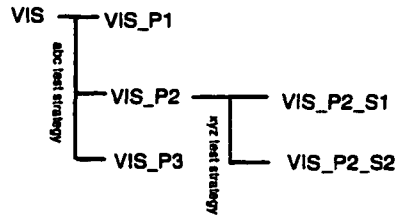

Figure 1.3 Test Template Hierarchy Example.

graph which can be used to generate test data for the particular partition. Properties (such as coverage, partition validity, partition equivalence, etc.) of the test templates can be analyzed under a structured framework. In addition, the effectiveness of the testing strategies can be compared under a common structured framework. The analysis of test template properties and their relative effectiveness will not be explored further in this project because the usage of the framework is most relevant at this point in the project.

A more concrete example will now be presented to demonstrate the application of TTF to a specification in order to derive instances of test data for the test suites. Our example will be a modified and shortened version of an example presented in [39]. A program is to take three natural numbers as inputs and determine if these values can be the lengths of the sides of a triangle. If the values can be the lengths of the sides of a triangle, the program should determine what type of triangle it could be: equilateral, isosceles (but not equilateral), or scalene (but not isosceles). The following function is used to specify the desired program behavior:

$$triangle(x,y,z) = \begin{cases} EQ & \text{if } (x < y+z \wedge y < x+z \wedge z < x+y) \wedge (x = z \wedge x = y \wedge x \neq 0) \\ ISO & \text{if } (x < y+z \wedge y < x+z \wedge z < x+y) \wedge [(x \neq 0 \wedge y = z \wedge x \neq y) \vee \\ & \quad (y \neq 0 \wedge x = z \wedge y \neq z) \vee (z \neq 0 \wedge x = y \wedge z \neq x)] \\ SCA & \text{if } (x < y+z \wedge y < x+z \wedge z < x+y) \wedge (x \neq y \wedge y \neq z \wedge z \neq x) \\ INVALID & \text{if } \neg(x < y+z \wedge y < x+z \wedge z < x+y) \\ & \\ & \text{where x, y, and z are natural numbers} \end{cases}$$

In the function, the binary operations of AND, OR, and NOT are represented using $\wedge, \vee$, and $\neg$, respectively. The domain of triangle() or its VIS as defined in the *where* clause of the function definition is the natural numbers, N. A function which generates the domain of a function, f(), is now defined as

dom(f) and will be useful later in this discussion. To generate a first partitioning of dom(triangle), a testing strategy needs to be selected. For this example, let's begin by using a cause-effect (CE) testing strategy to derive the first partitioning of the hierarchy. The cause-effect testing strategy maps effects (outputs) to causes (inputs). First, all specified outputs are determined, then the sub-domain of the VIS which causes a specific output is determined. We can now use dom(triangle), $D_{triangle}$, and the cause-effect strategy as parameters to the TTH generation operator: TTH($D_{triangle}$,cause-effect). From the function definition, the specified outputs are the abstract values representing invalid (INVALID), a scalene triangle (SCA), an isosceles triangle (ISO), and an equilateral triangle (EQ). The effects are caused by the *if* clauses in the function definition which determine which inputs (the (x,y,z) triples) will generate the given output. The original function triangle() is now respecified as four independent functions based on the causes generated by the input domains.

$$triangle_{EQ}(x,y,z) = \left\{ \; EQ \quad \text{where } (x < y+z \wedge y < x+z \wedge z < x+y) \wedge (x = z \wedge x = y \wedge x \neq 0) \right.$$

$$triangle_{ISO}(x,y,z) = \left\{ \; ISO \quad \text{where } (x < y+z \wedge y < x+z \wedge z < x+y) \wedge [(x \neq 0 \wedge y = z \wedge x \neq y) \vee (y \neq 0 \wedge x = z \wedge y \neq z) \vee (z \neq 0 \wedge x = y \wedge z \neq x)] \right.$$

$$triangle_{SCA}(x,y,z) = \left\{ \; SCA \quad \text{where } (x < y+z \wedge y < x+z \wedge z < x+y) \wedge (x \neq y \wedge y \neq z \wedge z \neq x) \right.$$

$$triangle_{INVALID}(x,y,z) = \left\{ \; INVALID \quad \text{where } \neg(x < y+z \wedge y < x+z \wedge z < x+y) \right.$$

Now, the domains of the four functions are extracted: $CE_{EQ} = dom(triangle_{EQ}), CE_{ISO} = dom(triangle_{ISO}), CE_{SCA} = dom(triangle_{SCA})$, and $CE_{INVALID} = dom(triangle_{INVALID})$. These represent the first level test templates of the TTH. This is represented by the following expression: TTH($D_{triangle}$, cause-effect)=$\{CE_{EQ}, CE_{ISO}, CE_{SCA}, CE_{INVALID}\}$. Figure 1.4 shows a graphical representation of the resulting test template hierarchy. Another application of the TTH generating operation will be done on the CE_{ISO} test template using a different testing strategy to demonstrate how one can further partition the VIS. This time we will use the disjunctive normal form (DNF) partitioning. DNF is used to reduce a statement into a combination of disjoint statements. For example, the

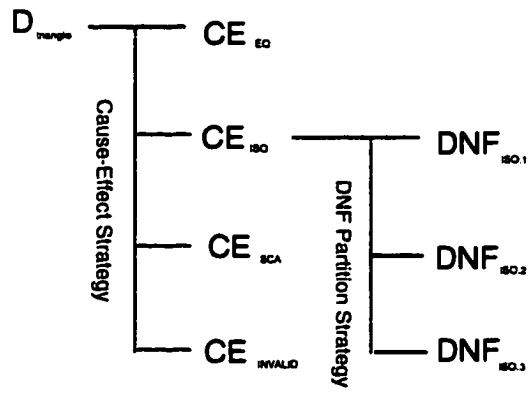

Figure 1.4 Triangle Test Template Hierarchy.

DNF of $(a \vee b) \wedge c$ would be $(a \wedge c) \vee (b \wedge c)$. As the example demonstrates, the Boolean OR operation is used to concatenate the disjoint statements. Applying the DNF partitioning to CE_{ISO} results in the following three test templates.

$$DNF_{ISO.1} = [CE_{ISO} | (x \neq 0) \wedge (y = z) \wedge (x \neq y)]$$

$$DNF_{ISO.2} = [CE_{ISO} | (y \neq 0) \wedge (x = z) \wedge (y \neq z)]$$

$$DNF_{ISO.3} = [CE_{ISO} | (z \neq 0) \wedge (x = y) \wedge (z \neq x)]$$

Represented by $TTH(CE_{ISO}, DNF) = \{DNF_{ISO.1}, DNF_{ISO.2}, DNF_{ISO.3}\}$, this results in the second level test templates of the TTH. This process of refinement can continue using different testing strategies until the desired level of refinement is reached. The abstract test hierarchy generated using the framework provides descriptions of test suites and the nodes of the hierarchy can be used to provide a specific set of test data. For example, a test suite used to test the isosceles triangle part of the function (program) would be composed of at least three test cases based on $DNF_{ISO.1}, DNF_{ISO.2}$, and $DNF_{ISO.3}$. Example test data from each test template are (2,4,4), (6,1,6), and (5,5,4) for $DNF_{ISO.1}, DNF_{ISO.2}$, and $DNF_{ISO.3}$, respectively. To test the equilateral part of the function, test data would be generated based on CE_{EQ} such as (1,1,1), (4,4,4) or (5,5,5). The test data generated by the TTF can be used in the development and implementation of a specific test suite.

Thesis Overview and Contributions

In this thesis, a new structured framework for test suite development will be introduced that builds on previous research in test suite development, specifically the TTF. The new structured framework realizes that a practical structured framework should be able to translate abstract test specifications into an implementable test suite with minimal loss of information. The new structured framework will follow the TTF by focusing on the use of partition testing techniques of an application's input space. The TTF and the other structured frameworks have focused on deriving sets of input test data from an application's specification but do not investigate issues that arise when using the set of input test data for a specific implementation of a test suite. For example, some test data may not be suitable for used by a test suite given the limitations of a specific implementation of an application to be tested. In addition, an application's interactions with other applications are not easily expressed with the TTF and other structured frameworks. The inability of these structured frameworks to easily capture interactions between applications has become very important as client–server applications propagate as more computers become networked together.

The structured framework will be applied to the Minimum Interoperability Specification For Public Key Infrastructure Components (MISPC) specification to develop and implement a test suite for a reference implementation of the MISPC in order to provide a worked example of the application of the new framework. The MISPC is a hybrid specification using natural language to describe component interaction while Abstract Syntax Notation One (ASN.1) is used to describe data structures. The structured framework will be applied assuming that only black–box testing techniques can be used to test the components of the MISPC reference implementation. Because of the design of the MISPC reference implementation components, the test suite configuration will be a modified version of Figure 1.2(a). The modification to the original configuration is that the test suite to be developed will only interact with the IUT, the MISPC reference implementation components. To begin the development of a test suite for the MISPC, the TTF will be applied to the ASN.1 data structures defined by the MISPC. However, the MISPC's component interactions are quickly shown not to be easily expressed with the TTF. In addition, the fact that the TTF is applied to a "valid" input space of an application places limitations on the objectives a test suite developed under the TTF can achieve. A new structured framework is proposed, based on the concepts and ideas of the TTF, to resolve the drawbacks of the TTF and to translate an abstract test suite description into an specific implementation of a test suite. The new structured framework is used to complete the development of a test suite for MISPC components for this project.

The remaining chapters in the thesis are organized as follows:

Chapter 2. Overview of the Minimum Interoperable Specification For Public Key Infrastructure Components. This chapter gives a general introduction to public key infrastructures (PKIs) and presents a detailed description of the Minimum Interoperable Specification for Public Key Infrastructure Components for use by this thesis.

Chapter 3. Test Suite Development For The MISPC. This chapter demonstrates the use of the TTF for the development of a test suite for the MISPC and the TTF's limitations. A new structured framework for test suite development is proposed to overcome the TTF's limitations and is applied to MISPC.

Chapter 4. Application Of The Proposed Framework To The MISPC. This chapter presents a general overview of the MISPC reference implementation and how a test suite is developed and implemented for a MISPC Certificate Authority (CA) using the new structured framework. In addition, the results of executing the test suite on the MISPC CA reference implementation are presented.

Chapter 5 Conclusions And Future Work. This chapter highlights the advantages of using the new structured framework for test suite development and presents areas of the new structured framework that needs to be investigated further.

2 OVERVIEW OF THE MINIMUM INTEROPERABILITY SPECIFICATION FOR PUBLIC KEY INFRASTRUCTURE COMPONENTS

Public Key Infrastructure and the Minimum Interoperability Specification for Public Key Infrastructure Components

Public Key Infrastructure (PKI) is the general term used to describe the distribution environment of public key information of a public key cryptographic system. A public key cryptographic system is a cryptographic system where one key is used to encipher information and another is used to decipher information. One of the keys must be keep secret and is known as a private key while the other key can be distributed to anyone and is known as a public key. Within a PKI, a data structure called a certificate is used to bind a specific identity to specific public key information. A Certificate Authority (CA) is a trusted entity that issues and revokes certificates within a PKI. An Organizational Registration Authority (ORA) is an entity trusted by a CA that performs identity and public key verification that relieves the CA of this function. A certificate holder (CH) and Clients are entities that use certificates issued by CAs they trust (directly or indirectly) to perform cryptographic operations.

The Minimum Interoperability Specification For Public Key Infrastructure Components (MISPC)[1] specifies components of a PKI which should be interoperable with other components which meet the requirements of the specification regardless of implementation details. The specification defines four components by their functionality and the data structures (certificates, Certificate Revocation Lists (CRLs), and PKI transaction messages) transferred between the components. In the specification, one component known as a repository (REP), used to store certificates and CRLs, is referenced but is beyond the scope of the specification to be completely described. The specification does not address some implementation issues such as the transport mechanism used to transfer information between components, error recovery mechanisms, decision algorithms (except for the path validation engine

[1]The MISPC version 1 does not include support for repositories and confidentiality. However, these features are being investigated and may be incorporated into subsequent versions of the MISPC.

(PVE) described later), and out-of-band transactions.

The MISPC is designed to support both hierarchical and networked trust models as shown in Figures 2.1 and 2.2, respectively. In a hierarchical trust model, trust is developed by a CA certifying subordinate CAs which implies the existence of root CA that is trusted by all nodes of the infrastructure [7]. In Figure 2.1, the top- most CA is the root CA which issues certificates (indicated by a single-headed arrow) to subordinate CAs that may in turn issue certificates to other CAs or CHs. All subordinate CAs and CHs must trust the root CA because all trust developed within the infrastructure originates and relies on the notion that the root CA can be trusted without question. Without this, no trust can be developed within the hierarchical trust model. The network trust model develops trust by certification between CAs in a peer relationship that removes the notion of a root CA trusted by all entities within the infrastructure. In Figure 2.2, the two CAs on the right trust each other as a result of their cross certification (indicated by double-headed arrow) of one another. The same situation exists with the two CAs on the left. The result is that two independent domains of trust have been developed without the need for every entity in the domains placing their trust in a root CA. Because both the hierarchical and networked trust models are supported by the MISPC, a hybrid trust model could be constructed if desired.

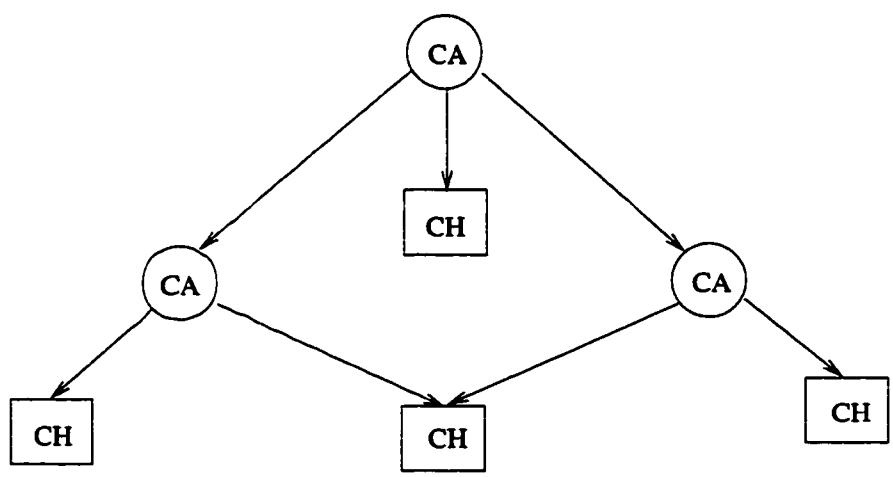

Figure 2.1 Hierarchical Trust Model.

The MISPC uses the natural language English to specify the functionality required (and some which are optional) by each component while using a formal description technique, ASN.1, to describe the data structures of certificates, CRLs, and PKI transaction messages. This subsection will review all the basic MISPC components and the transactions (required and optional) used by each. In addition, the

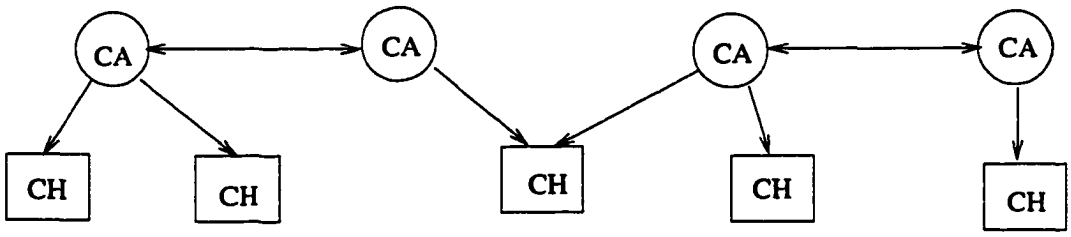

Figure 2.2 Networked Trust Model.

cryptographic systems, Path Validation Engine (PVE), and data structures for the certificates, CRLs, and PKI messages used by the basic components of the MISPC will be presented.

MISPC Components: Client, Certificate Holder, Certification Authority, and Organizational Registration Authority

The most basic MISPC component is the Client which performs the functions of (a) retrieving certificates and CRLs from repositories, (b) validating certification paths using the PVE, and (c) verifying signatures in certificates and CRLs. Certificates and CRLs are retrieved by the Client from repositories using the Lightweight Directory Access Protocol (LDAP) version 2, [41]. LDAP is a directory service protocol which operates over the Transmission Control Protocol (TCP) and is based on X.500. It contains a subset of the X.500 directory service operations (search, add, delete, modify, bind, unbind, abandon) and uses X.500's naming convention [27, 25]. LDAP uses ASN.1 to describe its data structures which are encoded using the Basic Encoding Rules[2] (BER). The use of LDAP is a minimal requirement which means that other directory service protocols can be used by a Client, however LDAP must be present in any MISPC compliant Client application. Certificates are identified by the certificate's subject name or a serial number/issuer identification pair. The certificate's subject name is an X.500 distinguished name used to identify the owner of the certificate. CRLs are retrieved based on the CA that issued them or by a serial number used to identify a specific CRL. The Client uses the PVE to validate a certification path when it does not directly trust a component that generates a signature using a certificate issued by a CA the Client does not trust. The Client will try to validate a path from the certificate in question to a certificate issued by a CA that it trusts. Details of the PVE operation as well as the specific procedure used to verify signatures of certificates and CRLs will be reviewed later in this section.

[2]The MISPC states that components must use Distinguished Encoding Rules (DER), a subset of BER, for the encoding of ASN.1 elements.

The Certificate Holder (CH) role adds to the capabilities of the basic Client role by allowing an entity to perform functions required for the maintenance of certificates. In addition to the Client functions, a CH can perform the following four functions: (a) generate signatures, (b) request certificates, (c) revoke certificates, and (d) renew certificates. The CH can only perform these certificate maintenance functions on certificates for which it is the owner (subject). A certificate can be requested by either of two different transactions: (a) ORA-Generated Registration or (b) Self Registration. In an ORA-Generated Registration transaction, the CH must interact with an ORA to request a certificate while a Self Registration transaction allows the CH to request a certificate directly from a CA. The details of a CA and ORA are given in the following two paragraphs, respectively. The revocation and renewal of certificates are done using the Certificate Revocation and Certificate Renewal transactions, respectively. Both transactions require the CH to interact with the CA directly. Finally, all of the information used in the CH's transactions needs to be authenticated to prove the origins of the data transmitted during the transactions. The authentication is done using signatures which means that a CH must be able to generate signatures as well as verify them. The specific procedure used to generate signatures will be discussed when cryptographic systems are reviewed later in this section.

The Certificate Authority (CA) role adds to the capabilities of the CH role by allowing an entity to perform functions required to manage certificates that it has issued to different PKI components. The CA has the functionality to (a) issue certificates, (b) revoke certificates, and (c) distribute information about certificates. The CA can only perform these certificate management functions on certificates which it has issued. CAs issue certificates by accepting ORA-Generated Registration or Self Registration requests from ORAs or CHs. The CA can revoke a certificate upon receiving a Certificate Revocation request from a CH or ORA which has the required authorization. Additionally, the CA can renew a certificate by accepting a Certificate Renewal request from the subject of the certificate to be renewed. The CA will process the different certificate management requests and will issue, revoke, or renew a certificate when appropriate and generate a reply about the status of the request to the originating component. A CA distributes information about certificates which it has issued using the Post Certificate and Post CRL transactions. The Post Certificate and Post CRL transactions require the CA and repository to interact; however, this interaction is not specified by the MISPC. A CA is only required to post certificates it has issued to other CAs but is not prevented from posting other types of certificates. A CA periodically uses the Post CRL transaction to distribute information about certificates it has revoked.

The Organizational Registration Authority (ORA) role extends a CH's capabilities by allowing it to

vouch for the identity of an entity by signing certificate requests on behalf of the entity or by providing information used to authenticate the entity to a CA. An ORA assists an entity in obtaining a certificate by using the ORA-Generated Registration transaction. The ORA accepts certificate requests from entities and, when appropriate, signs the requests using its private key. The ORA then forwards the signed requests to a CA to be processed. In addition, the ORA passes information required to verify a certificate's authenticity to the requesting entity, usually the issuing CA's public key. The ORA helps in the Self Registration transaction by providing secret information to an entity which is used to authenticate the entity to the CA. The interaction between the ORA and the entity wishing to obtain a certificate is beyond the scope of the MISPC but must exist. However, the ORA interaction with the CA is specified by the MISPC. An ORA can request revocation of certificates using the Certificate Revocation transaction on behalf of an entity's organization (company, university, agency, etc.) or for an entity which no longer possesses its private key.

Transactions Specified by the MISPC

The MISPC describes nine transactions which Clients, CHs, CAs, ORAs, and REPs participate in when appropriate. Table 2.1 provides a listing of these transactions, which have been mentioned during the component descriptions. The table divides each transaction into functions based on the component's role in the transaction. The table shows that a given MISPC component may be required to be able to perform more than one role for a given transaction. For example, in the ORA Generated Registration transaction, a CA must be able to perform the CA's role of accepting certificate requests and issuing certificates as well as the CH's role of generating certificate requests. The CA is required to be able to perform the CH's role in order to obtain certificates from other CAs for itself. The table also reflects the MISPC's ambiguity about which components must be able to participate in PKCS#10 Self Registration transactions. The PKCS#10 Self Registration transaction is basically the same as the Self Registration transaction with similar information presented in a different format. The PKCS#10 Self Registration transaction message profiles are included later for completeness but will not be expanded upon because of the MISPC's ambiguity and its similarity to the Self Registration transaction.

When a CH wishes to obtain an initial certificate, it can use either the Self Registration or ORA Generated Registration transaction. From Table 2.1, CH components are required to be able to participate in ORA Generated Registration transactions. However, participating in Self Registration transactions is optional. Both transactions involve the participation of a CH, a CA, and an ORA. An initial Out-Of-Band (OOB) transaction between the CH and ORA is required by both transactions. The OOB

Table 2.1 MISPC Transaction to Component Relationship.

Transaction	Roles	Components			
		CA	ORA	CH	CL
ORA Generated Registration	CA	M			
	ORA		M		
	CH	M	M	M	
Certificate Revocation	CA	M			
	ORA		M		
	CH	M	M	M	
Self Registration	CA	M			
	CH	O	O	O	
Certificate Renewal	CA	M			
	CH	O	O	O	
Post Certificate	CA	M			
	REP				
Post CRL	CA	M			
	REP				
Retrieve Certificate	CL	M	M	M	M
	REP				
Retrieve CRL	CL	M	M	M	M
	REP				
PKCS#10 Self Registration	CA	A			
	CH	A	A	A	

Key:

CA	Certificate Authority
ORA	Organization Registration Authority
CH	Certificate Holder
CL	Client
REP	Repository
CRL	Certificate Revocation List
M	Mandatory
O	Optional
A	Ambiguous

transactions are used to exchange any (public and/or secret) key information about the issuing CA as well as any extra information that may be required by a CH to authenticate itself to the issuing CA. The details of the OOB transactions are beyond the scope of the MISPC, but must convey at least the aforementioned information. Once the initial OOB transaction occurs, the Self Registration transaction consists of two steps: (1) a certificate request sent to a CA by the CH and (2) a certificate reply sent back to the requesting CH by the CA. The certificate request contains information (public key, preferred subject name, etc.) about the certificate which the CH wishes to obtain as well as information (signature, shared secret, etc.) to authenticate itself to the CA. The certificate reply will either contain the certificate or an error code indicating why the CA was unable to issue the certificate. A listing of the possible error codes and their meaning is shown in Table 2.2. The certificate reply includes the CA's digital signature or some other information (such as a nonce) to authenticate itself to the CH.

Table 2.2 MISPC Error Codes and Meanings.

Error Code	Definition
badAlg	Cannot validate signature because algorithm identifier is unrecognized or unsupported.
badMessageCheck	Protection field checked but did not match expected value.
badPoP	Proof-of-Possession field checked but did not match expected value.
badRequest	The responder does not permit or support transaction.
badTime	The time field in the message header was not sufficiently close to the responder's system time.
badCertId	No certificate could be found matching the non-zero serial field.

After the initial OOB transaction, the ORA Generated Registration transaction consists of two steps: (1) a certificate request is sent by the ORA to a CA on behalf of the CH and (2) a certificate reply is sent back by the CA to the ORA which submitted the certificate request and optionally to the subject of that requested certificate. Once the ORA is satisfied about the requesting CH's identity through the initial OOB transaction, it accepts the presented certificate request, signs the request, and forwards the request to a CA. The certificate request contains similar information to that which is used in the certificate request of the Self Registration transaction. If the CA does not issue a certificate, it sends back a certificate reply to the ORA which submitted the certificate request. The certificate reply contains the reason a certificate could not be issued, the CA's signature, and other identification and authentication information (such as transaction request number). If the CA issues a certificate, it sends a certificate reply back to the ORA containing the new certificate, the CA's signature, and other identification and authentication information (such as a nonce). In either case, the CA may optionally

send the same certificate reply to the subject of the certificate request.

The Certificate Revocation transaction is used to make a specific certificate invalid before its natural period of validity elapses. A component (CA, ORA, or CH) may wish to immediately invalidate a certificate which it discovers has been compromised or which is associated with a component that it no longer trusts. A CH may request that certificates it owns to be revoked. For example, a CH may have been issued a certificate which it no longer requires but will be valid for a period of time. The CH may wish to revoke the certificate immediately in order to reduce the risk of the certificate being used inappropriately and itself being held liable. ORAs may request that certificates be revoked on behalf of a CH or CH's organization. For example, an ORA may be required by its organization to revoke all current certificates of CHs which are no longer associated with the organization. The CH or ORA sends a revocation request to a CA including information about the certificate to be revoked and the reason for its revocation. The revocation request is signed using either the CH's or ORA's private key depending on who creates the revocation request. A CA sends a revocation reply back to the component that created the revocation request indicating whether or not the certificate was revoked. If the certificate is not revoked, an error code is returned indicating the reason for the failure. The revocation reply includes the CA's signature and other identification and authentication information (such as a nonce).

The Certificate Renewal transaction is used by a CH which currently possesses a valid certificate whose validity period is about to elapse and which wishes to obtain a new certificate. The CH interacts directly with the CA that issued the certificate to request that a new certificate be issued. The Certificate Renewal transaction is a two step renewal process where (1) the CH sends a renewal request to the issuing CA and (2) the CA sends a reply back to the CH. The renewal request contains information about the new certificate to be issued by the CA and is similar to the certificate request of the Self Registration transaction. If the CA issues a certificate, it sends back a reply to the CH containing the new certificate. Otherwise, the CA sends back a reply indicating why the CA was unable to issue a new certificate. The renewal reply includes the CA's signature and some identification and authentication information (transaction number, nonce, etc.) to allow the CA to authenticate itself to the requesting CH.

The Post Certificate/CRL and Retrieve Certificate/CRL transactions are beyond the scope of the MISPC due to the fact that they require participation with a repository which is not a completely specified component. However, all specified MISPC components (Clients, CHs, CAs, and ORAs) are required to be able to retrieve certificates and CRLs from repositories because they are specified to

include the Client functionality (see the previous description of Client component). The posting of certificates and CRLs is the responsibility of the CAs (see the previous description of CA component). However, the MISPC gives no guidance for the Post Certificate/CRL transactions beyond stating that the functionality must exist and CA and cross certificates as well as CRLs issued by a CA must be posted to a repository.

MISPC Cryptographic Systems

The MISPC specifies two classes of cryptographic algorithms: (1) digital signature algorithms and (2) message authentication codes (MACs). Digital signature algorithms are used by components to sign information. There are several ways of creating a digital signature but in general they are generated using a two step process: (1) creation of a message digest using a one-way hash function and (2) the encryption of the message digest with the signer's private key using a cryptographic algorithm. A one-way hash function is used to create a short string of bits (a message digest) used to fingerprint a large amount of information. A one-way hash function has the property of being one-way and may demonstrate the desirable characteristic of being collision-free. The one-way property means that, given a specific output of the hash function, it is difficult to determine an input that would create the same output. The desired characteristic of being collision-free means that two different inputs to the function will not result in the same output being produced by the hash function. The reason for using a one-way hash function for digital signatures is to address performance issues found in some cryptographic algorithms. For some cryptographic algorithms, encryption or decryption of a complete document for the purpose of signing the information would require too much computation time relative to the performance requirements of an application. One-way hash functions create a short piece of data which is encrypted and decrypted by a cryptographic algorithm for the purpose of signing longer amounts of information. To validate a digital signature, the validator creates the message digest for the information received, decrypts the message digest contained with the information, and compares the two resulting message digests. If the message digests match, the digital signature is considered valid. Otherwise, the digital signature is invalid.

The Secure Hash Algorithm (SHA-1) is the one-way hash function required by all MISPC components. The details of SHA-1 can be found in [29]. The three public-key cryptographic algorithms specified in the MISPC are RSA [35], Digital Signature Standard (DSS) [30], and (3) Elliptic Curve Digital Signature Algorithm (ECDSA) [2]. For interoperability[3] and verification purposes, one or more of

[3]MISPC components that use the same cryptographic algorithm(s) should be able to interact with each other in a

the three public-key cryptographic algorithms must be implemented by MISPC components to generate the required digital signatures.

The other type of cryptographic algorithm mentioned in the MISPC are MACs which are key-dependent one-way hash functions. MACs are similar to digital signatures, but use secret keys and do not allow for non-repudiation. The MISPC requires the use of Data Encryption Standard Message Authentication Code (DES-MAC) for MAC generation. The details of DES-MAC can be found in [31].

The Path Validation Engine Specified by the MISPC

The Path Validation Engine (PVE) is used by MISPC components to validate a sequence of certificates. The last certificate in the sequence is called the end certificate while the others are known as intermediate certificates. If only an end certificate is present, it was issued by a CA which the Client trusts and the Client goes through one iteration of the validation procedure. If intermediate certificates are present, the end certificate to be validated was issued by a CA which the Client does not directly trust. Thus, a path between a certificate issued by a CA trusted by the Client and the end certificate issued by the non-trusted CA must be validated. The MISPC defines a PVE by reference to the International Standards Organization (ISO)–Open Systems Interconnection–The Directory: Authentication Framework specification recommendation X.509 which gives a natural language description of a state-based validation engine [26].

The PVE is a state machine composed of six inputs, five outputs, and seven state variables. The input variables of the PVE are: (I1) a set of certificates making up a certificate path, (I2) a trusted public key to validate the first certificate in the path, (I3) an initial set of policies acceptable to the validator, (I4) an initial indicator used to explicitly indicate that at least one of the acceptable policies must be present in all certificates in the path, (I5) an initial policy mapping inhibitor indicator which tells if policy mapping is not allowed in the certificate path, and (I6) the current time/date. In X.509 version 3 certificates, some extensions have been defined to indicate under what conditions a certificate can be used. The certificate's extensions can also define a certificate's policies. Policy mapping is the act of processing these certificate extensions for use by the PVE. The output variables of the PVE are: (O1) an indicator telling whether the certificate path was validated or not, (O2) an error code (when a path is invalid), (O3) details on any policy mapping that occurred during path evaluation, and (O4) the set of all acceptable policies which are constrained by CAs in the path and all qualifiers of the policies or a special value indicating that any policy is acceptable. If (O4) has the special value

productive manner (i.e. issue certificates, revoke certificates, etc.).

indication "any policy", another output, (O5), is generated containing a set of all acceptable policy qualifiers encountered in the path. The state variables of the PVE are: (S1) the set of policies currently acceptable to the validator, (S2) the set of policies acceptable to the CAs in the certification path, (S3) the set of sub-trees within which all subject names of subsequent certificates must fall, (S4) the set of sub-trees within which no subject names of subsequent certificates can fall. (S5) an indicator used to tell that at least one acceptable policy must be in every certificate of the path, (S6) an indicator used to prohibit policy mapping, and (S7) an indicator used to show that a policy constraint or policy mapping suspension is pending and the number of certificates which will be processed before the pending requirement becomes effective.

The process used by the PVE to validate a certificate path for a set of certificates begins by the initialization of the state variables: (S1) is set to the value of (I3), (S2) is set to the special value which indicates any policy is acceptable, (S3) is set to allow any sub-tree (i.e. all possible name-space), (S4) is set to the empty set, (S5) is set to the value of (I4), (S6) is set to the value of (I5), and (S7) is cleared to indicate a policy constraint or policy mapping is not pending. Each certificate in the path is processed by checking that (a) the digital signature and date of the certificate are valid, (b) the certificate's subject and issuer relationships are correct, and (c) the certificate has not been revoked. If the certificate passes (a)-(c), it is further processed using the state variables. If (S5) is set and (S1) is not set to accept any policy, the policy extensions of the certificate are checked to see that at least one of the policies in (S1) is present. If any policy extensions are present in the certificate and indicated to be critical, a new value of (S2) is calculated by taking the intersection of the policies found in the certificate and (S2). The intersection of (S2) and (S1) is checked to ensure it is non-empty. The certificate's subject name is checked to see that it is falls into the allowable name-space (S3) and outside the disallowed name-space (S4). For an intermediate certificate, two additional processing checks are required. If the basic constraints extension is present in the certificate, the ca field of the extension is checked for a setting of "true". If the path length constraint field of the basic constraint extension is present in the certificate, a check is made to see that the current certificate path length does not violate the constraint. If any one of the aforementioned checks fails, the certificate validation process halts and a failure is returned with the appropriate error code. If all the certificate's checks pass, the certificate validation process terminates by returning (1) a successful indication, (2) a set of policy identifiers, (3) the required policy qualifiers, and (4) details of the policy mapping that occurred during the processing.

When the PVE is processing an intermediate certificate, the state variables are updated. The PVE state variables updated are the set of acceptable policies of the validator (S1), the set of acceptable

policies of the CAs (S2), the allowed and disallowed name-spaces (S3 and S4, respectively), the value of the indicator used to show that at least one acceptable policy must be present in the certificate (S5), the value of the indicator used to inhibit policy mapping (S6), and the values of the indicators used to show that a policy constraint or policy mapping suspension is pending and the number of certificates to be processed before the pending requirement takes effective (S7). The state variables which are the most straightforward to update are (S3) and (S4). If the permitted name-space extension is present in the certificate, (S3) is updated by simply taking the intersection of current value and the permitted name-space found in the certificate. (S4) is updated in a similar fashion by taking the union of current value and the prohibited name-space, if the prohibited name-space extension is present in the certificate. (S5) can only be changed from cleared to set during the processing of any specific certificate path. If (S5) is cleared and the explicit policy pending part of (S7) is set, the number of certificates to skip part of (S7) is decremented. If zero is the result of decrementing the number of certificates to skip part of (S7), then (S5) is set. The number of certificates to skip part of (S7) is set to a value found within a certificate containing either the explicit policy or inhibit policy mapping constraint extension. If the value found in the certificate is zero, (S5) is set immediately. Otherwise, the explicit policy pending part of (S7) is set and the number of certificates to skip is set to the non-zero value found in the certificate. If the number of certificates to skip part of (S7) currently is a non-zero value, the value is updated to the lesser of the current value and the value found in the certificate.

Finally, (S1), (S2)[4], and (S6) are updated only if (S6) is cleared. Similar to (S5) in the previous paragraph, (S6) can only be changed from clear to set during the processing of any specific certificate path. If (S6) is not set, any policy mapping extensions are processed with respect to the set of acceptable policies of the validator and CAs ((S1) and (S2), respectively) and added to the appropriate set as required. If the number of certificates to skip in the certificate is zero, then (S6) is set immediately. Otherwise, the inhibit policy mapping pending part of (S7) is set and the number of certificates to skip is set to the non-zero value found in the certificate. If the number of certificates to skip part of (S7) currently is a non-zero value, the value is updated to the lesser of the current value and the value found in the certificate.

[4]This update is supplemental to the update described when a certificate policy's extension is present and indicated to be critical.

MISPC Data Structures: Certificates, Certificate Revocation List, and PKI Messages

The certificates, CRLs, and PKI messages will be reviewed to complete this overview of the MISPC. The certificates, CRLs, and PKI messages used by the MISPC are specified using ASN.1. It should be noted that the ASN.1 definitions for the certificates, CRLs, and PKI messages are a subset of the data structures found in [23] and [1]. As a result, some fields defined as optional are required while other non-optional fields are not used or are ignored by the MISPC. The certificates and CRLs are based on X.509 and are shown in Figures 2.3 and 2.4, respectively. Even though the ASN.1 specifications indicate a type of **SEQUENCE** for the data structures, the actual order of the fields specified may be different when synthesized due to the tagging of fields done when encoding and decoding ASN.1 specified data structures. To clarify the notation for the data structures to be presented in the text of this section, data structure **Types** will be in **bold** and begin with a capital letter. Data structure *fields* will begin with a lower case letter and be in *italics*.

```
Certificate          ::=    SIGNED    { SEQUENCE {
  version                   [0] VersionDEFAULT v1,
  serialNumber              CertificateSerialNumber,
  signature                 AlgorithmIdentifier
  issuer                    Name,
  validity                  Validity,
  subject                   Name,
  subjectPublicKeyInfo      SubjectPublicKeyInfo,
  issuerUniqueIdentifier    [1] IMPLICIT UniqueIdentifier OPTIONAL,
  subjectUniqueIdentifier   [2] IMPLICIT UniqueIdentifier OPTIONAL,
  extensions                [3] Extensions Optional }}
```

Figure 2.3 ASN.1 Specification of an MISPC Certificate.

The certificates specified in the MISPC are composed of ten fields, of which three are optional. The optional fields need not be used in order for the PKI to work properly but are included to support advanced functionality, such as conditions under which the public key found in a certificate may or may not be used. All the fields of the certificate are covered by the digital signature generated by the issuing CA as indicated by the user defined type **SIGNED**. The *version* field is used to indicate the version of the certificate. Currently, there are three versions of X.509 based certificates; however, the MISPC specifies the use of version 3 certificates. The *serialNumber* field is an integer value assigned by

```
CertificateList      ::=    SIGNED  { SEQUENCE {
  version                   Version OPTIONAL,
  signature                 AlgorithmIdentifier,
  issuer                    Name,
  thisUpdate                ChoiceOfTime,
  nextUpdate                ChoiceOfTime OPTIONAL,
  revokedCertificates       SEQUENCE OF SEQUENCE {
    userCertificate                 CertificateSerialNumber,
    revocationDate                  ChoiceOfTime,
    crlEntryExtensions      Extensions OPTIONAL } OPTIONAL,
  crlExtensions             [0] Extensions OPTIONAL }}
```

Figure 2.4 ASN.1 Specification of an MISPC CRL.

the CA that issued the certificate. The integer should be unique with respect to the issuing CA so that an issuer name/serial number pair can be used to uniquely identify a certificate. The *signature* field is used to indicate the digital signature algorithm (RSA, DSA, or ECDSA) used by the CA to sign the certificate. The *issuer* field indicates the globally unique identifier associated with the issuing CA. The *subject* field is the globally unique identifier of the subject of the certificate. Both the *issuer* and *subject* fields are of type **Name** which indicates that X.500 distinguished names should be used. The *validity* field indicates the dates the certificate is valid. It is composed of a sequence of two fields, a beginning date and an ending date. The dates are to be expressed in Coordinated Universal Time, **UTCTime**. referenced to Greenwich Mean Time with a resolution of seconds. The *subjectPublicKeyInfo* field is composed of a sequence of two fields: one indicates the public key cryptographic algorithm including any public parameters and the other contains the actual public key. The *issuerUniqueIdentifier* and *subjectUniqueIdentifier* are optional fields used to handle the reuse of subject and/or issuer names over time. Finally, *extensions* is an optional field composed of a sequence of three fields. The fields indicate the name, criticality, and value of the extension included in the certificate. A certificate can contain any number of standard or private extensions. One standard extension is used to indicate the allowed uses of the public key contained in the certificate (e.g. encipherment only).

The CRLs specified by the MISPC contain seven fields of which three are optional. Just like a certificate, all the fields of a CRL are covered by the digital signature generated by the issuing CA as indicated by the user defined type **SIGNED**. The *version* field is used to indicate which version of the X.509 CRL is used. Currently, the MISPC specifies the use of X.509 version 2 CRLs. The *signature* field

indicates the digital signature algorithm (RSA, DSA, or ECDSA) used to sign the CRL. The *issuer* field indicates the globally unique identifier of the issuing CA. The *thisUpdate* field indicates the date the CRL was generated by the CA and the *nextUpdate* is a field that indicates the date by which the next CRL will be issued. The dates are to be expressed in **UTCTime** referenced to Greenwich Mean Time with a resolution of seconds. The *revokedCertificates* field contains the list of revoked certificates. Each revoked certificate listed contains the serial number of the certificate, the date of the revocation, and optional CRL extensions used to indicate additional information such as why the certificate was revoked. Finally, *crlExtensions* is an optional field composed of a sequence of three fields. The fields indicate the name, criticality, and value of the extension to the CRL. The CRL can contain any number of standard or private extensions. The standard CRL extensions are used to associate additional attributes to a given CRL such as the distribution point of the CRL.

The general layout of the PKI messages specified by the MISPC is shown in Figure 2.5. The PKI messages are composed of four fields of which two are optional. The *protection* field is an optional field containing a digital signature of the *header* and *body* fields of the message, but is required by the MISPC. The *extraCerts* is an optional field that is not required by the MISPC. The *header* and *body* fields of a PKI message are shown in Figures 2.6 and 2.7, respectively. The *header* field contains a sequence of 11 fields of which all but three are optional. The three required fields are *pvno*, *sender*, and *recipient* fields. The *pvno* field is used to indicate the version format of the PKI message. For version 1 of the MISPC, the *pvno* field value is specified to be zero. The *sender* and *recipient* fields are used to identify the sender and receiver of the PKI message by using X.500 distinguished names or Internet electronic mail names. *messageTime* is an optional field used to indicate the time the message was generated, but is required by the MISPC. The format of **GeneralizedTime** type is more general than the Coordinated Universal Time (**UTCTime**) type. A date's resolution can vary (seconds, minutes, hours, days, etc.) and can be relative to a local (Eastern, Central, etc.) or universal (Greenwich Mean Time) reference. However, the MISPC requires the *messageTime* field be expressed in Greenwich Mean Time with a resolution of seconds. Another optional field that is required by the MISPC is the *protectionAlg* field used to indicate the digital signature algorithm (RSA, DSA, or ECDSA) or message authentication code (DES-MAC) used to ensure that the information in the PKI message has not been modified. This field is used in coordination with the PKI message *protection* field mentioned previously. The *senderKID* and *recipKID* are optional fields used to indicate the key used to protect the PKI message. This field is useful when entities have multiple keys and specific keys need to be identified. The *transactionID* is an optional field used to identify with which transaction the given PKI message is associated when a

```
PKIMessage      ::=     SEQUENCE {
  header                PKIHeader,
  body                  PKIBody,
  protection            [0] PKIProtection OPTIONAL,
  extraCerts            [1] SEQUENCE OF Certificate OPTIONAL }
```

Figure 2.5 ASN.1 Specification of an MISPC PKI Message.

```
PKIHeader       ::=     SEQUENCE {
  pvno                  INTEGER     {fpki-version1 (0)},
  sender                GeneralName,
  recipient             GeneralName,
  messageTime           [0] GeneralizedTime OPTIONAL,
  protectionAlg         [1] AlgorithmIdentifier OPTIONAL,
  senderKID             [2] KeyIdentifier OPTIONAL,
  recipKID              [3] KeyIdentifier OPTIONAL,
  transactionID         [4] OCTET STRING OPTIONAL,
  senderNonce           [5] OCTET STRING OPTIONAL,
  recipNonce            [6] OCTET STRING OPTIONAL,
  freeText              [7] PKIFreeText OPTIONAL }
```

Figure 2.6 ASN.1 Specification of PKIHeader Element.

component is conducting multiple PKI transactions. The *transactionID* field is required by the MISPC for ORA Certificate Registration and Certificate Revocation transaction messages. The *senderNonce* and *recipNonce* are optional fields used to provide protection from replay attacks and to provide a level of authentication. The *senderNonce* is created by the entity that is generating the current PKI message. The *recipNonce* is a nonce that was generated by the entity that will be receiving the current PKI message and that was passed to the sender in the *senderNonce* field previously. Finally, *freeText* is an optional field used to include additional information (for human consumption) about the PKI message.

The second required PKI message field is *body* which can contain any one of 24 different PKI message types (see Figure 2.7). However, only six of the 24 PKI message types (**in bold**) are required by the MISPC. The **CertReqContent** is used by an entity to request a certificate. It is defined as a sequence of elements of type **FullCertTemplate** which allows for the processing of multiple requests by a single transaction. However, the feature of multiple certificate requests in a single transaction is

```
PKIBody      ::=    CHOICE {
  ir                [0] InitReqContent,
  ip                [1] InitRepContent,
  cr                [2] CertReqContent,
  cp                [3] CertRepContent,
  p10cr             [4] PKCS10CertReqContent,
  popdecc           [5] POPODecKeyChallContent,
  popdecr           [6] POPDecKeyRespContent,
  kur               [7] KeyUpdReqContent,
  kup               [8] KeyUpdRepContent,
  krr               [9] KeyRecReqContent,
  krp               [10] KeyRecRepContent,
  rr                [11] RevReqContent,
  rp                [12] RevRepContent,
  ccr               [13] CrossCertReqContent,
  ccp               [14] CrossCertRepContent,
  ckuann            [15] CAKeyUpdAnnContent,
  cann              [16] CertAnnContent,
  rann              [17] RevAnnContent,
  crlann            [18] CRLAnnContent,
  conf              [19] PKIConfirmContent,
  nested            [20] NestedMessageContent,
  infor             [21] PKIInfoReqContent,
  infop             [22] PKIInfoRepContent,
  error             [23] ErrorMsgContent }
```

Figure 2.7 ASN.1 Specification of PKIBody Element.

not supported by the MISPC, so only one **FullCertTemplate** will be found in **CertReqContent**. Six fields compose the **FullCertTemplate** type as shown in Figure 2.8. However, only the *certReqId*, *oldCertId*, *POPOSigningKey*, and *certTemplate* fields (**in bold**) are specified in the MISPC. The *certReqId* field is used to match a response to a given request while *oldCertId* field identifies a current or expired certificate for the subject. *POPOSigningKey* field provides information to assure the subject has possession of the private key for the new certificate's public key. *certTemplate* field contains information about the specific certificate being requested as defined by the **CertTemplate** type shown in Figure 2.9. The **CertTemplate** type has 10 optional fields used to request specific characteristics for the certificate to be generated. The *publicKey* field which contains the public key and the specific cryptographic algorithm (RSA, DSA, or ECDSA) of the certificate is the only required field. The *issuerUID*

```
CertReqContent          ::=     FullCertTemplates

FullCertTemplates       ::=     SEQUENCE OF FullCertTemplate

FullCertTemplate        ::=     SEQUENCE {
  certReqId                     INTEGER,
  certTemplate                  CertTemplate,
  popoSigningKey                [0] POPOSigningKey OPTIONAL,
  archiveOptions                [1] PKIArchiveOptions OPTIONAL,
  publicationInfo               [2] PKIPublicationInfo OPTIONAL,
  oldCertId                     [3] CertId OPTIONAL }
```

Figure 2.8 ASN.1 Specification of CertReqContent, FullCertTemplates, and FullCertTemplate Elements.

and *subjectUID* fields are not supported by the MISPC. The *version* field is used to request the version of the X.509 certificate to be generated which is version 3 for the MISPC. The *serial* field is used to request a particular serial number for a certificate or to indicate a request is related to a previously issued certificate. The *signingAlg* field is used to indicate the cryptographic signing algorithm (RSA, DSA, or ECDSA) that will be used to sign the generated certificate. The *subject* and *issuer* fields are used to indicate a specific subject name and issuing CA name for the certificate using the X.500 distinguished naming convention. The *validity* field is used to indicate a specific time interval for which the new certificate will be valid. The *extensions* field is used to indicate a request for certificate extensions such as public key usage.

As an alternative to requesting certificates using PKI messages with a *body* field of type **CertReqContent**, an entity can request certificates using the type **PKCS10CertReqContent** which is composed of three fields as shown in Figure 2.10. The *signatureAlgorithm* field indicates the cryptographic algorithm used to generate the digital signature contained in the *signature* field. The *certificateRequestInfo* field of **PKCS10CertReqContent** contains the information about the certificate to be generated. *certificateRequestInfo* includes the preferred certificate version (*version*), a preferred subject name (*subject*), the public key and cryptographic algorithm (*subjectPublicKey*) (RSA, DSA, or ECDSA), and any other *attributes* about the certificate. The *attributes* field may be ignored by MISPC compliant components.

In response to a PKI message with a *body* field of **CertReqContent** or **PKCS10CertReqContent**, a PKI message with a *body* field of **CertRepContent** is generated (See Figure 2.11). It is composed of

```
CertTemplate        ::=     SEQUENCE {
  version                   [0] Version OPTIONAL,
  serial                    [1] INTEGER OPTIONAL,
  signingAlg                [2] AlgorithmIdentifier OPTIONAL,
  subject                   [3] Name OPTIONAL,
  validity                  [4] OptionalValidity OPTIONAL,
  issuer                    [5] Name OPTIONAL,
  publicKey                 [6] SubjectPublicKeyInfo OPTIONAL,
  issuerID                  [7] UniqueIdentifier OPTIONAL,
  subjectID                 [8] UniqueIdentifier OPTIONAL,
  extensions                [9] Extensions OPTIONAL }
```

Figure 2.9 ASN.1 Specification of CertTemplate Element.

```
PKCS10CertReqContent        ::=     SEQUENCE {
  certificationRequestInfo          CertificationRequestInfo,
  signatureAlgorithm                SignatureAlgorithmIdentifier,
  signature                         Signature }

CertificationRequestInfo    ::=     SEQUENCE {
  version                           Version,
  subject                           Name,
  subjectPublicKeyInfo              SubjectPublicKeyInfo,
  attributes                        [0] IMPLICIT Attributes }
```

Figure 2.10 ASN.1 Specification of PKCS10CertReqContent and CertificationRequestInfo Elements.

the *response* field and the optional *caPub* field which is a certificate. The *response* field is a sequence of **CertResponse** type elements which allows for multiple certificates to be issued by a single transaction; however, the MISPC only supports messages for a single certificate per transaction. Each **CertResponse** element contains the request identification number (*certReqID*), the status of the certificate request (accepted, rejected, etc.), any reason codes needed (*certRepStatus*), and the certified key pair (*certifiedKeyPair*). The *certifiedKeyPair* field is composed of four fields which are specified as optional. However, the MISPC requires that the *certificate* field be present as it is used to hold the newly issued certificate. The *encryptCert*, *privateKey*, and *publicationInfo* fields are not used by the MISPC.

A PKI message with a *body* field type of **RevReqContent** is used by an entity to request the revocation of a certificate. The **RevReqContent** is composed of a sequence of **RevDetails** elements

```
CertRepContent      ::=     SEQUENCE {
  caPub                     [1] Certificate Optional,
  response                  SEQUENCE OF CertResponse }

CertResponse        ::=     SEQUENCE {
  certReqId                 INTEGER,
  certRepStatus             PKIStatusInfo,
  certifiedKeyPair          CertifiedKeyPair }

CertifiedKeyPair    ::=     SEQUENCE {
  certificate               [0] Certificate OPTIONAL,
  encryptedCert             [1] EncryptedValue OPTIONAL,
  privateKey                [2] EncPrivKey OPTIONAL,
  publicationInfo           [3] PKIPublicationInfo OPTIONAL}
```

Figure 2.11 ASN.1 Specification of CertRepContent, CertResponse, and CertifiedKeyPair Elements.

which allows for multiple certificates to be revoked by a single transaction: however, the MISPC only supports messages that revoke a single certificate per transaction. Each **RevDetails** element gives specific information on the revocation of a given certificate (See Figure 2.12). **RevDetails** contains four fields, of which one is optional. The optional field *badSinceDate* indicates how long a certificate has been bad, in **GeneralizedTime** type format, to the best of the requesting entity's knowledge. The *certDetails* field specifies information about the certificate which is to be revoked (subject/issuer name, serial number, type of cryptographic algorithm, etc.). The *revocationReason* field is used to indicate why a certificate is being revoked (key compromised, issuer CA compromised, hold placed on certificate, etc.). In response to a certificate revocation request, a PKI message with a *body* field of type **RevRepContent** is generated (See Figure 2.13). **RevRepContent** is composed of three fields of which two are optional. The optional *crls* field is not required by MISPC components. The *status* field is used to indicate the status of the revocation request (accepted, rejected, etc.). The optional *revCerts* field is specified as a sequence of **CertIds** types. Each **CertId** contains the identification of a certificate that was revoked by the current revocation transaction. The final PKI message *body* type is **PKIConfirmContent** which has a NULL value indicating that all required information is contained in the *header* of the PKI message (See Figure 2.14).

```
RevReqContent       ::=     SEQUENCE OF RevDetails

RevDetails          ::=     SEQUENCE {
  certDetails               CertTemplate,
  revocationReason          ReasonFlags,
  badSinceDate              [0] GeneralizedTime OPTIONAL.
  crlEntryDetails           Extensions}
```

Figure 2.12 ASN.1 Specification of RevReqContent and RevDetails Elements.

```
RevRepContent       ::=     SEQUENCE {
  status        PKIStatusInfo,
  revCerts      [0] SEQUENCE OF CertId OPTIONAL,
  crls          [1] SEQUENCE OF CertificateList OPTIONAL }
```

Figure 2.13 ASN.1 Specification of RevRepContent Element.

```
PKIConfirmContent   ::=     NULL
```

Figure 2.14 ASN.1 Specification of PKIConfirmContent Element.

3 TEST SUITE DEVELOPMENT FOR THE MISPC

Limitations of Applying the TTF to MISPC

The application of the TTF to the MISPC will be done to demonstrate the limitations of framework. This demonstration will provide a guide as to what needs to be improved and enhanced when a new structured framework is proposed. As shown in Chapter 1, the TTF is applied to the valid input space, VIS_X, as defined by the specification of application X. For the MISPC, a general input space, IS_{MISPC}, can be defined using the ASN.1 data structures shown in Figures 2.3-2.14. The IS_{MISPC} can be further partitioned into a valid and invalid input space for the MISPC (VIS_{MISPC} and IIS_{MISPC}, respectively) based on requirements found within the MISPC such as fields defined as optional in the ASN.1 description but are required to be present by the MISPC. This preliminary partitioning of the MISPC's input space, shown in Figure 3.1, already starts to highlight some of the limitations of the TTF.

The first limitation of the TTF is that it only works with the defined valid input space of an application and ignores an application's invalid input space. When the TTF was proposed, it was correctly argued that nothing can be concluded about an application's behavior when unspecified or invalid inputs are presented. However, this limits the TTF from being used in the development of test suites that are not concerned about behavior when invalid or unspecified inputs are presented to an application. In addition, an argument can be made that considering only the valid input space does not explore a significant amount of the general input space that is defined by the application's specification. A simple example will be given focusing on the header data structure of the MISPC shown in Figure 2.6 to illustrate this argument. The header data structure consists of three required fields and eight optional fields. If all the fields are considered optional, that results in 2^{11}-1 or 2047 possible header formats, excluding the header which contains no fields. This set of possible header formats (F_{Header}) can be partitioned into the set of valid and invalid formats (VF_{Header} and IF_{Header}, respectively) based on the text of the specification. The specification states that, in addition to the three required fields, two of the optional fields are required by the MISPC. This results in VF_{Header} having $2^6 - 1$ or 63 formats

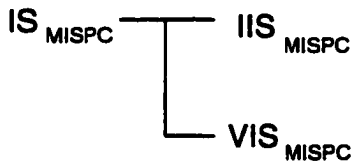

Figure 3.1 MISPC Input Space Partitioning.

and IF_{Header} having 1984 formats from F_{Header}, respectively. This means that the VF_{Header} makes up 3.08 percent of F_{Header} while IF_{Header} makes up 96.92 percent. This simple analysis only considers the formats based on a field's presence or absence and not the values contained by the fields, but it demonstrates how much of a well defined general input space is not explored by the TTF.

Another limitation of the TTF is its ability to express the interaction between different components defined by a specification. When the TTF was first proposed, this limitation was acknowledged. The MISPC provides an excellent example of the difficulty the TTF has capturing the interactive nature of a specified system. The MISPC is a collection of application specifications (CA, ORA, CH, Client) that compose an overall system specification. In addition to specifying each component, the MISPC describes how each component should interact with each other. An MISPC system can be partitioned into different sub–components in a similar fashion as the input space. Figure 3.2 shows one particular partitioning where the first level of division is based on the different components described by the MISPC. The next level of partitioning is provided by dividing a component into the roles that it may assume. This example concludes by partitioning each role into the transactions for which it can be a participant. The resulting partition is desired when developing a test suite with multiple components to capture the relationship of the different components to each other. This extension to the TTF shows the desired relationships but is not able to tie them back to the input space partitions. This makes it difficult to use the TTF in a unified way when developing test suites for a system composed of multiple components even though it can be applied to each component individually.

Finally, two fixable limitations of the TTF which can be corrected by modifying the framework will now be presented. The relationship between IS_{MISPC} and the general input space, IS, is not explicitly expressed by the TTF because it begins with the IS_{MISPC} input space. This limitation is overcome by providing an additional level of partitioning above the original partitioning and is shown in Figure 3.3. The modified partition shows that IS can be partitioned into IS_{MISPC} as defined by the MISPC and $IS_{\overline{MISPC}}$ which is not described by the MISPC. The significance of this refined partitioning is that

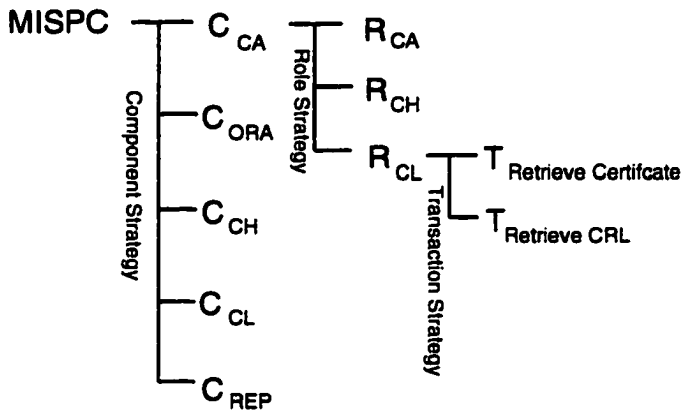

Figure 3.2 MISPC Component Partitioning.

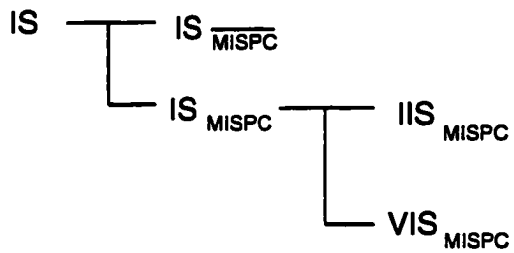

Figure 3.3 Modified MISPC Input Space Partitioning.

two distinct invalid input spaces are represented, $IS_{\overline{MISPC}}$ and IIS_{MISPC}, and their relationship is explicitly demonstrated in a global context. The invalid space of $IS_{\overline{MISPC}}$ is the set of inputs that do not conform to the MISPC general input space, IS_{MISPC}, as defined by the ASN.1 description of the MISPC data structures. This would include any ASN.1 data structure descriptions that could be created but are not explicitly mappable to the MISPC. The set of inputs that conform to IS_{MISPC} but do not meet the additional MISPC requirements are captured in IIS_{MISPC} partition. This would include MISPC ASN.1 data structures that are missing required fields. Another limitation of the TTF that can be corrected is the framework's ability to generate a sequence of inputs. This limitation is overcome not by modifying the framework but by using elements from the partitions to generate sequences. For example, two elements of the VIS_{MISPC} can be concatenated together to form an input sequence. After the IS_{MISPC} is sufficiently partitioned, it might be shown that both elements of the sequence

are certificate request messages. However, the framework is still unable to capture which component generates an input sequence. The sequence could be generated by a CH, ORA, CA or any combination of these MISPC components. Likewise, the input sequence can be presented to either a CA, ORA or both.

The limitations of the TTF that are not correctable require that a new framework be proposed. The new framework should address the invalid input space of a specification to allow for test suite development when reliability and assurance is required. In addition, the framework should be able to capture the interaction between components of a specification, so test suites for multi-component systems can be directly derived.

A Proposed Structured Framework for Test Suite Development

The proposed structured framework presented in this section will use the partitioning concepts of the TTF but apply them in a more general framework. The result should be a framework that can derive test suites for both multi-component systems and provide tests for completeness and robustness. In addition, the proposed structured framework will provide the context to understand the relationships of the input space and the components that compose a system. The proposed framework uses basic set notation to express the components of a test suite. The fundamental idea behind the proposed framework is that a set of input-output pairs defines a specific test suite. Once these input-output pairs are defined by the proposed framework, a specific test suite can be implemented following the results of the framework. The framework initially develops a test suite at the abstract level, then instantiates a specific test suite for implementation. By describing a test suite in these two levels, the framework can unify the abstract/theoretical and implementation aspects of test suite development. Where the abstract test suite is based on the theoretical testing techniques applied, the instantiation of a test suite depends on the implementation of the application to be tested. The result is that the framework provides a basis by which test data can be selected and generated for the implementation of a specific test suite.

The basic definition of the proposed framework begins using the variable y to denote a particular specification. The specification can be expressed by formal or informal specification techniques, but the framework views any specification as simply an abstract description. The framework denotes the abstract test suite, S, with respect to specification y by S(y). S(y) is a set of input-output pairs which is the fundamental view the proposed framework has of a test suite. By defining a test suite as a set of input-output pairs, it does not express how an input value is transformed into an output value which

makes the proposed framework a black-box testing strategy. The input–output pairs of S(y) are created from the set of abstract inputs and outputs defined by specification y and denoted by I(y) and O(y), respectively. The elements of the sets I(y) and O(y) can be a single abstract input/output or a sequence of them. The result is that the abstract test suite can initially be written as the expression shown in Equation 3.1. The proposed framework further defines two more sets that can be defined directly from a specification. The set C(y) is used to denote the set of abstract components defined by specification y. This allows the framework to easily express a specification composed of multiple components. In addition, the set of all abstract transactions specified by the specification y is denoted by T(y). An element of T(y) is an abstract transaction t which consists of a set of abstract messages used to implement the transaction it and is denoted by $T_M(t)$. An abstract message, m, of a transaction is either generated or processed by abstract components of the specification which are determined by the operations $M_G(m)$ and $M_P(m)$, respectively. $M_G(m)$ defines the set of abstract components that can generate the abstract message m. $M_P(m)$ defines the set of abstract components that can process the abstract message m. This allows the framework to capture the interactive relationships between the different abstract components of a specification. The framework provides a notation that allows attributes about an abstract message m to be expressed. The syntax $< message > . < valid/invalid >_{format} (y)$ is used to indicate the validity of an abstract message's format with respect to the specification y. The syntax can be extended to indicate the validity of an abstract message's values based on the specification y, $< message > . < valid/invalid >_{format} (y). < valid/invalid >_{value} (y)$. This notation allows the framework to express more information about the test data of an abstract test suite. For example, $m.invalid_{format}(y)$ indicates that an abstract message's format is invalid with respect to specification y–not all of the required fields as mandated by the specification y for the message are present. $m.valid_{format}(y).valid(y)$ would indicate that an abstract message's format is valid with respect to specification y and that the abstract values of the message are valid - all of the required fields as mandated by the specification y for the message are present and the abstract values contained in the fields are correct. Finally, two sets are defined to unify the abstract components, transactions, inputs, and outputs of the specification. The operation T(y,c) defines the set of abstract transactions for an abstract component c. This allows the set T(y) to be written as the expression shown in Equation 3.2. The operation $P_{Pair}(t, c)$ defines the set of input–output pairs for an abstract transaction t with respect to abstract component c. The operation $P_{Pair}(t, c)$ can be written as the expression shown in Equation 3.3. The expression for S(y) can be rewritten using the defined sets and operations as the expression in Equation 3.4. The expression shown in Equation 3.4 defines an abstract test suite for all

components of the specification but can be modified to reflect an abstract test suite for a particular component c as shown in Equation 3.5.

$$S(y) = \{[i, o] \mid i \in I(y) \text{ and } o \in O(y)\} \tag{3.1}$$

$$T(y) = \{t \mid t \in T(y, c),\ \forall c \in C(y)\} \tag{3.2}$$

$$P_{Pair}(t, c) = \{[i, o] \mid \exists m \in T_m(t) \text{ such that } (o = m \text{ AND } c \in M_C(m)) \text{ OR } (i = m \text{ AND } c \in M_P(m))\} \tag{3.3}$$

$$S(y) = \{[i, o] \mid [i, o] \in P_{Pair}(t, c),\ \forall t \in T(y, c).\ \forall c \in C(y)\} \tag{3.4}$$

$$S(y, c) = \{[i, o] \mid [i, o] \in P_{Pair}(t, c)\ , \forall t \in T(y, c)\} \tag{3.5}$$

A partitioning of the basic sets of the proposed framework can be done based on the required and optional functionality defined by the specification. If all the functionality described by the specification is required, then the optional set becomes empty. An abstract test suite, S(y), can be partitioned into test suites that test the required and optional functionalities of the specification, $S_R(y)$ and $S_O(y)$, respectively. This allows the abstract test suite to be written as the expression in Equation 3.6. The abstract test suites $S_R(y)$ and $S_O(y)$ are constructed using the set of abstract transactions, T(y). T(y) can be partitioned like the abstract test suite into the required and optional transactions of the specification, $T_R(y)$ and $T_O(y)$, respectively. Likewise, the operation T(y,c) used to define the set of abstract transactions for an abstract component can be refined into two operations, $T_R(y, c)$ and $T_O(y, c)$. The operation $T_R(y, c)$ is used to determine the required abstract transactions for abstract component c as described by the specification y while $T_O(y, c)$ determines the abstract component's optional transactions. This allows the abstract test suites $S_R(y)$ and $S_O(y)$ to be written as the expressions in Equations 3.7 and 3.8, respectively. The expressions shown in Equations 3.7 and 3.8 define required and optional abstract test suite for all components of the specification but can be modified to reflect a required and optional abstract test suite for a particular component c as shown in

Equations 3.9 and 3.10, respectively.

$$S(y) = S_R(y) \cup S_O(y) \tag{3.6}$$

$$S_R(y) = \{[i,o] \mid [i,o] \in P_{Pair}(t,c),\ \forall t \in T_R(y,c),\ \forall c \in C(y)\} \tag{3.7}$$

$$S_O(y) = \{[i,o] \mid [i,o] \in P_{Pair}(t,c),\ \forall t \in T_O(y,c),\ \forall c \in C(y)\} \tag{3.8}$$

$$S_R(y,c) = \{[i,o] \mid [i,o] \in P_{Pair}(t,c),\ \forall t \in T_R(y,c)\} \tag{3.9}$$

$$S_O(y,c) = \{[i,o] \mid [i,o] \in P_{Pair}(t,c),\ \forall t \in T_O(y,c)\} \tag{3.10}$$

The proposed framework has defined the structure for the development of an abstract test suite that will be used to instantiate a test suite for an implementation. The instantiation of the test suite requires some knowledge about the implementation of the application to be tested. In general, the amount of information known about an implementation varies and this variation will affect the instantiation of a test suite and the way it can be implemented. The framework uses z to denote a specific implementation of an application. The set of logical inputs and outputs of the implementation z is denoted by $I_I(z)$ and $O_I(z)$, respectively. The behavior, B, for an implementation of an application can be described by its input-output pairs and can be written as the expression shown in Equation 3.11. The elements of the sets $I_I(z)$ and $O_I(z)$ can be a single instance of an input/output or a sequence of them. The framework can use the implementation z to define two more sets. The set $C_I(z)$ is used to denote the set of components instantiated by the implementation of application z. The set $T_I(z)$ is the set of transactions instantiated by the implementation z. An element of $T_I(z)$ is an instantiation of a transaction t' which consists of a set of messages used to implement the transaction and is denoted by $T_{I_M}(t')$. A message, m', of a transaction is either generated or processed by the implemented components of the specification and can be determined using the operations $M_{I_G}(m')$ and $M_{I_P}(m')$, respectively. $M_{I_G}(m')$ defines the set of instantiated components that can generate the instantiated message m'. $M_{I_P}(m')$ defines the set of instantiated components that can process the instantiated message m'. The proposed framework provides a notation that allows attributes about a message m' or a set of messages M' to be expressed. In general, the syntax takes the following form: $< message >'$

. $<$ *valid/invalid* $>_{<attribute>}$ (y). The syntax $<$ *message* $>'$. $<$ *valid/invalid* $>_{format}$ (y) is used to indicate the validity of instantiated message formats based on the specification y. The syntax can be extended to indicate the validity of instantiated message values based on the specification y, $<$ *message* $>'$. $<$ *valid/invalid* $>_{format}$. $<$ *valid/invalid* $>_{value}$ (y). This notation allows the framework to express more detailed information about the instantiation of the test data for a test suite that is to be implemented. For example, $m'.invalid_{format}(y)$ indicates that an instantiated message's format is invalid with respect to specification y–not all of the required fields as mandated by the specification y for the message are present. $m'.valid_{format}.valid_{value}(y)$ would indicate that an instantiated message's format is valid with respect to specification y and that the values of the message are valid–all of the required fields as mandated by the specification y for the message are present and the values contained in the fields are correct. Finally, two functions are defined to unify instantiations of components, inputs, outputs, and transactions implemented by an application. The operation $T_I(z, c')$ defines the set of transactions implemented by the instantiated component c'. This allows the set $T_I(z)$ to be written as the expression shown in Equation 3.12. The operation $P_{I_{Pair}}(t', c')$ defines the set of input–output pairs for an implemented transaction t' with respect to an implemented component c' and can be written as the expression shown in Equation 3.13.

$$B(z) = \{[i', o'] \mid i' \in I_I(z) \text{ and } o' \in O_I(z)\} \tag{3.11}$$

$$T_I(z) = \{t' \mid t' \in T_I(z, c'),\ \forall c' \in C_I(z)\} \tag{3.12}$$

$$P_{I_{Pair}}(t', c') = \{[i', o'] \mid \exists m' \in T_{I_M}(t') \text{ such that } (o = m' \text{ AND } c' \in M_{I_C}(m)) \text{ OR } (i = m' \text{ AND } c' \in M_{I_P}(m'))\} \tag{3.13}$$

The relationship of the implementation z to a given specification y begins by defining the operation $C_I(z, y)$, which is used to determine the set of all components instantiated by implementation z with respect to specification y as shown in Equation 3.14. When the set is empty, the implementation does not implement any component described by the specification. If $C_I(z, y)$ is not empty, then a relationship between the specification y and $T_I(z)$ exists and can be expressed as shown in Equation 3.15. It should be noted the intersection operator, $\cap$, in Equation 3.14 and Equation 3.15 is performing the

intersection on two different types of sets: the abstract set of components/transactions specified by y and the set of components/transaction implemented by z. In addition, the intersection operator is performing a mapping between the specification z and implementation y. Once the relationship between the implementation and specification is established, an instantiation of a test suite can be defined for an implementation z with respect to specification y as shown in Equation 3.16. Equation 3.16 can be refined to reflect an instantiation of a test suite for a particular component c' of implementation z as shown in Equation 3.17.

$$C_I(z,y) = \{c' \mid c' \in (C_I(z) \cap C(y))\} \tag{3.14}$$

$$T_I(z,y) = \{t' \mid t' \in (T_I(z,c') \cap T(y)),\ \forall c' \in C_I(z,y)\} \tag{3.15}$$

$$S_I(z,y) = \{[i',o'] \mid \exists t' \in T_I(z,y) \textit{ such that } [i',o'] \in P_{I_{Pair}}(t',c'),\ \forall c' \in C_I(z,y)\} \tag{3.16}$$

$$S_I(z,c',y) = \{[i',o'] \mid \exists t' \in T_I(z,y) \textit{ such that } [i',o'] \in P_{I_{Pair}}(t',c')\} \tag{3.17}$$

A partitioning of the instantiation of the test suite can be done based on the required and optional functions defined by the specification y. This is similar to the partitioning done for the abstract test suite in Equations 3.7–3.10. The partitioning of $T_I(z)$ into required and optional transactions implemented by application z with respect to specification y, $T_{I_R}(z,y)$ and $T_{I_O}(z,y)$, respectively, allows the instantiation of the test suite to be partitioned. Equations 3.18–3.19 show how the partitions can be written as expressions. A further refinement to Equations 3.18–3.19 can be done that allows for the proposed framework to focus an instantiated test suite on a particular component c' of implementation z. Equations 3.20–3.21 show how the refined partitions can be written as expressions.

$$S_{I_R}(z,y) = \{[i',o'] \mid \exists t' \in T_{I_R}(z,y) \textit{ such that } [i',o'] \in P_{I_{Pair}}(t',c'),\ \forall c' \in C_I(z,y)\} \tag{3.18}$$

$$S_{I_O}(z,y) = \{[i',o'] \mid \exists t' \in T_{I_O}(z,y) \textit{ such that } [i',o'] \in P_{I_{Pair}}(t',c'),\ \forall c' \in C_I(z,y)\} \tag{3.19}$$

$$S_{I_R}(z,c',y) = \{[i',o'] \mid \exists t' \in T_{I_R}(z,y) \textit{ such that } [i',o'] \in P_{I_{Pair}}(t',c')\} \tag{3.20}$$

$$S_{I_O}(z,c',y) = \{[i',o'] \mid \forall t' \in T_{I_O}(z,y) \textit{ such that } [i',o'] \in P_{I_{Pair}}(t',c')\} \tag{3.21}$$

The development of a test suite using the proposed structured framework is done in two parts. First, the abstract test suite can be developed based on a specification of an application without specific implementation information. Once some implementation information of the specified application is known, an instance of the test suite can be derived and implemented. The proposed framework can be used to express properties a test suite may wish to demonstrate. For example, the test suite may wish to demonstrate an implementation of an application conforms to a particular specification. A instantiation of a conformance test suite can be expressed by modifying Equation 3.18 as shown in Equation 3.22 and using Equation 3.19. Equation 3.23 states that an instance of a test suite which is used to determine if an implementation z conforms to specification y should contain the input-output pairs for all the transactions required by the specification, $CS_{I_R}(z,y)$, and only the optional transactions implemented by the application being tested, $S_{I_O}(z,y)$. Using the notation for the attributes of an abstract message, more information about the instantiation of a conformance test can be expressed. Equations 3.19 and 3.22 can be modified to indicate that valid and invalid input formats of the transactions can be included and are shown in Equations 3.24 and 3.25, respectively. This results in the instantiation of the conformance test suite of Equation 3.23 to include input-output pairs that contain messages with valid and invalid formats. An example of the framework being applied to MISPC is given in the next section to demonstrate how the framework can be used to derive an abstract test suite.

$$CS_{I_R}(z,y) = \{[i',o'] \mid [i',o'] \in P_{I_{Pair}}(t',c'),\ \forall t' \in T_{I_R}(z,y),\ \forall c' \in C_I(z,y)\} \tag{3.22}$$

$$CS_I(z,y) = \{[i',o'] \mid [i',o'] \in [CS_{I_R}(z,y) \cup S_{I_O}(z,y)]\} \tag{3.23}$$

$$S_{I_O}(z,y) = \{[i',o'] \mid \exists [i',o'] \in P_{I_{Pair}}(t',c'), \\ \forall t' \in T_{I_O}(z,y),\ \forall c' \in C_I(z,y)\ such\ that \\ i'.valid_{format}(y)\ and\ i'.invalid_{format}(y)\} \tag{3.24}$$

$$CS_{I_R}(z,y) = \{[i',o'] \mid \exists [i',o'] \in P_{I_{Pair}}(t',c'), \\ \forall t' \in T_{I_R}(z,y),\ \forall c' \in C_I(z,y)\ such\ that \\ i'.valid_{format}(y)\ and\ i'.invalid_{format}(y)\} \tag{3.25}$$

Application of the Proposed Structured Framework to the MISPC

The proposed framework will be applied to the specification of the MISPC that was described in Chapter 2. This example demonstrates how the proposed framework is used in the development of a test suite. In Chapter 4, an instantiation of a test suite for an implementation of a Certificate Authority (CA) component specified by the MISPC will be created using the results from this section. The proposed framework starts by defining the basic sets which will be used to define the abstract test suite for the MISPC, S(MISPC). The set of components and transactions, C(MISPC) and T(MISPC) respectively, are enumerated by the MISPC. The elements of the C(MISPC) consist of the Certificate Authority (CA), Organization Registration Authority (ORA), Certificate Holder (CH), Client (CL), and Repository (REP) and can be written as the expression shown in Equation 3.26.

The proposed framework defines the set of transactions, T(MISPC), using the MISPC. The MISPC defines the following transactions: certificate requests by a CH (self_registration), certificate requests using an ORA (ora_registration), certificate requests using a PKCS#10 request message (pkcs10_registration), renewal of certificates (cert_renewal), the retrieval and posting of certificates and CRL (cert_retrieval, cert_post, crl_retrieval, crl_post), the generation and validation of signatures (signature_validation, signature_generation), and validation of certifcate paths (path_validation). The set T(MISPC) can be written as the expression shown in Equation 3.27. The MISPC defines the messages that make up the 12 transactions both formally using ASN.1 and informally using an English language description. The transactions that have messages defined formally using ASN.1 are: ora_registration, cert_revocation,

self_registration, cert_renewal, and pkcs10_registration. The ASN.1 defined data structures for the messages used by these transactions are shown in Figures 2.5-2.14. The ora_registration transactions include the following messages: a certificate request message from CH to the ORA (ch_ora_cr), a certificate request message from ORA to CA (ora_ca_cr), a certificate reply message from CA to ORA (ca_ora_cp), and optionally a certificate reply message from CA to CH (ca_ch_cp). The rest of the transactions consist of two ASN.1 defined messages: a transaction request and reply message. The cert_revocation transaction consists of a revocation request and reply message, rr and rp respectively. The self_registration transaction consists of a certificate request and reply message, self_cr and self_cp respectively. The cert_renewal transaction consists of a certificate renewal request and reply message, renewal_cr and renewal_cp respectively. The pkcs10_registration consists of a pkcs#10 certificate request and reply message, pkcs10_cr and pkcs10_cp respectively. The transactions that have messages implicitly defined are: cert_retrieval, crl_retreival, cert_post, crl_post, path_validation, signature_validation, and signature_generation. The signature_validation transaction consists of a signed message or certificate (signed_mess_s and signed_cert_s, respectively) and the result of the validation (sign_result). The path_validation transaction consists of a signed message or certificate (signed_mess_p and signed_cert_p, respectively) and the result of the validation (path_result). The signature_generation transaction consists of an unsigned message or certificate (unsigned_mess and unsigned_cert, respectively) and the resulting signed message or certificate (signed_mess_g and signed_cert_g, respectively). The cert_retrival, crl_retrival, cert_post, and crl_post transactions are also not addressed using ASN.1 by the MISPC and involve the repository which the specification does not completely describe. The cert_retrieval and crl_retrieval transactions consist of a retrieval request message (cert_query and crl_query, respectively) and retrieval reply message (cert_query_reply and crl_query_reply, respectively). The cert_post and crl_post transactions can be asynchronous and synchronous in nature. If the cert_post and crl_post transactions are asynchronous in nature, they consist of a single message (cert_post_mess and crl_post_mess, respectively). If the cert_post and crl_post transactions are synchronous in nature, they consist of two messages: a post request message (cert_post_req and crl_post_req, respectively) and post reply messages (cert_post_rep and crl_post_rep, respectively). The sets of messages for all the transactions can be written as the expressions shown in Equations 3.28–Equations 3.41.

$$C(MISPC) = \{CA, ORA, CH, CL, REP\} \tag{3.26}$$

$$T(MISPC) = \{ora\_registration, cert\_revocation, self\_registration, cert\_renewal, pkcs10\_registration, cert\_retrieval, crl\_retrieval, cert\_post, crl\_post, path\_validation, signature\_validation, signature\_generation\} \quad (3.27)$$

$$T_M(ora\_registration) = (CH\_ORA\_CR \cup ORA\_CA\_CR \cup CA\_ORA\_CP \cup CA\_CH\_CP) \quad (3.28)$$

$$T_M(cert\_revocation) = (RR \cup RP) \quad (3.29)$$

$$T_M(self\_registration) = (SELF\_CR \cup SELF\_CP) \quad (3.30)$$

$$T_M(cert\_renewal) = (RENEWAL\_CR \cup RENEWAL\_CP) \quad (3.31)$$

$$T_M(pkcs10\_registration) = (PKCS10\_CR \cup PKCS10\_CP) \quad (3.32)$$

$$T_M(cert\_retrieval) = (CERT\_QUERY \cup CERT\_QUERY\_REPLY) \quad (3.33)$$

$$T_M(crl\_retrieval) = (CRL\_QUERY \cup CRL\_QUERY\_REPLY) \quad (3.34)$$

$$T_M(path\_validation) = (SIGNED\_MESS\_P \cup SIGNED\_CERT\_P \cup PATH\_RESULT) \quad (3.35)$$

$$T_M(signature\_validation) = (SIGNED\_MESS\_S \cup SIGNED\_CERT\_S \cup SIGN\_RESULT) \quad (3.36)$$

$$T_M(signature\_generation) = (UNSIGNED\_MESS \cup UNSIGNED\_CERT \cup SIGNED\_MESS\_G \cup SIGNED\_CERT\_G) \quad (3.37)$$

$$T_M(cert\_post_{asynchronous}) = CERT\_POST\_MESS \quad (3.38)$$

$$T_M(crl\_post_{asynchronous}) = CRL\_POST\_MESS \quad (3.39)$$

$$T_M(cert\_post_{synchronous}) = (CERT\_POST\_REQ \cup CERT\_POST\_REP) \quad (3.40)$$

$$T_M(crl\_post_{synchronous}) = (CRL\_POST\_REQ \cup CRL\_POST\_REP) \quad (3.41)$$

The proposed framework will continue to be applied to the MISPC but will focus on the CA component because it is the component for which an instantiation of the abstract test suite will be implemented in Chapter 4. Applying the proposed framework similarly to the ORA, CH, CL components of the MISPC would allow an abstract test suite to be developed for them which can be combined to form a complete abstract test suite for an MISPC system. The proposed framework allows for the abstract test suite for the CA component to be written as the expression in Equation 3.42 which is a modification of Equation 3.6. Equation 3.42 shows that the abstract test suite for the CA component can be partitioned into test suites for the required and optional transactions/functionality defined by the MISPC, $S_R(MISPC, CA)$ and $S_O(MISPC, CA)$, respectively. Of the two abstract test suites, $S_O(MISPC.CA)$ provides a better initial starting point than $S_R(MISPC, CA)$. $S_O(MISPC, CA)$ is the set of input-output pairs for all optional transactions of the CA component and is shown in Equation 3.43. The set of optional transactions for the CA component, $T_O(MISPC, CA)$, consists of the pkcs10_registration and can be written as the expression shown in Equation 3.44. Using Equation 3.44, $S_O(MISPC, CA)$ can be simplified to the expression shown in Equation 3.45. The message operations $M_G(m)$ and $M_P(m)$ can be applied to the messages, pkcs10_cr and pkcs10_cp, which make up the pkcs10_registration transaction and are shown in Equation 3.46. Equation 3.46 demonstrates that a CA must be able to generate a pkcs10_cp message as output and accept pkcs10_cr message as

input. The MISPC further states that if a CA is presented a pkcs10_cr message, it should generate a pkcs10_cp message. This results in the input-output pairs shown in Equation 3.47 and can be used to simplify $S_O(MISPC, CA)$ to the expression shown in Equation 3.48.

$$S(MISPC, CA) = S_R(MISPC, CA) \cup S_O(MISPC, CA) \tag{3.42}$$

$$S_O(MISPC, CA) = \{[i, o] \mid [i, o] \in P_{Pair}(t, CA),\ \forall t \in T_O(MISPC, CA)\} \tag{3.43}$$

$$T_O(MISPC, CA) = \{pkcs10\_registration\} \tag{3.44}$$

$$S_O(MISPC, CA) = \{[i, o] \mid [i, o] \in P_{Pair}(pkcs10\_registration, CA)\} \tag{3.45}$$

$$\begin{aligned} CA \in M_G(pkcs10\_cp) &\Rightarrow o \in (M_G^{-1}(CA) \cap T_M(pkcs10\_registration)) \\ CA \in M_P(pkcs10\_cr) &\Rightarrow i \in (M_P^{-1}(CA) \cap T_M(pkcs10\_registration)) \end{aligned} \tag{3.46}$$

$$P_{Pair}(pkcs10\_registration, CA) = \{[i, o] \mid i \in PKCS10\_CR\ AND\ o \in PKCS10\_CP\} \tag{3.47}$$

$$S_O(MISPC, CA) = PKCS10\_CR \times PKCS10\_CP \tag{3.48}$$

In a similar fashion, the abstract test suite for the required transactions, $S_R(MISPC, CA)$, of an MISPC CA component can be derived. $S_R(MISPC, CA)$ is the set of input-output pairs for all required transactions of the CA component and is shown in Equation 3.49. The set of required transactions for the CA component, $T_R(MISPC, CA)$, consists of ora_registration, cert_revocation, self_registration, cert_renewal, cert_post, and crl_post which can be written as the expression shown in Equation 3.50. Using Equation 3.50, $S_O(MISPC, CA)$ can be rewritten as to the expression shown in Equation 3.51. Applying the message operations, $M_G(m)$ and $M_P(m)$, to the messages (shown in Equation 3.28) used by the ora_registration tranaction results in the expressions shown in Equation 3.52. Equation 3.52

demonstrates that a CA must be able to generate ca_ora_cp and ca_ch_cp messages as output and accept ora_ca_cr messages as input. The MISPC states that if a CA receives an ora_ca_cr message, it should respond with ca_ora_cp and possibly a ca_ch_cp message and this results in the set of input-output pairs shown by Equation 3.53[1].

$$S_R(MISPC, CA) = \{[i, o] \mid [i, o] \in P_{Pair}(t, CA),\ \forall t \in T_R(MISPC, CA)\} \tag{3.49}$$

$$T_R(MISPC, CA) = \{ora\_registration, cert\_revocation, self\_registration, cert\_renewal, cert\_post, crl\_post\} \tag{3.50}$$

$$\begin{aligned} S_R(MISPC, CA) = \{[i, o] \mid [i, o] \in & P_{Pair}(ora\_registration, CA) \cup \\ & P_{Pair}(cert\_revocation, CA) \cup P_{Pair}(self\_registration, CA) \cup \\ & P_{Pair}(cert\_renewl, CA) \cup P_{Pair}(cert\_post, CA) \cup \\ & P_{Pair}(crl\_post, CA)\} \end{aligned} \tag{3.51}$$

$$\begin{aligned} CA \in M_G(ca\_ora\_cp) &\Rightarrow o \in (M_G^{-1}(CA) \cap T_M(ora\_registration)) \\ CA \in M_G(ca\_ch\_cp^*) &\Rightarrow o \in (M_G^{-1}(CA) \cap T_M(ora\_registration)) \\ CA \in M_P(ora\_ca\_cr) &\Rightarrow i \in (M_P^{-1}(CA) \cap T_M(ora\_registration)) \end{aligned} \tag{3.52}$$

$$P_{Pair}(ora\_registration, CA) = \{[i, o] \mid i \in ORA\_CA\_CR\ AND\ o \in CA\_ORA\_CP \cup CA\_CH\_CP\} \tag{3.53}$$

Applying the message operations to the messages (shown in Equation 3.29) used by the cert_revocation transaction results in the expressions shown in Equation 3.54. Equation 3.54 demonstrates that a CA

[1] * indicates that the input-output is optional.

must be able to generate rp messages as output and accept rr messages as input. The MISPC states that if a CA receives an rr message, it should respond with an rp message and this results in the set of input-output pairs shown by Equation 3.55.

$$CA \in M_G(rp) \Rightarrow o \in (M_G^{-1}(CA) \cap T_M(cert\_revocation)) \quad (3.54)$$
$$CA \in M_P(rr) \Rightarrow o \in (M_P^{-1}(CA) \cap T_M(cert\_revocation))$$

$$P_{Pair}(cert\_revocation, CA) = \{[i, o] \mid i \in RR \ AND \ o \in RP\} \quad (3.55)$$

Applying the message operations to the messages (shown in Equation 3.30) used by the self_registration transaction results in the expressions shown in Equation 3.56. Equation 3.56 demonstrates that a CA must be able to generate self_cp messages as output and accept self_cr messages as input. The MISPC states that if a CA receives a self_cr message, it should respond with a self_cp message and results in the set of input-output pairs shown by Equation 3.57.

$$CA \in M_G(self\_cp) \Rightarrow o \in (M_G^{-1}(CA) \cap T_M(self\_registration)) \quad (3.56)$$
$$CA \in M_P(self\_cr) \Rightarrow i \in (M_P^{-1}(CA) \cap T_M(self\_registration))$$

$$P_{Pair}(self\_registration, CA) = \{[i, o] \mid i \in SELF\_CR \ AND \ o \in SELF\_CP\} \quad (3.57)$$

Applying the message operations to the messages (shown in Equation 3.31) used by the cert_renewal transaction results in the expressions shown in Equation 3.58. Equation 3.58 demonstrates that a CA must be able to generate renewal_cp messages as output and accept renewal_cr messages as input. The MISPC states that if a CA receives a renewal_cr message, it should respond with a renewal_cp message and results in the set of input-output pairs shown by Equation 3.59.

$$CA \in M_G(renewal\_cp) \Rightarrow o \in (M_G^{-1}(CA) \cap T_M(cert\_renewal)) \quad (3.58)$$
$$CA \in M_P(renewal\_cr) \Rightarrow i \in (M_P^{-1}(CA) \cap T_M(cert\_renewal))$$

$$P_{Pair}(cert\_renewal, CA) = \{[i, o] \mid i \in RENEWAL\_CR \ AND \ o \in RENEWAL\_CP\} \quad (3.59)$$

The cert_post and crl_post transactions can either be asynchronous or synchronous in nature. Applying the message operations to the messages (shown in Equation 3.38) used by the $cert\_post_{asynchronous}$ transaction results in the expressions shown in Equation 3.60. Equation 3.60 demonstrates that a CA must be able to generate cert_post_mess messages as output; however, the MISPC does not state under what conditions a CA should post a certificate. This results in a set of input-output pairs shown in Equation 3.61 which is not completely specified. The γ is used to indicate that an unknown input is used to generate the cert_post_mess. If the cert_post transaction is synchronous in nature, applying the message operations to the messages (shown in Equation 3.40) used by $cert\_post_{synchronous}$ results in the expressions shown in Equation 3.62. Equation 3.62 demonstrates that a CA must be able to generate cert_post_req message as output and accept a cert_post_rep messages as input. Again, the MISPC does not state under what conditions a CA should post a certificate which results in a set of input-output pairs shown in Equation 3.63 which is not completely specified. The γ is used to indicate that an unknown input is used to generate the cert_post_req. The κ is used to indicate that an unknown output is produced when the CA is given a cert_post_rep message.

$$CA \in M_G(cert\_post\_mess) \Rightarrow o \in (M_G^{-1}(CA) \cap T_M(cert\_post_{asynchronous})) \quad (3.60)$$

$$P_{Pair}(cert\_post_{asynchronous}, CA) = \{[i, o] \mid i = \gamma \ AND \ o \in CERT\_POST\_MESS\} \quad (3.61)$$

$$CA \in M_G(cert\_post\_req) \Rightarrow o \in (M_G^{-1}(CA) \cap T_M(cert\_post_{synchronous})) \quad (3.62)$$
$$CA \in M_P(cert\_post\_rep) \Rightarrow i \in (M_P^{-1}(CA) \cap T_M(cert\_post_{synchronous}))$$

$$P_{Pair}(cert\_post_{synchronous}, CA) = \{[i, o] \mid i \in (\gamma \cup CERT\_POST\_REP) \ AND \quad (3.63)$$

$$o \in (CERT\_POST\_REQ \cup \kappa)\}$$

Applying the message operations to the messages (shown in Equation 3.39) used by the $crl\_post_{asynchronous}$ transaction results in the expressions shown in Equation 3.64. Equation 3.64 demonstrates that a CA must be able to generate crl_post_mess messages as output; however, the MISPC does not state under what conditions a CA should post a CRL. This results in a set of input–output pairs shown in Equation 3.65 which is not completely specified. Again, γ is used to indicate that an unknown input is used to generate the crl_post_mess. If the crl_post transaction is synchronous in nature, applying the message operations to the messages (shown in Equation 3.41) used by $crl\_post_{synchronous}$ results in the expressions shown in Equation 3.66. Equation 3.66 demonstrates that a CA must be able to generate a crl_post_req message as output and accept a crl_post_rep messages as input. Again, the MISPC does not state under what conditions a CA should post a CRL which results in a set of input–output pairs shown in Equation 3.67 which is not completely specified. The γ is used to indicate that an unknown input is used to generate the crl_post_req. The κ is used to indicate that an unknown output is produced when the CA is given a crl_post_rep message.

$$CA \in M_G(crl\_post\_mess) \Rightarrow o \in (M_G^{-1}(CA) \cap T_M(crl\_post_{asynchronous})) \tag{3.64}$$

$$P_{Pair}(crl\_post_{asynchronous}, CA) = \{[i, o] \mid i = \gamma \ AND \ o \in CRL\_POST\_MESS]\} \tag{3.65}$$

$$CA \in M_G(crl\_post\_req) \Rightarrow o \in (M_G^{-1}(CA) \cap T_M(cert\_post_{asynchronous})) \tag{3.66}$$
$$CA \in M_P(crl\_post\_rep) \Rightarrow i \in (M_P^{-1}(CA) \cap T_M(cert\_post_{asynchronous}))$$

$$P_{Pair}(crl\_post_{synchronous}, CA) = \{[i, o] \mid i \in (\gamma \cup CRL\_POST\_REP) \ AND \tag{3.67}$$
$$o \in (CRL\_POST\_REQ \cup \kappa)\}$$

Using the elements from the sets defined in Equations 3.53, 3.55, 3.57, 3.59, 3.61, 3.63, 3.65, and 3.67, $S_R(MISPC, CA)$ can be rewritten as the set of input-output pairs shown in Equation 3.68. † indicates that the input-output pairs are only necessary when using the synchronous form of the cert_post and crl_post transactions. ‡ indicates that the input-output pairs are only necessary when using the asynchronous form of the cert_post and crl_post transactions.

$$\begin{aligned} S_R(MISPC,CA) = \{&(ORA\_CA\_CR \times CA\_ORA\_CP \times CA\_CH\_CP^*), \\ &(SELF\_CR \times SELF\_CP) \cup (RR \times RP) \cup (RENEWAL\_CR \times RENEWAL\_CP), \\ &(\gamma \times CERT\_POST\_MESS)^\ddagger \cup (\gamma \times CERT\_POST\_REQ)^\dagger \cup CERT\_POST\_REP \times \kappa)^\dagger \cup \\ &(\gamma \times CRL\_POST\_MESS)^\ddagger \cup (CRL\_POST\_REP \times \kappa)^\dagger \cup (\gamma \times CRL_POST\_REQ)^\dagger\} \end{aligned} \tag{3.68}$$

Using Equations 3.68 and 3.48, the overall MISPC CA component abstract test suite of Equation 3.42 can be rewritten as Equation 3.69. A more descriptive MISPC CA component abstract test suite, $S_{more}(MISPC, CA)$, can be specified which indicates both valid and invalid abstract message formats for inputs to the MISPC CA component are part of the test suite, and it can be written as Equation 3.70. The MISPC CA abstract test suite allows the interactions between components to be explicitly demonstrated by applying the $M_G(m)$ and $M_P(m)$ operations on the abstract inputs and outputs of the input-output pairs of the test suite. For example, $M_G(rr) = \{ORA, CH\}$ and $M_P(rp) = \{CA\}$. for the input-output pair [rr.rp] which makes up a certificate revocation transaction. shows that a CA receives rr messages from and sends rp messages to ORAs and CHs. This abstract test suite of the MISPC CA component derived will be used to derive an instantiation of the test suite to be implemented in section 2 of Chapter 4.

$$\begin{aligned} S(MISPC,CA) = \{&(ORA\_CA\_CR \times CA\_ORA\_CP \times CA\_CH\_CP^*) \cup \\ &(SELF\_CR \times SELF\_CP) \cup (RR \times RP) \cup (RENEWAL\_CR \times RENEWAL\_CP) \cup \\ &(\gamma \times CERT\_POST\_MESS)^\ddagger \cup (\gamma \times CERT\_POST\_REQ)^\dagger \cup (CERT\_POST\_REP \times \kappa)^\dagger \cup \\ &(\gamma \times CRL\_POST\_MESS)^\ddagger \cup (CRL\_POST\_REP \times \kappa)^\dagger \cup \\ &(\gamma \times CRL\_POST\_REQ)^\dagger \cup (PKCS10\_CR \times PKCS10\_CP)\} \end{aligned} \tag{3.69}$$

$$S_{more}(MISPC, CA) \subset S(MISPC, CA)\ AND$$
$$\exists (i, o) \in S_{more}(MISPC, CA)\ such\ that\ i.invalid_{format}(MISPC)\ AND \tag{3.70}$$
$$\exists (i, o) \in S_{more}(MISPC, CA)\ such\ that\ i.valid_{format}(MISPC)$$

4 APPLICATION OF THE PROPOSED FRAMEWORK TO THE MISPC

Overview of the MISPC Reference Implementation

A reference implementation of the MISPC was developed for NIST by CygnaCom Solutions. The reference implementation provides a proof of concept for the MISPC by having a working implementation of a Certificate Authority (CA), Organization Registration Authority (ORA), and Client based on the specification. Both the source and executable codes for the components implemented were provided, as well as a simple stand-alone application which generates and verifies signatures. The components of the reference implementation can operate together as a PKI by adding a repository service application that uses the Lightweight Directory Access Protocol (LDAP) to store issued certificates and certificate revocation lists (CRLs), an e-mail server to transport MISPC messages, and the use of a floppy disk to transport out-of-bound information (such as shared secrets) between components. An overview of the configuration of this MISPC PKI implementation can be seen in Figure 4.1. The subject to be exercised by the test suite developed from the newly proposed framework will be the CA component of the MISPC reference implementation. This section will provide a technical description of how the various components of the MISPC reference implementation were implemented [12]. This will provide the background needed to understand the following test suite implementation issues which will be described later in this chapter: how the test suite subject differs from the original MISPC reference implementation of the component, how the test suite will interface with its test subject, and where the perimeter of the test suite is located.

The components of the reference implementation were specifically designed to run on personal computers with a TCP/IP network connection running the Windows 95 operating system. The primary programming language used to implement the MISPC components was C++; however, C was used to implement some of the underlying functions used by the components. In addition to C/C++, the following development libraries were used: Microsoft Foundation Class (MFC), Object Linking and

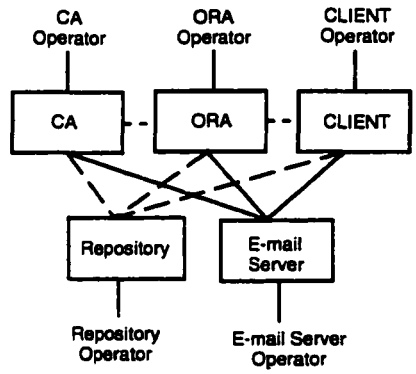

Figure 4.1 MISPC Reference Implementation PKI Configuration.

Embedding (OLE), and Component Object Model (COM) [40, 33, 6, 34]. The MFC library offers a framework for developing an application by providing a collection of C++ classes for Graphical User Interfaces (GUIs), input devices, message handlers, etc. OLE is a set of standards for building software components that are connectable. The standard used to define interfaces that allow for connections between OLE components is the COM specification. The COM interfaces allow for connections to be made between different applications with OLE components on the same or different hosts (i.e. inter-process communication) as well as OLE components within a single application. Another important feature of OLE components is that a component can set and obtain variables that it uses by registering itself and the variables in the system registry. The system registry stores system-wide state information for the Windows operating system (95, NT, and 3.1). Typically, an OLE component will register configuration variables such as directory paths, host names, IP addresses, user names, etc. The registered variables can be modified by an application program that uses the system registry application programming interface (API). One stand-alone program that interfaces with the system registry API is REGEDIT.EXE. REGEDIT.EXE is able to access and display the entries of the system registry and is provided with the standard installation of a Microsoft Windows operation system. REGEDIT.EXE allows a user to manually inspect and modify entries of the system registry.

One or more of the MISPC reference implementation components uses the following libraries in the form of Dynamically Linked Libraries (DLLs) or OLE components in the form of an OLE control: gcs.dll[1], certific.dll, sldap32.dll, certpath.dll, CygnaComSMTP.dll, and CygnaComPOP3.dll. These libraries provide services required for the components of the MISPC reference implementation and will

[1]The source code for this DLL was not provided for distribution with the MISPC reference implementation due to U.S. Government Export Control Regulations.

be reviewed individually.

The gcs.dll is a DLL based on the Generic Cryptographic Services (GCS) standard and provides the cryptographic functionality required by the different components of the MISPC reference implementation. The GCS standard provides cryptographic services to applications by defining a standard API for cryptographic services such as providing confidentiality, integrity, and authenticity of data [32]. The GCS DLL provides data integrity for the components of the MISPC reference implementation by using the Standard Hashing Algorithm (SHA-1) and Data Encryption Standard-Message Authentication Code (DES-MAC). For data authentication, the GCS DLL uses the Digital Signature Algorithm (DSA) public key cryptographic system. The GCS DLL stores the private keys in an encrypted format and requires authentication to access the keys.

The certific32.dll provides the Abstract Syntax Notation One (ASN.1) elements required by all components of the MISPC reference implementation. In addition to specifying data structures using a well defined notation, ASN.1 specifies how the data structures should be represented when translated into streams of bytes [37]. The certific32.dll defines specific C data structures for the ASN.1 data structures specified by the MISPC. In addition, it provides the functions to encode and decode the defined C data structures following the Distinguished Encoding Rules (DER) specified by ASN.1.

The CygnaComSMTP.dll and CygnaComPOP3.dll provide the transport mechanism used to exchange MISPC-defined PKI messages between the components of the reference implementation. The MISPC reference implementation was designed to transport PKI messages as Multipurpose Internet Mail Exchange (MIME) encoded attachments of electronic mail (e-mail) messages. MIME encoding specifies how e-mail messages should be formatted to minimize the possibility that their contents (for example images, audio, executable code, etc.) will be corrupted as they are transported through different e-mail systems [15]. The CygnaComSMTP.dll is used to send the e-mail messages generated by the components to an e-mail server. Similarly, CygnaComPOP3.dll is used to retrieve e-mail messages for a component from an e-mail server.

The certpath.dll is only used by the simple stand-alone application that generates and verifies signatures [11]. Given a list of certificates (or certificate chain), the DLL verifies that a valid chain of certificates exists between a certificate issued by a CA trusted by the verifier and a certificate used to generate a valid signature. For example, CA1 is trusted by the verifier but CA 2 is not directly trusted by the verifier. If CA1 issues a certificate to CA2, then a valid certificate path could exisit between the verifier and a signature generated using a certificate issued by CA2. The act of CA1 issuing a certificate to CA2 implies that CA1 trusts CA2. The result is that the verifier will trust certificates issued by CA2

because CA1 trusts CA2.

Finally, the sldap32.dll provides the MISPC reference implementation components with the functionality to access repositories in order to search, retrieve, and post certificates and CRLs. The DLL uses the Lightweight Directory Access Protocol (LDAP) to access repositories as specified by the MISPC.

In Figure 4.2, the integration of the previously described DLLs as well as two executables, ManagementApp.exe and TransProc.exe, shows how the CA component of the MISPC reference implementation is formed. The ManagementApp.exe application is launched by the operator of the CA [8]. It provides a GUI to the CA component which allows an operator to set configuration variables, provide authentication information to access the CA's private keys, add ORAs which it has certified, and create, start, and stop instantiations of the CA's transaction processor (TransProc.exe). The ManagementApp.exe interacts with TransProc.exe by OLE–COM interfaces that begin and halt it, presents it with DER encoded PKI messages for processing, and returns either a DER encoded PKI message or an error message as a result of processing a PKI message. The certific32.dll provides the DER encoding and decoding functionality required by the transaction processor to generate and parse PKI messages. The transaction processor uses the sldap32.dll to interact with a repository as it issues, revokes, and searches for certificates and CRLs. To exchange DER encoded PKI messages between other MISPC components, the ManagementApp.exe uses the CygnaComSMTP.dll and CygnaComPOP3.dll to interact with an e–mail server. Finally, the ManagementApp.exe creates and manages a local database (powered by Microsoft Access using the OLE module for Access) of keys it has certified.

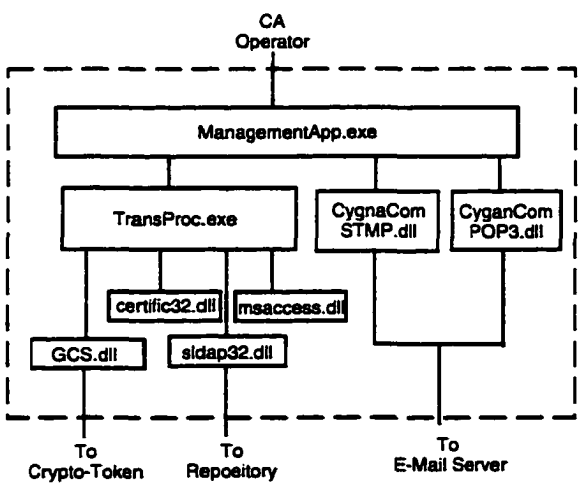

Figure 4.2 Overview of the MISPC Reference Implementation CA.

The ORA and Client components of the MISPC reference implementation are formed by the integration of the aforementioned DLLs which are shown in Figure 4.3. The code executed by the ORA and Client components are the exact same except for the GUI, which is presented to an operator of the component. The installMode variable entry found in the registry in the HKEY_LOCAL_MACHINE\-SOFTWARE\CYGNACOM SOLUTIONS, INC\NIST\ORA folder is set to a zero for the Client GUI to be presented and to a one for the ORA GUI. These GUIs provide the operator with the means to set configuration variables, present authentication information to access the components private keys, and request PKI transactions to be performed [9, 10]. The ORA and Client components generate and process DER encoded PKI messages with the help of the certific32.dll. To search for certificates or CRLs, the ORA and Client components use the sldap32.dll to interact with repository service applications. The ORA and Client components exchange DER encoded PKI messages with other MISPC components by using the CygnaComSMTP.dll to interact with an e-mail server. However, the ORA and Client components do not receive PKI messages directly from an e-mail server. A stand-alone e-mail client is used by both to retrieve e-mail messages which contain the MIME attached PKI messages. Once the e-mail messages containing the PKI messages are downloaded to the host system, the ORA and Client components process the MIME attachments containing the PKI messages.

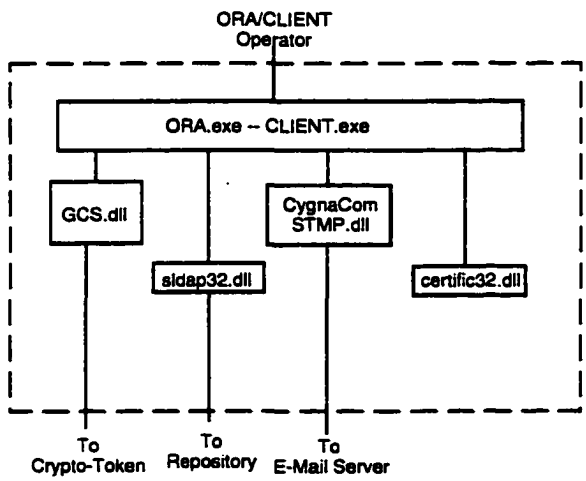

Figure 4.3 Overview of the MISPC Reference Implementation ORA and Client.

Test Suite Instantiation for the MISPC Reference Implementation CA Component

An instantiation of a test suite for the CA component of the MISPC will be derived for implementation based on the abstract test suite derived in Section 3 of Chapter 3 and the specific details of the MISPC reference implementation of the CA component presented in the previous section. In general, the proposed framework begins to instantiate a test suite by describing an implementation of an application as the set of input-output pairs that it generates, as previously shown in Equation 3.11. Using Equation 3.11, the MISPC reference implementation of the CA component application, CA_{ref}, can be written as Equation 4.1. The proposed framework continues to describe an application by determining the set of transactions instantiated by an implementation of an application, as previously shown in Equation 3.12. Using Equation 3.12, the set of transactions instantiated by CA_{ref} can be written as Equation 4.2. Because CA_{ref} only implements the CA component of the MISPC, Equation 4.2 can be simplified to Equation 4.3. An enumeration of the set of CA transactions implemented by CA_{ref} can be expressed by simplifying Equation 4.2 as shown in Equation 4.4.

$$B(CA_{ref}) = \{[i', o'] \mid i' \in I_I(CA_{ref}) \text{ and } o' \in O_I(CA_{ref})\} \tag{4.1}$$

$$T_I(CA_{ref}) = \{t' \mid t' \in T_I(CA_{ref}, c'),\ \forall c' \in C_I(CA_{ref})\} \tag{4.2}$$

$$T_I(CA_{ref}) = \{t' \mid t' \in T_I(CA_{ref}, CA')\} \tag{4.3}$$

$$T_I(CA_{ref}) = T_I(CA_{ref}, CA') = \{ora\_registration', cert\_revocation', self\_registration', cert\_renewal', cert\_post', crl\_post'\} \tag{4.4}$$

Once the specifics of an implementation of an application are expressed by the proposed framework, the relationship between the implementation and a specification can be demonstrated. The proposed framework begins by determining the relationship between components implemented by an application and the components described by a specification using Equation 3.14. The implementation of CA_{ref} only instantiates the CA component described by the MISPC and can be expressed as Equation 4.5.

Because of the relationship that exists between CA_{ref} and the MISPC, Equation 3.15 can be used to demonstrate a relationship between $T_I(CA_{ref})$ and the MISPC and is shown in Equation 4.6.

$$C_I(CA_{ref}, MISPC) = C_I(CA_{ref}) \cap C(MISPC) = CA' \tag{4.5}$$

$$\begin{aligned} T_I(CA_{ref}, MISPC) = T_I(CA_{ref}) \cap T(MISPC) = \\ \{ora\_registration', cert\_revocation', self\_registration', \\ cert\_renewal', cert\_post', crl\_post'\} \end{aligned} \tag{4.6}$$

The instantiation of the abstract test suite, $S_I(CA_{ref}, MISPC)$, can be expressed based on Equation 3.16. Equation 3.16 can be simplified because $C_I(CA_{ref}, MISPC)$ contains a single element and can be rewritten as shown in Equation 4.7. The framework will allow $S_I(CA_{ref}, MISPC)$ to be partitioned based on the required and optional transactions instantiated by CA_{ref}, $T_I(CA_{ref}, MISPC)$. The partitioning of $T_I(CA_{ref}, MISPC)$ into the required and optional transactions implemented by CA_{ref}, $T_{I_R}(CA_{ref}, MISPC)$ and $T_{I_O}(CA_{ref}, MISPC)$ are shown in Equations 4.8 and 4.9, respectively. Because the set of $T_{I_O}(CA_{ref}, MISPC)$ is empty, $S_I(CA_{ref}, MISPC)$ degenerates into the instantiation of the test suite for the required transactions, $S_{I_R}(CA_{ref}, MISPC)$. A simplification of $S_I(CA_{ref}, MISPC)$ can be made using Equations 4.7 and 4.8 and is shown in Equation 4.10. The set of messages instantiated for use by the instantiated transactions of CA_{ref}, shown in Equation 4.10, can be written as the expressions in Equations 4.11–4.16.

$$S_I(CA_{ref}, MISPC) = \{[i', o'] \mid \exists t' \in T_I(CA_{ref}, MISPC) \text{ such that } [i', o'] \in P_{I_{Pair}}(t', CA')\} \tag{4.7}$$

$$\begin{aligned} T_{I_R}(CA_{ref}, MISPC) = T_I(CA_{ref}) \cap T_R(MISPC) = \\ \{ora\_registration', cert\_revocation', self\_registration', \\ cert\_renewal', cert\_post', crl\_post'\} \end{aligned} \tag{4.8}$$

$$T_{I_O}(CA_{ref}, MISPC) = \{\emptyset\} \tag{4.9}$$

$$\begin{aligned} S_I(CA_{ref}, MISPC) = \{[i', o'] \mid [i', o'] \in P_{I_{Pair}}(ora\_registration', CA') \cup \\ P_{I_{Pair}}(cert\_revocation', CA') \cup P_{I_{Pair}}(self\_registration', CA') \cup \\ P_{I_{Pair}}(cert\_renewal', CA') \cup P_{I_{Pair}}(cert\_post', CA') \cup P_{I_{Pair}}(crl\_post', CA')\} \end{aligned} \tag{4.10}$$

$$T_{I_M}(ora\_registration') = (CH\_ORA\_CR' \cup ORA\_CA\_CR') \tag{4.11}$$

$$T_{I_M}(cert\_revocation') = (RR' \cup RP') \tag{4.12}$$

$$T_{I_M}(self\_registration') = (SELF\_CR' \cup SELF\_CP') \tag{4.13}$$

$$T_{I_M}(cert\_renewal') = (RENEWAL\_CR' \cup RENEWAL\_CP') \tag{4.14}$$

$$T_{I_M}(cert\_post'_{asynchronous}) = CERT\_POST\_MESS' \tag{4.15}$$

$$T_{I_M}(crl\_post'_{asynchronous}) = CRL\_POST\_MESS' \tag{4.16}$$

The instantiated message operations $M_{I_G}(m')$ and $M_{I_P}(m')$ are applied to the messages that make up the *ora_registration'* transaction and are shown in Equation 4.17. Equation 4.17 demonstrates that the CA_{ref} is capable of generating a *ca_ora_cp'* message as output and accepting an *ora_ca_cr'* message as input. Using Equation 4.5 and what is stated about a CA in the MISPC, if the CA_{ref} is presented an *ora_ca_cr'* message, it should generate a *ca_ora_cp'* message. The resulting input-output instantiated pair is shown in Equation 4.18.

$$CA' \in M_{I_G}(ca\_ora\_cp') \Rightarrow o' \in (M_{I_G}^{-1}(CA) \cap T_M(ora\_registration')) \tag{4.17}$$

$$CA' \in M_{I_P}(ora\_ca\_cr') \Rightarrow i' \in (M_{I_P}^{-1}(CA) \cap T_M(ora\_registration'))nonumber$$

$$P_{I_{Pair}}(ora\_registration', CA') = \{[i', o'] \mid i' \in ORA\_CA\_CR' \ AND \ o' \in CA\_ORA\_CP'\} \quad (4.18)$$

Applying the instantiated message operations to the messages used by the *cert_revocation'* transaction (shown in Equation 4.12) results in the expression shown in Equation 4.19. Equation 4.19 demonstrates that the CA_{ref} is capable of generating rp' messages as output and accepting rr' messages as input. Using Equation 4.5 and what is stated about a CA in the MISPC, if the CA_{ref} is presented a rr' message, it should generate a rp' message. The resulting input-output instantiated pair is shown in Equation 4.20.

$$\begin{aligned} CA' \in M_{I_G}(rp') &\Rightarrow o' \in (M_{I_G}^{-1}(CA) \cap T_M(cert\_revocation')) \\ CA' \in M_{I_P}(rr') &\Rightarrow i' \in (M_{I_P}^{-1}(CA) \cap T_M(cert\_revocation')) \end{aligned} \quad (4.19)$$

$$P_{I_{Pair}}(cert\_revocation', CA') = \{[i', o'] \mid i' \in RR' \ AND \ o' \in RP'\} \quad (4.20)$$

Applying the instantiated message operations to the messages used by the *self_registration'* transaction (shown in Equation 4.13) results in the expression shown in Equation 4.21. Equation 4.21 demonstrates that the CA_{ref} is capable of generating $self\_cp'$ messages as output and accepting $self\_cr'$ messages as input. Using Equation 4.5 and what is stated about a CA in the MISPC, if the CA_{ref} is presented a $self\_cr'$ message, it should generate a $self\_cp'$ message. The resulting input-output instantiated pair is shown in Equation 4.22.

$$\begin{aligned} CA' \in M_{I_G}(self\_cp') &\Rightarrow o' \in (M_{I_G}^{-1}(CA) \cap T_M(self\_registration')) \\ CA' \in M_{I_P}(self\_cr') &\Rightarrow i' \in (M_{I_P}^{-1}(CA) \cap T_M(self\_registration')) \end{aligned} \quad (4.21)$$

$$P_{I_{Pair}}(self\_registration', CA') = \{[i', o'] \mid i' \in SELF\_CR' \ AND \ o' \in SELF\_CP'\} \quad (4.22)$$

Applying the instantiated message operations to the messages used by the *cert_renewal'* transaction (shown in Equation 4.14) results in the expression shown in Equation 4.23. Equation 4.23 demonstrates that the CA_{ref} is capable of generating *renewal_cp'* messages as output and accepting *renewal_cr'* messages as input. Using Equation 4.5 and what is stated about a CA in the MISPC, if the CA_{ref} is presented a *renewal_cr'* message, it should generate a *renewal_cp'* message. The resulting input-output instantiated pair is shown in Equation 4.24.

$$\begin{aligned} CA' \in M_{I_G}(renewal\_cp') &\Rightarrow o' \in (M_{I_G}^{-1}(CA) \cap T_M(cert\_renewal')) \\ CA' \in M_{I_P}(renewal\_cr') &\Rightarrow i' \in (M_{I_P}^{-1}(CA) \cap T_M(cert\_renewal')) \end{aligned} \quad (4.23)$$

$$P_{I_{Pair}}(cert\_renewal', CA') = \{[i', o'] \mid i' \in RENEWAL\_CR' \ AND \ o' \in RENEWAL\_CP'\} \quad (4.24)$$

Applying the instantiated message operations to the messages used by the *cert_post'* transaction (shown in Equation 4.15) results in the expression shown in Equation 4.25. Equation 4.25 demonstrates that the CA_{ref} is capable of generating *cert_post_mess'* messages as output. The MISPC does not address the questions of when a *cert_post_mess'* message should be generated, which leads to the use of an unknown input, γ', being used to indicate that some input is required to cause the *cert_post_mess'* message to be generated. The resulting input-output instantiated pair is shown in Equation 4.26. When applying the instantiated message operations to the messages used by the *crl_post'* transaction (shown in Equation 4.16), a similar result occurs as demonstrated by Equations 4.27 and 4.28. Using the Equations 4.18, 4.20, 4.22, 4.24, 4.26, and 4.28, Equation 4.10 can be simplified to the expression shown in Equation 4.29. The $[\gamma', cert\_post\_mess']$ and $[\gamma', crl\_post\_mess']$ elements can be eliminated from the instantiated test suite because the undefined input make it impossible for the test suite to generate the input. The result is a further simplification of Equation 4.29 to the expression shown in Equation 4.30.

$$CA' \in M_{I_G}(cert\_post\_mess') \Rightarrow o' \in (M_{I_G}^{-1}(CA) \cap T_M(cert\_post'_{asynchronous})) \quad (4.25)$$

$$P_{I_{Pair}}(cert\_post'_{asynchronous}, CA') = \{[i', o'] \mid i' = \gamma' \ AND \ o' \in CERT\_POST\_MESS'\} \quad (4.26)$$

$$CA' \in M_{I_G}(crl\_post\_mess') \Rightarrow o' \in (M_{I_G}^{-1}(CA) \cap T_M(crl\_post'_{asynchronous})) \quad (4.27)$$

$$P_{I_{Pair}}(crl\_post'_{asynchronous}, CA') = \{[i', o'] \mid i' = \gamma' \ AND \ o' \in CRL\_POST\_MESS'\} \quad (4.28)$$

$$S_I(CA_{ref}, MISPC) = \{(ORA\_CA\_CR' \times CA\_ORA\_CP') \cup \quad (4.29)$$
$$(SELF\_CR' \times SELF\_CP') \cup (RR' \times RP') \cup (RENEWAL\_CR' \times RENEWAL\_CP') \cup$$
$$(\gamma' \times CERT\_POST\_MESS') \cup (\gamma' \times CRL\_POST\_MESS')\}$$

$$S_I(CA_{ref}, MISPC) = \{(ORA\_CA\_CR' \times CA\_ORA\_CP') \cup (SELF\_CR' \times SELF\_CP') \cup \quad (4.30)$$
$$(RR' \times RP') \cup (RENEWAL\_CR' \times RENEWAL\_CP')\}$$

Using Equation 4.30, the definition of the test suite to be instantiated can be further specified. For example, a test suite to be instantiated can include conditions on messages instantiated by the CA_{ref}. Equations 4.31 and 4.32 demonstrate how a test suite to be instantiated can be further expressed. Equation 4.31 expresses an instantiated test suite that includes test cases with input-output pairs where the input is correctly and incorrectly formatted with respect to the MISPC for all instantiated transactions. Similarly, Equation 4.32 expresses a instantiated test suite that includes test cases with input-output pairs where the input is not specified with respect to the MISPC. These two instantiated test suites can be combined into one test suite as expressed in Equation 4.33. The instantiated test suite expressed in Equation 4.33 is the test suite to be implemented by this project. The details of the

instantiated test suite to be implemented by this project, including test data selection and evaluation of test cases, are presented in the next section.

$$IS_1(CA_{ref}, MISPC) = \{[i', o'] \mid [i', o'] \in S_I(CA_{ref}, MISPC)\ AND \\ \exists i'.invalid_{format}(MISPC)\ AND\ \exists i'.valid_{format}(MISPC)\} \tag{4.31}$$

$$IS_2(CA_{ref}, MISPC) = \{[i', o'] \mid \exists i' \in I(CA_{ref})\ AND\ i' \notin T_{I_M}(t'), \\ \forall t' \in T(CA_{ref}, MISPC)\} \tag{4.32}$$

$$IS(CA_{ref}, MISPC) = IS_1(CA_{ref}, MISPC)\ \cup\ IS_2(CA_{ref}, MISPC) \tag{4.33}$$

Test Suite Application Implementation and Execution Configuration

The instantiation of the test suite, IS_p, to be implemented by this project will focus on testing the certificate revocation transaction for the MISPC reference implementation of the CA component. Equation 4.34 shows the expression that can initially be used to describe the test suite. Equation 4.34 is a modified version of Equation 3.16 where an extra parameter has been introduced to describe the general test suite so that specific transactions to be tested can be expressed by the proposed framework. Using Equations 4.20 and 4.34, the expression of the test suite to be implemented can be simplified as shown in Equation 4.35 because the input-output pair of the certificate revocation transaction is the only transaction to be tested. Using the notation to specify attributes about transaction messages, the test suite can be further enhanced as shown in Equation 4.36. Equation 4.36 shows that the test suite will contain test data where the revocation request message is an element from either a set of messages that have an incorrectly formatted header with the body formatted correctly; or a set of messages that have an incorrectly formatted body with the header formatted correctly; or a set of messages where both the header and body are correctly formatted. The test suite to be implemented will not test the validation of cryptographic functions such as signature validation. To express the testing of the cryptographic functions by the test suite, a modification to the sets expressed in Equation 4.36 would be required. For example, $RR'.invalid_{format\_header}.valid_{format\_body}.valid_{value\_protection}(MISPC)$ would be the set

of messages that have an invalid header format, a validly formatted body, and a valid protection field value.

$$IS_p(CA_{ref}, MISPC, cert\_revocation') = \{[i', o'] \mid [i', o'] \in S_I(CA_{ref}, MISPC) \; AND \; i' \in T_{I_M}(cert\_revocation')\} \tag{4.34}$$

$$IS_p(CA_{ref}, MISPC, cert\_revocation') = RR' \times RP' \tag{4.35}$$

$$IS_p(CA_{ref}, MISPC, cert\_revocation') = \{[rr', rp'] \mid rr' \in RR'.invalid_{format\_header}.valid_{format\_body}(MISPC) \; OR \; rr' \in RR'.valid_{format\_header}.invalid_{format\_body}(MISPC) \; OR \; rr' \in RR'.valid_{format}(MISPC)\} \tag{4.36}$$

The elements of the sets described in Equation 4.36 are generated by the implemented test suite based on specific input values to the test suite. The test suite reads an input file that contains a vector of values required to generate the specified segments of the PKI message. A sample of the contents of an input file is contained in Appendix A. The input file contains input values required to generate the fields of the header (hdr_version, hdr_sender, etc.), body (bdy_type, bdy_rr_number, etc.), and protection (protection_signature, protection_gen_id, etc.) segments of a PKI message. Currently, if NULL is the input value for a field, the field is skipped when a value cannot be automatically generated for it by the test suite. For example, the test suite will obtain the current time from the system clock for use in a time dependent field. if the field has a NULL as its input value. However, a X.500 distinguished name value that is NULL would result in the field not being assigned a value because the test suite does not have the capability to generate random X.500 distinguished names. Based on the values present in the input file, the test suite automatically generates the sets for $RR'.invalid_{format\_header}.valid_{format\_body}(MISPC)$, $RR'.valid_{format}(MISPC)$, and $RR'.valid_{format\_header}.invalid_{format\_body}(MISPC)$. For the sets $RR'.valid_{format}(MISPC)$ and

$RR'.invalid_{format\_header}.valid_{format\_body}(MISPC)$, the test suite will generate $2^n - 1$ permutations of the header formats for testing, where n is the number of header fields that can be filled in, given the input values. The adjustment of minus one is used to eliminate the header format that contains no fields. In the input file of Appendix A, all 11 fields of the header can be filled, which results in $2^{11} - 1$ or 2047 header permutations being generated. Of the 2047 header format permutations, 32 of the header formats are valid and 2015 of the header formats are invalid. Each header format is placed in a PKI message with a valid body to be used as test data for the test suite. This results in 32 elements for the set $RR'.valid_{format}(MISPC)$ and 2015 elements for the set $RR'.invalid_{format\_header}.valid_{format\_body}(MISPC)$. Similarly for the sets, $RR'.valid_{format}(MISPC)$ and $RR'.valid_{format\_header}.invalid_{format\_body}(MISPC)$, the test suite will generate $2^m - 1$ permutations of the revocation request body, where m is the number of revocation request body fields that can be filled in, given the values of the input file. Using the input file of Appendix A, all four fields of the revocation body request can be filled in resulting in $2^4 - 1$ or 15 permutations of the revocation request body formats being generated. Of the 15 revocation request body formats, 4 of the body formats are valid and 11 of the body formats are invalid according to the MISPC. Each revocation request format is placed in a PKI message with a valid header to be used as test data for the test suite. This results in 4 elements for the set $RR'.valid_{format}(MISPC)$ and 11 elements for the set $RR'.valid_{format\_header}.invalid_{format\_body}(MISPC)$.

The test suite uses test data from the defined sets and presents the ASN.1 DER encoding of each to the transaction processor of the MISPC CA reference implementation. If the transaction processor cannot process the test data, it returns an implementation-specific error code and message. The error code and message are recorded by the test suite into a results file. Examples of the contents of a results file are contained in Appendix B and Appendix C. If the transaction processor can successfully process the test data, it returns an MISPC ASN.1 DER encoded revocation response to the test data. If an MISPC revocation response is returned by the transaction processor, it is analyzed to determine if the format of the header and body of the response conform to the MISPC. For example, the header must contain sender and recipient fields. In addition, the values of revocation response fields are inspected to verify that their contents are valid with respect to the MISPC, when possible. For example, the header version field should contain a value of zero. Finally, the values of the revocation response fields are compared to the revocation request fields where appropriate. For example, the sender field of the header for the revocation response should match the recipient field of the header for the revocation request. The results of the revocation response analysis are recorded in the results file. The results files

presented in Appendices B and C will be reviewed in detail in the next section of this chapter.

The execution environment used by this project for the test suite application requires two host systems as shown in Figure 4.4. Host System1 is where the test suite application and the MISPC reference implementation of the CA component are hosted while Host System2 hosts a repository service application. This configuration could be simplified by running the repository service application on the Host System1 or by eliminating the repository service application all together. The development contract for the components of the MISPC reference implementation specified that they would utilize Netscape's Directory Server version 1.03.[2] The TransProc.exe module for the CA component of the MISPC reference implementation is to be tested for conformance to the MISPC by this project. The TransProc.exe module of the CA was chosen for testing because it allows the core MISPC functionality provided by the CA to be tested without overhead of testing the CA application's GUI for the CA operator. This is achieved by having the CA operator GUI and the core MISPC functionality in separate executable modules. The ORA/Client components of the MISPC reference implementation do not provide this advantage because their operator GUI and the core MISPC functionality are within the same executable module. To achieve the same level of test suite and subject application interaction, the source code for the ORA/CA component would need to be modified. In addition, by testing the TransProc.exe module directly, the elimination of the e-mail server as part of the execution environment application is achieved which simplifies the design of the test suite application.

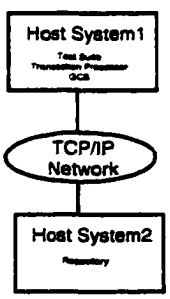

Figure 4.4 System Configuration for Execution of Test Suite.

Figure 4.5 shows the test suite related elements that are hosted on Host System1 and which elements communicate with each other. The test suite application logs into the GCS module (gcs.dll) in order to access an existing cryptographic token containing a DSA private key for use by the instantiation of the TransProc.exe module that will be created by the test suite. When an instantiation of the TransProc.exe

[2]Attempts have been made to use version 1.1 of the reference implementation with later versions of the Netscape Directory Server, but they have not yielded successful results.

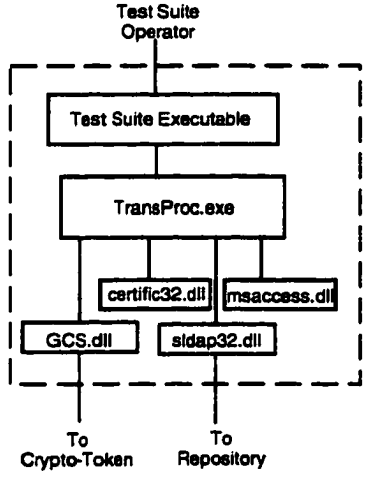

Figure 4.5 Host System Configuration for the Test Suite.

modules is created, the test suite application presents and receives MISPC DER encoded PKI messages as test data to the module. After TransProc.exe has been tested, the test suite application destroys the instance of TransProc.exe it previously created. Figure 4.5 also shows that to eliminate the repository service application from the configuration would require that a new version of the sldap32.dll be created for the test suite to intercept repository queries by TransProc.exe.

The test suite was implemented using the C/C++ programming languages. These programming languages were selected because the MISPC reference implementation components were originally programmed with them. The object-oriented aspects of C++ are required for the test suite application to be able to interact with the TransProc.exe. The test suite application uses the OLE-COM interfaces defined by TransProc.exe to interact with it. As different versions of TransProc.exe are developed using the same OLE-COM interfaces, they can be tested without modification to the test suite application. The development library used to implement the ASN.1 defined data structures for the test suite application also dictated the use of the C++ programming language. The BBN ASN.1 compiler and library were used to synthesize the MISPC PKI data structures defined in ASN.1 into C++ object classes. The BBN ASN.1 library contained useful methods to manipulate the C++ objects, such as to DER encode and decode method for ASN.1 objects. The final test suite application was compiled to target a personal computer running Windows 95 using version 5.01 of Borland's C/C++ compiler. The only DLL required by the test suite application is the GCS module, gcs.dll, which was not developed by this project but is required for testing TransProc.exe.

The test suite application and TransProc.exe were installed and run on a DELL personal computer with 16M of RAM, an Intel Pentium processor running at 100MHz, and running Microsoft's Windows 95. The repository was installed and run on a DELL personal computer with 128M of RAM, an Intel Pentium Pro running at 180MHz, and running Microsoft's Windows NT. Both host systems were networked using a TCP/IP Ethernet connection. Other important configuration variables for the test suite application are given in Table 4.1. The existing cryptographic token used by the test suite application was generated by the gcs.dll before the test suite was executed. The next section will present details of the results of executing this test suite on the TransProc.exe module.

Table 4.1 Configuration Variables Of The Test Suite Application.

Configuration Variable	Variable Setting
Test Suite Host	129.6.52.15
Test Suite Default Directory	C:\PROGRA~1\MISPC\CA
GCS Token Default Directory	C:\PROGRA~1\MISPC\CA\CC_DB
GCS Token Username	ref_ca
GCS Token Password	trustme2
LDAP Sever Host	129.6.52.23
LDAP Port	389
LDAP Username	cn=Directory Manager, ou=nist, o=gov, c=US
LDAP Password	trustme2

Results Analysis of the Test Suite Application's Execution

The test suite described in the previous section was executed on the TransProc.exe module of the MISPC CA reference implementation. Using the input file of Appendix A, the test suite generated 2057 individual test cases which were presented to the TransProc.exe module. The test suite could have generated 2062 individual test case; however, the TransProc.exe module was unable to handle five of the test cases. The five test cases that TransProc.exe module could not handle related to the following variations of the header formats: 1) only the version field is present, 2) only the sender field is present, 3) only the recipient field is present, 4) only the sender and recipient fields are present, and 5) only the version and recipient fields were present. The TransProc.exe module responded to these five test cases by causing a fatal system error that terminated the execution of the TransProc.exe module and the test suite. The time required by the test suite to execute the 2057 test cases was about 2.5 hours. The possibility of increasing the number of test cases was explored by looking at creating permutations

of the CertDetails field of the revocation request body, but this was shown to be not practical for this project. The additional permutation would have increased the number of the test cases to 16384 for revocation request body. This would have resulted in a test suite execution time of about 22 hours. which was determined to be excessive for this experiment.

The number of elements in the sets $RR'.invalid_{format\_header}.valid_{format\_body}(MISPC)$, $RR'.valid_{format}(MISPC)$, and $RR'.valid_{format\_header}.invalid_{format\_body}(MISPC)$ can be determined. The test suite generated 2011 elements for the set $RR'.invalid_{format\_header}.valid_{format\_body}(MISPC)$, which composed 97.76% of the total test cases executed by the test suite. The test suite generated 12 elements for the set $RR'.valid_{format\_header}.invalid_{format\_body}(MISPC)$, which was .58% of the total test cases executed by the test suite. The test suite generated 34 elements for the set $RR'.valid_{format}(MISPC)$, which was 1.65% of the total test cases executed by the test suite. This result is interesting because it demonstrates how limiting structured frameworks that only consider valid input spaces can be with regards to the generation of test cases. If only the valid input space were considered by the implemented test suite, only 34 test cases would have been developed. In addition. the five test cases that caused the fatal system error for the TransProc.exe module would not have been generated, resulting in this error not being revealed. This underscores how structured frameworks limited to valid input spaces are not well suited for the development of test suites that intend to demonstrate an application's robustness.

The results of executing the test cases for the set $RR'.valid_{format}(MISPC)$ generated by the test suite are given in Appendix B and Appendix C. The results in Appendix B were generated when the test suite was generating the permutations of the header formats. The results in Appendix C were generated when the test suite was generating the permutations of the revocation request body formats. The results of the test cases for the sets $RR'.invalid_{format\_header}.valid_{format\_body}(MISPC)$ and $RR'.valid_{format\_header}.invalid_{format\_body}(MISPC)$ are similar but were not included in this report because of the volume of the results. These results yielded some interesting information about the behavior of the TransProc.exe module. The implementation specific errors encountered by the test suite were numbered 6316, 4230, and 4245. Error 6316 indicates that the TransProc.exe module could not parse the DER encoded PKI message it received because it did not know where to place a particular field in its internal data structure. Error 4230 indicates that the TransProc.exe module encountered an ASN.1 tag which it did not recognize and therefore did not know where to place the information in its internal data structure. Error 4245 indicates that a null value was found for a field that requires a value when the TransProc.exe module was parsing the DER encoded PKI message. Using the meanings

of these implementation specific errors and the results contained in Appendix B, the TransProc.exe module was shown not to be able to parse PKI messages that contained the FreeText field of the PKI message header even though this forms a valid MISPC PKI message format. Additionally, if the sender or recipient field of the header contained an e-mail address instead of a X.500 distinguished name, the test suite showed that the TransProc.exe module could not parse the PKI message header. The MISPC requires that any conforming MISPC component must be able to understand both X.500 distinguished names and e-mail addresses for the sender and recipient fields of the PKI message header. To verify that the TransProc.exe module was really having ASN.1 parsing problems, a few of the PKI messages generated by the test suite that could not be parsed were inspected at the byte level to make sure the test suite was DER encoding the PKI messages correctly. These PKI messages were then compared to the messages being generated by the components of the MISPC reference implementation. In some preliminary tests, it was shown that the components of the MISPC reference implementation were incorrectly generating DER encoded self registration request messages. In most cases, it was verified that the test suite was correctly DER encoding the PKI messages but the TransProc.exe module was unable to parse the message. Finally, the test suite revealed that the TransProc.exe module was not generating valid PKI message header formats for revocation responses when the header of a revocation request message contained a sender nonce. The MISPC requires that if a revocation request header contains a sender nonce, the revocation response header must contain that same nonce value in the recipient nonce field. This part of the MISPC is to protect PKI components from replay attacks using old transaction messages. The results contained in Appendix C show that the TransProc.exe module appears to be processing the permutations of revocation request bodies correctly and generating valid revocation response messages.

5 CONCLUSIONS AND FUTURE WORK

Conclusions

The proposed structured framework for test suite development was initially introduced by this thesis to address some of the weaknesses found in the TTF structured framework for test suite development. Test case generation based on a specification's invalid input space is not being performed by the TTF structured framework. The TTF structured framework begins by defining the valid input space and acknowledging the existence of an invalid input space. However, the TTF structured framework focuses on manipulation of the valid input space by partition testing techniques and uses the results to generate test cases for a test suite. By not considering the invalid input space, the TTF structured framework cannot be used effectively in the development of test suites that desire to determine an application's robustness. This thesis showed that over 90% of the test cases used by the test suite were considered by the MISPC to be invalid. This result underscores how important it is for structured frameworks to use both valid and invalid input spaces when generating test cases so that the framework will be useful in the development of test suites with various objectives.

Another weakness of the TTF structured framework is its acknowledged inability to express interactions between applications. The ability to express interactions between applications has become more important as applications are being designed as a single application that interacts with multiple modules and the number of client-server applications increases. A structured framework for test suite development that can capture interactions between applications can develop relationships between the applications that are important in the design of a test suite. The proposed framework addresses this weakness by expressing interactions (or transactions) between applications as a set of messages. The messages are organized as input-output pairs with the use of the processing and generating operations which allow relationships between applications to be expressed. The relationships between applications help in the development and implementation of test suites. For example, in this thesis understanding that MISPC ORA and CH components can both request certificates to be revoked, gives insight to the test suite designer that the MISPC CA component must be able to process certificate revocation

requests from both ORA and CH components.

Finally, an unexpected result of this thesis is the idea of expressing a test suite on two levels, abstractly and concretely (an instantiation). The abstract test suite is created based on theoretical notions about how a test suite should be described and is developed using abstract methodologies. The instantiation of a test suite is created based on real-world issues, taking into account the specific application to be tested while the development and implementation of a test suite is being done. The proposed structured framework recognizes the benefits of each view and tries to provide a structured framework under which both can be unified. Thus, the proposed structured framework introduces a method to describe an abstract test suite which can then be used to create an instantiation of the original abstract test suite. In short, the proposed structured framework enhances structured frameworks for test suite development by using both valid and invalid input spaces for test data generation, is able to express interactions between applications, and provides a unifying framework under which both abstract test suites and instantiations of test suites can be developed.

Future Work

The proposed structured framework for test suite development introduced in this thesis offers several opportunities for further research. The continued development and extension of the proposed structured framework provides one area for further research. Investigating and developing a method for expressing an application's state is not currently addressed by the proposed structured framework. A starting point may be to concatenate input-output pairs to develop sequences of test data which might be used to represent the state of the application under test. The development of a statistical method for sampling test data that is generated by the proposed framework would help when the set of test data generated becomes very large. The statistical method should maximize the effectiveness of the test suite when using a limited number of test data. The continued application of the proposed structured framework to other specifications and applications will help in furthering the development of the structured framework and extending its capabilities. One opportunity would be to apply the structured framework to the FIPS-140-1 Security Requirements For Cryptographic Modules standard and its Derived Test Requirements (DTRs), which describe the requirements for a cryptographic module to become validated. The DTRs could be expressed elegantly with the proposed structured framework because of its set nature which is similar to that of the DTRs. Finally, the test suite for the MISPC reference implementation that was developed and implemented in this thesis could be made more complete, which also would continue to help the development and extension of the structured framework.

APPENDIX A EXAMPLE INPUT FILE OF THE TEST SUITE APPLICATION

hdr_version=0

hdr_sender=c=US@o=gov@ou=nist@cn=nelson hastings

hdr_sender_type=DIRNAME

hdr_issuer=c=US@o=gov@ou=nist@cn=ref ca

hdr_type=DIRNAME

hdr_messageTime=NULL

hdr_protection=id_dsa_with_sha1

hdr_senderKID=232abd

hdr_recipKID=332312

hdr_transactionID=1234567

hdr_senderNonce=23233

hdr_recipNonce=34324

hdr_freeText=For User Consuption....

hdr_freeTextType=IA5STRING

bdy_type=RR

bdy_rr_number=1

bdy_rr_ct_version=2

bdy_rr_ct_serialNumber=121289

bdy_rr_ct_signingAlgID=id_dsa_with_sha1

bdy_rr_ct_subject=c=US@o=gov@ou=nist@cn=nelson hastings

bdy_rr_ct_validity_notBefore=NULL

bdy_rr_ct_validity_notAfter=NULL

bdy_rr_ct_issuer=c=US@o=gov@ou=nist@cn=ref ca

bdy_rr_ct_pubKeyInfo_id=id_dsa_with_sha1

bdy_rr_ct_pubKeyInfo_dsa_parameter_p=ec1a364759a3e339d6dcf2d710c96a968fcf14bece314f3
6045f95ba55c8bc256d93b3d0971d2b3523944c72313
142e39b80c9bbb332dc5884f3f2c8e1846ad3

bdy_rr_ct_pubKeyInfo_dsa_parameter_q=87bf6ee5868ef6314ee8da2cbda6cd72f5002045

bdy_rr_ct_pubKeyInfo_dsa_parameter_g=6e3ba6373fd92f195b40e6f45b8bed03726ccca1872f5
b1f3e97db76fbc60cd769e10ae016d4b433c7cb3de4
b863357a63eeb3f90455930ae89bfe088aeeb82f

bdy_rr_ct_pubKeyInfo_dsa_parameter_seed=42806313e0226d9f7d489c7ede1eacd3ed95fbd4

bdy_rr_ct_pubKeyInfo_key=NULL

bdy_rr_ct_issuerUID=894823423

bdy_rr_ct_subjectUID=3312223

bdy_rr_ct_number_ex=2

bdy_rr_ct_ex_id=id_ce_subjectKeyIdentifier

bdy_rr_ct_ex_value=43423423

bdy_rr_ct_ex_critical=TRUE

bdy_rr_ct_ex_id=id_ce_authorityKeyId

bdy_rr_ct_ex_value=48929304

bdy_rr_ct_ex_critical=FALSE

bdy_rr_reason=434324

bdy_rr_badSince=NULL

bdy_rr_number_ex=2

bdy_rr_ex_id=id_ce_subjectKeyIdentifier

bdy_rr_ex_value=43423423

bdy_rr_ex_critical=TRUE

bdy_rr_ex_id=id_ce_authorityKeyIdentifier

bdy_rr_ex_value=48929304

bdy_rr_ex_critical=FALSE

protection_sigature=NULL

protection_gen_id=id_dsa_with_sha1

protection_parameter_dsa_p=ec1a364759a3e339d6dcf2d710c96a968fcf14bece314f3
6045f95ba55c8bc256d93b3d0971d2b3523944c72313
142e39b80c9bbb332dc5884f3f2c8e1846ad3

protection_parameter_dsa_q=87bf6ee5868ef6314ee8da2cbda6cd72f5002045

protection_parameter_dsa_g=6e3ba6373fd92f195b40e6f45b8bed03726ccea1872f5
b1f3e97db76fbc60cd769e10ae016d4b433e7cb3de4
b863357a63eeb3f90455930ae89bfe088aeeb82f

protection_parameter_dsa_seed=42806313e0226d9f7d489c7edc1eacd3ed95fbd4

protection_dsa_key=NULL

protection_des_mac_key=8980989

APPENDIX B RESULTS FILE FOR THE CERTIFICATE REVOCATION TRANSACTION: VALID HEADER FORMATS

Header Testcase:: 2047

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHMID,
SUBJECT_KEY_ID,RECIPIENT_KEY_ID,TRANSACTION_ID,
SUBJECT_NONCE,RECIPIENT_NONCE,FREE_TEXT

Result :: 2047

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 2015

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,RECIPIENT_KEY_ID,TRANSACTION_ID,SUBJECT_NONCE,
RECIPIENT_NONCE,FREE_TEXT

Result :: 2015

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1983

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHMID,
SUBJECT_KEY_ID,NULL,TRANSACTION_ID,SUBJECT_NONCE,
RECIPIENT_NONCE,FREE_TEXT

Result :: 1983

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1951

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME.ALGORITHM ID,
NULL.NULL,TRANSACTION_ID,SUBJECT_NONCE,
RECIPIENT_NONCE.FREE_TEXT

Result :: 1951

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1791

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,RECIPIENT_KEY_ID,TRANSACTION_ID,NULL,
RECIPIENT_NONCE,FREE_TEXT

Result :: 1791

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1759

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,RECIPIENT_KEY_ID,TRANSACTION_ID,NULL.
RECIPIENT_NONCE,FREE_TEXT

Result :: 1759

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1727

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,NULL,TRANSACTION_ID,NULL,
RECIPIENT_NONCE,FREE_TEXT

Result :: 1727

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1695

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,NULL,TRANSACTION_ID,NULL,
RECIPIENT_NONCE,FREE_TEXT

Result :: 1695

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1535

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,RECIPIENT_KEY_ID,TRANSACTION_ID,
SUBJECT_NONCE,NULL,FREE_TEXT

Result :: 1535

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1503

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,RECIPIENT_KEY_ID,TRANSACTION_ID,SUBJECT_NONCE,
NULL,FREE_TEXT

Result :: 1503

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1471

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,NULL,TRANSACTION_ID,SUBJECT_NONCE,
NULL,FREE_TEXT

Result :: 1471

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1439

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,NULL,TRANSACTION_ID,SUBJECT_NONCE,
NULL,FREE_TEXT

Result :: 1439

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1279

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,RECIPIENT_KEY_ID,TRANSACTION_ID,NULL,
NULL,FREE_TEXT

Result :: 1279

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1247

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,RECIPIENT_KEY_ID,TRANSACTION_ID,NULL,
NULL,FREE_TEXT

Result :: 1247

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1215

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,NULL,TRANSACTION_ID,NULL,
NULL,FREE_TEXT

Result :: 1215

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1183

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,NULL,TRANSACTION_ID,NULL,
NULL,FREE_TEXT

Result :: 1183

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Header Testcase:: 1023

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,RECIPIENT_KEY_ID,TRANSACTION_ID,
SUBJECT_NONCE,RECIPIENT_NONCE,NULL

Header Analysis Result :: 1023

Error :: WRONG_DYNAMIC_CONTENT

Field :: RECIPIENT_NONCE

RP Body Analysis Result :: 1023

Error :: NONE

Field :: N/A

Header Testcase:: 991

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,RECIPIENT_KEY_ID,TRANSACTION_ID,
SUBJECT_NONCE,RECIPIENT_NONCE,NULL

Header Analysis Result :: 991

Error :: WRONG_DYNAMIC_CONTENT

Field :: RECIPIENT_NONCE

RP Body Analysis Result :: 991

Error :: NONE

Field :: N/A

Header Testcase:: 959

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,NULL,TRANSACTION_ID,SUBJECT_NONCE,
RECIPIENT_NONCE,NULL

Header Analysis Result :: 959

Error :: WRONG_DYNAMIC_CONTENT

Field :: RECIPIENT_NONCE

RP Body Analysis Result :: 959

Error :: NONE

Field :: N/A

Header Testcase:: 927

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,NULL,TRANSACTION_ID,SUBJECT_NONCE,
RECIPIENT_NONCE,NULL

Header Analysis Result :: 927

Error :: WRONG_DYNAMIC_CONTENT

Field :: RECIPIENT_NONCE

RP Body Analysis Result :: 927

Error :: NONE

Field :: N/A

Header Testcase:: 767

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,RECIPIENT_KEY_ID,TRANSACTION_ID,NULL,
RECIPIENT_NONCE,NULL

Header Analysis Result :: 767

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 767

Error :: NONE

Field :: N/A

Header Testcase:: 735

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,RECIPIENT_KEY_ID,TRANSACTION_ID,
NULL.RECIPIENT_NONCE,NULL

Header Analysis Result :: 735

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 735

Error :: NONE

Field :: N/A

Header Testcase:: 703

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,NULL,TRANSACTION_ID,NULL,
RECIPIENT_NONCE,NULL

Header Analysis Result :: 703

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 703

Error :: NONE

Field :: N/A

Header Testcase:: 671

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,NULL,TRANSACTION_ID,NULL,
RECIPIENT_NONCE,NULL

Header Analysis Result :: 671

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 671

Error :: NONE

Field :: N/A

Header Testcase:: 511

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,RECIPIENT_KEY_ID,TRANSACTION_ID,
SUBJECT_NONCE,NULL,NULL

Header Analysis Result :: 511

Error :: WRONG_DYNAMIC_CONTENT

Field :: RECIPIENT_NONCE

RP Body Analysis Result :: 511

Error :: NONE

Field :: N/A

Header Testcase:: 479

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,RECIPIENT_KEY_ID,TRANSACTION_ID,SUBJECT_NONCE,
NULL,NULL

Header Analysis Result :: 479

Error :: WRONG_DYNAMIC_CONTENT

Field :: RECIPIENT_NONCE

RP Body Analysis Result :: 479

Error :: NONE

Field :: N/A

Header Testcase:: 447

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,NULL,TRANSACTION_ID,SUBJECT_NONCE,
NULL,NULL

Header Analysis Result :: 447

Error :: WRONG_DYNAMIC_CONTENT

Field :: RECIPIENT_NONCE

RP Body Analysis Result :: 447

Error :: NONE

Field :: N/A

Header Testcase:: 415

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,NULL,TRANSACTION_ID,SUBJECT_NONCE,
NULL,NULL

Header Analysis Result :: 415

Error :: WRONG_DYNAMIC_CONTENT

Field :: RECIPIENT_NONCE

RP Body Analysis Result :: 415

Error :: NONE

Field :: N/A

Header Testcase:: 255

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,RECIPIENT_KEY_ID,TRANSACTION_ID,NULL,
NULL,NULL

Header Analysis Result :: 255

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 255

Error :: NONE

Field :: N/A

Header Testcase:: 223

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,RECIPIENT_KEY_ID,TRANSACTION_ID,NULL,
NULL,NULL

Header Analysis Result :: 223

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 223

Error :: NONE

Field :: N/A

Header Testcase:: 191

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
SUBJECT_KEY_ID,NULL,TRANSACTION_ID,NULL,
NULL,NULL

Header Analysis Result :: 191

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 191

Error :: NONE

Field :: N/A

Header Testcase:: 159

Error :: NONE

Header Format :: VERSION,SENDER,RECIPIENT,MESSAGE_TIME,ALGORITHM ID,
NULL,NULL,TRANSACTION_ID,NULL,
NULL,NULL

Header Analysis Result :: 159

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 159

Error :: NONE

Field :: N/A

APPENDIX C RESULTS FILE FOR THE CERTIFICATE REVOCATION TRANSACTION: ALL BODY FORMATS

Revocation Request Testcase :: 15

Error :: NONE

Revocation Request Format :: CERT_TEMPLATE,REASON_CODE,BAD_SINCE_DATE,
CRL_EXTENSIONS

Header Analysis Result :: 15

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 15

Error :: NONE

Field :: N/A

Revocation Request Testcase :: 14

Error :: MISSING_FIELD

Revocation Request Format :: NULL,REASON_CODE,BAD_SINCE_DATE,
CRL_EXTENSIONS

Result :: 14

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Revocation Request Testcase :: 13

Error :: MISSING_FIELD

Revocation Request Format :: CERT_TEMPLATE,NULL,BAD_SINCE_DATE,
CRL_EXTENSIONS

Result :: 13

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Revocation Request Testcase :: 12

Error :: MISSING_FIELD

Revocation Request Format :: NULL,NULL,BAD_SINCE_DATE,CRL_EXTENSIONS

Result :: 12

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Revocation Request Testcase :: 11

Error :: NONE

Revocation Request Format :: CERT_TEMPLATE,REASON_CODE,NULL,
CRL_EXTENSIONS

Header Analysis Result :: 11

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 11

Error :: NONE

Field :: N/A

Revocation Request Testcase :: 10

Error :: MISSING_FIELD

Revocation Request Format :: NULL,REASON_CODE,NULL,CRL_EXTENSIONS

Result :: 10

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Revocation Request Testcase :: 9

Error :: MISSING_FIELD

Revocation Request Format :: CERT_TEMPLATE,NULL,NULL,CRL_EXTENSIONS

Result :: 9

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Revocation Request Testcase :: 8

Error :: MISSING_FIELD

Revocation Request Format :: NULL,NULL,NULL,CRL_EXTENSIONS

Result :: 8

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Revocation Request Testcase :: 7

Error :: MISSING_FIELD

Revocation Request Format :: CERT_TEMPLATE,REASON_CODE,BAD_SINCE_DATE,
NULL

Header Analysis Result :: 7

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 7

Error :: NONE

Field :: N/A

Revocation Request Testcase :: 6

Error :: MISSING_FIELD

Revocation Request Format :: NULL,REASON_CODE,BAD_SINCE_DATE,NULL

Result :: 6

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Revocation Request Testcase :: 5

Error :: MISSING_FIELD

Revocation Request Format :: CERT_TEMPLATE,NULL,BAD_SINCE_DATE,NULL

Result :: 5

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Revocation Request Testcase :: 4

Error :: MISSING_FIELD

Revocation Request Format :: NULL,NULL,BAD_SINCE_DATE,NULL

Result :: 4

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Revocation Request Testcase :: 3

Error :: MISSING_FIELD

Revocation Request Format :: CERT_TEMPLATE,REASON_CODE,NULL,NULL

Header Analysis Result :: 3

Error :: NONE

Field :: N/A

RP Body Analysis Result :: 3

Error :: NONE

Field :: N/A

Revocation Request Testcase :: 2

Error :: MISSING_FIELD

Revocation Request Format :: NULL,REASON_CODE,NULL,NULL

Result :: 2

Error :: 6316

Error Message :: The CA could not parse the request: general transaction processor error

Revocation Request Testcase :: 1

Error :: MISSING_FIELD

Revocation Request Format :: CERT_TEMPLATE,NULL,NULL,NULL

Result :: 1

Error :: 4245

Error Message :: The CA could not parse the request: general input error

BIBLIOGRAPHY

[1] C. Adams and S. Farrell. *Internet Draft: Internet Public Key Infrastructure Part III: Certificate Management Protocols.* Internet Engineering Task Force, Reston, VA, June 1997.

[2] American National Standards Institute X9.62-199x. *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm*, June 1996. Working Draft.

[3] Boris Beizer. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold Company, Inc., New York, NY, 1984.

[4] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold Company, Inc., New York, NY, 2nd edition, 1990.

[5] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* John Wiley and Sons, Inc., New York, NY, 1995.

[6] Kraig Brockschmidt. *Inside OLE.* Microsoft Press, Redmond, WA, 1995.

[7] W.E. Burr. *TWG-96-102: Public Key Infrastructure (PKI): Technical Specification (Version 2.3): Part C - Concept of Operations.* Federal PKI Technical Working Group, Washington, DC, November 1996.

[8] CygnaCom Solutions, McLean, VA. *Operations and Maintenance Manual: Certificate Authority*, April 1998.

[9] CygnaCom Solutions, McLean, VA. *Operations and Maintenance Manual: Organizational Registration Authority*, April 1998.

[10] CygnaCom Solutions, McLean, VA. *Operations and Maintenance Manual: Registration Client*, April 1998.

[11] CygnaCom Solutions, McLean, VA. *Operations and Maintenance Manual: Signing and Verifying Client*, April 1998.

[12] CygnaCom Solutions, McLean, VA. *Reference Implementation of the Minimum Interoperability Specification for PKI Components Final System Design*, April 1998.

[13] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming.* Academic Press, New York, NY, 1972.

[14] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Lecture Notes in Computer Science*, New York, NY, 1993. Springer-Verlag.

[15] N. Freed and N. Borenstein. *RFC-2045 Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* Internet Engineering Task Force, Reston, VA, November 1996.

[16] Roy S. Freeman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6), June 1991.

[17] Matthew Geller. Test data as an aid in proving program correctness. *Communications of the ACM*, pages 368–375, May 1978.

[18] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, pages 156–173, June 1975.

[19] John S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, pages 686–709, November 1983.

[20] John S. Gourlay. Introduction to the formal treatment of testing. In *Software Validation*, New York, NY, 1984. Elsevier Science Publishers.

[21] P.A.V. Hall. Toward testing with respect to formal specifications. In *Second IEE/BCS Conference on Software Engineering '88*, 1988.

[22] Dieter Hogrefe. Conformance testing based on formal methods. In *Formal Description Techniques III*, New York, NY, 1991. Elsevier Science Publishers.

[23] R. Housley, W. Ford, and D. Solo. *Internet Draft: Internet Public Key Infrastructure Part I: X.509 Certificate and CRL Profile.* Internet Engineering Task Force, Reston, VA, March 1997.

[24] William E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, pages 208–215, September 1976.

[25] Timothy A. Howes. The lightweight directory access protocol: X.500 lite. Technical Report 95–8, Center for Information Technology Integration, University of Michigan, July 1995.

[26] International Telecommunications Union (ITU formerly CCITT), Geneva, Switzerland. *ITU-T Recommendation X.509: Information Technology - Open Systems Interconnection - The Directory: Authentication Framework*, 1997. Draft work-in-progress.

[27] International Telegraph and Telephone Consultative Committee (CCITT), Geneva, Switzerland. *CCITT Recommendation X.500: The Directory*, 1993.

[28] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software*. International Thomson Computer Press, Boston, MA, 2nd edition, 1993.

[29] National Institute of Standards and Technology, Gaithersburg, MD. *Secure Hash Standard: FIPS-PUB-180-1*, April 1985.

[30] National Institute of Standards and Technology, Gaithersburg, MD. *Digital Signature Standard: FIPS-PUB-186*, May 1994.

[31] National Institute of Standards and Technology, Gaithersburg, MD. *Computer Data Authentication, FIPS-PUB-113*, May 1995.

[32] The Open Group, Menlo Park, CA. *Open Group Preliminary Specification: Generic Cryptographic Services*, June.

[33] Charles Petzold. *Programming Windows 95*. Microsoft Press, Redmond, WA, 1996.

[34] Dale Rogerson. *Inside COM*. Microsoft Press, Redmond, WA, 1997.

[35] RSA Data Security Inc., San Mateo, CA. *PKCS #1: RSA Encryption Standard, Version 1.4*, June.

[36] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., New York, NY, 2nd edition, 1996.

[37] Douglas Steedman. *Abstract Syntax Notation One (ASN.1): The Tutorial and Reference*. Technology Appraisals, Ltd., Twickenham, Middlesex, United Kingdom, 1993.

[38] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[39] Philip Alan Stocks. *Applying formal methods to software testing*. PhD thesis, The University of Queensland, Brisbane, Queensland, Australia, December 1993.

[40] Bryan Walters. *Writing Windows Applications using Microsoft Foundation Classes: Covers MFC Version 3.* Henry Holt and Company, New York, NY.

[41] W. Yeong, T. Howes, and S. Kille. *Request For Comments 1777: Lightweight Directory Access Protocol.* Internet Engineering Task Force, Reston, VA, March 1995.