Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)—A New Standard for System Diagnostics

John W. Sheppard
ARINC
2551 Riva Road
Annapolis, MD 21401
jsheppar@arinc.com

Leslie A. Orlidge
AlliedSignal Aerospace
Guidance and Control Systems
Teterboro, NJ 07608
leslie.orlidge@alliedsignal.com

Abstract

We describe a recently approved IEEE standard for exchanging diagnostic information and embedding diagnostic reasoners in any test environment. We describe the defined formats and services, an example application, and current industry acceptance.

1. Introduction

Recent initiatives by the Institute of Electrical and Electronics Engineers (IEEE) on standardizing test architectures have provided a unique opportunity to improve the development of test systems. The IEEE P1232 "Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)[1]" initiative is attempting to usher in the next generation in product diagnostics by standardizing diagnostic services and development tool interfaces. By using problem encapsulation, defining interface boundaries, developing exchange formats and specifying standard services, AI-ESTATE provides a methodology for developing diagnostic systems that will be interoperable, have transportable software, and move beyond vendor and product specific solutions.

The concepts in the AI-ESTATE standard are not limited to the arena of automatic test equipment, but apply to manual, automatic, and semi-automatic test, as well as the domains of electronic, mechanical, pneumatic, and other types of systems. The AI-ESTATE subcommittee designed the P1232 standards to abstract specific test and product details out of the diagnostic models and tie these models to domain-specific models as needed to complete the test system.

In this paper, we describe the AI-ESTATE architecture [1] and two "component" standards of the AI-ESTATE family of standards. We discuss a neutral exchange format for diagnostic models standardized in IEEE Std 1232.1-1997 [2]. We also discuss several software services to be provided by an AI-ESTATE conformant diagnostic system standardized in IEEE Std P1232.2 [3]. In addition, we describe example application scenarios using these standards and suggest alternative uses of the standards in various test environments. Finally, we discuss industry involvement and acceptance of the standards and provide directions for future work on AI-ESTATE.

[1] Previously, Artificial Intelligence and Expert System Tie to Automatic Test Equipment.

2. Background

Increasing complexity and cost of current systems, the inability to consistently diagnose and isolate faults in the system using conventional means, and the advances in artificial intelligence technology have fostered the growth of AI technology in test and diagnosis. The proliferation of diagnostic reasoners and tools necessitates establishing standard interfaces to these tools and formal data specifications to capture relevant diagnostic information. Current test standards (e.g., Boundary Scan and STIL) [4,5] provide no guidance for using AI technology in test applications. Proposed AI standards (e.g., KIF) do not specifically address the concerns of the test community [6]. Thus, no standard exists, currently, addressing the use of AI systems in test environments. The AI-ESTATE standards are intended to fill this void.

The AI-ESTATE subcommittee has established several ambitious goals for the AI-ESTATE standards that include:

- Provide a standard interface between diagnostic reasoners and other functional elements that reside within an AI-ESTATE system.
- Provide formal data specifications to support the exchange of information relevant to the techniques commonly used in system test and diagnosis.
- Maximize compatibility of diagnostic reasoning system implementations.

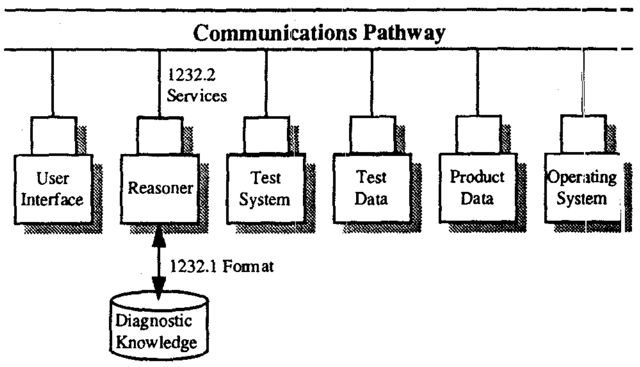


Figure 1. AI-ESTATE Architectural Concept

- Accommodate embedded, coupled, and stand-alone diagnostic systems.
- Facilitate portability, reuse, and sharing of diagnostic knowledge.

To achieve these goals, the AI-ESTATE subcommittee proceeded to define an architecture for a standard diagnostic system and then defined component standards for information exchange and software interfaces.

3. An Architecture for Diagnosis

The AI-ESTATE architecture presented in Figure 1 shows a conceptual view of an AI-ESTATE-conformant system. AI-ESTATE applications may use any combination of functional elements and interfunction communication as shown in the figure. The service specification (P1232.2), or other specifications relevant to the particular functional element, define the form and method of communication between reasoning systems and other functional elements. AI-ESTATE identifies reasoning services provided by a diagnostic reasoner so that transactions between test system components and the reasoner are portable. AI-ESTATE assumes a client-server or cooperative processing model in defining the diagnostic services.

As indicated in Figure 1, AI-ESTATE includes two component standards focusing on two distinct aspects of the stated objectives. The first aspect concerns the need to exchange data and knowledge between conformant diagnostic systems. By providing a standard representation of test and diagnostic data and knowledge and standard interfaces between reasoners and other elements of a test environment, test, production, operation, and support costs will be reduced.

Two approaches can be taken to address this need: providing interchangeable files (P1232.1) and providing services for retrieving the required data or knowledge through a set of standard accessor services (P1232.2). AI-ESTATE is structured such that either approach can be used [2,3].

The second aspect concerns the need for functional elements of an AI-ESTATE conformant system to interact and interoperate. The AI-ESTATE architectural concept provides for the functional elements to communicate with one another via a "communications pathway." Essentially, this pathway is an abstraction of the services provided by the functional elements to one another. Thus, implementing services of a reasoner for a test system to use results in a communication pathway being established between the reasoner and the test system.

AI-ESTATE services (P1232.2) are provided by reasoners to the other functional elements fitting within the architecture illustrated by Figure 1. These reasoners may include (but are not necessarily limited to) diagnostic systems, test sequencers, maintenance data feedback analyzers, intelligent user interfaces, and intelligent test programs. The current focus of the standards is on diagnostic reasoners. In addition to providing services to the test system, the human presentation system, a maintenance data collection system, and possibly the unit under test, the reasoner also uses services provided by these other systems as required. These services are not specified by the standards.

4. Models for Diagnosis

The current version of IEEE Std 1232.1 defines three models for use in diagnostic systems—a common element model, a fault tree model, and an enhanced diagnostic inference model. All of the models were defined using ISO 10303-11, EXPRESS [7]. EXPRESS is a language for defining information models and has received wide-spread acceptance in the international standards communities of ISO and IEC. For example, EDIF 3 0 0 and EDIF 4 0 0 were defined using EXPRESS.

The common element model defines information entities, such as a test, a diagnosis, an anomaly, and a resource, which are expected to be needed by any diagnostic system. The common element model also includes a formal specification of costs to be considered in the test process. A graphical view of the common element model is shown in Figure 2. The cost model associated with the common element model is shown in Figure 3. These models are shown graphically in EXPRESS-G [7]. Entity relationships are shown with lines terminated by circles. For example, from Figure 2, we see that `diagnostic_model` is composed of sets of `model_anomaly`, `model_test`, `model_resource`,

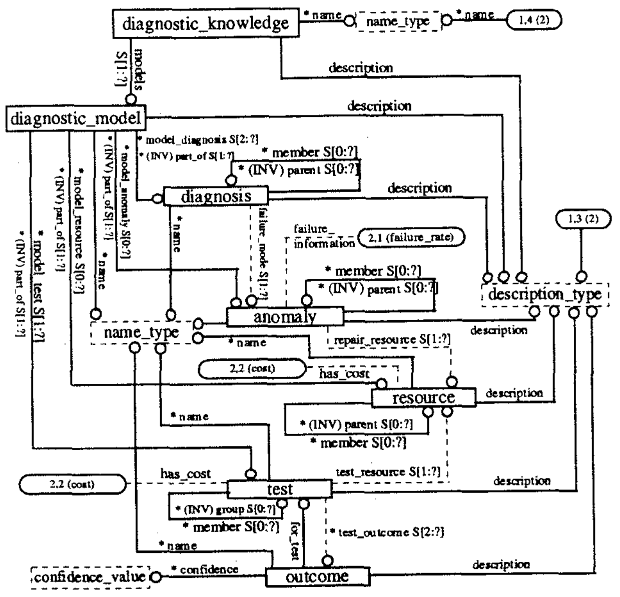


Figure 2. Common Element Model

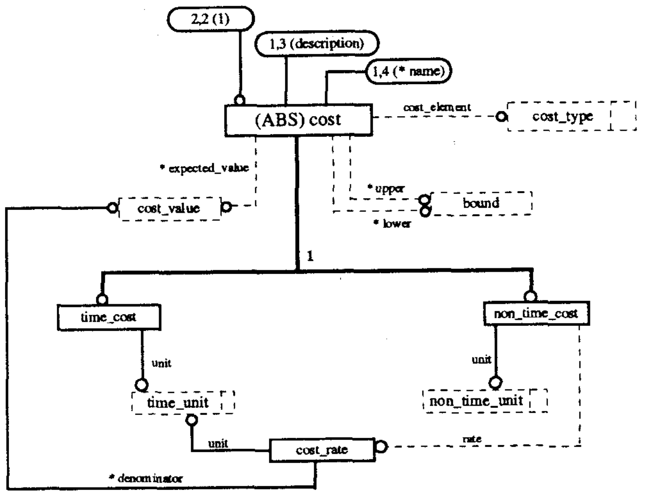
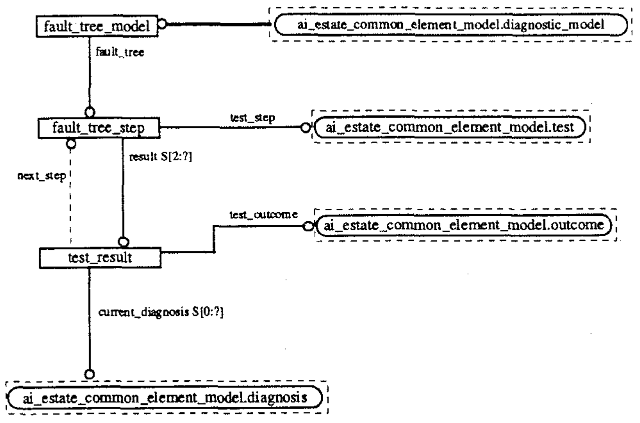


Figure 3. Cost Model

and `model_diagnosis`. Dashed lines indicate the associated relationships are optional, and dashed boxes indicate defined types. Heavy lines indicate a supertype relationship. For example, `cost` is a supertype of one of `time_cost` or `non_time_cost`. The "(ABS)" notation indicates `cost` is an abstract supertype, meaning the `cost` entity cannot be instantiated without one of its subtypes.



fault_tree_model
ai_estate_common_element_model.diagnostic_model
fault_tree
fault_tree_step
test_step
ai_estate_common_element_model.test
next_step
result S[2:?]
test_outcome
ai_estate_common_element_model.outcome
test_result
current_diagnosis S[0:?]
ai_estate_common_element_model.diagnosis

Figure 4. Fault Tree Model

The remaining two models represent knowledge that may be used by specific types of diagnostic systems. The fault tree model defines a decision tree based on outcomes from tests performed by the test system [8]. Each node of the tree corresponds to a test with some set of outcomes. The outcomes of the tests are branches extending from the test node to other tests or to diagnostic conclusions (such as *No Fault*). Typically, test programs are designed around static fault trees; therefore, the AI-ESTATE subcommittee decided to include a representation for a fault tree in the standard, even though fault trees are not typically considered to be AI systems. The EXPRESS-G representation of the fault tree model is shown in Figure 4.

The AI-ESTATE fault tree model imports elements and attributes from the common element model. Typically, test systems process fault trees by starting at the first test step, performing the indicated test, and traversing the branch corresponding to the test's outcome. The test program follows this procedure recursively until it reaches a leaf in the tree, indicating it can make a diagnosis.

The enhanced diagnostic inference model (EDIM) is based on the dependency model. Historically, test engineers used the dependency model to map relationships between functional entities in a system under test and tests that determine whether or not these functions are being performed correctly [9]. In the past, the model characterized the connectivity of the system under test from a functional perspective using observation points (or test points) as the junctions joining the functional entities together. If a portion of the system fed a test point, then the model assumed that the test associated with that test point *depended* on the function defined by that part of the system. This type of model is also referred to as a causal model [10,11].

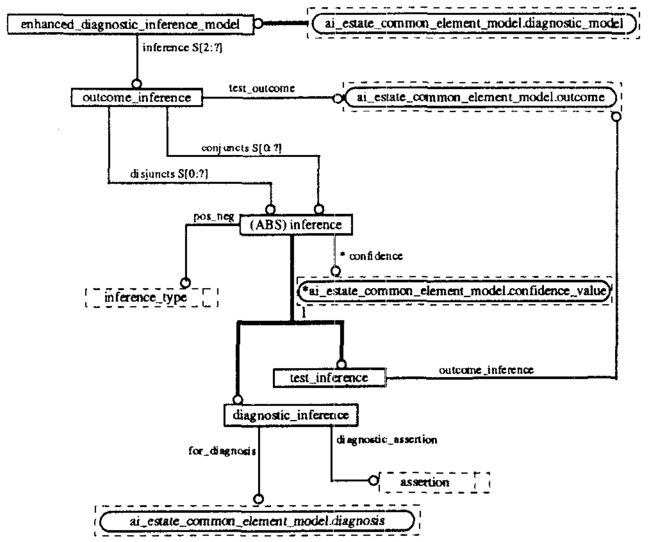


Figure 5. Enhanced Diagnostic Inference Model.

Recently, researchers and practitioners of diagnostic modeling found that the functional dependency approach to modeling was problematic and could lead to inaccurate models. Believing the algorithms processing the models were correct, researchers began to identify the problems with the modeling approach and to determine how to capitalize on the power of the algorithms without inventing a new approach to model-based diagnosis. They found that the focus of the model should be on the tests and the faults those tests detect rather than on functions of the system [12]. In particular, the focus of the model shifted to the inferences drawable from real tests and their outcomes, resulting in a new kind of model called the "diagnostic inference model."

The *enhanced* diagnostic inference model, defined by AI-ESTATE, generalizes the diagnostic inference model by capturing hierarchical relationships and general logical relationships between tests and diagnoses. The EXPRESS-G of the resulting model is shown in Figure 5.

The information models defined in the AI-ESTATE standard, by themselves, provide a common way of talking about the information used in diagnosis, but this is not enough for a standard. In AI-ESTATE, these models also provide the basis for a neutral exchange format. Using this neutral format, multiple vendors can produce diagnostic models in the format to enable their use by other tools that understand that format.

To specify the neutral exchange format, the AI-ESTATE subcommittee decided to use an instance language defined by the ISO STEP (Standards for the Exchange of Product data) community based on EXPRESS—EXPRESS-I. EXPRESS-I is an instance

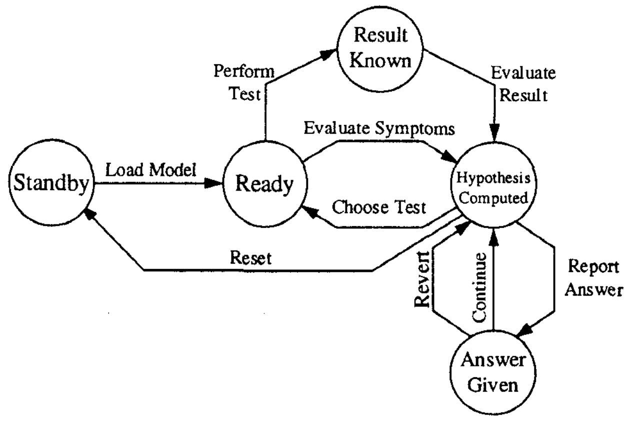


Figure 6. State Diagram for Diagnostic Process

language defined to facilitate developing example instances of information models and to facilitate developing test cases for these models.

As an alternative, the ISO STEP community has defined a standard physical file format derived from EXPRESS models [13]. Unfortunately, the STEP physical file format is very difficult for a human to read but very easy for a computer to process. The AI-ESTATE subcommittee found added benefit in EXPRESS-I over the STEP physical file format in that the language is both computer-processable and human-readable. For example, the following defines the inferences drawable from a particular test of a half-adder that has passed:

```
t4_pass_implies =
    outcome_inference(
        test_outcome -> @t4_pass;
        conjuncts ->
            (@x_sa0_absent,@y_sa1_absent,
            @y_sa0_absent,@S_sa1_absent,
            @C_sa0_absent);
        disjuncts -> ();
    );
```

5. Services for Diagnosis

In addition to defining models for knowledge exchange, the AI-ESTATE standard defines several software services to be provided by a diagnostic reasoner. The nature of these services enables the reasoner to be embedded in a larger test system; however, it is possible that the diagnostic system is a stand-alone application connected to a graphical user interface of some kind.

In defining the standard services for AI-ESTATE, the AI-ESTATE subcommittee began by considered several use cases of diagnostic applications. The most general use case is shown as a state diagram in Figure 6. This diagram shows five states with several transitions based on service requests to the reasoner. These states are intended to be internal states for the reasoner, but it may be necessary for a client of a reasoner to also maintain knowledge of the current state. This will define the set of services available to the client at that point in the diagnostic process.

The first thing to note is that there are two states in Figure 6 that correspond to mode settings defined in 1232.2 [3]. In 1232.2, the reasoner is assumed to be in one of three states: NULL, INACTIVE, or ACTIVE. In Figure 6, STANDBY would correspond to NULL, and READY would cover both INACTIVE and ACTIVE. For our examples, we can consider transitioning to the READY state being equivalent to passing through INACTIVE straight to ACTIVE. It is assumed that all services "descendent from" the READY state occur within the ready mode of the reasoner. These other states are not explicitly identified in 1232.2 and, in some sense, may indicate a particular implementation

The process illustrated in Figure 6 can be interpreted as follows. At the start of any diagnostic process, the reasoner would be in a quiescent (or STANDBY) state. Prior to taking any action, a diagnostic model must be available to the reasoner. The service indicated is *load_model*; however, it is possible that the model may be processed in a "lazy" manner where the model itself need not be resident. Either way, the model must be identified, and this identification is performed via the *load_model* transition.

Once a model is identified and (optionally) loaded, the reasoner enters the READY state from which it can control the diagnostic process. In one scenario, it is possible for several test results to be available at startup. For the sake of discussion, we will call these test results, "symptoms." In Figure 6, the symptoms are processed in batch form (i.e., all of the test results are evaluated in the *evaluate_symptoms* transition). We would expect this transition to consist of a sequence of low-order services. A particular application might offer this derived service, but it is an open issue whether such a powerful service would be appropriate for the standard. One sequence of service requests for loading a model and processing a set of symptoms follows.

```
model_id = attach_model(model_name)
status_code = set_mode(INACTIVE)
status_code = set_mode(ACTIVE)
for all test_id in symptoms do
    get_test_outcome(test_id,&outcome,&confidence)
    status_code =
        apply_test_outcome(model_id,test_id,
            outcome,confidence)
od
```

At any point in the process, it might be desirable to determine the current "hypothesis." Obviously, for this to be possible, the reasoner must have the ability to generate a hypothesis. The current belief is that a service such as *get_current_hypothesis* might be reasoner-specific; however, all of the models currently specified in 1232.1 [2] provide the ability to compute a hypothesis. Further, it is possible the service, *get_most_likely_diagnoses* could be used to generate a hypothesis. Using primitive services, one method of computing a hypothesis is as follows:

```
hypothesis_set = $\varnothing$
step_no = get_number_steps(model_id)
diagnoses = get_all_diagnoses(model_id)
for all diagnosis_id in diagnoses do
    confidence =
        get_diagnosis(model_id,diagnosis_id,step_no)
    if $confidence \geq \theta$,
        $hypothesis = hypothesis \cup \{diagnosis\}$
od
```

This method assumes some threshold, θ, has been defined and confidence values for each of the diagnoses in the current model are computed as test outcomes are applied. It is the latter assumption that is reasoner-specific, thus justifying the claim that a *get_current_hypothesis* service is not necessarily appropriate. Further, this sequence demonstrates that such a service may not be needed. For the remainder of this paper, we will assume our implementation has this derived service at its disposal and that the underlying reasoner is able to provide the required confidence information.

According to the state diagram in Figure 1, hypothesis computation only occurs following the evaluation of a test result. A more general scenario would tie a service request (or transition) off the READY state to compute the hypothesis. This leads naturally to the service *report_answer* that would be a service provided by the user of the reasoner rather than the reasoner itself. Once an answer is available (in the *hypothesis* set), the application that requested the hypothesis can then use services, perhaps from a presentation system, to actually report the answer. Depending on the hypothesis, the user may then request the reasoner to revert some number of steps or to continue its diagnosis. For reverting to a previous state, the service call, *revert(num_steps)* would be used.

From some point in the diagnosis, the test environment may be required to perform an additional test. One of the intended roles for the diagnostic reasoner is as an optmizer that selects tests to be performed, minimizing some cost function. A possible sequence of

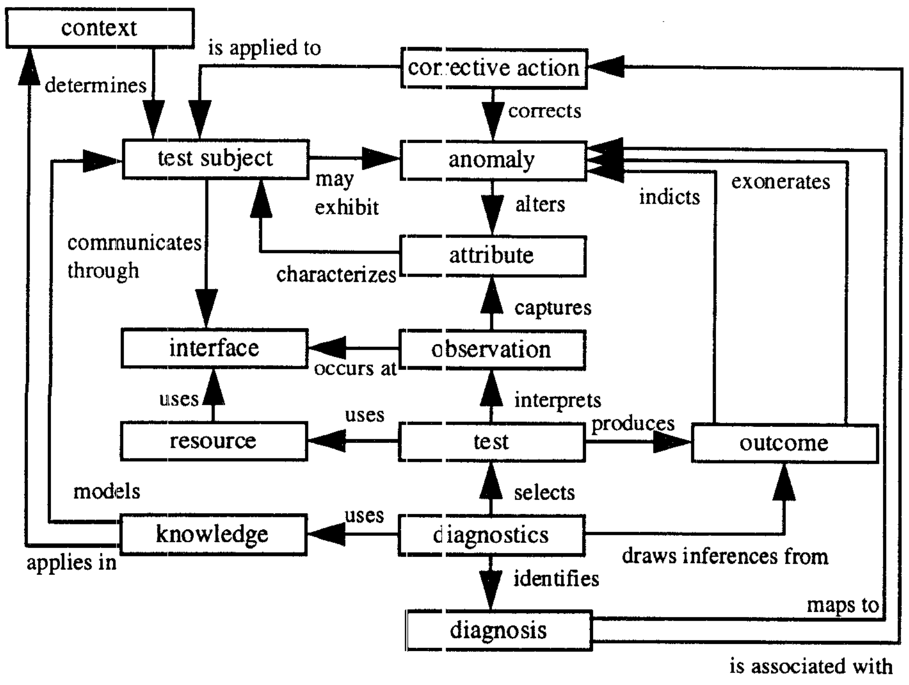


Figure 7. Model of System Test.

service calls in which this optimization capability would be used might follow the READY — *evaluate_symptoms* — HYPOTHESIS-COMPUTED — *choose_test* — READY — *result_known* — HYPOTHESIS-COMPUTED — *choose_test* — READY cycle in the state diagram. Following this cycle, the set of symptoms would be empty, and then a sequence of tests would be selected, evaluated, applied, etc., until an answer is obtained. The corresponding services for this procedure (assuming we have already attached a model and determined the desired set of cost attributes) might include the following:

```
symptoms = $\emptyset$
evaluate_symptoms(model_id,symptoms)
get_current_hypothesis(model_id)
put_active_cost_attributes(model_id,cost_attributes)
do while ((test_id = select_test(model_id)) $\neq$ NULL)
    perform_test(test_id)
    get_test_outcome(test_id,&outcome,&confidence)
    status_code =
        apply_test_outcome(model_id,
            test_id,outcome,confidence)
    get_current_hypothesis(model_id)
od
```

As with the data models, providing a list of software services such as *evaluate_symptoms*, *apply_test_outcome*, and *get_current_hypothesis* are not sufficient to define a standard. The interface itself must be standardized with appropriate bindings (or mappings) to implementations in a programming language.

The approach taken by the AI-ESTATE subcommittee to standardizing the software services is somewhat unique. AI-ESTATE decided to specify the services using the functional and procedural notation of EXPRESS. Typically, this aspect of EXPRESS is used only within the context of defining rules and constraints on information entities. For consistency, the AI-ESTATE subcommittee decided these facilities would work equally well with the services too.

To define the services, the models specified in 1232.1 provided the data types of the information to be passed into and out of the services. Next the services themselves were specified in EXPRESS as if they were used by constraints on the information models. For example, *apply_test_outcome* was specified as follows:

```
PROCEDURE apply_test_outcome(
   model_id : diagnostic_model;
   test_id : test;
   outcome_id : outcome;
   confidence : confidence_value);
```

From this specification, bindings were created in the target programming language. For example, in the target application language C, the function prototype for *apply_test_outcome*, derived from the EXPRESS specification, is as follows:

```
void apply_test_outcome(
    diagnostic_model *,
    test *,
    outcome *,
    confidence_value);
```

This also defines the function call and expected parameters, with types, for an implementation of the service in the target reasoner.

6. Applying AI-ESTATE

Since the purpose of testing is to gain information about the system under test, and diagnosis is the process of interpreting the test information, we assert all testing is done in the context of diagnosis. More simply, the only purpose a test serves is to provide an outcome that can be used to infer something about the system being tested. This premise broadens the view of diagnosis beyond the process for only isolating faults. Rather, diagnosis is considered to be the process of determining the state of the system under test relative to some anticipated state. Therefore, we envision AI-ESTATE being applied in a wide variety of test contexts.

To capture this expanded role of a diagnostic system in test, we developed a model of the test process. This model is described in detail in [14] and is reproduced here in Figure 7. In interpreting this model, note that we have not applied any of the standard object [15,16], activity [17], or information modeling [18] techniques commonly in use today.

To read this model, each of the blocks represents entities or objects within the test environment. The relationships between the entities provide "processes" or "constraints" that one entity performs or imposes on another. In some cases, the processes are highly dynamic (e.g., diagnostics selects test), and in others they simply define a relationship (e.g., knowledge applies_in context). In all cases, the entity to entity relationships can be read as a simple sentence (subject verb object). Given this approach, we can now read the model as a narrative description of the test process.

The element of this model that is of the greatest interest to AI-ESTATE is the diagnostics. We believe diagnostics drives the test process. As we said before, the purpose of testing is information discovery and drawing conclusions about the test_subject. The process of drawing conclusions from test information is referred to as "diagnostics." Thus we consider the diagnostic process centering on the entity labeled diagnostics. Here diagnostics uses knowledge to draw_inferences_from outcome to then identify diagnosis. As one might expect, diagnostics selects test which in turn produces outcome. Then outcome either indicts or exonerates anomaly. Once the diagnosis has been determined, one can take appropriate action since diagnosis maps_to anomaly and is_associated_with corrective_action. The whole process of performing a diagnosis is fully dependent on what diagnostics knows about the test_subject. This is captured by knowledge that models test_subject and applies_in context.

Testing can be performed under a wide variety of contexts. Indeed, the context completely defines the requirements and objectives of the test process. For example, in the context of specification compliance, testing can determine whether or not a design or a manufactured unit conforms to specifications. In the context of acceptance testing or product delivery, testing can determine if the unit satisfies a set of customer requirements. During performance or operational evaluation, testing can be used to determine optimal performance parameters for the system. In maintenance, testing can assist in detecting and isolating faults.

The primary advantage of providing an abstract model for test and diagnosis and associating that model with a "system view" of the test subject is that it provides a means for abstracting details out of the test problem until needed. This in turn permits optimizing the structure of the test problem, thus managing the complexities inherent in manipulating large, heterogeneous systems. For this to work, we need to be able to see how to map real systems into the model. This in turn enables us to identify the critical elements of the test problem and to optimize the solution. In this section, we map two "systems" into the model to illustrate.

The first system we consider is very simple. Consider an integrated circuit containing four D flip-flops (e.g., an SN5475). The test_subject would correspond to the this IC, and we would identify the attributes of the IC as the nominal behavior of the chip (as we might find in the truth table for each of the flip-flops). In addition, we might add characteristics such as voltage, temperature range, orientation of the pins, size and color of the packaging, etc. The actual set of attributes of interest to us would depend on the context under which we are testing. If we only care about failure modes associated with the pins and the logic of the chip, we may restrict our focus to the logic values observed at the pins. If we are interested in manufacturing issues, we might include characteristics of packaging.

For the sake of discussion, we limit the context to be assessing the logical performance of the chip at the pins.

Thus we will restrict our attention to the logic specification and to the set of stuck-at faults. This set of faults would define the `anomalies` that might be exhibited by the chip. If we use a digital tester with a fault dictionary, then `knowledge` would correspond to the fault dictionary itself and `diagnostics` would include the test controller and the matching algorithm in the dictionary. The `tests` would be defined by the vectors in the fault dictionary (actually, each of the output values for each vector would correspond to a different `test`). Whenever a vector is processed, the associated tests would either pass or fail (thus we know the `outcomes`), and these outcomes would point to the possible anomalies of the chip. In th event the chip is faulty, possible `corrective_actions` include replacing the chip or re-setting (or re-soldering) the chip in the socket.

The role AI-ESTATE plays in this environment is two-fold. First, a model conforming to the EDIM defined by 1232.1 can be used to capture the diagnostic knowledge resident in the fault dictionary [19]. This means that the diagnostic knowledge inherent in fault dictionaries can be exchanged in a standard way between test systems by using IEEE Std 1232.1. When coupled with another standard such as P1029.1 (WAVES), P1445 (DTIF), or P1450 (STIL), complete digital test information can be tied to an encapsulated diagnostic system to facilitate efficient and accurate diagnostics.

The second role for AI-ESTATE facilitates a means for replacing the traditional, but often proprietary, fault dictionary signature matching process. By providing standard services for applying and interpreting test results, an AI-ESTATE-conformant reasoner can replace the signature matcher.

For the second system, we consider a case at the other end of the spectrum. Consider a network of satellites in orbit that provide the backbone for a communications network. One test `context` of interest would be to verify that all of the defined links in the network are present (including satellite-to-satellite cross links and ground links). Thus we have defined our `test_subject` to be the connectivity of the network. `Attributes` of this system might include transmission of messages between two points in the network in a reasonable period of time, and we could define `tests` to be a set of messages that traverse the network in some predictable way. The `anomalies` would correspond to links being down.

For this system, the `knowledge` required for diagnosis and the associated `diagnostics` might be significantly different from the fault dictionary of the first system. For example, it is possible we would be able to rely on a set of SNMP (simple network management protocol) traps to signify when a link drops in the network (assuming the endpoints of the links are treated as SNMP managed objects). In this case, the `knowledge` would correspond to the MIB (managed-object information base), and the `diagnostics` would be a passive network monitoring system or alarm correlator.

Here, the AI-ESTATE application is also two-fold. Many network systems and associated network management systems utilize heterogeneous software technology. To date, they have focused on information provided by standard protocols and specifications of managed objects and not on capturing information relating network alarms to possible causes. Similar to the fault dictionary discussed above, models to be used in alarm correlation can be developed, merged, and exchanged between applications in a network environment using IEEE Std 1232.1. Similarly, the alarm correlation and fault isolation services needed in network fault management and trouble ticketing systems can utilize IEEE Std 1232.2.

7. Industry Acceptance

The AI-ESTATE standard has been under development for six years under the sponsorship of three IEEE societies—the Computer Society, the Aerospace Electronic Systems Society, and the Instrumentation and Measurement Society. Current membership on the subcommittee includes representatives from academia (University of Connecticut and Johns Hopkins University), government (U.S. Navy, U.S. Air Force, U.S. Army), and industry (McDonnell Douglas, Boeing, NCR, AlliedSignal, IET Intelligent Electronics, ARINC, Etec). In addition, the subcommittee has representatives from several countries other than the United States (United Kingdom, France, Germany).

Recently, with the approval of IEEE Std 1232-1995 and IEEE Std 1232.1-1997, the International Electrotechnical Commission's Technical Committee 93 (IEC/TC93), which focuses on standards for design automation, accepted a proposal to advance the AI-ESTATE standard for fast track standardization at the IEC level. Current membership of IEC/TC93 includes representatives from Denmark, Finland, France, Germany, Japan, the United Kingdom, and the United States. Sponsorship for AI-ESTATE fast track came from the French delegation. The IEC/TC93 is best know for standardizing EDIF 3 0 0 and VHDL at the international level.

In addition to the broad support within the standards community, several tool vendors are committed to enhancing or providing tools that conform to the AI-ESTATE standard. Several tools already implement pre-standard versions of the AI-ESTATE models, including

ARINC's System Testability and Maintenance Program (STAMP) and Portable Interactive Troubleshooter (POINTER), Detex's System Testability Analysis Tool (STAT), IET's TechMate, Giordano Associates' Diagnostician, Qualitech's Testability Engineering and Maintenance System (TEAMS), and the Navy's Integrated Diagnostic Support System (IDSS). These tools are likely to be upgraded to the AI-ESTATE standards in the near future.

8. Future Work

In spite of the large amount of work that has been done on the AI-ESTATE standards and the wide acceptance of the standards in government and industry, much additional work needs to be done to keep the standards in pace with technology advances. Currently, several projects are underway within the AI-ESTATE subcommittee to do just that.

First, we recognize that there are more approaches to performing diagnosis than using fault trees and diagnostic inference models. Currently, we are in the process of defining a constraint model that will capture temporal, logical, and resource constraints in testing and diagnosing a system. Related to this is work underway in the ABBET (A Broad Based Environment for Test) subcommittee and EDIF committee defining a test requirements model (TeRM). TeRM that includes constraints for capturing information about the behavior of a product and a test resource. We expect to be able to work with the constraints defined in TeRM to facilitate reasoning with constraint-based systems.

In addition to the constraint model, we anticipate developing models for rule-based systems and for connectionist systems. Rule-based systems provided the first success stories in diagnosis and artificial intelligence. While both the EDIM and the constraint model would be able to capture the logical information contained in a rule base, a separate model is required that is tailored to the rule-based architecture.

In addition, neural networks and fuzzy systems have shown tremendous promise in diagnostics, especially in the presence of noise and error. While it is unclear at this time what a standard representation of a neural network would look like, we feel it is a worthwhile endeavor to explore the possibility of defining such a standard.

Upon defining additional models for knowledge exchange, we will also need to define standard services for accessing and manipulating these models. Thus we see that the development of services and the development of data models cannot be done independently. While objectives for these two activities can be separated, the common role the information plays in both contexts necessitates developing the models and the services together.

As discussed in section 6, diagnosis occurs within some context. In fact, the context is central to determining the scope of the test and diagnosis problem. Unfortunately, capturing information about context in a standard way is problematic. The number of variables associated with context is excessive, and the relationships between those variables are frequently unknown. Nevertheless, we find that proper interpretation of diagnostic and test information relies upon a common understanding of the context in which testing takes place. As a result, we are beginning to develop a model of context for diagnosis. Recent work in non-monotonic reasoning and categorical reasoning offer promise in formalizing context for this problem [20].

9. Conclusion

Reasoning system technology has progressed to the point where electronic systems are employing artificial intelligence as a primary component in meeting system test and verification requirements. This is giving rise to a proliferation of AI-based design, test, and diagnostic tools. Unfortunately, the lack of standard interfaces between these reasoning systems is increasing the likelihood of significantly higher product life-cycle cost. Such costs would arise from redundant engineering efforts during design and test phases, sizable investment in special-purpose tools, and loss of system configuration control.

The AI-ESTATE standard promises to facilitate ease in production testing and long-term support of systems as well as reducing overall product life-cycle cost. This will be accomplished by facilitating portability and knowledge reuse and sharing of test and diagnostic information, among embedded, automatic, and stand-alone test systems within the broader scope of product design, manufacture, and support.

Acknowledgments

We would like to thank the members of the AI-ESTATE subcommittee of SCC20 and to the many reviewers of the AI-ESTATE standards. Especially, we thank Randy Simpson, Greg Bowman, Tim Bearse, Tim Wilmering, Rick Maguire, Mark Kaufman, Tony Bartolini, Jack Taylor, Sharon Goodall, Mukund Modi, and Helmut Scheibenzuber for their assistance in bringing this standard to fruition.

References

[1] IEEE Std 1232-1995. 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.

[2] IEEE Std 1232.1-1997. 1997. *Trial Use Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE): Data and Knowledge Specification*, Piscataway, New Jersey: IEEE Standards Press.

[3] IEEE P1232.2. 1996. *Trial Use Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE): Service Specification*, Draft 2.2.

[4] IEEE Std-1149.1-1990. 1990. *Standard Test Access Port and Boundary Scan Architecture*, Piscataway, New Jersey" IEEE Standards Press.

[5] IEEE P1450. 1996. *Standard Test Interface Language (STIL)*, Draft 0.23.

[6] IEEE P1252. 1993. *Standard for a Frame Based Knowledge Representation*, Draft 2.1.

[7] ISO 10303-11:1994. 1994. *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 11: EXPRESS Language Reference Manual*, Geneva: ISO Press.

[8] Simpson, W. R. and J. W. Sheppard. 1993. "Fault Isolation in an Integrated Diagnostic Environment," *IEEE Design and Test of Computers*, 10(1):52–66.

[9] Simpson, W. R. and H. S. Balaban. 1982. "The ARINC System Testability and Maintenance Program (STAMP)," *Proceedings of AUTOTESTCON '82*, Dayton, Ohio.

[10] Peng, Y. and J. A. Reggia. 1990. *Abductive Inference Models for Diagnostic Problem-Solving*, New York: Springer-Verlag.

[11] Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, San Mateo, California: Morgan-Kaufmann Publishers.

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

[13] ISO 10303-21:1994. 1994. *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 21: Clear Text Encoding of the Exchange Structure*, Geneva: ISO Press.

[14] Sheppard, J. W. and W. R. Simpson. 1996. "A Systems View of Test Standardization," *Proceedings of AUTOTESTCON '96*, Dayton, Ohio, pp. 384–389.

[15] Booch, G. 1994. *Object-Oriented Analysis And Design With Applications*, 2nd Ed. Benjamin Cummings.

[16] Schlaer, S., and Mellor, S. L. 1992. *Object Lifecycles: Modeling the World in States*, Englewood Cliffs, New Jersey: Yourdon Press.

[17] FIPS-183. 1993. *Integrated Definition for Function Modeling (IDEF0)*. National Institute of Standards and Technology.

[18] Schenk, D. A. and P. R. Wilson. 1994. *Information Modeling: The EXPRESS Way*, New York: Oxford University Press.

[19] 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.

[20] Akman, V. and M. Surav. 1996. "Steps Toward Formalizing Context," *AI Magazine*, 17(3):55–72.