Prototyping a Diagnostic Interface

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

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

***Abstract:** The ARI is planning to prototype several potential interface implementations related to TPS transportability and interoperability. For the diagnostic interface, examination of available commercial standards has identified only one viable candidate—IEEE Std 1232, AI-ESTATE (Artificial Intelligence Exchange and Service Tie to All Test Environments) and its component standards. In this paper, we will discuss how to apply the AI-ESTATE standards to satisfy the diagnostic interface requirements for an ARI prototype.*

I. Introduction

According to the Automatic Test System (ATS) Research and Development (R&D) Integrated Product Team (IPT) system engineering plan, "the primary goal of the ARI is to define a generic ATS open systems architecture from which specific hardware, software, and system implementation architectures can derive [1]." The identified objectives of defining an open system architecture include:

- Supporting life-cycle test
- Supporting TPS rehost
- Supporting and promoting product and test information reuse
- Facilitating commercial acceptance

To that end, the ARI has decided to apply a spiral development approach to defining this generic ATS open architecture. The approach follows four major evolutions with prototyping activities occurring in each evolution. These four evolutions have been identified as:

1. Instrument Interchangeability and Interoperability
2. TPS Transportability and Interoperability
3. Life-cycle Information Exchange
4. Processes and Tools

Within the Department of Defense, there is direction to move procurement activities toward commercial processes and standards. To facilitate this process, the use of commercial standards is being evaluated in defense ATS environments. When the standards are successfully evaluated, they will be included in the ATS Executive Agent's acquisition guidance requirements for implementing the ATS critical interfaces [2].

Currently, the ATS R&D program is analyzing the results of prototyping Evolution 1 interfaces and is beginning to consider issues related to Evolution 2. Moving into Evolution 2 entails incorporating technology to address requirements on capturing product design data and on implementing the diagnostic interface (DIA). This document discusses issues related to the requirements for the DIA interface [2].

II. The Diagnostic Interface

According to the ATS Critical Interfaces report, the DIA interface is "the interface protocol linking execution of a test with software diagnostic processes that analyze the significance of test results and suggest conclusions or additional actions that are required [2]." The key issues identified by this definition include:

- The relationships between test and diagnosis,
- The analysis and interpretation of test results,
- Diagnostic inference resulting from analyzing test results,
- Recommendations of appropriate actions based on inference.

Further, the recommendations resulting from diagnoses may address issues of

- Test setup or preparation,
- Test selection,
- Corrective action.

During Evolution 2, the ARI is planning to prototype several potential interface implementations related to TPS transportability and interoperability. For the DIA interface, examination of available commercial standards has identified only one viable candidate—IEEE Std 1232, AI-ESTATE (*Artificial Intelligence Exchange and Service Tie to All Test Environments*) and its component standards[3–5]. This paper provides a detailed discussion of how to apply the AI-ESTATE standards to satisfy the DIA interface requirements for the Evolution 2 prototype.

0-7803-4420-0/98/$10.00 ©1998 IEEE

A. Diagnosis

The current approach to testing intertwines the test procedures, the sequence of tests, and the diagnostic outcomes. To use the DIA interface effectively, details on the test procedures must be separated from the details on diagnosis, and vice versa. This intertwining of the test procedures with diagnostic information inferred from tests creates an obstacle in achieving test reuse. By separating test procedures and "encapsulating" them, the diagnostic information can be related to them as determined by the system under test without requiring significant re-work of the test procedures themselves [6].

To fully understand the importance of this separation of test and diagnosis, several technical disciplines must come to bear. The key engineering discipline having a direct impact on diagnostics and the ultimate efficacy of the diagnostic interface is termed, *testability*. In the context of the ARI, testability is directly related to the ability to isolate faults. As such, it extends beyond the traditional view of the ability to control and observe to include the ability to infer the status of the system.

One of the primary objectives of the ARI is to facilitate communication between design and test activities to capitalize on the impact of design on ultimate test effectiveness. Thus, it becomes useful to consider testability within the broader context of *integrated diagnostics*, which is considered a part of the process of *concurrent engineering*. While generally regarded more as a "buzzword" than as a real discipline, integrated diagnostics includes an important concept of applying structure to the process of system support. The ultimate goal of integrated diagnostics is maximizing the effectiveness of diagnostics, but this goal can only be achieved in an environment where design and test elements are utilized in an integrated (i.e., coordinated) fashion.

The word "diagnosis" is derived from two Greek words for "to discern apart." Thus the goal of diagnosis is the differentiation among multiple things (e.g., faults). In the context of the ARI, diagnosis is the process of drawing conclusions about the system under test and can be considered at one of three levels.

1. *Detection*. This is the ability of a test, combination of tests, or a diagnostic strategy to identify that a failure in a subsystem has occurred.
2. *Localization*. This is the ability to say that a detected fault is restricted to some subset of the possible causes. Specifically, if the unit is not functioning properly, then localization focuses the attention of the tester to determine what is wrong.
3. *Isolation*. This is the identification of a specific cause of a fault or anomaly through some test, combination of tests, or diagnostic strategy. The focus for isolation is determining what must be done to return the unit to service.

B. Diagnostic Inference Models

Many approaches exist for performing fault diagnosis. In fact, methods for improving the diagnostic process have been studied for many years, with particular attention coming from the artificial intelligence community. One formal approach from artificial intelligence that facilitates developing diagnostics is derived from the principles of formal logic. This approach uses *diagnostic inference models* (DIMs) to capture logical relationships between tests and faults in a unit.

DIMs are constructed to identify causal relationships between diagnoses and tests. In other words, when considering only Pass/Fail outcomes, the DIM captures which faults "cause" which tests to fail. In the more general case where a test may have three or more outcomes, the conclusions "causing" the corresponding test outcomes are identified.

Logically, DIMs provide "rules" for inferring candidate conclusions (e.g., faults) from a set of test results. In other words, given a set of test results, the process of diagnosis with a DIM is to determine which conclusions are consistent with these results.

Since DIMs only provide the logical relationships between tests and conclusions, one can see that the DIM is an abstraction of the test information and does not include specific details of the tests or the faults themselves. DIMs do not include procedural information or physical information about the unit under test.

The process of combining information from multiple, possibly heterogeneous sources and drawing some conclusion from this information is called *information fusion*. The diagnostic problem can be posed as an information fusion problem in that information from multiple tests is being combined to draw conclusions about the health of the unit being tested. Information fusion carries with it a formal set of mathematical tools and techniques that are readily applied to the diagnostic problem.

From the view of information fusion, a test is "any signal, observation, or other event that may be caused to happen [6]" and provides information about the system. Thus, simple observations or formal stimulus-response procedures are regarded as tests. In addition, a diagnosis is any conclusion to be drawn about the health-state of the system; thus, no-fault is a diagnosis. Further, diagnoses *may* or *may not* relate to faults, failure modes, or nominal and anomalous behaviors of a system.

C. DIA Interface Requirements

To satisfy the requirements for the DIA interface, we have analyzed the factors affecting test and diagnostic re-use and measurement capabilities. From this we draw the following conclusions:

1. It is necessary to separate test from diagnosis to facilitate re-use of test and diagnostic *methodologies*

0-7803-4420-0/98/$10.00 ©1998 IEEE

and *tools*. Such separation permits independent analysis and development. It also allows a plug-and-play environment to change only the relevant elements in each of the domains.

2. It is necessary to address issues of incorporating context and state dependence. These are aspects of diagnostic problems that have either been embedded or ignored. Such dependence severely limits the ability to re-use either test or diagnostic elements.
3. Measurement theory dictates that, to obtain an accurate diagnosis, reasoning under uncertainty must be incorporated into diagnostic models and processes [9, 10]. Calibration only addresses one element of measurement theory, namely bias. Random error can best be addressed by statistical approaches.
4. The largest measure of cost savings will be obtained through test synthesis. Related to this will be derivation of diagnostic models. The largest single cost element in the development of a TPS is labor. Providing a reasonable tie to design environment will promote synthesis of tests and diagnostic models; thereby reducing labor cost. These factors will be addressed in Evolution 3; therefore, it is essential that the DIA interface (defined in Evolution 2) be capable of being tied to the design environment.
5. Even utilizing effective reasoning under uncertainty and test/diagnostic synthesis, variation between systems and contexts will lead to inaccuracy in diagnosis. In the long term, the best method for combating this inaccuracy will be through machine learning. Machine learning environments will be dealt with in Evolution 3; therefore, it is essential that the DIA interface support a machine learning framework.

We have examined several alternative standards for satisfying DIA interface requirements, including AI-ESTATE (P1232), DTIF (P1445), Boundary Scan (P1149), and STIL (P1450). Based on the above, it is apparent that an IEEE 1232-compliant reasoning system should be used in that it is capable of incorporating methods addressing each of the issues raised above. Further, AI-ESTATE can incorporate implementations of the above-mentioned standards.

III. AI-ESTATE

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)" 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 systems, 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 to tie these models to domain-specific models as needed to complete the test system.

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

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

A. AI-ESTATE Architecture

The AI-ESTATE architecture presented in Figure 1 shows a conceptual view of an AI-ESTATE-conformant system [3]. AI-ESTATE applications may use any combination of functional elements and inter-function 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. One significant difference between a traditional client-server model and AI-ESTATE is the assumption of an abstract application executive mediating service requests. In some sense, this application executive can be regarded as a service broker for the subsystems in the test environment. Services are "published" to the application executive, and service

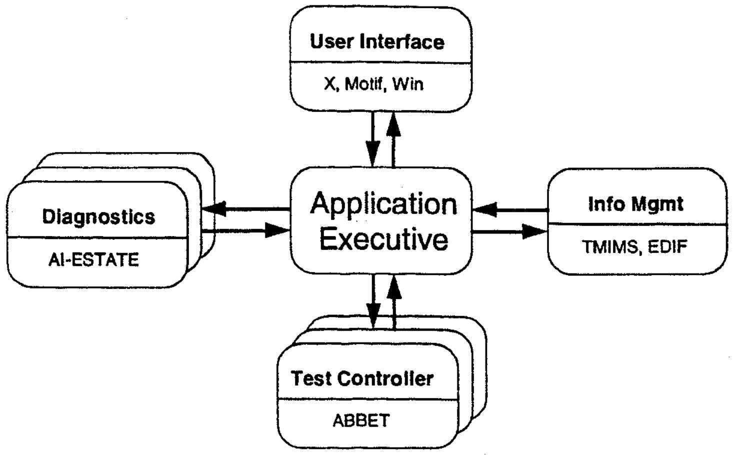

Figure 1. AI-ESTATE Architecture

requests from other subsystems are matched to the available services to satisfy the request. This idea is analogous to the CORBA architecture except with no underlying assumption of being object-oriented. Further, since the application executive is abstract, it is still possible for subsystems to interact directly with other subsystems using their published services.

From the vantage point of an AI-ESTATE-conformant diagnostic reasoner, one sees the interaction with the application executive from two views. First, the AI-ESTATE-conformant reasoner makes available several services (as defined by IEEE P1232.2) to the application executive for traversing diagnostic models or actually performing diagnosis given test results. Second, the diagnostic reasoner interfaces with other subsystems in the test environment (e.g., the test system) by requesting services from the application executive. For example, while the reasoner will not perform any tests, it is likely to request certain tests be performed in certain contexts. The application executive will be used to submit the request to the test system.

In addition to interfacing with the application executive, it is assumed the AI-ESTATE-conformant reasoner has direct access to diagnostic models. IEEE Std 1232.1 provides a means for exchanging models between conformant reasoners, and this exchange can either be accomplished using model traversal services or using the interchange format defined in 1232.1.

B. Diagnostic Models

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 [4]. All of the models were defined using ISO 10303-11, EXPRESS [7]. EXPRESS is a language for defining information models and has received widespread 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. The common element model also includes a formal specification of costs to be considered in the test process. 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. 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*). Currently, 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 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. 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.

Recently, researchers and practitioners of diagnostic modeling found that the functional dependency approach to modeling 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. 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 information models defined in the AI-ESTATE standard 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 [8]. EXPRESS-I is an instance 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. Unfortunately, the STEP physical file format is difficult for a human to read but 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.

C. Diagnostic Services

The AI-ESTATE standard defines several software services to be provided by a diagnostic reasoner [5]. 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. Currently, the services defined by AI-ESTATE are classified as either static model traversal services, reasoner state services, and reasoner control services. Following the object-oriented programming paradigm, we found that most services could be represented in one of four forms: **`create`**, **`get`**, **`put`**, or **`delete`**.

With the publication of the data and knowledge specification, the first set of services defined for the service specification focused on the existing models. As described above the data and knowledge specification defines three models—a common element model, a fault tree model, and an enhanced diagnostic inference model (EDIM). The common element model provides definitions of basic entities expected to be used by any diagnostic reasoner, and the fault tree model and EDIM organize these entities in a way to facilitate diagnostic reasoning.

The model traversal services provide the means for a reasoner to process the model on-line. As an example, IEEE Std 1232.1-1997 defines, using EXPRESS, a diagnostic model to be the following entity:

```
ENTITY diagnostic_model;
    name              :    name_type;
    description       :    description_type;
    model_test        :    SET [1:?] OF test;
    model_diagnosis   :    SET [2:?] OF diagnosis;
    model_resource    :    SET [0:?] OF resource;
    model_anomaly     :    SET [0:?] OF anomaly;
END_ENTITY;
```

Based on this entity, we can define services for traversing the model to obtain information about a particular diagnostic model. For example, consider the set of tests defined in a particular model. To find the set of tests, we must first have access to the model itself. To obtain this access, we need to know either the name or the internal identifier of the model.

The service, **`get_diagnostic_model(name)`**, returns an identifier for a diagnostic model identified by its name. Once we have the identifier, we can then access the set of tests defined in the model with the service **`get_model_tests(model_id)`**. This service would return a list of tests that can then be used with another service to reference the actual test entities.

Associated with each entity in the model will be the four types of services given above (i.e., **`create`**, **`get`**, **`put`**, and **`delete`**). EXPRESS includes the ability to define functions and procedures (similar to a programming language), but the intent is for these functions and procedures to be used in defining constraints on model entities. The AI-ESTATE committee observed, however, that EXPRESS provides a natural fit between information modeling and service definition since the models provide the type system. As a result, the committee decided to use EXPRESS to define the services as well. For example, **`get_model_tests`** is defined in P1232.2 as follows:

```
FUNCTION get_model_tests(model_id:diagnostic_model):
    SET [1:?] OF test;
END_FUNCTION;
```

Given a model has been loaded and a system is being tested, the model needs to be processed for diagnosis to be performed. Simply traversing the models defined in P1232.1 is not useful since the model only defines expectations rather than reality. Any reasoner will maintain an internal state of its reasoning process that will be used to report a diagnosis or to explain its reasoning. For diagnostic reasoners to be plug-and-play compatible, certain diagnostic state information needs to be in common between the reasoners.

To facilitate defining services for accessing or modifying the state of the reasoner, the AI-ESTATE committee made some assumptions about common diagnostic information. These assumptions were derived from the common element model and extended to support various types of reasoners. Specifically, the AI-ESTATE committee has defined a new information model, called the *dynamic context model* (DCM), that provides the type structure for reasoner state.

The "starting point" for the model is given by a diagnostic **session**. The session corresponds to a sequence of steps in which the state of the reasoner changes at each **step**. At each step, the reasoner keep track of what it knows about the state of models (i.e., whether or not a model is available for processing), the resources, the tests, and the diagnoses. In addition, for each resource and test, the actual cost incurred at that step (both time and non-time) is recorded.

This model can be traversed similarly to the models defined in P1232.1. More interesting is the fact that entities can be created dynamically and the states of these entities vary from step to step. Another interesting byproduct of this model is that, by organizing the state according to steps in a session, the information captured can be fed to a maintenance data collection system and retained as a diagnostic history.

Another issue related to reasoner state is how one "explains" the reasoning process. During committee discussion, this was a highly contentious issue since no one could agree on a good method for explanation. The debate was settled when it was observed that the process of traversing reasoner state provides a rudimentary approach to explanation. Then, if tool developers want to provide "higher order" services, they can use the standard services to collect the necessary information. For example, explaining the currently inferred value for a diagnosis can be accomplished by identifying the tests in the model that affect the diagnosis and showing the step in the session where one of these tests was performed that led to the current assertion.

IV. EXAMPLES

The first "reasoner" model developed for AI-ESTATE was the fault tree model. This model provides a baseline for AI-ESTATE since most test environments in existence today follow some kind of fixed test strategy that can be modeled as a fault tree. The AI-ESTATE fault tree also extends the notion of the traditional fault tree in two ways. First it permits test confidences to be processed to establish a level of "belief" in the reported diagnosis. This provides a simple facility for satisfying DIA requirement to reason with uncertain information, even in the fault tree. Second, it provides the capability of reporting "intermediate diagnoses" in the interior of the tree rather than waiting until the leaves of the tree to report diagnostic information.

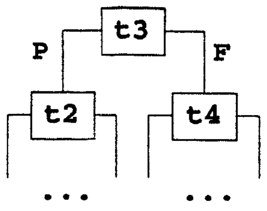

Figure 2. Example Fault Tree

The following example shows a small portion of a fault tree specified in EXPRESS-I (Figure 2). Note the fault tree model assumes the incorporation of entities from the common element model. These entities are not shown in the example. In this example, the first entity, given as **FT**, identifies the fault tree and points to the first step (**Step1**) of the tree.

```
FT = fault_tree_model
{
    SubOf(@model1);
    fault_tree -> @Step1;
};

Step1 = fault_tree_step
{
    test_step -> @t3;
    result -> (@S1_Result1,@S1_Result2);
};

S1_Result1 = test_result
{
    test_outcome -> @t3_pass;
    next_step -> @Step2;
    current_diagnosis -> ();
};
```

Step1 shows we are considering test **t3** that has two results, **S1_Result1** and **S1_Result2** corresponding to the test passing or failing respectively. Assuming the test passes, we go to **S1_Result1** that points to the next step in the tree, **Step2**.

The "primary" reasoning model defined in AI-ESTATE is the enhanced diagnostic inference model (EDIM). Philosophically, the EDIM is derived from the assumption that information provided by tests is what matters rather than focusing on the diagnoses that might be drawn. The EDIM does not explicitly include sequence information but instead records logical relationships between tests and diagnoses.

Since sequencing can be a significant concern as a part of the diagnostic context, AI-ESTATE provides a means for coupling EDIMs and fault trees together by way of the **diagnostic_knowledge** entity in the common element model. At any point, if a test is selected that is simultaneously a test in the EDIM and an initial test in a fault tree, the application executive can shift from the EDIM to the fault tree and follow the predefined sequence. When the fault tree terminates, if tests remain that could further resolve the diagnoses, the application executive can shift back to the EDIM to continue. This mechanism partially fulfills the DIA requirement for addressing context in diagnostics.

The following example shows a small portion of an EDIM derived from a digital fault dictionary. As described earlier, the EDIM is a set of inferences as shown in the **inference** attribute of **edim**.

```
edim = enhanced_diagnostic_inference_model
{
    SubOf(@model1);
    inference -> (@t1_pass_implies,
              @t1_fail_implies,
              @t2_pass_implies,
              @t2_fail_implies,
              @t3_pass_implies,
              @t3_fail_implies,
              @t4_pass_implies,
              @t4_fail_implies);
};

t1_pass_implies = outcome_inference
{
    test_outcome -> @t1_pass;
    conjuncts ->
              (@x_sa1_absent,@y_sa1_absent,
              @S_sa1_absent,@C_sa1,absent);
    disjuncts -> ();
};

x_sa1_absent = inference
{
    SupOf(@x_sa1_elim);
    pos_neg -> inference_type{!positive};
    confidence -> confidence_value(0.99);
};
```

A particular inference (**t1_pass_implies**), specifies the identification of the actual test outcome (**t1_pass**) and the inferences that can be drawn. Note the test outcome points to the respective test as one of its attributes thus eliminating the need for the inference to point directly to the test. In this example, the inferences are limited to a set of AND diagnostic inferences in which several candidate faults are eliminated from consideration.

Three example services are provided to demonstrate the tie to the models and the specification in EXPRESS. The service **get_test_outcomes** is a model traversal service that returns the set of outcomes associated with the identified test. The service **put_actual_outcome** demonstrates how an attribute can be updated using the services but is tied to the dynamic context model rather than the knowledge model. Thus there are no modifications of the set of outcomes tied to a test, but the specific outcome received for a test is stored with the active test evaluated at the current step. The service **get_hypothesis** is also tied to the dynamic context model and simply returns the set of diagnoses that are identified at the current step in the process.

The following example demonstrates using model traversal services to process fault tree model.

```
curr_step = get_fault_tree(model_id);
while (curr_step != NULL)
{
    test = get_test_step(curr_step);
    outcome = perform_test(test);
    outcomes = get_test_outcomes(test);
    for (i=0;i<len(outcomes);i++)
        if (outcome == outcomes[i])
        {
            result = get_result(curr_step,i);
            curr_step = get_next_step(result);
            break;
        }
}
answer = get_current_diagnosis(result);
```

Technically, it is not necessary to use the dynamic context model with the fault tree. However, from the perspective of consistency in reasoning and the ability to track reasoner state (for historical or other purposes), it is still advisable to use the DCM. Note that the service **perform_test** is *not* an AI-ESTATE service. All of the other services correspond to services defined by the AI-ESTATE standard. It is assumed the test environment through the application executive is providing this service.

The third example demonstrates the use of the dynamic services to process an EDIM.

```
curr_step = get_step(session_id,0);
while (test_available(curr_step))
{
    test = choose_test(curr_step);
    outcome = perform_test(test);
    next_step = create_step();
    put_step(session_id,next_step);
    put_actual_outcome(next_step,test,outcome);
    update_state(session_id,next_step);
    curr_step = next_step;
}
answer = get_hypothesis(session_id,curr_step);
```

Several services in addition to the AI-ESTATE model traversal services (either persistent or non-persistent) are required, including **test_available**, **choose_test**, **perform_test**, and **update_state**. The services **test_available**, **choose_test**, and **update_state** are services tied to reasoner control. The service **test_available** determines if any tests have unknown outcome (or perhaps low confidence) and can be performed. The service **choose_test** selects a test from the set of available tests to be performed. The service **update_state** uses the active model to update the states of tests and diagnoses in the model and assigns the new state to the current step in the DCM.

V. SUMMARY

Given the assumptions of the AI-ESTATE architecture, the models and services of AI-ESTATE have broad applicability to advanced diagnostics. As claimed in the acronym, it is believed that AI-ESTATE covers all essential elements of diagnostic reasoners in all test environments. Nothing in the standard limits the applicability of the standard to a particular approach to test; however, the generality covers the diagnostic requirements for ATS as needed in Evolution 2 and 3.

The principal assumption of AI-ESTATE is separation of test and diagnosis. Without this separation, the standards cannot be used effectively, nor can the DIA requirements be satisfied. By separating the diagnostics from the test process, it also provides a means of developing more accurate diagnostics and a means for understanding and validating the diagnostics.

The advantage to AI-ESTATE is that it provides a consistent framework for incorporating diagnostic knowledge and services in any test environment. For example, the standards provide facilities for reasoning with multiple models and for coupling fixed fault trees and dynamic EDIMs, which may be required in Evolution 2 depending on the UUTs chosen for the prototype. In addition, this framework supports a plug-and-play approach for incorporating diagnostic reasoners into the ATS architecture.

REFERENCES

[1] ATS R&D IPT. (1997, April). *Systems Engineering Plan for a Generic ATS Open System Architecture*, Version 1.01.

[2] ATS R&D IPT. (1996, September). *Automatic Test System Critical Interfaces Report*, Release 1.

[3] IEEE Std 1232-1995. *Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE): Overview and Architecture*, Piscataway, NJ: IEEE Standards Press.

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

[5] IEEE P1232.2. (1998, January). *Trial Use Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE): Service Specification* Draft 4.2.

[6] W. R. Simpson and J. W. Sheppard. 1994.*System Test and Diagnosis*, Kluwer Academic Publishers.

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

[8] ISO 10303-12. Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 12: *EXPRESS-I Language Reference Manual* Technical Report.

[9] William R. Simpson 1998, "Inaccurate Diagnosis and What to Do about Them"' digest of the 1998 IEEE International Workshop on System Test and Diagnosis (IWSTD '98), Alexandria, Virginia.

[10] J. W. Sheppard. and W. R. Simpson 1998. "Managing Conflict in System Diagnosis", Computer, Volume 31, Number 3 IEEE Computer Society , Alameda, California