Test Strategy Component of an Open Architecture for Electronics Design and Support Tools

Leonard Haynes, Ph.D.*
Sharon Goodall*
Floyd Phillips*

*Intelligent Automation, Inc
Rockville, MD

William Simpson, Ph.D.**
John Sheppard**

**ARINC Research Corp.
Annapolis, MD

ABSTRACT

This paper describes work being carried out under the auspices of the Artificial Intelligence, Expert System Tie to Automatic Test Equipment (AI-ESTATE) Committee of the IEEE Standards Coordinating Committee 20 (SCC-20), the goal of which is to develop a formal data model for dependency related information called an Information Flow Model (IFM). The paper includes the most important ENTITIES in the model, and a brief description of the model. The introduction attempts to place the IFM in the larger context of the Ada-Based Environment for Test, and briefly describes other related efforts. The model included in this paper is **not** an approved draft of SCC-20, and should be viewed solely as contributions of the authors. Readers having comments on the model are invited to provide those comments to the authors, an address for which is included in section 1 of the paper.

1. Introduction

Modern systems are becoming increasingly complex. Because of this increasing complexity, it has become essential to develop automated tools to assist a designer in the development of reliable, testable, and maintainable equipment, and to develop automated tools to help the maintainer test and diagnose that equipment. During the equipment design phase, there is great advantage to being able to predict the testability and maintainability of a design before final commitment is made to full scale production. Once hardware is fielded, the complexity of a modern system may necessitate automated tools which can assist a technician in both test and diagnosis of suspect equipment.

One of the most popular approaches to predicting the testability and maintainability of systems including those with both mechanical and electronic components is based on what is called a "dependency model." See references 1 and 2. The dependency model of equipment captures the relationship between the tests which can be performed at specific test points, and what is learned from each test. If data concerning the failure rates of individual components, the time and cost to perform tests, and other similar data is available, then analysis systems can use that dependency model and other data to compute the expected time and cost to diagnose the equipment, sparing requirements, etc. Hence dependency model based systems can provide valuable information during design time to predict life cycle costs, and to identify problems in the design which can be modified to reduce life cycle costs, and to increase system availability.

Dependency models can also be used to dynamically compute an optimal test strategy to diagnose a specific piece of equipment. Using a dependency model and the associated information as described above, the test sequence used to diagnose equipment can be optimized based on the equipment symptoms, technicians and test equipment available, current priorities, etc.

There are many tools available commercially which exploit dependency models. Companies which market dependency model based tools include ARINC Research Corporation (reference 1 and 2), DETEX Incorporated (reference 4), Automated Reasoning Corp (reference 5), Harris Corp., and BITE Inc. None of these tools interoperate, and the models used by one of these companies will not work with the products of any other companies. The lack of interoperability or portability of models is not due to complex technical problems. All of the tools which have been evaluated have similar dependency models. There are, however, enough differences between the model formats used by the available tools that models cannot be ported from one tool to another.

The remainder of this paper describes the efforts of the Artificial Intelligence, Expert System Tie to Automatic Test Equipment (AI-ESTATE) Committee of the IEEE Standards Coordinating Committee 20 (SCC-20) to develop a formal data model for dependency related information. The model for a Unit Under Test (UUT) must specifically encode the names of all tests, and for each test, encode what is learned from each of the possible test outcomes. In addition, related information includes the cost and time to perform the test, test equipment and technician skill levels required, and test reliability. The related information also includes component information including failure rates, cost to replace the component, groups of parts which normally fail together, etc. We call the total model an Information Flow Model (IFM) and we will use that name henceforth. The IFM must be generic and neutral in that it is adequate to model the information required by all the commercially available and U.S. Department of Defense tools which use dependency information.

The IFM standard will allow portability of dependency models between tools. The larger goal is to allow reasoners which use the IFM model to be "plug compatible," in the true sense of the concept of an open architecture. This will require a set of SERVICES and a set of PROTOCOLS to be developed and standardized. Applications which request only the standard set of services, and which adhere to the protocols will be able to use any compliant reasoner. This will still allow reasoners to provide additional non-standard capabilities. Normally, these non-standard capabilities would be in the form of additional information which would be provided in response to the standard service requests. In this case, plug compatibility would not be lost even though applications might use the non-standard information, as long as they could function without the additional information if it were not available. Work on the details of a SERVICES interface for IFM based reasoners is progressing, but space does not allow inclusion of additional information related to that effort.

The Information Flow Model is an abstract model, and will be represented in the International Standards Organization (ISO) standard language called EXPRESS. EXPRESS is an implementation independent language. It does not specify how the data of a particular model will be stored in a computer, what database system will be used to access the data, etc. In order to allow the physical exchange of models, some physical structure must be agreed upon, and at this point, it appears that we will use ISO STEP 10303.21, the physical file format normally used with EXPRESS. Given an EXPRESS model, and data for a particular device, STEP 10303.21 provides the actual exchange format for data.

Figure 1 diagrams the architecture for the work of the AI-ESTATE Committee, plus icons which represent the IFM and other formal data models (the data model icons are not included in the approved AI-ESTATE architecture diagram.) In addition to the IFM data model, a fault tree model has been approved for distribution beyond SCC-20, and initial work has been done to define a standard SERVICES interface for both an IFM based reasoner, and a fault tree reasoner. This ongoing work is indicated in the figure.

The Information Flow Model is only one small component of the standards which will be required to achieve an open system architecture for system test and diagnosis, and for testability analysis. There are other standards being developed by the AI-

CH3148-4/92/0000-0049 $1.00©1992 IEEE

ESTATE Committee, and the work of the AI-ESTATE Committee will hopefully fit within the larger framework of the Ada Based Environment for Test (ABET) effort. In this sense, the AI-ESTATE work can be thought of generally as the Test Strategy Layer of ABET.

The AI-ESTATE Committee is eager to receive comments regarding the IFM model, and for expanded participation in all its efforts. Interested persons can contact Dr. Leonard Haynes, President, Intelligent Automation, Incorporated, 1370 Piccard Drive, Suite 210, Rockville, MD 20850 or call at (301) 990-2407.

2. Benefits

2.1 Benefits of the IFM Model

Adoption of standards for information models will allow portability of models between tools. It would allow dependency model based tools to generate output which could be used directly by electronic test equipment to automatically perform tests. It would facilitate feedback of field experience to be used to update the models to provide improved diagnosis. It would encourage development of tools for automatically generating information models because a standard model format would increase the marketability of such a tool (it would be usable by many dependency model based tools.) Adoption of the proposed class of standards would also facilitate integration of "Interactive Electronic Technical Manuals" with the test strategy optimization tools since the interface to the tools would be standardized. Today, an information model might be developed during initial design to facilitate concurrent engineering analysis, but there is little likelihood that that same model would be usable on field service diagnosis equipment, again because of the incompatibility of the models used in the various available tools.

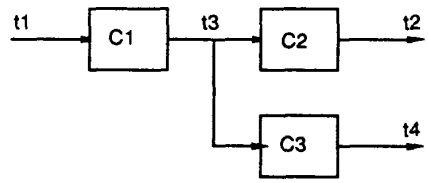
For Concurrent Engineering to be effective, design data must be sharable and usable by design engineers, by test and maintenance system developers, by the logistics support community, by production personnel, and by training and related personnel. The standards we propose will help facilitate this sharing because it will allow dependency related information to be portable over a wide range of tools for both testability and maintainability analysis, and then will allow the same models to be used in the field for optimization of diagnosis.

2.2 Benefits of a standard interface for SERVICES

The development of standards appears to be most effective when individual standards aggregate together into an overall superstructure which relates each individual standard to the intent of the entire system. Layered architectures have been very effective in providing that superstructure and there are several examples of a family of standards which fit within a layered, hierarchical architecture. The best example of a layered architecture is the International Standards Organization (ISO) Open System Interconnection (OSI) architecture. OSI is composed of seven layers, each of which defines specific services and protocols provided at that level. Each layer provides a set of standard services to the structure above, independent of the particular implementation of that layer, and uses a standard set of services provided by the structure below, again independent of the particular implementation of that layer. Compliant implementations are plug compatible, with many proven advantages. In the sense that an Information Flow Reasoner responds to commands from above, such as "run the next test" and uses the services of software and automatic test equipment at lower layers, its analogy with the OSI model is clear. Our goal is to achieve the same portability and interchangability which has been achieved by the OSI model. Work is progressing on the services and protocols for two classes of reasoners, however this work is just beginning.

3. Brief Example of a Dependency Model

In dependency models, each of the available tests which can be performed is identified, and for each of those tests, the information learned from that test in terms of components known to be good as a result of a good test outcome, or components still suspected of being faulty as a result of a bad test outcome are tabulated. The t1, t2, and t3 shown below are tests performed on units C1, C2, and C3. In the figure, we see that t4 depends on component C3 and test t3. Also, test t3 depends on C1 and t1. These are referred to as first order dependencies. By inference, t4 also depends on c1 and t1. This is an example of a higher order dependency.



The concept of dependency modeling is very powerful, partly because it can be applied hierarchically. At one level, components can be single integrated circuits, switches, or similar individual components. At the next level, components can be larger aggregates such as a multiplexer, power supply, floating point multiplier circuit, etc. At yet the next level of aggregation, the same exact modelling concept can be used to model subsystems, and assess the testability of the system at that level. References 1 and 2 given much more detail on dependency models, and on the tools which use these models.

It should be mentioned that dependency models are not limited to electronic equipment. Dependency-based testability analysis has been used in many domains with equal success.

The term "dependency" is often used in the context of "fault dictionary" approaches to electronic system diagnosis, and this has caused considerable problems in describing the standard we are proposing to those familiar with fault dictionary based tools. There are several very fundamental differences between a dependency model as described in this paper, and dependency as used in fault dictionary approaches. In fault dictionary approaches, an extensive sequence of test vectors are applied to a system under test, and a set of system outputs are simultaneously monitored. In the presence of faults, some of the outputs will be incorrect at various points in time, and the resulting fault vector is analyzed after the entire test sequence is applied to the device under test. In general, for each failed output, analysis techniques look for which components affect the failed outputs, and hence for those components whose failure could cause the output in question to fail. The dependency in this context is between system output and components, independent of which test is being executed. There is only one "test" and it tests all components simultaneously. There is only one set of outputs and it remains the same through the entire analysis process. For dependency models as we use the term, there are many tests, each with its own distinct set of components which are tested by that test. We assume that for each test, what is learned from each of the distinct outcomes of that test is known a priori and encoded in the model. The primary function of a dependency model is to select an optimal next test in terms of a set of criteria. In a fault dictionary approach, there is only one test sequence so even at the most basic level, the two approaches are entirely different. In order to meet the objective of determination of the optimal sequence of tests to perform, special inference types, groups, weights, etc also need to be considered. When this additional information is added to dependency information, we call the result an information flow model.

4. The Information Flow Model

The IFM is quite compact and elegant, with only 29 ENTITIES. The model is included below. Only the consistency rules have been excluded to meet the paper page limit, and the format of normal EXPRESS has been compressed to save space.

```
SCHEMA IFM_model;
  TYPE status = ENUMERATION OF (bad, good);
  TYPE yes_or_no = ENUMERATION OF (yes, no);
  TYPE reference_point_type = ENUMERATION OF
                (input, output, inout, internal_location);
  TYPE operator_type = ENUMERATION OF
                (and_op, or_op, not_op);
  TYPE relation_type = ENUMERATION OF
                (before, equals, after);
  TYPE multiple_failure_group_id = INTEGER;
  TYPE arbitrary_group_id = INTEGER;
  TYPE unit_id = INTEGER;
  TYPE unit_type_id = INTEGER;
  TYPE aspect_id = INTEGER;
  TYPE encapsulated_test_id = INTEGER;
  TYPE test_id = INTEGER;
      (* identifies either an encapsulated test or a unit test *)
  TYPE setup_operation_id = INTEGER;
  TYPE access_operation_id = INTEGER;
  TYPE test_outcome_id = INTEGER;
  TYPE reference_point_id = INTEGER;
  TYPE term_id = INTEGER;

ENTITY model;
    header_information:          header;
    unit_model:                  unit;
    type_library:                SET [0:?] OF unit_type;
    test_library:                SET [0:?] OF unit_test;
    group_information:           groups;
    test_related_libraries:      libraries;
END_ENTITY;

ENTITY header;
    preparer:                    STRING;
    organization:                OPTIONAL STRING;
    description:                 OPTIONAL STRING;
    model_requirements:          OPTIONAL STRING;
    model_creation_date:         STRING;
    last_modified_date:          STRING;
    classification:              STRING;
    replacement_time_units:      OPTIONAL STRING;
    replacement_cost_units:      OPTIONAL STRING;
    test_time_units:             OPTIONAL STRING;
    test_cost_units:             OPTIONAL STRING;
    failure_units:               OPTIONAL STRING;
    retest_ok_units:             OPTIONAL STRING;
    setup_time_units:            OPTIONAL STRING;
    setup_cost_units:            OPTIONAL STRING;
    access_time_units:           OPTIONAL STRING;
    access_cost_units:           OPTIONAL STRING;
END_ENTITY;

ENTITY groups;
    arbitrary_groups:      SET [0:?] OF arbitrary_group;
    mult_failure_groups:   SET [0:?] OF mult_failure_group;
END_ENTITY;

ENTITY libraries;
    setup_operations:            SET [0:?] OF setup_operation;
    access_operations:           SET [0:?] OF access_operation;
    reference_points:            SET [0:?] OF reference_point;
    terms:                       SET [0:?] OF  term;
END_ENTITY;

ENTITY named_element
  ABSTRACT SUPERTYPE ;
    name:                        STRING;
    description:                 OPTIONAL STRING;
    id:                          INTEGER;
UNIQUE id;
END_ENTITY;

ENTITY failable_element
  ABSTRACT SUPERTYPE ;
    replacement_level:           OPTIONAL STRING;
    replacement_time:            OPTIONAL REAL;
    replacement_cost:            OPTIONAL REAL;
    replacement_skill_level:     OPTIONAL INTEGER;
    criticality:                 OPTIONAL INTEGER;
    failure_rate:                OPTIONAL REAL;
    retest_ok:                   OPTIONAL REAL;
END_ENTITY;

ENTITY unit
    SUBTYPE OF(named_element, failable_element);
    has_type:                    STRING;
    of_type:                     unit_type_id;
    has_parts:                   SET [0:?] OF unit;
    has_aspects:                 SET [0:?] OF aspect;
END_ENTITY;

ENTITY unit_type
    SUBTYPE OF(named_element, failable_element);
END_ENTITY;

ENTITY aspect
    SUBTYPE OF(named_element);
    failure_rate:                OPTIONAL REAL;
    relative_likelihood:         OPTIONAL REAL;
END_ENTITY;

ENTITY arbitrary_group
    SUBTYPE OF(named_element);
    member_aspects:              SET [0:?] OF aspect_id;
    member_units:                SET [0:?] OF unit_id;
END_ENTITY;

ENTITY multiple_failure_group
    SUBTYPE OF(named_element, failable_element);
    member_aspects:              SET [0:?] OF aspect_id;
    member_units:                SET [0:?] OF unit_id;
END_ENTITY;

ENTITY unit_test
    SUBTYPE OF(named_element);
    has_unit_tests:              SET [0:?] OF unit_test;
    has_encapsulated_tests:      SET [0:?] OF encapsulated_test;
END_ENTITY;

ENTITY encapsulated_test
    SUBTYPE OF (named_element);
    is_measurable:               yes_or_no;
    time_to_perform:             OPTIONAL REAL;
    cost_to_perform:             OPTIONAL REAL;
    req_setup_operations:        SET[0:?] OF setup_operation_id;
    reqs_access_ops:             SET[0:?] OF access_operation_id;
    reqs_technicians:            SET [0:?] OF  technician;
    reqs_equipment:              SET [0:?] OF STRING;
    has_stimulation_pts:         SET [0:?] OF reference_point_id;
    has_info_pts:                SET [0:?] OF reference_point_id;
    has_outcomes:                SET [1:?] OF test_outcome;
    has_rel_to_other_tests:      SET [0:?] OF test_relation;
END_ENTITY;

ENTITY operation_element
  ABSTRACT SUPERTYPE;
    cost:                        OPTIONAL REAL;
    time:                        OPTIONAL REAL;
END_ENTITY;

ENTITY setup_operation
    SUBTYPE OF(named_element, operation_element);
END_ENTITY;

ENTITY access_operation
    SUBTYPE OF(named_element, operation_element);
END_ENTITY;

ENTITY technician;
```

```
    requires_skill_level:          INTEGER;
    requires_clearance:            INTEGER;
    requires_authority:            INTEGER;
END_ENTITY;

ENTITY reference_point
     SUBTYPE OF(named_element);
    has_type:      OPTIONAL reference_point_type;
    location:      SET [0:?] OF reference_point_location;
END_ENTITY;

ENTITY reference_point_location;
    on_unit:                       unit_id;
    at_label:                      OPTIONAL STRING;
END_ENTITY;

ENTITY test_outcome
    SUBTYPE OF(named_element);
    prob_false_outcome:            OPTIONAL REAL;
    base_confidence:               OPTIONAL REAL;
    implies_good_list:             OPTIONAL expression;
    implies_bad_list:              OPTIONAL expression;
    exclude_for_safety:            SET [0:?] OF test_id;
END_ENTITY;

ENTITY expression;
    isa_term:                      OPTIONAL term_id;
    isa_parenthesized_expr:        OPTIONAL expression;
    isa_tuple:                     OPTIONAL tuple;
  WHERE
    only_one_entry:
    EXISTS(isa_term) XOR
    EXISTS(isa_parenthesized_expr)
    XOR EXISTS(isa_tuple);
END_ENTITY;

ENTITY tuple;
    expression1:                   expression;
    operator:                      operator_type;
    expression2:                   OPTIONAL expression;
WHERE
    not_is_unary_operator:
    (EXISTS(expression2) AND operator <> not_op) OR
    (operator = not_op) AND (NOT (EXISTS (expression2))));
END_ENTITY;

ENTITY term
    SUBTYPE OF(named_element);
    is_test_status:                OPTIONAL test_status;
    is_aspect_status:              OPTIONAL aspect_status;
    is_unit_status:                OPTIONAL unit_status;
    is_arb_group_status:           OPTIONAL
                                     arb_group_status;
    is_mult_failure_group_status:  OPTIONAL
                                     mult_failure_group_status;
WHERE
    only_one_entry:
    EXISTS(is_test_status) XOR
    EXISTS(is_aspect_status) XOR
    EXISTS(is_unit_status) XOR
    EXISTS(is_arbitrary_group_status)
    XOR EXISTS(is_multiple_failure_group_status);
END_ENTITY;

ENTITY test_status;
    for_test:                      encapsulated_test_id;
    certainty:                     OPTIONAL REAL;
    has_outcome:                   test_outcome_id;
END_ENTITY;

ENTITY aspect_status;
    for_aspect:                    aspect_id;
    certainty:                     OPTIONAL REAL;
    has_status:                    status;
END_ENTITY;

ENTITY unit_status;
    for_unit:                      unit_id;
    certainty:                     OPTIONAL REAL;
    has_status:                    status;
END_ENTITY;

ENTITY multiple_failure_group_status;
    for_mult_failure_group:        multiple_failure_group_id;
    certainty:                     OPTIONAL REAL;
    has_status:                    status;
END_ENTITY;

ENTITY arbitrary_group_status;
    for_arbitrary_group:           arbitrary_group_id;
    certainty:                     OPTIONAL REAL;
    has_status:                    status;
END_ENTITY;

ENTITY test_relation;
    occurs:                        relation_type;
    related_tests:                 SET [1:?] OF test_id;
END_ENTITY;
END_SCHEMA;
```

5. Discussion of the IFM Model

A few key points related to the model are included below:

UNIT

The most essential element of the IFM is the Unit. Since Unit is a Named Element, it includes attributes such as name, a textual description of the equipment, and a unique id (see ENTITY Named_Element in the IFM). The structure of the equipment being modelled is expressed recursively: Units can be composed of parts which are themselves unique units. The format of the has_parts attribute for Units forces the hierarchy of Unit entities to be unique, exclusive and untangled. A Unit which is a part of one particular Unit entity will never appear as part of another Unit entity.

Each unit entity is associated with a Unit Type entity. Type information for units in the IFM is stored in the library of Unit Types. General information common to all units of the particular type relating to replacement of the unit and statistics on failure rates is included in the Unit Type entity (both Unit and Unit Type entities are subtypes of the Failable Element entity). Multiple Unit entities may reference the same Unit Type entity. Sometimes it may be necessary to override the Unit Type information which is automatically inherited by a Unit entity. For this purpose optional information about replacement level, replacement cost, replacement skill level, criticality, failure rate and Retest OK rate may be specified for a Unit entity.

UNIT TYPE

A library of Unit Type information is part of the Information Flow Model. Type information includes attributes related to failure such as the maintenance level at which it is replaceable, time, cost and skill level required to replace units of this type and the criticality of this type of unit. Additional type information pertains to unit failure statistics and includes attributes such as failure rate, and Retest OK rate of units of this type. This information is inherited by Units of this type unless the Unit entity specifically includes values for the replacement and failure statistic related attributes which are common to both entities.

ASPECT

Units are modelled by dividing their failure modes from a test perspective into aspects. An example of an aspect would be a counter/shift register combination which can function as a counter or shift register. That unit can fail by not being able to count, or by not being able to shift, or both. Certain tests will test the counter function and others will test the shift register function and still other

tests may test both so the model must allow modelling of the single physical unit as two separate "virtual units." An Aspect is a subtype of the Named Element entity and has additional attributes for a failure rate and a relative likelihood. The failure rate is a book value for this aspect. The relative likelihood indicates how likely this unit is to fail in this failure mode with respect to other failure modes for the unit.

ARBITRARY GROUP

Entity Arbitrary Group provides a means of grouping Unit entities in the IFM. Note that while units and aspects of the Unit hierarchy are distinct entities, the concept of an arbitrary group in the IFM is specifically designed to provide an alternative partitioning of the Units and Aspects in the Unit hierarchy. Having the has_parts attribute contain a set of unit_ids instead of Units, forces the Arbitrary Group to reference Unit entities previously defined in the Unit hierarchy; the same holds true for Aspects.

The Arbitrary Group entity includes attributes which identify its name, a description of the group, a unique identifier number and the list of Aspect and Unit entities which define the group. Dependencies in the model can be established between test outcomes and arbitrary groups.

MULTIPLE FAILURE GROUP

Entity Multiple Failure Group provides another means of grouping Unit entities. In this case, however, the grouping is not purely arbitrary but indicates that the Unit and Aspect entity group members tend to fail together within the system being modelled. By grouping the set of Unit and Aspect failures together, they can be treated as a single failure in the model. Attributes of the Multiple Failure Group include the group name, a description of the group, a unique identifier number (used in establishing dependencies within test outcomes) and a list of Unit and Aspect entities which comprise the group. Also included in the Multiple Failure Group entity are attributes for describing failure statistics and replacement information for the group.

UNIT TEST

The third major category of Entities in this model are those related to specific tests. The basic premise on which the IFM is based is that tests are encapsulated tests. This means that a test is identified primarily by a unique identifier and that the details of the test itself are contained in other models which are referenced through a symbolic name, but not otherwise included in the IFM.

As far as any inference mechanism which uses the model is concerned, all tests are encapsulated tests. Entity Unit Test allows encapsulated tests to be identified, for naming purposes, with higher level aggregations. This is solely for convenience and the information in entity Unit Test will not be used during inference. Entity Unit Test allows tests to be identified as a subtype of Named Element, and any Unit Test can be composed of other Unit Tests and/or Encapsulated Tests.

ENCAPSULATED TEST

An encapsulated test is an atomic element. It is modelled as a subtype of the Named Element entity, along with optional information regarding the time and cost to perform the test.

Reference points have been added to the IFM so that the co-locality of various tests can be identified. Each encapsulated test may optionally have a set of stimulation (input) points and a set of information gathering (output) points. The Reference Point may have a type identified (one of input, output, inout, or internal_location) and optionally have a set of Locations. Each Reference Point Location optionally identifies a particular unit on which the point is located and a textual description further labelling the reference point.

An encapsulated test includes entities which define access time and cost, and setup time and cost. The model allows any number of access operation and setup operation attributes to be specified. The physical interpretation of the sets is that several operations may be feasible to provide access to a specific test. Some may also provide access to other tests so there is an optimization issue. The assumption is that any of the operations identified in the set of access_operations will provide access for the required test. The same is true for setup_operations.

Encapsulated Tests also include a set of required technicians and a set of required test equipment. This information is required by the inference mechanism in order for it to be aware of what resources are required to execute what tests. In the event that some resources are not available, the inference mechanism can still proceed with the diagnosis using the tests for which the required resources are available. An attribute identifying whether a test is a built-in test or not has not been included in the IFM since this information is derivable from the model: if a test requires neither technicians nor equipment it must be a built-in test.

The attribute has_relation_to_other_tests provides a means of expressing how a test relates temporally to other tests. This attribute contains a set of Test Relation entities which specify the set of related tests and the temporal relationship to those tests. The tests identified in the list of related tests may either refer to specific encapsulated tests or to unit tests.

The kernel relationships for the Encapsulated Test is the set of two or more Test Outcomes. It is in this relationship that the dependency information is modelled. This will be explained under the entity Test Outcome.

TEST_OUTCOME

The IFM does not restrict test outcomes to GOOD or BAD. It allows any number of test outcomes, each identified as a Named Element, along with attributes identifying the probability of this outcome occurring falsely and the base confidence in this particular outcome.

The essence of a dependency model is "what is learned from each possible outcome of each test." IFM models this learned information in the most general way, so that redundant systems can be modelled as well as conventional systems. The two attributes which capture the dependency information are requires_good_list and requires_bad_list. The interpretation of the good list is that the resulting expression defines what is learned to be good as a result of that particular test outcome. The interpretation of the bad list is that the resulting expression defines what is learned to be bad as a result of that particular test outcome.

Test outcomes are formulated in logical expressions in the good list and bad list attributes. The entities Expression, Tuple and Term implement the logical expression, shown below in BNF form, in the EXPRESS language:

```
expression := term | ( expression ) |tuple
tuple := unary_operator expression | expression
binary_operator expression
term := test_status | aspect_status | unit_status |
multiple_failure_group_status |
        arbitrary_group_status
unary_operator := NOT
binary_operator := OR | AND
```

where AND has precedence over OR, NOT has precedence over AND and parentheses have the highest precedence. The three attributes of the entity Expression capture the three possible forms of a logical expression in this syntax: an expression is either a Term, a parenthesized Expression or a Tuple.

The notion of a "symptom" is not included in the model as a unique type of Test Outcome. There appears to be great differences between the way different tools handle symptoms, and even in the meaning of the word. The model attempts to be as general as

feasible, so we have decided not to distinguish symptoms as a separate category because the data which is generally used to identify symptoms is already available in the model. Tests include the information as to the number and skill levels of technicians required to perform a test, the time and cost of each test, and the test equipment required to perform a test. If a particular tool (WSTA for example) defines a Symptom as the result of a "cheap test" by some definition of "cheap" then this information is available and "symptoms" can be distinguished from other tests by these measures. Information available at no cost will be modelled as a test regardless of other information, symptom not withstanding.

ASPECT STATUS, UNIT STATUS,
MULTIPLE FAILURE GROUP STATUS,
ARBITRARY GROUP STATUS

All of these entities allow specification of the particular aspect, unit or group as being GOOD or BAD, and the certainty with which this information is known.

TEST STATUS

This entity allows the results of tests to be included in the logical expression described above, permitting first order dependencies to be described in the IFM. It is not clear at this time whether a certainty value is required.

TECHNICIAN
TEST EQUIPMENT

Tests shall be further modelled to include the set of Test Equipment and the set of Technicians required to perform the test. For entity Technician, the number and skill levels of the technicians required to perform a test is included.

The equipment required to perform a test is included in the model so that in the event specific equipment is not available, the system can proceed with diagnosis by recommending other tests which can be performed. Links to other models for description of test equipment can be provided external to the IFM.

6. Services Interface

As described in section 2.2 of this paper, one of the goals of the AI-ESTATE Committee of SCC-20 is to develop a set of standards which support plug compatibility between components of an AI-ESTATE compliant system. In order to achieve this goal, it is essential to define the SERVICES which components of the system provide to other components. The following discussion deals with the SERVICES interface to an IFM-based reasoner which will provide test strategy services to the levels above, and exploit the test equipment at the lower levels to actually perform tests.

The SERVICES interface to an IFM-based reasoner must not restrict the manner in which services are provided, nor can it prevent components from providing non-standard capabilities. Our standards should allow competitive advantage and flexibility to be creative, yet conforming components must still be plug compatible. We believe these goals can be achieved, and our initial approach is discussed below.

There are several paradigms possible to specify SERVICES, and the AI-ESTATE Committee is still evaluating possibilities. No formal decisions have been made, even as to the basic paradigm for the specification of SERVICES. There has also been no formal discussion as to what SERVICES should be included in the standard. There has, in fact, not been a formal vote even as to the need for a SERVICES interface so this work is in its infancy. With these strong caviots, the following paragraphs describe the **authors'** current thinking on these issues. Specifically, the following is not approved by the SCC-20 or by the AI-ESTATE Committee.

6.1 Services Paradigm

Our current view of the services interface is that an IFM-based reasoner provides information regarding optimal test strategy, and can be viewed as an abstract knowledge base. We can represent the required ENTITIES and their relationships in EXPRESS just as we described the IFM model itself in EXPRESS. EXPRESS is implementation independent. Any language can be used to store the actual data, and any query language can be used to access the data. In order to achieve plug compatibility, some decisions must be made as to the specific command formats, but this is a trivial problem compared to the issue of what information must be provided to an IFM reasoner, and what results are produced. SQL, for example, could be adopted as the standard method for accessing the IFM-reasoner.

Using this paradigm, all requests for service are equivalent to reads and writes into the "knowledge base." The writes to this data base are used to pass parameters to the reasoner, and to establish the criterion for a given session. As an example, assume it is desired to set the priority for diagnosis to emphasize the time required to diagnose a problem. Setting this priority would be equivalent to writing a value into the priority values defined below. An SQL command could be used to effect this write.

```
ENTITY reasoner_parameter_data;
    controller_model_#:                 STRING;
    maintenance_level:                  STRING;
    max_cost_for_callout:               cost;
    max_amb_group_size_for_callout:     INTEGER;
    max_repl_time_for_callout:          time;
    min_certainty_for_callout:          percent;
    priority_on_repair_cost:            REAL;
    priority_on_repair_time:            REAL;
    priority_on_diagnosis_cost:         REAL;
    priority_on_diagnosis_time:         REAL;
    priority_on_total_time:             REAL;
    priority_on_total_cost:             REAL;
    priority_on_repair_accuracy:        REAL;
    non_standard_pars:                  non-standard_d;
END_ENTITY;
```

A command to compute the next test to execute would be equivalent to a read from the value next_test in the following entity structure. The reasoner would then compute the value which was returned as a result of the query.

```
ENTITY reasoner_diagnosis_data;
    predicted_time_to_diag:     time;
    predicted_time_to_repair:   time;
    predicted_cost_to_diag:     cost;
    predicted_cost_to_repair:   cost;
    current_amb_group:          current_amb_group_d;
    unit_type:                  unit_type_d;
    unit_serial_#:              unit_serial_#_d;
    current_best_callout:       callout_d;
    explaination:               STRING;
    last_test:                  test_id;
    last_test_result:           test_result_d;
    next_test:                  test_id;
    test_result:                test_result_d;
    non-standard_data:          non-standard_d;
END_ENTITY;
```

It can be seen that the ENTITY reasoner_diagnostic_data includes non-standard_data as an attribute. This non-standard data can represent additional functionality, or it can represent additional information provided in response to the standard functionality beyond the standardized response data. If the non-standard features are in the form of additional information beyond the required answers, then these non-standard capabilities do not even result in a loss of plug compatibility. As an example, a reasoner computes and then provides the test_id of the **next test to execute** in response to a "read" from the next_test attribute. This is the standard response. A specific reasoner uses non-standard

attributes to provide a set of other tests which could be executed next with close to equal efficiency. An application using this capability would read these values and might exploit one of the alternative choices. If the reasoner were replaced with a compliant reasoner which did not provide that additional information, then when the reasoner tried to access the alternative choices, it would receive "null" responses which it would then ignore, hence plug compatibility has not been lost.

The current "strawman" list of "services" is shown below although we must emphasize again that this list is incomplete, has not been approved by any Committee, and is to be interpreted only as the opinion of the authors.

6.2 Service Interface ENTITIES

```
ENTITY reasoner_admin_data;
    reasoner_id:                STRING;
    user_list:                  SET[0:?] OF user_list_d;
    current_user_logon:         user_list_d;
    current_time:               current_time_d;
    current_date:               current_date_d;
    replacement_level:          STRING;
    non_standard_admin_data:    non_standard_d;
END_ENTITY;

ENTITY user_list_d;
    user_name:                  STRING;
    user_id:                    STRING;
    user_password:              STRING;
    user_skill_level:           INTEGER;
    clearance:                  INTEGER;
    authority:                  INTEGER;
    non_standard_user_data:     non_standard_d;
END_ENTITY;

ENTITY reasoner_id;
    reasoner_name:              STRING;
    reasoner_version_number:    INTEGER;
    reasoner_serial_number:     STRING;
    reasoner_contact_name:      STRING;
    reasoner_contact_phone_#    STRING;
END_ENTITY

ENTITY user_admin_data;
    user_logon_data:            user_list_d:
END_ENTITY;

ENTITY reasoner_status_data;
    status:                     status_d;
    current_task:               current_task_d;
    expected_time_to_complete:  time;
    time_expended_current_task: time;
    time_expended_this_session: time;
    time_since_last_backup:     time;
    allowable_user_options:     SET[1:?] OF command_id;
    non_standard_status_data:   non_standard_d;
END_ENTITY;

ENTITY status_d;
    status:  ENUMERATION_OF(executing, waiting for task,
                 error requires restart, no model loaded);
END_ENTITY

ENTITY current_task_d;
    task_id;                    INTEGER;
    task_name:                  STRING;
    task_description:           STRING;
END_ENTITY

ENTITY user_resource_data;
    tech_available:             tech_available_d;
    equipment_available:        SET[1:?] OF equipment_id;
    parts_inventory:            parts_inventory_d;
    equipment_unavailable:      SET[1:?] OF equipment_id;
    non_standard_resource_data: non_standard_d;
END_ENTITY;

ENTITY technician_available_d;
    available_technician:       SET[1:?] technician;
END_ENTITY

ENTITY reasoner_archive_data;
    session_archive:            SET[0:?] OF archive_portion;
END_ENTITY;

ENTITY archive_portion;
    archive_time:               time;
    archive_event:              archive_event_d;
    archive_text:               STRING;
    archive_annotation:         STRING;
    non_standard_archive_data:  non_standard_d;
END_ENTITY;

ENTITY archive_event_d;
    test_event:                 OPTIONAL test_id;
    callout_event:              OPTIONAL unit_id;
    logon_event:                OPTIONAL user_id:
    command_event:              OPTIONAL command_id:
END_ENTITY

ENTITY reasoner_mdc_data;
    session_mdc:                session_mdc_d;
    annotation:                 annotation_d;
    mdc_since_last_sent:        mdc_since_last_sent_d;
END_ENTITY;

ENTITYsession_mdc_d;
    has_session_number:         session_no;
    identifies_reasoner_id;     reasoner_id;
    identifies_IFM_model_id;    model_id;
    identifies_controller_id;   controller_id;
    has_mdc_data:               SET[0:?] OF mdc_unit_data;
END_ENTITY;

ENTITY mdc_unit_data;
    specifies_unit_id_to_repair:        string;
    specifies_unit_serial_no_to_repair; string;
    specifies_unit_type_to_repair:      string;
    specifies_technician_id:            string;
    has_unit_test_data:         SET[0:?] OF mdc_test_data;
    initiation_time:                    time;
    total_isolation_time:               time;
    total_repair_time:                  time;
    units_replaced:                     SET[0:?] OF unit_id;
    final_unit_status:                  status;
    non_standard_mdc_data:              non_standard_d;
END_ENTITY;

ENTITY mdc_data;
    reasoner_id:                string;
    IFM_model_id:               string;
    unit_id:                    string;
    unit_type:                  string;
    technician_id:              string;
    session_#:                  integer;
    test_data:                  set [0:?] of mdc_test_data
    initiation_time:            time;
    total_isolation_time:       time;
    total_repair_time:          time;
    units_replaced:             set[0:?] of unit_id;
    final_unit_status:          status;
END_ENTITY;

ENTITY mdc_test_data;
    test_name:                  string;
    test_id:                    string;
    result:                     outcome_id;
    callout:                    set [0:?] of unit_callout;
    action_taken:               set [0:?] of unit_repair;
```

```
    note:                          string;
    unit_status_after_action:      status;
    action_time:                   time;
    action_cost:                   cost;
    non_standard_mdc_test_data:    non_standard_d;
END_ENTITY;

ENTITY unit_repair;
    component_id:                  string;
    component_type:                string;
    old_component_serial_no:       string;
    new_component_serial_no:       string;
    replaced:                      yes_or_no;
    other_repair_action:           string;
    repair_time:                   time;
    repair_cost:                   cost;
    non_standard_repair_data:      non_standard_d;
END_ENTITY;

ENTITY unit_callout;
    unit_id:                       string;
    unit_type                      string;
END_ENTITY;

ENTITY reasoner_parameter_data;
    controller_model_#:            controller_model_#_d;
    maintenance_level:             STRING;
    max_cost_for_callout:          cost;
    max_amb_group_size_for_callout:    INTEGER;
    max_repl_time_for_callout:     time;
    min_certainty_for_callout:     min_certainty_for_callout_d;
    priority_on_repair_cost:       priority;
    priority_on_repair_time:       priority;
    priority_on_diagnosis_cost:    priority;
    priority_on_diagnosis_time:    priority;
    priority_on_total_time:        priority;
    priority_on_total_cost:        priority;
    priority_on_repair_accuracy:   priority;
    non_standard_parameter_data:   non_standard_d;
END_ENTITY;

ENTITY reasoner_diagnosis_data;
    predicted_time_to_diag:        time;
    predicted_time_to_repair:      time;
    predicted_cost_to_diag:        cost;
    predicted_cost_to_repair:      cost;
    current_ambituity_group:       current_ambituity_group_d;
    unit_type:                     unit_type_d;
    unit_serial_#:                 unit_serial_#_d;
    current_best_callout:          callout_d;
    explaination:                  STRING;
    last_test:                     test_id;
    last_test_result:              encapsulated_test_outcome;
    next_test:                     test_id;
    non_standard_diag_data:        non_standard_d;
END_ENTITY;

ENTITY callout_data;
    unit:                          unit_name;
    certainty:                     REAL;
    statistical_prob_of_failure:   REAL;
    expected_time_to_repair:       time;
    expected_cost_to_repair:       cost;
    non_standard_callout_data:     non_standard_d;
END_ENTITY

ENTITY user_diagnosis_data;
    run_test:                      encapsulated_test_id;
    manual_test_result:            manual_test_result_d;
END_ENTITY;

ENTITY ATE_diagnosis_data;
    run_test:                      encapsulated_test_id;
    test_result:                   encapsulated_test_outcome;
END_ENTITY
```

```
ENTITY non_standard_d;
    non_std_item_name:             STRING;
    any_strings:                   SET[0:?] of STRING;
    any_reals:                     SET[0:?] of REAL;
    any_integers:                  SET[0:?] of INTEGER;
END_ENTITY;
```

7. Conclusions

The Information Flow Model and the Services model are two key elements required to achieve an Open Architecture for dependency-based test strategy tools. Similar models have also been developed for fault tree-based reasoners. Other work will attempt to develop standard models and services for other types of reasoners including rule-based reasoners, neural-net reasoners, fault dictionary-based reasoners, etc. Most test strategy reasoners will provide similar services, and the union of these services will become the general interface for test strategy reasoners, with conformance classes defined for the individual reasoner types.

Hopefully, products which conform to the standards developed by the AI-ESTATE Committee will then be plug compatible test strategy components within the larger Ada Based Environment for Test.

8. Acknowledgements

The research which provides the foundation for the model development described in this paper is funded by the Wright Laboratory, Air Force Systems Command, Wright-Patterson AFB, Ohio. Technical Direction is provided by Mr. James Poindexter, (MRL/C/MTR). Mr. Poindexter has made many important suggestions regarding our work, and his contributions are in part responsible for the project's success to date.

References

1. Simpson, W., and Sheppard, J., "System Complexity and Integrated Diagnostics," IEEE Design and Test of Computers, Volume 8, number 3, Sept. 91, pgs 16-30.

2. Sheppard, J., and Simpson, W., "A Mathematical Model for Integrated Diagnostics," IEEE Design and Test of Computers, Volume 8, number 4, Dec. 91, pgs 25-38.

3. Keiner, W., "A Navy Approach to Integrated Diagnostics," AutoTestCon 90, San Antonio, TX Sept. 1990.

4. "STAMP User's Manual," DETEX Systems Inc., Orange, CA.

5. Cantone, R., and Caserta, P., "Evaluating the Economic Impact of an Expert Fault Diagnosis System: The ICAT Experience," Proc of the 3rd IEEE Symposium on Intelligent Control, Los Alamitos, CA, 1988.

Figure 1. AI-ESTATE Architectural Concept