NISTIR 7210

Conformance Testing of the Government Smart Card

Elizabeth Fong
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg MD 20899

February 2005

U.S. Department of Commerce
Technology Administration
National Institute of Standards and Technology

Conformance Testing of the Government Smart Card

Elizabeth Fong
National Institute of Standards and Technology
Information Technology Laboratory
Software Diagnostic and Conformance Testing Division

ABSTRACT

A conformance test suite helps to ensure consistency between a specification and the behavior of a product. This paper presents the conformance testing methodology for the Government Smart Card Interoperability Specification. It starts with some basic terminology in the area of testing and discusses a methodology on how to design conformance test. The test strategy used for the design of this conformance test suite uses the eXtended Markup Language (XML) which is a declarative, implementation-neutral markup language. Finally, the paper explores the benefits and limitations with the conformance testing approach for the Government Smart Card Interoperability Specification.

KEY WORDS: Conformance testing, smart card, specification, test assertions, test suite, XML.

ACKNOWLEDGEMENTS

NIST Computer Security Division (CSD) has the lead for developing technical interoperability and security standards for smart cards for use in the US Government. CSD asked Software Diagnostics and Conformance Testing Division (SDCT) to partner with them to create conformance testing program. Barbara Guttman, manager of Interoperability Group provided technical direction to this project. I thanked my team members, Alan Goldfine and Eric Dalci for their contributions, without which, this project will not be successful.

TABLE OF CONTENTS

DISCLAIMER

1. Introduction

Smart cards have been used throughout the world in a variety of applications. The US Government has been trying to use smart cards to improve security efficiency for a long time. The Administration wants to adopt 'smart card' technology so that, ultimately, every Federal employee will be able to use one card for a wide range of purposes, including travel, small purchases, and building access. Smart cards are also an important part in the implementation of the Homeland Security Presidential Directive (HSPD) #12 "Policy for a Common Identification Standard for Federal Employees and Contractors," Aug, 2004

In 1999, the National Institute of Standards and Technology (NIST) agreed to lead development of specifications and standards related to the Government Smart Card program. A General Services Administration (GSA) Smart Access Common ID Contract was awarded in 2000. This effort has resulted in the publication of the Government Smart Card Interoperability Specification (GSC-IS).

Since specifications are typically written in narrative form, they can often be ambiguous and subject to interpretations. To ensure that implementations faithfully adhere to the specifications, it is necessary to have conformance tests that methodically compare the implementation results against the syntax (the structure) and semantics (the intent) contained in the specification.

In 2001, NIST Software Diagnostics and Conformance Testing Division was tasked to develop a comprehensive Government smart card conformance testing program. The GSC-IS was revised and resulted in GSC-IS Version 2.1 published in July 2003 as NISTIR 6887 [1]. This version was used as basis for the development of a conformance testing program.

The purpose of this paper is to present the conformance testing program for the GSC-IS Version 2.1. We begin with some basic terminology and methodology on how to do conformance testing. We then describe our approach to the design of the conformance testing architecture of the Government Smart Card Interoperability Specification.

2. CONFORMANCE TESTING AND RELATED TERMINOLOGY

The term "**testing**" is used in the general sense of finding errors, but in the broad sense, testing means the activity of evaluating operational software products.

Software testing is the technical operation that determines whether a given software system works or contains errors. There are many types of software testing [2,3,4,5]. This paper only addresses the conformance test methodology.

Conformance testing activities assess whether a software product meets the requirements of a particular specification or standard. To do conformance testing, there must be a written specification or a standard. The specification must contains conformance clauses, that is a section of a specification that states all the requirements or criteria that must be satisfied for an implementation to claim conformance to the specification. Therefore, conformance testing is the process of determining whether or not an implementation exhibits deviations from the specification.

Interoperability testing activities assesses whether a software product will exchange and share information (interoperate) with other products. Conformance testing facilitates interoperability between various products by confirming that each product meets an agreed-upon standard or specification. Therefore, when two implementations of a given specification behave the same to client applications, then the two implementations can interoperate. However, interoperability does not necessarily mean that an implementation is conformant. Two implementations may interoperate while both being non-conformant with the specification. Also, they may be conformant and not interoperate if the specification or standard is not sufficiently detailed.

Validation is the result of evaluating a product and seeing that it meets the requirements as stated in the specification.

Certification is a formal validation program with the establishment of a certificate authority to issue a certificate to the organization whose implementation was tested. The certificate denotes an acknowledgement that a validation has been completed and the criteria established by the certifying organization have been met.

Accreditation is the procedure by which an authoritative body gives formal recognition that an organization or person is competent to carry out validation.

3. COMPONENTS OF CONFORMANCE TESTING

A conformance testing program usually includes the following components:

- A standard or a specification which includes conformance clauses,
- A validation and certification process, and
- An executable test suite.

3.1 Conformance Clause

The conformance clause of a standard or a specification is a high-level description of what is required of implementations to be considered in conformance with the specification. The conformance clause may specify minimal requirements for certain functions and may specify the permissibility of extensions, options, and alternative approaches and how they are to be handled.

The conformance clause appears in Section 1.3 of the GSC-IS Version 2.1 [1], and this forms the basic requirements for the development of the GSC conformance test suite.

3.2 Validation and Certification Process

Government smart card buyers need to be confident that they are procuring smart cards that conform to the GSC-IS. Smart card vendors, in order to provide certified products must submit their implementations to a conformance testing process and earn a certificate of conformance.

The validation and certification process is the agreed-upon process whereby buyer and seller can get the product validated. There are varying degrees of formality in the formulation of the validation program. The most informal of programs occurs when implementers self-test with publicly availably test suites. This is now becoming a common approach in today's internet-time environment. The most formal validation program requires independent, third-party testing by nationally accredited testing laboratories.

For the Government smart card validation and certification process, we recommended a framework involving the following roles:

- Testing Laboratory
- Certification Authority
- Certificate Issuer
- Control Board

These roles and a recommendation of establishing a validation and certification process for the GSC-IS appeared in [6] and will not be repeated in this paper.

3.3 Executable Test Suite

An executable test suite is the software, consisting of code and data, to be executed on the implementation under test (IUT). The approach for the development of the GSC-IS Version 2.1 test suites is described in this paper.

Together with the test suite, there must also be procedures for describing how the test is to be done and the instructions for the tester to follow. The procedures should be detailed enough so that testing of a given implementation can be repeated with no change in test results. The procedures should also contain information on what must be done when failures occur.

4. CONFORMANCE TEST SUITE DESIGN METHODOLOGIES

The design of conformance test approach must first consider the nature of the conformance points. If the conformance points are specified at the interface level, such as application programming interface (API), the design of test requires invoking a call with the API command and checking for the behavior of implementation under test. If the conformance points are defined as finite state machine or event-driven interaction, the design of the test requires testing for each state and the transitions for each state. The GSC-IS Version 2.1 defines conformance at the API level.

The sequence of steps in the development of the conformance test suite typically consists of the following. These are the steps used for the design of GSC-IS Version 2.1.

- The first step is to identify the series of **requirements** for conformance testing based upon the conformance clauses in the specification. In other words, requirements are statements of what the IUT must do or must not do.

- Each requirement translates into one or more **test assertions,** sometimes called abstract test specifications. Each test assertion typically states that if the IUT is in a particular start state, a specified atomic action is performed to the implementation under test, and the IUT must produce the expected result(s) as passing the test, else the IUT fails this assertion.

- To implement the test assertion, a **test scenario or test script** that describes the sequence of test steps to perform a test must be generated. The test sequencing generally consists of a start state condition(s), the instantiation action(s) to be performed, and the verification action(s), if any, to check after the action is executed. Each test scenario will generate one or more test cases.

- The **test case** is a single function to be tested such that the implementation under test must yield a pass/fail answer.

- All of the test cases are grouped logically into a complete **executable test suite.** The test suite can be executed upon the particular hardware/software platform, which the implementation under test could be accepted, and generate an observable output.

- A **test reporting** capability is an important element of the conformance test design. A good test result report should have easy to read and uniform headings. Typically, the test report should contain a header section listing the test environment in which the test was done and a repetitious section that lists each test case executed.

- Finally, for ease of testing, a **test driver or test harness** can be provided. The test harness typically includes a user interface, a sequentially automatic execution scripts, an automatic matching for expected results, and a pass/fail test report.

The above-described methodology for developing conformance test suite can be adjusted to tailor to a particular type of software domain that is being tested. Using the scenario test sequencing approach, one important factor is that there is an assumption that the pre-condition test must work before the actual instantiation test could be issued. Therefore, the ordering of test and some temporal mechanisms must be considered.

4.1 Test Suite Design

Test suite design is the process of detailing the overall test approach. In the case of Government Smart Card conformance testing, the conforming criteria are described in the form of application programming interface (API). API testing requires that the IUT must recognize the program call, behaves accordingly in order to perform the required actions, and produce a result. This type of testing, sometimes referred to as "**black box**" testing [7], involves exercising the IUT with an API call. This means that the IUT is treated as a black-box and its internal structure or source code of the implementation is not examined. The test only tests the external functionality of an implementation – how the implementation appears to client applications.

The **"Falsification Testing"** approach is also used for the GSC-IS Version 2.1 conformance testing. This means that the test assertions are designed as a collection of valid and invalid inputs to the implementation under tested. The first test case is a call to the IUT with valid parameters, and subsequent calls to the IUT consists of combinations and permutations of invalid inputs with the purpose to demonstrate that the IUT indeed produced the expected error code. For each valid and invalid call, the corresponding "expected results" are known and these expected results are compared with the outputs from the IUT to determine a pass/fail test result.

It is generally accepted that designing exhaustive test cases to cover every single legal or illegal cases can be prohibitively expensive, therefore, the Government Smart Card conformance test suite are designed with "reasonable" amount of test coverage.

A good conformance test design should also have the following attributes:

- **Atomicity**. Each test assertion only contains a single, independent purpose.

- **Repeatability.** Every test assertion can be executed repeatedly and produce the same result.

- **Tracebility**. Every test assertion can be trace back to a single requirement in the specification.

4.2 Test Development

Developing conformance testing code can be very time consuming and expensive. There are many test tools in the marketplace aimed at various types of tests [2]. There are also some tools that can automatically generate test code [8, 9]. The disadvantage of these automated tools are that they require the test assertions to be written in formal languages, and the generated test code is sometimes superfluous.

The approach for the development of the GSC-IS Version 2.1 conformance test is the use of eXtended Markup Language (XML) [10]. The test scenarios were first developed in narrative form. These test scenarios were then translated into well-defined and modular

XML format, producing the actual test cases. Each test case is coded in XML in accordance with the corresponding Document Type Definition. An example of the XML for GSC-IS Version 2.1 appears in Appendix A.

The XML files are parsed using JDOM [11], which is a Java API that can read or write XML. The execution of the test cases extracts data from XML, calls the command, and produces the output as HTML test reports.

4.3 Test Report Output

Typically, the test report should contain a header section listing the specific environment in which the test was done. This includes information such as date and place the test is conducted, the name of the operator who executed the test, the hardware, software, OS, the name of the IUT. In the case of the smart card test, the card reader name and the smart card identifier are all listed. The test report also must contain a repetitious section, one for each test case, with a test case number corresponding to the test case definition. The repetitious section should contain a narrative description and purpose or this test, the actual test actions to be performed and the input data values. Most importantly, the test report should list the expected returned codes and expected return values. The report should also list the tested returned codes and returned data values. For the GSC-IS Version 2.1, the test statuses consisting either pass or fail indictor, is given in color green or red. If the test cannot be completed, the test status will be shown in yellow. A narrative assessment or analysis of this test is produced. Depending upon the type of test, it is sometimes necessary to provide test log files.

The GSC-IS Version 2.1 test report is produced in HTML format. An example of the HTML test report for the GSC-IS Version 2.1 appears in Appendix B,

5. ARCHITECTURE OF GSC-IS

This portion of the paper describes the requirements for conformance for the Government Smart Card and the design of the conformance test suite including the XML approach.

The Government Smart Card Interoperability Specification Version 2.1 describes an architectural model (See Figure 1). The following components are discussed.

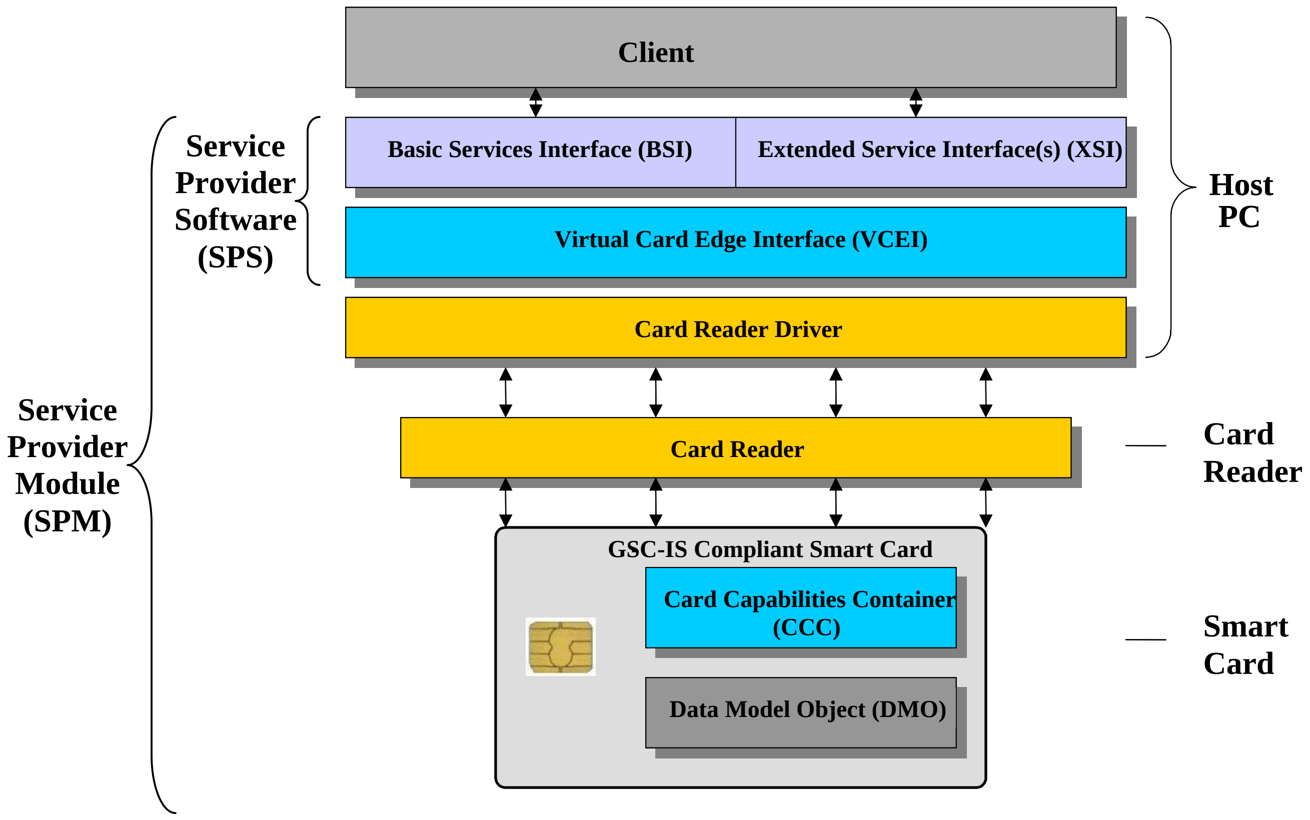

Client
Service Provider Software (SPS)
Basic Services Interface (BSI)
Extended Service Interface(s) (XSI)
Virtual Card Edge Interface (VCEI)
Host PC
Card Reader Driver
Service Provider Module (SPM)
Card Reader
Card Reader
GSC-IS Compliant Smart Card
Card Capabilities Container (CCC)
Smart Card
Data Model Object (DMO)

Figure 1 – GSC-IS Version 2.1 Architectural Model

- The Basic Services Interface (BSI) is a core set of smart card services at the Application Programming Interface (API) layer between client applications and the smart card service provider modules.

- The Extended Services Interface (XSI) which defines additional API functions specific to particular implementations. These additional functions are optional features and are outside of the scope of the conformance requirements.

- The Virtual Card Edge Interface (VCEI) defines from ISO 7816 sets of Application Protocol Data Units (APDUs). The ADPUs are commands sent and received from the card. These sets of ADPUs support two types of smart cards: file system cards and Java virtual machine smart cards.

- The Card Capabilities Container (CCC) is one of the required containers. The purpose of the CCC is to describe the differences between a card’s native APDU set and the set defined in the GSC-IS.

- The GSC Data Model object defines the user data, such as name and social security number, etc. It consists of a set of containers and associated data elements. The GSC-IS Version 2.1 defines two data models and expects future data models to be added. For GSC-IS Version 2.1, the conformant card may implement all, some or none of the data models. However, if the smart card uses any of the data elements defined in either data model then it must use the container and the format specified by the data model for that element.

5.1 Conformance Requirements for GSC-IS Version 2.1

The mandatory conformance requirements for implementations of the GSC-IS Version 2.1 are defined at two levels: the service call level and the card command (APDU) level. The mandatory requirement for the CCC is that it must exist in a specified location and can be retrieved using a standard procedure. The data model and the extended service interface are optional features and are not included in the conformance test.

The service call level is concerned with functional calls required to obtain various services from the card. The GSC-IS Version 2.1 addresses interoperability at this level by defining the Basic Service Interface (BSI) API.

The card command level is concerned with the exact APDUs that are sent to the card to obtain the requested service. The GSC-IS Version 2.1 addresses interoperability at this level by defining the Virtual Card Edge Interface. The VCEI consists of a card-independent, standard set of APDUs implemented by the Service Provider Module. The smart card themselves include the mandatory container called Card Capability Container (CCC). The CCC is a file that contains rules and procedures for translating between the smart card's native APDU set and the standard APDU set defined by the VCEI.

The complete conformance requirements for Government smart cards consist of four separate test suites: conformance test suites at the BSI level for the Java and C bindings, and test suites at the VCEI level for file system and virtual machine cards.

5.1.1 Requirements at the BSI Interface

The Basic Services Interface is a high level API specifying standard smart card services that can be used by client applications. The BSI defines 23 functions grouped into three functional modules: the utility provider module, the generic container provider module, and the cryptographic provider module.

The BSI, as specified in GSC-IS Version 2.1, has two programming language bindings: C and Java. The conforming IUT must support at least one of the language binding.

Conformance testing requirements for the BSI are:

- All GSC-IS conformant Service Provider Modules (SPM) must accept each of the defined BSI function calls.

- BSI functions exposed by the SPM must be callable from external applications.

- For every conformance test scenario, the SPM will behave correctly. That is, the SPM must give the expected results regarding access to and data on the smart card, and return the proper return code, as specified in the GSC-IS.

5.1.2 Requirements at the Virtual Card Edge Interface Level

The Virtual Card Edge Interface (VCEI) is a lower-level, APDU interface between an SPM and a smart card. The GSC-IS Version 2.1 specifies two sets of default APDU commands, each of which, for the most part, conforms to the ISO 7816-4 protocol for the formatting of APDU commands. These two sets are referred as File System (FS) or Virtual Machine (VM) APDU sets. Any differences between a vendor's native APDU set and the GSC-IS Version 2.1 default APDU set are accommodated by the rules and procedures documented in the mandatory Card Capability Container.

The conforming IUT must support both sets of APDU level commands, the file system APDU set and the virtual machine APDU set.

5.1.3 Requirements for Conformant Government Smart Cards

The complete requirements for a GSC-IS Version 2.1 conformant card are:

- The card must have a Card Capability Container whose address is known and contains data and structure as defined in Chapter 6 of the GSC-IS Version 2.1. The conformance test suite tests for the existence of the CCC.

- The card service provider module must implement the BSI with at least one of the language bindings, C or Java.

- The card must support the VCEI that defines default sets of interoperable APDU level commands for both files system and virtual machine Java cards.

6. DESIGN OF THE CONFORMANCE TEST SUITE

Based upon the above requirements for interoperability, a GSC-IS Version 2.1 implementation must be tested for conformance at both the BSI and VCEI levels. The GSC-IS Version 2.1 describes the BSI with two language bindings, Java and C and the VCEI supports two types of cards, file system and virtual machine cards. The complete GSC-IS Version 2.1 conformance test suite consists of 4 separate modules.

6.1 Conformance Testing System

Conformance test systems have been developed for both the C and Java BSI bindings. For each of C or Java, there are 23 sets of BSI functions, corresponding to the 23 commands defined in the GSC-IS Version 2.1. Each BSI function test assertion is defined in narrative form. Each assertion is a statement of the behavior expected of the candidate implementation as it processes the respective command under different conditions. The test assertions are combined together to produce test scenarios that are also written in narrative form [12,13,14,15]. The scenario consists of:

- a starting state,
- a function (C) or method (Java) call with a set of generic legal or illegal input parameters, and
- the return code, generic return values, and return conditions that are anticipated of the candidate implementation, based on the input parameters.

Similarly, the Application Protocol Data Unit (APDU) test assertions, defined for file system cards and for virtual machine Java cards, are written in narrative form. These APDU test assertions are combined together to produce test scenarios that are also written in narrative form [16,17,18.19].

Each test scenario is then coded in XML and validated with the corresponding document type definition (DTD).

6.2 Conformance Test Code

The well-defined and modular XML files are centered on a method call to the BSI implementation being tested, or an APDU call to the VCEI implementation being tested. Each test case is fully defined using XML tags which are defined by the DTD, such as “parameters”, “expected results”, etc.

A test program (Executer) was written in Java to do the parsing (data extraction from the XML files), the execution of the test cases, and the output of the results as HTML test reports.

When a test case is executed, the observed results are compared with the expected results. For example, if a bad card handle is given as input to a method, it is expected that the method will reply with the response code BSI_BAD_HANDLE.

The HTML output indicates the status of each test case (passed, failed, undetermined).

The following figure illustrates the architecture described in this document.

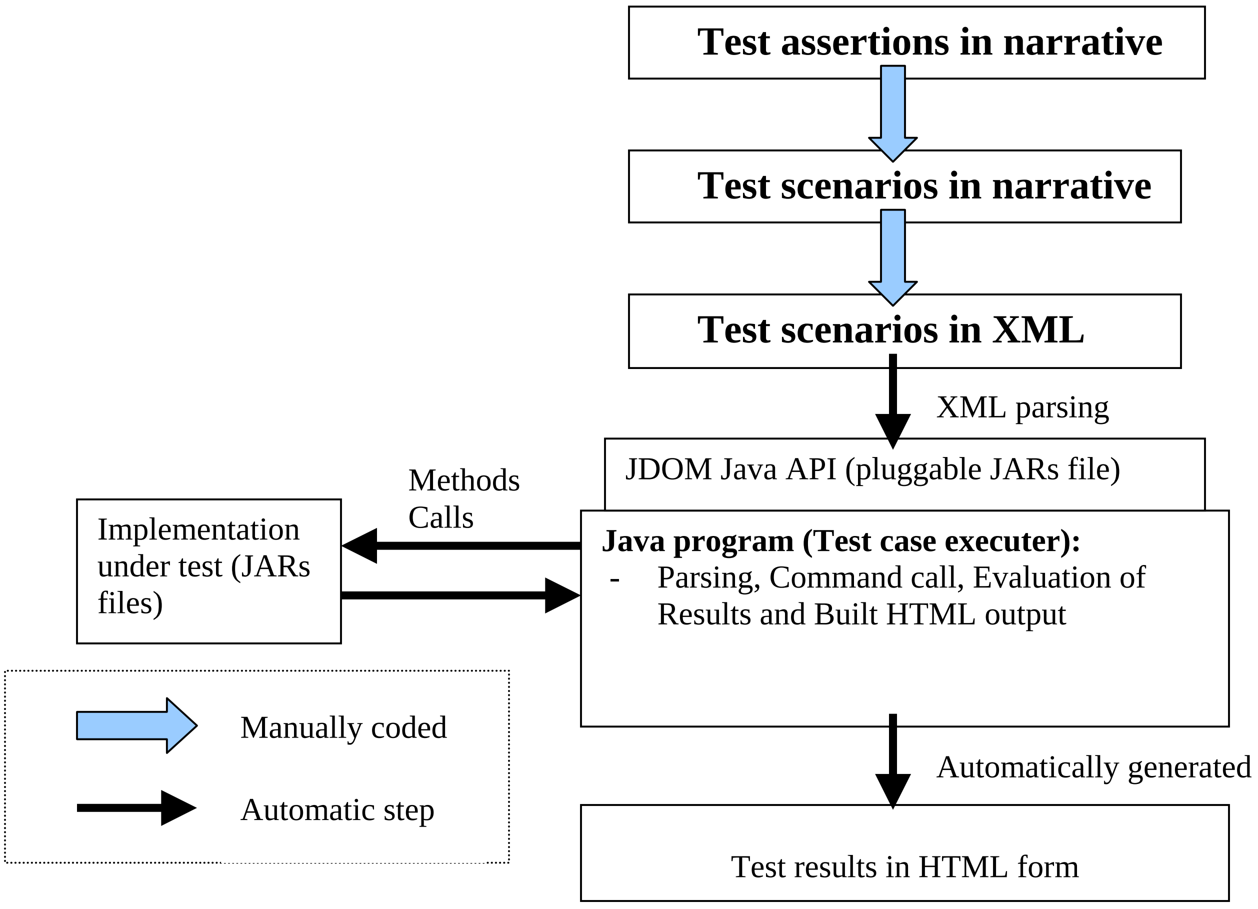

Test assertions in narrative
Test scenarios in narrative
Test scenarios in XML
XML parsing
JDOM Java API (pluggable JARs file)
Methods Calls
Implementation under test (JARs files)
Java program (Test case executer):
- Parsing, Command call, Evaluation of Results and Built HTML output
Manually coded
Automatic step
Automatically generated
Test results in HTML form

Figure 2 – GSC-IS Version 2.1 Conformance Test Architectural Model

6.3 Conformance Test Suite Reports

The test reports for the four test suites (C binding, Java binding, file system and virtual machine) are each constructed using HTML format. Each report contains a header information section specifying the software and hardware platform upon which the suite was executed, identification of the card reader and tested card, the date, and the name of the test operator. The body of the report consists of an account of the execution of each test case. For each test case,

- starting conditions and input parameters
- expected return codes and values, and

- the actual codes and values returned by the candidate implementation are noted. If the actual results are the same as the expected ones, the implementation is considered to have passed the test case, and a "test passed" notice is highlighted in green. Otherwise, the implementation is considered to have failed, and a "test failed" notice is highlighted in red.

In some cases, the test cannot be completed because the start state cannot be accomplished. Those tests notices are highlighted in yellow.

In some cases, the return code is the expected return code, however, the value of the test result requires a visual inspection to check whether the return value is correct. In this case, the test result gives an “inspection” icon to alert the tester for visual inspection.

7 ASSESSMENT OF THE TEST SUITE

This section assesses how well the use of XML to represent test case definition worked. This first version of this test suite design used eXtended Style Sheet Language (XSLT) embedded in the XML to automatically generate source code. This approach worked well with the generation of test report in HTML; however, the generated test source code was not optimal. The source code was quite long due to the fact that the source was carrying all the parameters and value extracting from the XML files with the XSLT engine. It was not easy to design the code through the XSLT style sheet because the code was mixing C, Java, XSLT and HTML. The handling of a returned value and comparison with a reference value later in the test flow was fastidious. The XSLT version was discarded to favor a more natural fit by using JDOM API with Java code.

The test scenario written in XML has many advantages. The general features with this test suite design are as follows:

- **The test suite architecture is modular**. The complete GSC-IS Version 2.1 conformance test suite consists of 4 separate modules. For each module, the commands are individually executable. For instance, the Basic Service Interface, Java binding defines 23 methods. Each method can be individually called to produce a separate test report. This allows for profiles or levels of conformance to be defined for the specification. The definition of profiles or levels of conformance levels means defining a subset of a general list of test requirements. This modular design of the test suite allows for flexibility for users to selectively assemble commands and produce a custom test report.

- **The test scenario is written in eXtended Markup Language (XML).** Because XML is a markup language, it is easy to read and write, thus maintaining the test code at the XML level is easier than going to the programming code level. XML also offers consistency because the XML document is validated according to a structure described by the DTD. The characteristic of well-formedness means every XML document is type-validated and tag-validated. Finally, XML provides portability because XML is language-independent and platform-independent. It is easy to move the test suites from one platform to another.

- **The test suite is highly configurable**. The scope of this test suite ranges from specifying all valid test scenarios as well as a reasonable number of invalid test scenarios. The number of test instance for each BSI command averaged around the number of parameters in the call command together with the number of error codes associated with this command. The XML tree-structured metadata declaration makes it easy to add or delete test cases or regroup test cases in various levels. For example, if one wishes to run only all commands with valid parameters and create a mini test suite, this can be easily done by selecting the appropriate portion of the XML code.

8 CONCLUDING REMARKS

This paper presents the general methodology for conformance testing design. Specifically, this paper presents the conformance testing approach for the Government Smart Card as specified in the GSC-IS Version 2.1. The application of XML to produce test code provides a clean and fast development of tests.

9 REFERENCES

[1] T. Schwarzhoff, J. Dray, J. Wack, E. Dalci, A. Goldfine, and M. Iorga, “Government Smart Card Interoperability Specification” Version 2.1, NISTIR 6887, July 16, 2003.

[2] National Institute of Standards and Technology (NIST), “Metrology for Information Technology,” NISTIR 6025, May 1977.

[3] National Institute of Standards and Technology (NIST), “The Economic Impacts of Inadequate Infrastructure for Software Testing,” Final Report Prepared for G. Tassey, Prepared by RTI, RTI Project Number 7007.011, May 2002.

[4] L. Gallagher, J. Offutt, and A. Cincotta, “Integration Testing of Object-Oriented Components using Finite State Machines,” to appear in Journal on Software Testing, Verification and Reliability (STVR), Draft submitted publication, March 2001, Revision resubmitted March 2002, Second revision September 2004.

[5] W. Oberkampf, T. Trucano, and C. Hirsch, “Verificartion, Validation, and Predictive Capbility in Computational Engineering and Physics, in J. Fong and R. deWit (eds.) Verification & Validation of Computer Models for Design and Performance Evaluation of High-Consequence Engineering System. November 2004.

[6] E. Fong, **“**Conformance Test Framework for Government Smart Card – A White Paper,” June 2002, http://xw2k.sdct.itl.nist.gov/smartcard/.

[7] B. Beizer, “Black-Box Testing – Techniques for Functional Testing of Software and Systems,” John Wiley & Sons, Inc. ISBN 0-471-12094-4, 1995.

[8] P. Black, “Automatic Test Generation from Formal Specification,” http://hissa.nist.gov/~Black/FTG/autotest.html.

[9] Chandramouli, R, Blackburn, M., “Automated Testing of Security Functions Using a combined Model & Interface-driven Approach" - Proceedings of the Thirty-Seventh Hawaii International Conference on System Sciences (HICSS-37), Big Island, HI, January 2004.

[10] Extensible Markup Language (XML) Conformance Test Suites, http://xw2k.sdct.itl.nist.gov/brady/xml/generate.asp?tech=XML

[11] The JDOM project, http://www.jdom.org.

[12] A. Goldfine,“Government Smart Card Interoperability Specification V2.1 (NISTIR 6887 – 2003 Edition) Basic Services Interface Java Binding Conformance Test Assertion.” http://xw2k.sdct.itl.nist.gov/smartcard/.

[13] A. Goldfine,“Government Smart Card Interoperability Specifcation V2.1 (NISTIR 6887 – 2003 Edition) Basic Services Interface Java Binding Conformance Test Instantiation, Verification, and Reporting Scenarios.” http://xw2k.sdct.itl.nist.gov/smartcard/.

[14] A. Goldfine,“Government Smart Card Interoperability Specification V2.1 (NISTIR 6887 – 2003 Edition) Basic Services Interface C Binding Conformance Test Assertion.” http://xw2k.sdct.itl.nist.gov/smartcard/.

[15] A. Goldfine,“Government Smart Card Interoperability Specifcation V2.1 (NISTIR 6887 – 2003 Edition) Basic Services Interface C Binding Conformance Test Instantiation, Verification, and Reporting Scenarios.” http://xw2k.sdct.itl.nist.gov/smartcard/.

[16] A. Goldfine,“Government Smart Card Interoperability Specification V2.1 (NISTIR 6887 – 2003 Edition) Virtual Card Edge Interface File System Cards Conformance Test Assertion.” http://xw2k.sdct.itl.nist.gov/smartcard/

[17] A. Goldfine,“Government Smart Card Interoperability Specifcation V2.1 (NISTIR 6887 – 2003 Edition) Virtual Card Edge Interface File System Cards Conformance Test Instantiation, Verification, and Reporting Scenarios.” http://xw2k.sdct.itl.nist.gov/smartcard/

[18] A. Goldfine,“Government Smart Card Interoperability Specification V2.1 (NISTIR 6887 – 2003 Edition) Virtual Card Edge Interface Virtual Machine Cards Conformance Test Assertion.” http://xw2k.sdct.itl.nist.gov/smartcard/

[19] A. Goldfine,“Government Smart Card Interoperability Specifcation V2.1 (NISTIR 6887 – 2003 Edition) Virtual Card Edge Interface, Virtual Machine Cards Conformance Test Instantiation, Verification, and Reporting Scenarios.” http://xw2k.sdct.itl.nist.gov/smartcard/

APPENDIX A

The following Document Type Definition is the schema for XML Java Binding of the BSI Test Scenario

```
<!--  August 16, 2004  Elizabeth Fong -->
<!-- This files represents the DTD used to build the XML Test Scenario for the Java Binding  of the
BSI -->
<!-- DTD start -->
<!ELEMENT test-method (method+)>
<!ELEMENT method (description+, reference+, BSI-definition, assertions+)>
<!ATTLIST method
    method-name CDATA #REQUIRED
    number CDATA #REQUIRED
>
<!--the attribute "number" is a reference number for parameter naming -->
<!ELEMENT description (#PCDATA)>
<!ELEMENT reference (#PCDATA)>
<!ELEMENT BSI-definition (commandBSI+)>
<!ELEMENT commandBSI (#PCDATA)>
<!ELEMENT assertions (purpose, parameters*, sequenceBSI*)>
<!ATTLIST assertions
    id CDATA #REQUIRED
    test CDATA #IMPLIED
>
<!-- the attribute "test" gives an optional information about the assertion -->
<!ELEMENT purpose (#PCDATA)>
<!ELEMENT parameters ANY>
<!ATTLIST parameters
    way (IN | OUT) #REQUIRED
    type CDATA #REQUIRED
    pname CDATA #REQUIRED
>
<!-- "pname" is the parameter's name -->
<!ELEMENT sequenceBSI (BSIset+)>
<!ELEMENT BSIset (test-type-description, local-param*, method-call, expected-results*, print*)>
<!ATTLIST BSIset
    caseNo CDATA #REQUIRED
    testType (CON | PRE | INST | TEXT | OTH | VER | END | DISC) #REQUIRED
    opMode (REPEAT | PAUSE | PAUSE_BAD_CARD_REQUIRED) #IMPLIED
>
<!-- testType :
    CON : Smart card Connection call
    PRE : Precondition call
    INST : Instanciation call
    TEXT : Extended error text call
    OTH : Other call (used for intermediate calls)
    VER : Verification call
    END : End call ("clean up" calls)
    DISC : Smart card Disconnection call
-->
<!-- opMode :
    REPEAT : indicates that the call should be repeated a few times (e.g PIN blocking)
    PAUSE : indicates that the test should pause and get the tester's input.
    PAUSE_BAD_CARD_REQUIRED: indicate that the test should pause and require the insertion of
a bad card
-->
<!ELEMENT test-type-description (#PCDATA)>
<!ELEMENT local-param ANY>
<!ATTLIST local-param
    way (IN | OUT) #REQUIRED
    type CDATA #REQUIRED
    pname CDATA #REQUIRED
>
<!ELEMENT method-call (BSIset-param*)>
<!ATTLIST method-call
    method-name (gscBsiUtilAcquireContext | gscBsiUtilConnect | gscBsiUtilDisconnect |
gscBsiUtilBeginTransaction | gscBsiUtilEndTransaction | gscBsiUtilGetVersion |
gscBsiUtilGetCardProperties | gscBsiUtilGetCardStatus | gscBsiUtilGetExtendedErrorText |
gscBsiUtilGetReaderList | gscBsiUtilPassthru | gscBsiUtilReleaseContext | gscBsiGcDataCreate |
gscBsiGcDataDelete | gscBsiGcGetContainerProperties | gscBsiGcReadTagList |
gscBsiGcReadValue | gscBsiGcUpdateValue | gscBsiGetChallenge | gscBsiSkiInternalAuthenticate |
gscBsiPkiCompute | gscBsiPkiGetCertificate | gscBsiGetCryptoProperties) #REQUIRED
>
<!ELEMENT BSIset-param (#PCDATA)>
<!ATTLIST BSIset-param
    way (IN | OUT) #IMPLIED
    type CDATA #REQUIRED
    pname CDATA #REQUIRED
>
<!ELEMENT expected-results (exception*, return-value*)>
<!ELEMENT exception (#PCDATA)>
<!ATTLIST exception
    messageF CDATA #IMPLIED
    messageP CDATA #IMPLIED
    messageNT CDATA #IMPLIED
>
<!-- messageF : message associated with an Exception which Fails the test
    messageP : message associated with an Exception which Passes the test
    messageNT : message associated with an Exception which makes the test Not Testable
-->
<!ELEMENT return-value ANY>
<!ATTLIST return-value
    type (String | int | ArrayByte | ArrayShort | Short | Vector | CardProperties | CryptoProperties |
ContainerProperties) #REQUIRED
    pname CDATA #IMPLIED
>
<!ELEMENT print (#PCDATA)>
<!ATTLIST print
    printpass CDATA #REQUIRED
    printfail CDATA #REQUIRED
>
<!--
     printpass : default message associated with a successful test
     printfail : default message associated with an unsuccessful test
-->
```

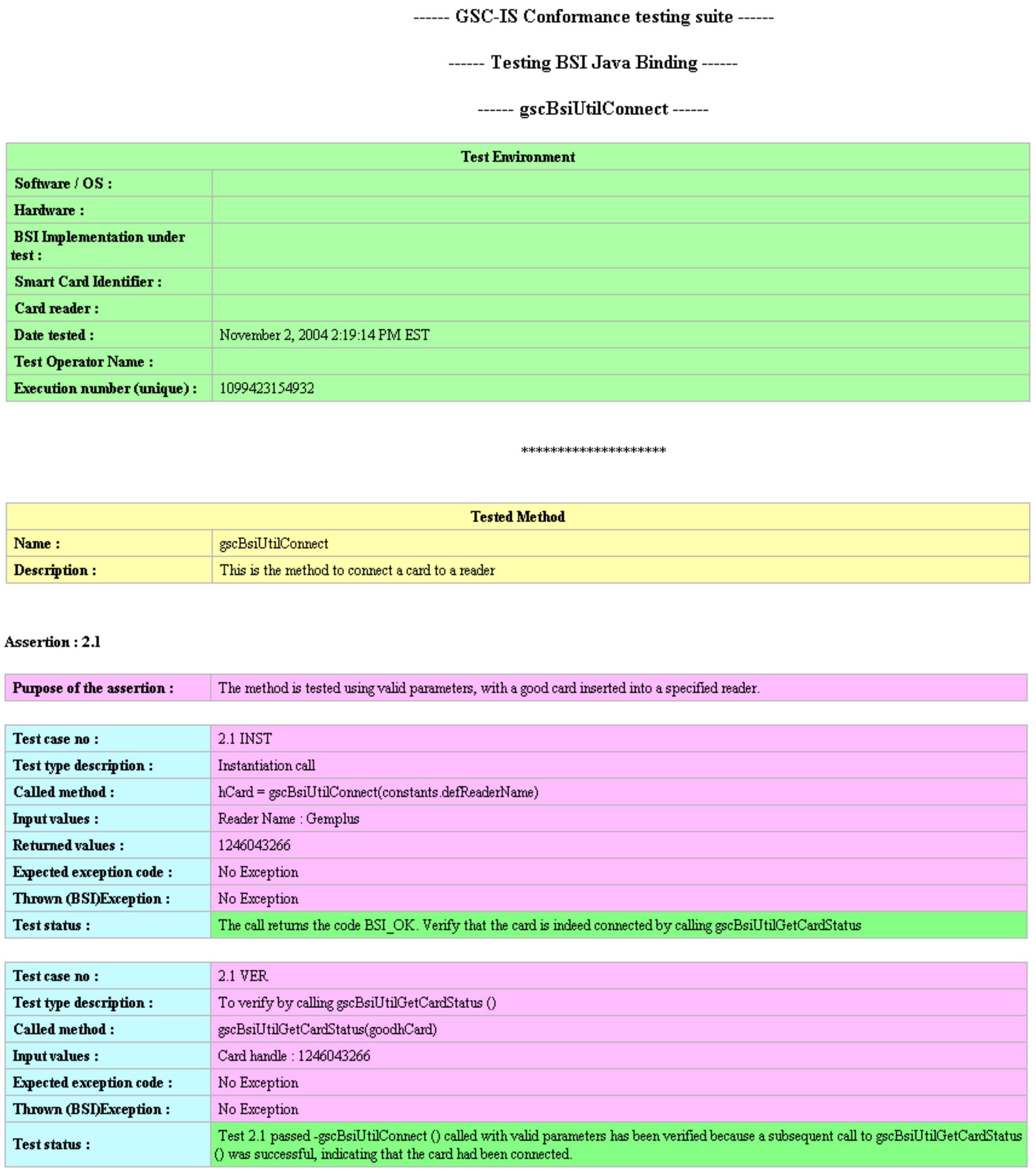
The following XML file is the Java BSI method gscBsiUtilConnect ()

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE test-method SYSTEM "jsdtd.dtd">
<!--Connect-  BSI  Java Binding- September 14, 2004 Elizabeth Fong- -->
<test-method>
    <method method-name="gscBsiUtilConnect" number="0200">
        <description>This is the method to connect a card to a reader</description>
        <reference>GSC-IS 4.4, F.2.2</reference>
        <BSI-definition>
            <commandBSI>gscBsiUtilConnect(readerName)</commandBSI>
        </BSI-definition>
        <assertions id="2.1">
            <purpose>The method is tested using valid parameters, with a good card inserted into a
specified reader.</purpose>
            <sequenceBSI>
                <BSIset caseNo="2.1 INST" testType="INST">
                    <test-type-description>Instantiation call</test-type-description>
                    <method-call method-name="gscBsiUtilConnect">
                        <BSIset-param type="String"
pname="readerName0200">constants.defReaderName</BSIset-param>
                    </method-call>
                    <expected-results>
                        <return-value type="String">hCard</return-value>
                    </expected-results>
                    <print printfail="Test 2.1 failed - gscBsiUtilConnect () called with valid parameters
with a good card inserted into a specified reader returned an incorrect code." printpass="The call
returns the code BSI_OK.  Verify that the card is indeed connected by calling
gscBsiUtilGetCardStatus"/>
                </BSIset>
                <BSIset caseNo="2.1 VER" testType="VER">
                    <test-type-description>To verify by calling gscBsiUtilGetCardStatus ()</test-type-
description>
                    <method-call method-name="gscBsiUtilGetCardStatus">
                        <BSIset-param type="int" pname="hCard">goodhCard</BSIset-param>
                    </method-call>
                    <print printfail="Test 2.1 failed - gscBsiUtilConnect ()  called with valid paramters has
not been verified because a subsequent call to gscBsiUtilGetCardStatus () was unsuccessful,
indicating that the card had not been connected. " printpass="Test 2.1 passed -gscBsiUtilConnect ()
called with valid parameters  has  been verified because a subsequent call to
gscBsiUtilGetCardStatus () was successful, indicating that the card had been connected. "/>
</BSIset>
            </sequenceBSI>
        </assertions>
        <assertions id="2.2">
            <purpose>The method is tested using valid parameters, with a good card inserted into a
non-specified reader.</purpose>
            <sequenceBSI>
                <BSIset caseNo="2.2 INST" testType="INST">
                    <test-type-description>Instantiation call</test-type-description>
                    <local-param way="OUT" type="int" pname="hCard0200"/>
                    <method-call method-name="gscBsiUtilConnect">
                        <BSIset-param type="String" pname="emptyString">emptyString</BSIset-
param>
                    </method-call>
                    <expected-results>
                        <return-value type="String">hCard0200</return-value>
                    </expected-results>
                    <print printfail="Test 2.2 failed - gscBsiUtilConnect () called with valid parameters
with a good card inserted into a non-specified reader returned an incorrect code." printpass="The call
returns the code BSI_OK.  Verify that the card is indeed connected by calling
gscBsiUtilGetCardStatus"/>
                </BSIset>
                <BSIset caseNo="2.2 VER" testType="VER">
                    <test-type-description>To verify by calling gscBsiUtilGetCardStatus ()</test-type-
description>
                    <method-call method-name="gscBsiUtilGetCardStatus">
                        <BSIset-param type="int" pname="hCard">goodhCard</BSIset-param>
                    </method-call>
                    <expected-results/>
                    <print printfail="Test 2.2 failed - gscBsiUtilConnect () called with valid parameters
with a good card inserted into a non-specified reader has not been verified because a subsequent call
to gscBsiUtilGetCardStatus () was unsuccessful, indicating that the card had not been connected."
printpass="Test 2.2 passed -gscBsiUtilConnect () called with valid parameters with a good card
inserted into a non-specified reader has been verified because a subsequent call to
gscBsiUtilGetCardStatus () was successful, indicating that the card had been connected "/>
    </BSIset>
            </sequenceBSI>
        </assertions>
        <assertions id="2.3">
            <purpose>The method is tested using a bad reader name.</purpose>
            <sequenceBSI>
                <BSIset caseNo="2.3 INST" testType="INST">
                    <test-type-description>Instantiation call</test-type-description>
                    <method-call method-name="gscBsiUtilConnect">
                        <BSIset-param type="String"
pname="badreaderName">constants.badReaderName</BSIset-param>
                    </method-call>
                    <expected-results>
                        <exception>BSI_UNKNOWN_READER</exception>
                        <return-value type="int">hCard</return-value>
                    </expected-results>
                    <print printfail="Test 2.3 failed - gscBsiUtilConnect () called with a bad reader name
returned incorrect code." printpass="Test 2.3 passed -gscBsiUtilConnect () called with a bad reader
name has been verified."/>
                </BSIset>
            </sequenceBSI>
        </assertions>
        <assertions id="2.5">
            <purpose>The method is tested using a bad inserted card.   Manual Instruction: BAD
CARD IN THE READER.</purpose>
            <sequenceBSI>
                <BSIset caseNo="2.5 INST" testType="INST"
opMode="PAUSE_BAD_CARD_REQUIRED">
                    <test-type-description>Instantiation call</test-type-description>
                    <local-param way="OUT" type="int" pname="hCard0200"/>
                    <method-call method-name="gscBsiUtilConnect">
                        <BSIset-param type="String"
pname="readerName0200">constants.defReaderName</BSIset-param>
                    </method-call>
                    <expected-results>
```

```
                        <exception>BSI_CARD_ABSENT</exception>
                        <exception>BSI_UNKNOWN_ERROR</exception>
</expected-results>
                    <print printfail="Test 2.5 failed -gscBsiUtilConnect () called with a bad inserted card
returned  an incorrect  code." printpass="Test 2.5 passed -gscBsiUtilConnect () called with a bad
inserted card has been verified."/>
                </BSIset>
            </sequenceBSI>
        </assertions>
        <assertions id="2.4">
            <purpose>The method is tested with no card in the reader.  Manual Instruction: NO CARD
IN THE READER.</purpose>
            <sequenceBSI>
                <BSIset caseNo="2.4 INST" testType="INST" opMode="PAUSE">
                    <test-type-description>Instantiation call</test-type-description>
                    <method-call method-name="gscBsiUtilConnect">
                        <BSIset-param type="String"
pname="readerName0200">constants.defReaderName</BSIset-param>
                    </method-call>
                    <expected-results>
                        <exception>BSI_CARD_ABSENT</exception>
                        <return-value type="int">hCard</return-value>
                    </expected-results>
                    <print printfail="Test 2.4 failed -gscBsiUtilConnect () called with no card in the reader
returned an  incorrect code." printpass="Test 2.4 passed -gscBsiUtilConnect () called with no card in
the reader has been verified. "/>
                </BSIset>
            </sequenceBSI>
        </assertions>
    </method>
</test-method>
```

APPENDIX B

HTML Test Report for the Basic Service Interface Java Binding



------ GSC-IS Conformance testing suite ------

------ Testing BSI Java Binding ------

------ gscBsiUtilConnect ------

Test Environment	
Software / OS :	
Hardware :	
BSI Implementation under test :	
Smart Card Identifier :	
Card reader :	
Date tested :	November 2, 2004 2:19:14 PM EST
Test Operator Name :	
Execution number (unique) :	1099423154932

****

Tested Method	
Name :	gscBsiUtilConnect
Description :	This is the method to connect a card to a reader

Assertion : 2.1

Purpose of the assertion :	The method is tested using valid parameters, with a good card inserted into a specified reader.

Test case no :	2.1 INST
Test type description :	Instantiation call
Called method :	hCard = gscBsiUtilConnect(constants.defReaderName)
Input values :	Reader Name : Gemplus
Returned values :	1246043266
Expected exception code :	No Exception
Thrown (BSI)Exception :	No Exception
Test status :	The call returns the code BSI_OK. Verify that the card is indeed connected by calling gscBsiUtilGetCardStatus

Test case no :	2.1 VER
Test type description :	To verify by calling gscBsiUtilGetCardStatus ()
Called method :	gscBsiUtilGetCardStatus(goodhCard)
Input values :	Card handle : 1246043266
Expected exception code :	No Exception
Thrown (BSI)Exception :	No Exception
Test status :	Test 2.1 passed -gscBsiUtilConnect () called with valid parameters has been verified because a subsequent call to gscBsiUtilGetCardStatus () was successful, indicating that the card had been connected.

Assertion : 2.2

Purpose of the assertion :	The method is tested using valid parameters, with a good card inserted into a non-specified reader.

Test case no :	2.2 INST
Test type description :	Instantiation call
Called method :	hCard0200 = gscBsiUtilConnect(emptyString)
Input values :	Reader Name :
Returned values :	1246043266
Expected exception code :	No Exception
Thrown (BSI)Exception :	No Exception
Test status :	The call returns the code BSI_OK. Verify that the card is indeed connected by calling gscBsiUtilGetCardStatus

Test case no :	2.2 VER
Test type description :	To verify by calling gscBsiUtilGetCardStatus ()
Called method :	gscBsiUtilGetCardStatus(goodhCard)
Input values :	Card handle : 1246043266
Expected exception code :	No Exception
Thrown (BSI)Exception :	No Exception
Test status :	Test 2.2 passed -gscBsiUtilConnect () called with valid parameters with a good card inserted into a non-specified reader has been verified because a subsequent call to gscBsiUtilGetCardStatus () was successful, indicating that the card had been connected

Assertion : 2.3

Purpose of the assertion :	The method is tested using a bad reader name.

Test case no :	2.3 INST
Test type description :	Instantiation call
Called method :	hCard = gscBsiUtilConnect(constants.badReaderName)
Input values :	Reader Name : FF
Expected exception code :	BSI_UNKNOWN_READER
Thrown (BSI)Exception :	BSI_UNKNOWN_READER
Test status :	Test 2.3 passed -gscBsiUtilConnect () called with a bad reader name has been verified.

Assertion : 2.5

Purpose of the assertion :	The method is tested using a bad inserted card. Manual Instruction: BAD CARD IN THE READER.

Test case no :	2.5 INST
Test type description :	Instantiation call
Called method :	gscBsiUtilConnect(constants.defReaderName)
Input values :	Reader Name : Gemplus
Expected exception code :	BSI_CARD_ABSENT or BSI_UNKNOWN_ERROR
Thrown (BSI)Exception :	BSI_CARD_ABSENT
Test status :	Test 2.5 passed -gscBsiUtilConnect () called with a bad inserted card has been verified.

Assertion : 2.4

Purpose of the assertion :	The method is tested with no card in the reader. Manual Instruction: NO CARD IN THE READER.

Test case no :	2.4 INST
Test type description :	Instantiation call
Called method :	hCard = gscBsiUtilConnect(constants.defReaderName)
Input values :	Reader Name : Gemplus
Expected exception code :	BSI_CARD_ABSENT
Thrown (BSI)Exception :	BSI_CARD_ABSENT
Test status :	Test 2.4 passed -gscBsiUtilConnect () called with no card in the reader has been verified.

****

Test result summary for method : gscBsiUtilConnect				
Assertion No	**Precondition call :**	**Instantiation call :**	**Verification call :**	**Final result of testing**
2.1	- -	2.1 INST gscBsiUtilConnect	2.1 VER gscBsiUtilGetCardStatus	Passed
2.2	- -	2.2 INST gscBsiUtilConnect	2.2 VER gscBsiUtilGetCardStatus	Passed
2.3	- -	2.3 INST gscBsiUtilConnect	- -	Passed
2.5	- -	2.5 INST gscBsiUtilConnect	- -	Passed
2.4	- -	2.4 INST gscBsiUtilConnect	- -	Passed