# Encapsulation and Diagnosis with Fault Dictionaries

William R. Simpson
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, Virginia 22311
rsimpson@ida.org

John W. Sheppard
ARINC
2551 Riva Road
Annapolis, Maryland 21122
sheppard@arinc.com

***Abstract*–To date, test and diagnosis has been domain knowledge driven. However, as system complexity grows and we strive to develop reusable components, the concept of encapsulation becomes increasingly important. Encapsulation embodies the concepts of separation and partitioning. In this paper we deal with encapsulation by illustration of the fault dictionary approach to digital electronics. We then extend the concept of encapsulation to the system test approach as well as the development of maintenance systems. Finally we develop the concept that encapsulation is a key element in achieving general standardization open system architectures.**

## I. Introduction

Of the work performed in the area of fault diagnosis of electronic systems, diagnosis of complex digital circuits continues to be a difficult problem because of the large number of possible conditions under which a circuit can operate and fail. Problems related to state dependence, timing and race conditions, circuit density, design errors, manufacturing flaws, field failures, and other sources result in the many failure modes that must be considered in testing a circuit.

When fully automatic testing is required, test engineers rely on the fault dictionary to provide the diagnostics. Test engineering for digital electronics involves using digital simulation to determine input and output vectors that detect various faults within the circuit. Through a process of fault insertion and pattern generation, input vectors and faulty output vectors can be combined and associated with the inserted faults that cause the changes in output to occur. These vectors can be assembled into fault dictionaries to use for circuit diagnosis.

Because of the complexity associated with building and running fault simulations, most digital simulations have been limited to the "single stuck-at" fault model. This assumes that most failures of a circuit can be detected (and isolated) using tests designed to look for single stuck-at faults.

Fault dictionaries work well when fault signatures are recognized by the fault dictionary. Unfortunately, the approach fails completely when faults that may be in the system are not included in the fault dictionary. Under ideal circumstances, tests applied to digital systems result in fault signatures that exactly match signatures in the fault dictionary. These matches uniquely identify the faults in the system. However, under many conditions, errors may be introduced, resulting in no exact match being found in the fault dictionary. This mismatch may be due to errors in testing, noise present in the system, modeling errors, violation of the analysis assumptions, the presence of indeterminate states, and many other factors that are not addressed here.

A large number of mismatches should be anticipated in sequential logic circuits where the presence of faults may actually mask the state of the system. Using the fault dictionary, configured by a good system paradigm with distance-based matching algorithms can lead to improper identification of failures in a system and ineffective repair.

We will first review the basis of fault dictionary approaches and a simple example where encapsulation is not an issue (the combinational circuit). We will then discuss why the encapsulation concept must be controlled in the sequential circuit and discuss ways to approach the building of fault dictionaries with encapsulation concepts.

## II. Diagnosis With Fault Dictionaries

Fault dictionaries define a mapping from combinations of input vectors and output vectors to faults. Formally, this is represented as $FD: I \times O \rightarrow F$ where $FD$ is the fault dictionary, $I$ is the space of input vectors, $O$ is the space of output vectors, and $F$ is the space of faults. At a more basic level, this can be represented as $FD: \{0,1\}^n \times \{0,1\}^m \rightarrow F$. This represents the fact that the vectors are binary. In the simplest case, diagnosis can be performed with a fault dictionary by finding a direct match between the input/output vectors and a fault signature in the dictionary. Indeed, with a proper model, high confidence tests, and a reasonable fault universe, many faults will be identified in this manner.

For illustration purposes, we use a simple digital circuit [1]. This circuit is given in Figure 1. From this figure, and assuming a single stuck-at fault model, we can identify 26 possible stuck-at faults. Each stuck-at fault is denoted as $x_i$

0-7803-3379-9/96 $5.00 © 1996 IEEE

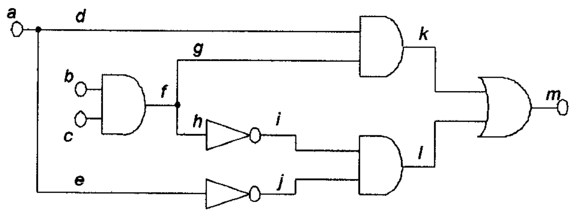

**Figure 1.** Sample combinational circuit.

where $x$ is a letter matching the line where the fault occurs, and $i$ is either 0 or 1 (denoting stuck-at-0 or stuck-at-1, respectively). We close the fault universe by defining a special "fault" in which no fault, *nf*, has been detected. The fault dictionary would then include the input vectors (i.e., the patterns applied to lines $a$, $b$, and $c$) and the expected response vector (in this case the value at $m$). Also associated with that entry would be the list of faults detected should the response be in error.

We see in this circuit (Figure 1) only three input lines and, therefore, only eight possible input vectors (disregarding timing and faults other than stuck-ats). For our example, we can examine all eight input vectors; however in general, enumerating all possible vectors would be too costly. If the circuit were sequential, the three input lines might require several additional tests because of the sensitivity of the circuit to the previous state of the circuit. Several tools, such as LASAR [2][3], provide assistance to the modeler in developing input vectors and detecting stuck-ats and other faults at output vectors.

Limiting ourselves to the combinational case (and the example in Figure 1), we begin constructing the fault dictionary by considering the possible input vectors. Each input vector can be regarded as a test. For example, one test might be the vector (0 1 1). Tracing through the circuit, we would expect the output of the circuit to be (0). If the value is (1), a fault must be present in the circuit. The question then becomes, what failure modes (i.e., stuck-at faults) can cause the erroneous output? Examining the circuit identifies $a_1$, $b_0$, $c_0$, $d_1$, $f_0$, $i_1$, $h_0$, $k_1$, $l_1$, or $m_1$ as possible causes.

By examining all eight input vectors, we find that several failure modes are "ambiguous," meaning no test vectors can differentiate them. These ambiguity groups, which were taken from [1] are listed in Table 1. The approach used for determining these ambiguous faults is called "fault collapsing" and consists of identifying lines in the circuit that will have identical values regardless of input or fault because of the logical nature of the gates in the circuit. For example, if we examine the initial AND gate with inputs $b$ and $c$, we note $b_0$, $c_0$, and $f_0$ are indistinguishable because either $b_0$ or $c_0$ (or both) will force $f$ to have a value of zero, whether or not $f$ is faulty. This approach to ambiguity analysis provides a first, but incomplete, cut on the ambiguity in the fault dictionary.

**Table 1.** Ambiguity groups for sample circuit.

| Number | Ambiguity Group | Number | Ambiguity Group |
|---|---|---|---|
| 1 | $a_0$ | 8 | $g_1$ |
| 2 | $a_1$ | 9 | $i_0$, $h_1$, $l_0$, $j_0$, $e_1$ |
| 3 | $b_1$ | 10 | $i_1$, $h_0$ |
| 4 | $c_1$ | 11 | $j_1$, $e_0$ |
| 5 | $d_1$ | 12 | $k_0$, $d_0$, $g_0$ |
| 6 | $f_0$, $b_0$, $c_0$ | 13 | $k_1$, $l_1$, $m_1$ |
| 7 | $f_1$ | 14 | $m_0$ |

An approach to representing the fault dictionary is to construct a table, such as Table 2, in which each test (i.e., input vector) corresponds to a row in the table. The columns of the table correspond to the first member of each ambiguity group in the circuit. Each cell in the table contains the expected output from the circuit. This dictionary assumes eight tests as follows:

$t_1: 0\ 1\ 1$ $t_2: 1\ 1\ 0$ $t_3: 1\ 0\ 1$ $t_4: 1\ 1\ 1$
$t_5: 0\ 0\ 1$ $t_6: 0\ 0\ 0$ $t_7: 0\ 1\ 0$ $t_8: 1\ 0\ 0$

### *A. Diagnosis with The Fault Dictionary*

Given the single stuck-at fault model, we assume that the circuit simulation accurately reflects the performance of the actual circuit. In other words, we assume that (1) the only faults of interest to us are stuck-at faults, (2) these faults are accurately represented in the circuit model, and (3) only one of these faults will be encountered at a time. Given these assumptions and the fact that digital circuit models are deterministic (i.e., the outputs are directly determined by the inputs and, in the case of sequential circuits, the internal state), whenever the fault signature fails to match any signatures in the fault dictionary, the circuit must be exhibiting behavior that was not represented in the model (i.e., the problem lies in the model, not the test results).

Diagnosis matches the results of running the tests with the columns in the table. For example, suppose we run all eight tests and get (1 0 0 1 1 1 1 0) as the set of responses. This pattern would match both $d_1$ and $i_1$, indicating ambiguity between the two associated groups. It is significant that ambiguity is determined by the actual tests used to test the circuit, and selecting a subset of possible test vectors could result in different ambiguity groups. For example, if we evaluated only $t_1$, $t_2$, $t_3$, and $t_4$, we would find that $a_1$ with $d_1$ and $i_1$ forms a new ambiguity group.

Debaney and Unkle [4] assert, "In practice, it is very seldom that an observed fault signature has an exact match in the fault dictionary." This assertion points to the need for as

Table 2. Fault dictionary for sample circuit.

| Test | Fault Signatures | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $a_0$ | $a_1$ | $b_1$ | $c_1$ | $d_1$ | $f_0$ | $f_1$ | $g_1$ | $i_0$ | $i_1$ | $j_1$ | $k_0$ | $k_1$ | $m_0$ | $nf$ |
| $t_1$ | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| $t_2$ | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| $t_3$ | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| $t_4$ | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| $t_5$ | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| $t_6$ | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| $t_7$ | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| $t_8$ | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

reconsideration of the fault dictionary process. The current practice for processing inexact matches in fault dictionaries applies various distance measures to find the column in the dictionary that most closely matches the target vector. In the case where the single nearest vector is used to identify the fault, we refer to the matching process as *single nearest neighbor classification* (i.e., 1-NN). In the general case, nearest neighbor can be performed by retrieving the *k* nearest neighbors (i.e., *k*-NN) and voting for one of the recommended diagnoses. The winner is reported as the diagnosis [5]. Improvements to this process have been reported by Sheppard and Simpson [6].

### *B. Sources of Error In Combinational Circuits*

The process of extracting a signature from mismatches only makes sense for the case of a "noisy" output. If the mismatch is due to loss of state, then the mismatch cannot be handled as a misidentification. If we examine the test vectors that are involved in the combinational circuit, we see that they are all independent of one another. There is no sequencing required and each test stands alone. Thus, as in the fault dictionary, information derived from each test can be added or combined with information from any other test without fear of misinformation. The resulting "fusion" algorithm is a straight forward intersection of sets, and any error present is probably due to noise in test outputs and/or inputs. Even so, it has been shown that using a mismatch algorithm to estimate an answer can lead unnecessarily to error and is not a good choice for a diagnostic algorithm [6].

### *C. Sources of Error In a Sequential Circuits*

Figure 2 shows a sample sequential circuit. It has been made deliberately simple for purposes of illustration. The memory associated with the flip-flop has changed a number of things about the testing of this circuit. First, a sequence of tests is no longer independent of its history unless we strive to make it so. Each time we change the value of a or c we may change the value of the memory element. Note that we detect an error in the output of the flip-flop, we have lost the state of the

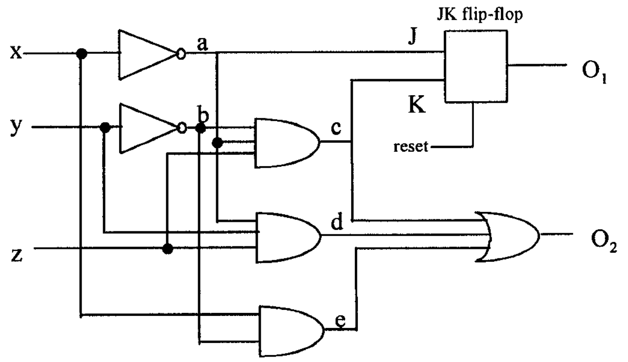

Figure 2. Sample Sequential Circuit

memory element. In a complex circuit we cannot readily identify all of the circumstances (or the enumeration is too high) under which this loss of state occurs. We may, in general say that the first detection destroys our ability to track state of the system. Given this, the "good system" paradigm used in developing the fault dictionary is no longer valid.

An unencapsulated set of test vectors, is only valid up to the first detection. This is why, after extensive testing of a circuit, the answer provided is often only a first step in the diagnostic problem and is followed by probing or technician-directed diagnostics.

## III. Evolving an Approach to Using Fault Dictionaries for Sequential Circuits

What has been missing in the sequential circuit is the concept of encapsulation. If we had encapsulated test sets we could then learn information in each instance and "fuse" that information to arrive at answers. When mismatches occur we could then make use of pattern matching, uncertainty or other algorithms to refine the answer. The most straightforward approach is to apply one of the many scan architectures [7]. Scan architectures separate the combinational and sequential parts of the circuit and allow them to be tested separately. However, for many circuits scan is not an option. The circuit may already exist, or scan may be too expensive.

A first approach might be to reset the state each time we apply a test vector. This does encapsulate the tests but may degrade our ability to detect faults in many circuits because we may need a sequence of vectors to exercise the memory elements. Specifically, this approach assumes we can set detect any of the faults in the circuit with a single vector from a known initial state. Further, it assumes no preconditioning of the circuit (other than reset) to put the circuit in that state. This, however, is unlikely to be the case in most sequential circuits.

In addition, we may not be able to arbitrarily reset the state. A faulty circuit may not even be initializable to start the testing process[8][9]. This makes the design for testability approaches of using a common reset and a separate rest line extremely important if we wish to do diagnosis as well as detection on sequential logic circuits. Even with common resets, the problem of state degradation in the presence of faults makes encapsulation extremely difficult. Since a fault might be detected at any point in the process, to assume that we will reset the state at that point leads to requiring an exponential number of paths to be explored, branching from each of these detections. Otherwise, we would have no way to match the results from subsequent tests since the state would no longer be predictable.

The most practical solution appears to take the generated fault dictionary and rerun the simulation with each inserted fault to produce a fault dictionary that reflects actual faults being present. This latter approach can be done incrementally by tracking the potential state changes generated by faults. It is anticipated that the fault dictionary created in this manner will have the same essential detection characteristics, but a more accurate fault isolation resolution. If unacceptable ambiguity exists after this rework of the dictionary, it is recommended that an encapsulation boundary be set and work proceed on breaking up the ambiguities within encapsulated sets. The key element of fault insertion, as opposed to simply backtracing, provides us with a diagnostic approach and not just a fault detection approach. Most recent releases of simulation tools such as LASAR may actually apply this process incrementally in the development of test vector sets.

Another possibility is the re-evaluation of the way in which the fault dictionary is applied and encapsulation at a convenient set of boundaries (perhaps after a sequence that tests a sensitized path in the circuit). The fault dictionary would then consist of a series of "encapsulated" test vector sets, whose results can be fused. The steps outlined in the previous paragraph should be followed for each encapsulated set. However, without such rework, the results must be computed after the first detection and subsequent elements in the fault dictionary must be ignored. Further, the inability to reset the circuit or otherwise encapsulate should be interpreted as a test and applied to the fusion. Once the circuit is unable to be reset, further work with the fault dictionary is not likely to be fruitful and probing may be the only way to proceed.

## IV. Extending the Concept of Encapsulation

While we have been using the digital fault dictionary as an example, it is clear that the concepts involved with encapsulation have a much broader application. Any system (digital or otherwise) can be viewed as a set of transfer functions with inputs and outputs between the subsystems. Figure 3 shows a functional diagram for a hypothetical system. Each of the transfer functions may be simple, complex, contain memory, or be based upon prior events. Testing this system can be based upon independent functions, (as in the combinational circuit) or dependent functions (as in the sequential circuit), and the testing of any such system may require encapsulated tests.

Encapsulation is a key element in a systems approach to problem solving. The general approach to encapsulation is to lay down a boundary that separates a piece of the problem from every other piece of the problem. The use of common resets will apply, even in non-digital systems. In fact, the technology of the system (digital, analog, hybrid, mechanical, pneumatic, etc.) is not a factor in the basic approach. This boundary we will call an interface. At the interface we will provide for data exchange with the other elements of the problem and provide services to each of the other elements. Each encapsulated piece will also demand services. A perfect encapsulation occurs when all external dependencies can be severed. This latter seldom happens, and underscores the importance of the precise definition of the interface and its information "dependencies".

For test and diagnosis, encapsulation means developing tests (or sets of tests) that can be used as independent information sources. The key is separation of one problem detail from another so that they can be treated independently. The independence means a reduction in overall complexity that would otherwise be unachievable because of the system structure, hierarchy and design philosophy. A key element in system level approaches is the management of the complexity issue which often dominates in modern systems. Effective encapsulation can help manage system complexity.

For maintenance systems, encapsulation can mean the development of an open architecture that allows multiple vendors to participate. The advantage is that what goes on within an encapsulation boundary is unimportant to the overall functioning of the system. Thus, for a maintenance system such as the Aviation Maintenance Integrated

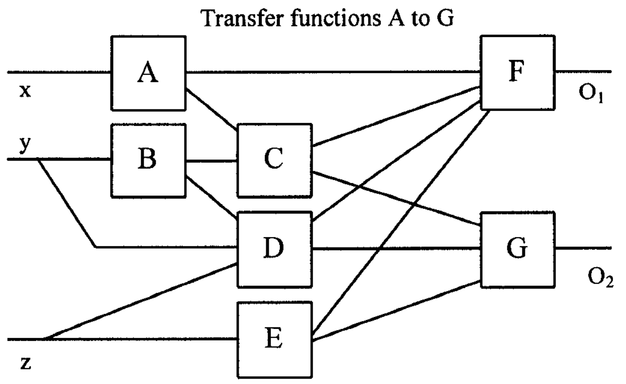

**Figure 3.** Functional Diagram for Hypothetical System

Diagnostics Demonstration (AMIDD) [10], encapsulation means being able to plug and play essential elements of the system. In the latter case, the diagnostic reasoner was encapsulated and as a result, the type of diagnostic reasoner was able to be changed during the demonstration without impact on the maintenance operations or pilot debriefs. The change-over was transparent. The encapsulation was not an accident, but specified. In fact, the diagnostic reasoners were not even of the same type with the former being a CLIPS rule set and associated software, and the latter being a model-based reasoner and its knowledge base.

## V. Harnessing the Power of Encapsulation

Encapsulation, as a concept, can point the way to open system architectures. Encapsulation can be at the hardware, software, and/or problem domain level. Open systems would allow multiple vendors to bring competing products into an integrated package. The encapsulation will specify the interface, information exchange, and services. Within encapsulation boundaries, products may be open or proprietary because understanding the details of how a vendor product may meet the encapsulation requirements is not relevant. Numerous consortium arrangements are beginning to develop the encapsulation concept in detail. For example, the *VXIPlugandPlay* consortium is putting together a set of specifications to allow test instruments interchange. The consortium arrangements will lead to *defacto* standards in test. However, the process does not stop there consensus standards' bodies are also developing encapsulated representations consisting of interfaces, information exchange, and services. IEEE standards for A Broad Based Environment for Test (ABBET) [11][12][13][14] and Artificial Intelligence Exchange and Service tie for All Test Environments (AIESTATE) [15][16][17] are defining encapsulated representation. The development of encapsulated areas within problems of interest, and their attendant information exchange and services, promotes the development of multiple vendor interoperability while allowing the vendors to compete on price performance and functionality within the encapsulation boundaries. Clearly encapsulation is the process by which we can achieve these often conflicting goals.

## VI. Conclusion

In this paper, we have covered a broad range of topics concerning the concept of encapsulation. Time and space only permitted one detailed example, that of the fault dictionary. In the fault dictionary, the concept of encapsulation can be used to refine the diagnostic capability. The extension to the system level through the state machine analogy is not only useful from a mathematical sense, but provides us a basis for the development of
a system level approach to testing. This approach is useful for managing the complexity of today's evolving systems. Finally, by extension, the encapsulation process allows us to develop objective criteria for evaluating standardization efforts. To increase the likelihood of success for a standard, the standards developers should strive, to the maximum extent practical, to utilize these concepts of encapsulation. The encapsulation not only provides interchangability, but opens the market to a broader and more profitable base than can be achieved through customized solutions. If proper encapsulation is achieved in the development of standards, then each implementation, vendor, or application should fit comfortably within the encapsulation boundaries. Otherwise, each implementation is likely to struggle to meet the standard and probably extend or subset its domain.

## References

[1] Abramovici, M., M. A. Breuer, and A. D. Friedman. 1990. *Digital Systems Testing and Testable Design*, New York: Computer Science Press.

[2] Richman, J., and K. R. Bowden. 1985. "The Modern Fault Dictionary," *Proc. International Test Conference*," Los Alamitos, California: IEEE Computer Society Press.

[3] Grant, F. 1986. "Noninvasive Diagnostic Technique Attacks MIL-SPEC Problems," *Electronics Test*, Miller-Freeman Publications.

[4] Debaney, W. H., and C. R. Unkle. 1995. "Using Dependency Analysis to Predict Fault Dictionary Effectiveness," New York: IEEE Press, *AUTOTESTCON '95 Conference Record.*

[5] Cover, T. M., and P. E. Hart. 1967. "Nearest Neighbor Classification," *IEEE Trans. on Information Theory*, Vol. IT-13, pp. 21–27.

[6] Sheppard, J. W., and W. R. Simpson. 1996., . "Improving the Accuracy of Diagnostics Provided by Fault Dictionaries," *Proceedings of the 14th IEEE VLSI Test Symposium*, Los Alamitos, California: IEEE Computer Society Press, pp. 180–185.

[7] Maunder, C., and Tulloss, R. 1990, *The Test Access Port and Boundary Scan Architecture*, Los Alamitos, California: IEEE Computer Society Press.

[8] Keim, M., B. Becker, and B. Stenner. 1996. "On the (Non-) Resetability of Synchronous Sequential Circuits," *Proceedings of the*

*14th IEEE VLSI Test Symposium*, Los Alamitos, California: IEEE Computer Society Press, pp. 240–245.

[9] Wehbeh, J. and D. Saab. 1996. "Initialization of Sequential Circuits and its Application to ATPG," *Proceedings of the 14th IEEE VLSI Test Symposium*, Los Alamitos, California: IEEE Computer Society Press, pp. 246–251.

[10] Gartner, D. 1996. "Intelligent Diagnostics for the F/A-18," *National Electronics Produces' Conference (NEPCON) - West, 1996*, Anaheim, California, pp. 241-250.

[11] IEEE Std 1226-1993. *Trial Use Standard for A Broad Based Environment for Test (ABBET™): Overview and Architecture*, Piscataway, New Jersey: IEEE Standards Press.

[12] IEEE P1226. *Standard for A Broad Based Environment for Test (ABBET™): Overview and Architecture*, Piscataway, New Jersey: IEEE Standards Press, Draft 16.

[13] IEEE Std P1226.3. *Standard for A Broad Based Environment for Test (ABBET™): Software Interface for Resource Management*, Piscataway, New Jersey: IEEE Standards Press (In Ballot).

[14] IEEE Std P1226.4. *Standard for A Broad Based Environment for Test (ABBET™): Software Interface for Instrument Drivers*, Piscataway, New Jersey: IEEE Standards Press (In Ballot).

[15] IEEE Std 1232-1995. *Trial Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Overview and Architecture*, Piscataway, New Jersey: IEEE Standards Press.

[16] IEEE P1232.1. 1996. *Trial Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Data and Knowledge Specification*, Draft 4.7.

[17] IEEE P1232.2. 1996. *Trial Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Service Specification*, Draft 2.0.