A View of the ABBET™ Upper Layers

John W. Sheppard
ARINC Incorporated
2551 Riva Road
Annapolis, MD 21401 USA
sheppard@arinc.com

William R. Simpson
Institute for Defense Anlayses
1801 N. Beauregard Street
Alexandria, VA 22311 USA
rsimpson@ida.org

Abstract: **Currently, the IEEE Std 1226-1993 (ABBET™) is undergoing significant revision in preparation for its release as a "full-use" standard. Much of this work is motivated by a need to define the interfaces between the various "layers" of the current architecture and prepare a road map for implementing those interfaces. To date, little work has been done on the upper layers of ABBET, yet it is believed that the upper layers offer the greatest potential for cost savings in developing advanced automatic test systems. In this paper, we address the issues of the ABBET upper layers in the context of a new architecture that is focused on addressing the needs of the upper layers.**

I. INTRODUCTION

Recent initiatives by the IEEE toward standardizing test architectures have provided opportunities to improve the development of test systems. The "A Broad Based Environment for Test" (ABBET) initiative (IEEE PAR 1226) is attempting to usher in the next generation of product test by standardizing test services and test development tools [1]. By using standardized methodologies in developing test executives, test programs, and communication protocols, ABBET conformant test systems will be interoperable, have transportable software, and move beyond vendor- and product-specific test systems.

Currently, the ABBET architecture is organized around five layers—a product description layer, a test strategy and requirements layer, a test procedure layer, a test resource management layer, and an instrument control layer [1]. To date, the greatest emphasis has been placed on the latter three layers due to a procedural decision to focus on the architecture from the "bottom up." We believe, however, that much of the cost savings that may be attributable to the ABBET architecture lies in capitalizing on standardization and integration in the upper two layers. Describing a process and architecture for developing upper layer standards is the first step in their completion. To develop this description, we reexamined the interfaces in the upper layers of the ABBET architecture from a different point of view. To facilitate this reexamination, we began by developing an alternative high-level information model of the ABBET architecture. In this paper, we present the new architecture and describe its impact on the upper layers.

II. APPROACH

In proceeding from a systems engineering approach, we focused on several characteristics of a broad test environment (the stated scope of the ABBET standards) and devised an architecture that has these characteristics. In particular, we focused on issues of information and resource reuse, knowledge encapsulation, test encapsulation, and test context. For test context, we recognized that testing concerns at least three dimensions: the product life cycle (e.g., design verification, factory acceptance, or maintenance), the position of a test subject in the product hierarchy (e.g., system level, board level, or component level), and the test philosophy applied (e.g., built-in test, automatic test, manual test, or combinations).

The perceived scope of the ABBET standards appears to be limited to maintenance test on automatic test equipment (ATE) in some application domain with *application domain* being defined as the level of test (e.g., card level or system level) [2]. Where this hierarchical view of test is appropriate, it is insufficient to cover the true, intended scope of the standard: design verification, manufacturing test, factory acceptance test, and operational evaluation *as well as* maintenance test [3,4]. This means that the information architecture must be structured to identify information common to these types of tests as well as information peculiar to a particular type of test. In this way, one can identify the source and type of information required for test throughout the product life cycle. Then this information can also be used in developing automatic test systems, built-in self-test systems, aided test systems, and manual test environments. Information can be developed throughout the life cycle as a "value-added" development process using previously available information and adding peculiar or previously unneeded data.

III. TESTING IN CONTEXT

In the broadest possible scope, testing is performed for various activities, including performance evaluation, mechanical/electrical integrity, periodic and unscheduled maintenance, process evaluation, operational readiness, and specification and compliance. In short, the underlying purpose of any testing process is information discovery. In this reexamination

0-7803-2621-0/95 $3.00 © 1995 IEEE

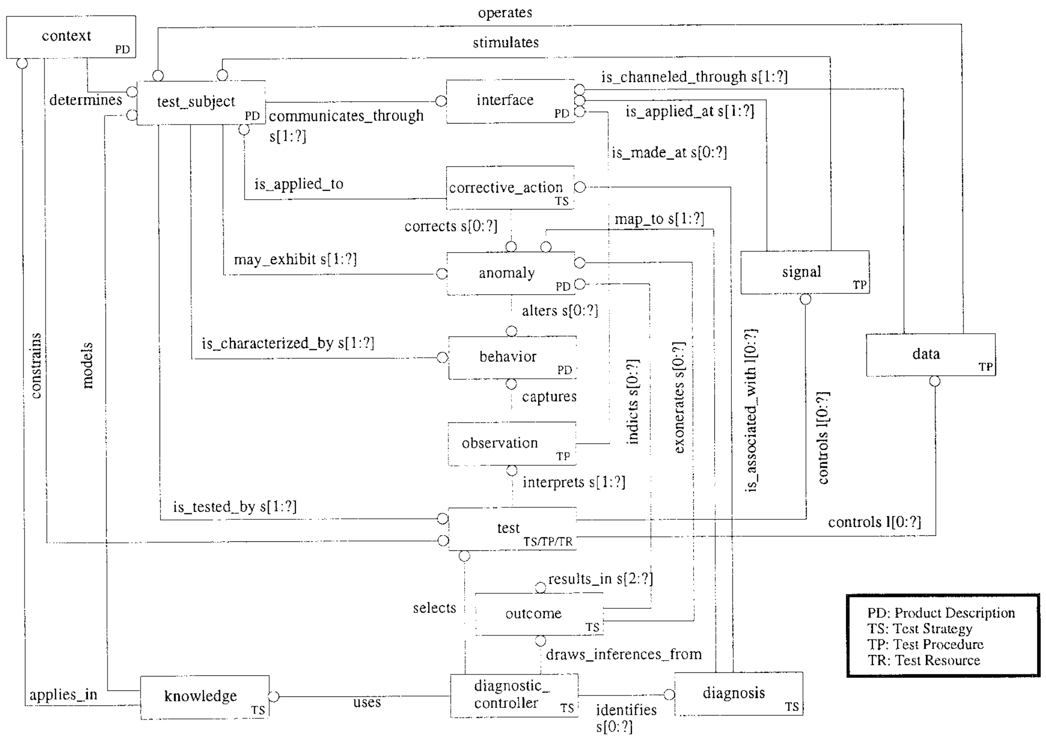

Fig. 1 Information model of proposed architecture.

of ABBET, we limited our view of testing to electronics test for design verification, manufacturing test, and maintenance test. In our concept, a test is a signal, indication, or other observable event that provides information about the system being tested [5]. The observation may be caused to happen (as with a stimulus/response test) or be part of a normal operational environment (as in the case of a symptom). The only purpose a test serves is to provide an outcome that can be used to infer something about the system being tested. As such, testing is driven by separate reasoning processes. The resultant reasoning on test results is context sensitive.

IV. A NEW INFORMATION ARCHITECTURE

Fig. 1 presents a new information model for the ABBET architecture. Several similarities exist between the proposed architecture and the architecture currently used by ABBET. In the following discussion, we focus on the differences, referring the interested reader to [1] for an explanation of the existing architecture and to [6] for more detail on the proposed architecture.

A. Context

The `context` entity in Fig. 1 is intended to define the context in which the test is to occur. We anticipate that the actual context will be defined along the three (at least) dimensions outlined above. Because the context defines the type and level of test, and the test subject is an abstract view of the physical product, we can claim that `context determines test_subject`. Further, the test process is wholly determined by the context, so we assert that `context constrains test`.

B. Test Subject

As indicated above, we intend the test subject to continue to be interpreted as the object of analysis in a test system. One definition of a *system* is "any aggregation of related elements that together form an entity of sufficient complexity for which it is impractical to treat all of the elements at the lowest level of detail [5]." From this definition, we see that a test system can be interpreted hierarchically to be a system, subsystem, or component within a system. If the test subject is a component, then diagnostics is less of a concern since it is sufficient to focus on detection rather than isolation at that point. Thus we focus our discussion on testing systems like those we have just defined, and we refer to those systems as the test subjects.

C. Behavior

What we know about our test subject comes from how the subject behaves. (Exceptions to this exist with respect to static and physical properties of the test subject, e.g., discoloration of a solder joint. Observing these properties can be very important in diagnostics. Here, however, we limit our discussion to the dynamic properties of the system, recognizing that *behavior* can be extended to cover the static properties as well.) By observing and evaluating the behavior of the test subject, we can evaluate the state of the test subject and determine whether a fault exists. Through subsequent testing, we hope to use behavior to further localize and isolate the faults. To make this explicit, we say that the `test_subject is_characterized_by SET [1:?] OF behavior`.

D. Observation

Because *behavior* is the manifestation of the functioning of the test subject, we need to be able to observe that behavior to test the test subject. Thus we say the `observation captures behavior`. A possible source of controversy centers on "where" an observation takes place. We provide for the possibility that the observation occurs at the defined interfaces of the test subject (because this is usually what happens in electronics testing) and state explicitly that `observation is_made_at SET [0:?] OF interface`. With the set having a lower bound of zero, we are allowing observations to occur in ways other than using the defined interfaces. It is likely, however, that the test interfaces to be standardized by ABBET will be limited to `interface`.

E. Anomaly

Consistent with the current model, we note that an `anomaly` defines the cause of misbehavior of the test subject. For any number of reasons, a `test_subject may_exhibit SET [1:?] OF anomaly`. (Note that the current model describes anomaly as the misbehavior itself.) We would prefer to identify the anomaly as the cause of misbehavior, so we say that an `anomaly alters SET [0:?] OF behavior` thus putting the burden of evaluating the test subject back on evaluating the behavior of the test subject. Then diagnosis is identification of the anomaly or anomalies that resulted in the misbehavior of the test subject. In addition, we can avoid treating "No Fault" as a special case by including it as an instance of anomaly. Also `diagnosis maps_to SET [1:?] OF anomaly`. Since anomalies alter behavior, we allow for "null" alteration in the model (thus the lower bound of zero).

F. Outcome

Diagnosis usually proceeds by drawing inferences from specific test outcomes. It is possible to diagnose based on raw test values, but most practical systems quantize the results of tests and assign outcomes (e.g., pass, fail low, and fail high). In this model, we note that a `test results_in SET [2:?] OF outcome`. It seems reasonable to expect the test to have at least two outcomes (e.g., expected and unexpected).

From this we can further assert that the `outcome indicts SET [0:?] OF anomaly` and the `outcome exonerates SET [0:?] of anomaly`. Both of the sets of anomalies are given a lower bound of zero because it is possible that a given outcome provides no information about the anomalies on any particular side. We call tests with these types of outcome inferences *asymmetric tests*. Usually (though not necessarily) a fail outcome indicts and a pass outcome exonerates; however, situations arise, albeit infrequently, where the opposite occurs. In any case, we want to allow for inferences (of indictment or exoneration) to be associated with arbitrary outcomes beyond simply pass and fail outcomes.

G. Knowledge

We use `knowledge` to refer to the collective diagnostic knowledge to be used to diagnose the test subject within an appropriate context. As such, knowledge includes the diagnostic knowledge and the historical data and knowledge. This collective knowledge base is a representation of the test subject and defines the test system view of the test subject. Thus we say that `knowledge models test_subject`. Further, a particular context determines applicable knowledge in a test problem. For the test subject, we say, therefore, that the `knowledge applies_in context`, referring to the applicable knowledge.

The relationship between a test outcome and the behavior the test observes is contained in the knowledge, not in the test or the test outcomes. Because the relationships between tests and behaviors are context-dependent, failure to separate this knowledge from the test entities will severely limit the reusability of those entities. For example, a test object that thoroughly examines a chip at the chip testing level may examine the chip and several "components" between that chip and an interface when testing at the board level. Further, those components considered at the board level may modify the information presented to the chip or received from the chip, thus limiting the testing of the chip. This has significant implications for hierarchical representations of knowledge. In particular, we find that ***the hierarchy applies to behavior rather than inference since inference is not transitive across the levels of the hierarchy***.

H. Diagnostic Controller

In the current ABBET architecture, the diagnostic controller (which resides in the test procedure layer) is responsible for invoking test procedures and evaluating the results. Our proposed architecture is consistent with this view, and we reinforce the importance of separating explicit test control (through the test procedures) from the diagnostic process. To accomplish this, we say that `diagnostic_controller uses knowledge` to control the diagnostic process which is implemented when the `diagnostic_controller selects test` and `diagnostic_controller draws_inferences_from outcome`. Finally, once a diagnosis is made, we say that `diagnostic_controller identifies SET [1:?] of diagnosis`. Because the objective of diagnostics is to identify the cause of anomalous behavior so as to correct the problem and restore the test subject to its proper state, diagnostics is central to the process of determining what the corrective action should be (by way of identifying the diagnosis). Also, by including No Fault as a diagnosis, we can tie the absence of a fault to the corrective action of return to service. This would correspond to certifying that a product is ready for issue. Of course, it is possible that given the data received, no diagnosis can be made, and we allow for a null diagnosis from the diagnostics (e.g., in triggering an event for maintenance data collection).

I. Diagnosis

The results of applying the diagnostics to the test process is the identification of a fault (or not) in the system. The result of the identification process is called the diagnosis. We distinguish the diagnosis from the anomaly by noting that the diagnosis is the conclusion drawn and the anomaly is the actual cause of behavior that leads to the diagnosis being made. Because the diagnosis leads to some action to restore the test subject to a nominal condition, we say `diagnosis is_associated_with LIST [0:?] OF corrective_action`. We use a list instead of a set because the order of the actions may be significant. Unfortunately, sometimes the diagnosis has no associated corrective action (i.e., it may not be possible to restore the test subject to nominal), and this is reflected by permitting an empty list of corrective actions.

J. Corrective Action

The last step in the process of applying a test environment occurs when a diagnosis is made. At this point, some corrective action (or actions) should be performed to restore the test subject to a nominal state. Thus we note that `corrective_action is_applied_to test_subject` and `corrective_action corrects SET [0:?] of anomaly`. Even when a corrective action is recommended by the diagnostics, it is possible that this corrective action will still fail to correct the anomaly, and this is the reason for providing a lower bound of zero on the anomaly set. Also we wish to abstract the corrective action from the diagnosis just as we abstract the test procedure from its outcome. That is what this model supports.

V. TEST ENCAPSULATION

An important concept being considered in the ABBET architecture (and in test architectures in general) is *test encapsulation*. In general, an object is *encapsulated* whenever it is wholly self-contained and independent of other objects of the same class. In the context of testing, this means an encapsulated test is self-contained and wholly independent of other tests in the test system. Encapsulation can occur at any phase: test requirement → test specification → test method → test development.

A. Test Requirement Encapsulation

Since the object of test encapsulation is to make the test we are encapsulating independent of other tests, we need to consider, at this level, how one makes test requirements independent. As described earlier, testing is dependent on context and on the test subject. It may be possible to specify high-level attributes of the context which can be inherited by a given requirement. The goal of standardizing the upper layers is to identify interfaces as pathways for deriving test requirements from product data. Identifying the product characteristics (i.e., behavior) and anomalies in a given test context is sufficient to define the test requirements for that product. By abstracting common elements of the test requirements to either the test subject or context, the requirements will be independent since dependence occurs at a higher level in the class structure.

B. Test Specification Encapsulation

Test specifications are derived from test requirements. By using encapsulated test requirements (i.e., requirements derived directly from product specifications), we have provided a mechanism for deriving specifications. A test specification should include no more than a description of input, expected output, and the context for test. This specification of context includes the types of interfaces available for the test. Under this formulation, the input and the output depend on the test specification, but the test specification depends on the interfaces as defined by the context for testing. Once again, common elements for the specifications can be elevated to the requirements (or above), thus preserving independence of the actual specifications.

C. Test Method Encapsulation

Test methods are used to meet the test specifications; therefore, there is a direct dependence on these specifications. We envision test methods as being high-level definitions of test algorithms that can be used in appropriate test contexts and that the appropriate methods can be derived or mapped from the specifications. For example, if we are testing the RAM on a digital signal processing board in the factory on a board tester, we can use a RAM test method to identify the way we will test the memory. This same test method may be applicable for testing a PC memory board in a maintenance shop. Attributes of the test method are inherited from the test specification that satisfies the test requirement, which is based on the context and test subject.

D. Test Procedure Encapsulation

At the lowest level of test (in this model), we are concerned with defining test procedures. The test procedures are implementations of test methods and implement the means of acquiring a test observation. Because the test observations are determined by the input and output of the test, and the test procedure produces the input, acquires the output, implements a test method, and satisfies one or more requirements, the observation ultimately depends on how the test was implemented in the procedure. Also, an outcome of a test ultimately depends on the test procedure because the test procedure defines how one determines the outcome. It is at this point that the advantage of encapsulation becomes most obvious, even though the greatest cost savings can be anticipated from encapsulation at higher levels.

E. A Recommendation

The goal of defining an architecture for encapsulation is twofold. First, we want to *maximize the level of reuse possible* in defining and implementing test assets. Using the dependency structure described in the previous sections, we can determine where one receives the information necessary for each of the entities in the test structure. This provides the needed mechanism for maximizing reuse. The second aspect of encapsulation involves *maximizing the effectiveness of the test process and associated diagnostics*. Recall that diagnostics refers to information that can be inferred from testing (whether to identify faults or to certify that no faults exist). It is absolutely essential that the diagnostic knowledge be separated from the test structure as much as possible. Diagnostic knowledge is wholly dependent on the test subject, the test context, and the tests performed. Tests do not depend on diagnostics. Yet current approaches to testing inextricably tie test and diagnosis together. For example, most current test requirements documents (TRDs) include the diagnostic strategy, thus severely limiting the ability to optimize and improve the diagnostics associated with a given test subject. A TRD may even tag a test that calls out an anomaly, but it is the totality of testing to that point that justifies calling out the anomaly.

To separate the diagnostics from the test, we recommend that test methods be developed such that the test inferences are not included. We believe inferences can be determined only by tracing the effects of anomalies on product behavior and identifying which aspects of the behavior are observed by a given test procedure. The implementation of a test procedure (for a given test subject in a given context) determines the inferences that can be drawn from associated outcomes. Test procedures should be derived from the test methods in the appropriate context, and the inferences can be tied to the procedures at that point.

An important aspect of test encapsulation relates directly to tying the diagnostics to the test procedure. We want to maximize reuse *and* flexibility; therefore, we want to minimize dependence of test procedures on the state of the test subject. This corresponds to the definition of test encapsulation from the perspective of the Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE) standards [7]. By limiting the dependence of diagnostic information to tests at the lowest level (i.e., at the level of the test procedure), we maximize our opportunity to define encapsulated test objects of this sort. This, in turn, provides AI-ESTATE with a testing process that permits flexible inference and test choice.

VI. A ROAD MAP FOR IMPLEMENTING UPPER LAYERS

In developing the proposed architecture, we want to promote the objectives of concurrent engineering by providing an environment of cooperation and communication between design activities and test activities. Ideally, we would like to capture existing design data and provide a definition of needed test data that can be generated during product design with the intent of ultimately reducing cost in test development, increasing test effectiveness, and providing a mechanism for iterative process improvement.

The proposed information architecture provides a mechanism whereby required engineering design data necessary for test can be identified and generated during the design phase or whenever it is created as a value-added process. Recall that the focus of the test process is on the behavior of the test subject, and this behavior fully depends on the definition of the product and impact of anomalies. Thus the key to determining the behavior of the test subject is defining the behavior in terms of the product characteristics and the anomaly characteristics.

Product design frequently provides the following data: system connectivity and structure, theory of operation, functional specification and definition, and simulation models. These data can be represented using many of the existing hardware description languages (HDLs) such as VHDL, MHDL, or AHDL. Unfortunately, such data are not sufficient for test. In addition to these data, the test engineer needs to understand the types of anomalies that can exist within the system. These anomalies can be defined with the following types of data: alterations in structure (e.g., shorts, sneak circuits, or bridging faults), fault propagation (does not necessarily follow signal flow paths), "functional" descriptions of the anomalies, and fault models. Although current use of HDLs is predominantly for describing structural characteristics, most HDLs have the ability to model behavior. As a result, we find that anomalies, too, can be represented using HDLs (either in their current form or with relatively minor extension or augmentation). Thus it should be straightforward for design teams to augment their data packages with specifications of the anomalies using the design tools already available to them. The data can then be used to determine the behavior of the product under nominal and anomalous conditions. Anomalies that are pertinent to each level of test can be developed at that level and reused (and added to) at each subsequent level. Thus the test process is one of "value-added" engineering, where knowledge bases are supplemented, not replaced.

Recall that the focus of the test is the observation of the test subject's behavior. We appear to have a way of representing the product and anomaly information, and now we need to process the data to define the observations that can be made. Once the observations are defined, the test definition follows. We know that the observation depends on the product behavior, and the behavior is directly influenced by signals, data, and the available interfaces in the test subject. The definitions of the behavior, in conjunction with specifications of the test requirements (which is the intent of the Test Requirement Specification Language [TRSL] standardization effort) can be used to derive test methods that include definitions of the signals and data to be applied at the product interfaces.

Combining the definition of the test subject, the anomalies, and the test provides all of the information necessary to construct a diagnostic knowledge base (or model). The test definition provides information on the observations being made, and the anomaly definitions explicitly identify the diagnostic conclusions to be drawn. The product definition is required for specialized diagnostics, such as constraint-based reasoning and testing devoted to determining that the product is functioning properly, as in factory acceptance testing or product certification. This view is consistent with the AI-ESTATE initiative [7].

VII. CONCLUSION

This paper provides an alternative architecture for ABBET that focuses on defining product and anomaly behavior during design to facilitate automatic synthesis of test and diagnostic information. The approach further focuses on the behavior as defining the current state of the test subject and characterizing whether the state is nominal. From this characterization, tests are defined to classify the behavior and diagnose problems in the test subject. The advantage of this architecture over the current architecture is that a road map for automating the test engineering process from product design is readily identifiable. This road map, with standard representations, will provide a "value-added" process, and in most cases, standards already exist to facilitate this process. Where the standards are lacking, standardization efforts are under way to fill the void. Since the goal of ABBET is to provide a framework and define interfaces between existing standards in a test environment (not necessarily tied to ATE), an architecture that facilitates integrating existing standards, such as the one proposed here, is paramount to success.

REFERENCES

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

[2] Robert L. McGarvey. "Object-Oriented Test Development in ABBET," *AUTOTESTCON '94 Conference Proceedings*, Piscataway, New Jersey: IEEE Press. 1994.

[3] Gary Hinkle and Terry Campbell. "Technical Approach for ABBET Upper Layers," ABBET Working Paper, ABBET Log #674. 1993.

[4] IEEE P1226.6. *Draft Guide to the Understanding of A Broad-Based Environment for Test (ABBET)*, Draft 4.2 (in ballot), New York: IEEE Press. 1994.

[5] William R. Simpson and John W. Sheppard. *System Test and Diagnosis*, Norwell, Massachusetts: Kluwer Academic Publishers. 1994.

[6] John W. Sheppard and William R. Simpson. "A View of the ABBET Upper Layers," ABBET Working Paper, ABBET Log #790. 1994.

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