

Application Scenarios For AI-ESTATE Services

Richard J. Maguire
IET Intelligent Electronics Ltd.
67 S. Bedford St., Suite 400W
Burlington, MA 01803
rickm@ietusa.com

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

Abstract—One of the principal objectives of the P1232 AI-ESTATE standardization project is the development of a standard set of services to be provided by a diagnostic reasoner within a test environment. Recently, considerable progress has been made defining an initial set of services for P1232.2 AI-ESTATE Service Specification. In this paper, we provide several possible scenarios under which such services may be used within a test environment. We describe these scenarios using state diagrams and then demonstrate how one might use the defined set of services to provide the functionality required by these scenarios.

I. INTRODUCTION

The P1232 AI-ESTATE [1] subcommittee of SCC20 is in the process of developing its P1232.2 Service Specification. The Service Specification will define a set of services to be provided by a reasoner that has been embedded within a test environment of some kind. This test environment may be a performance monitoring system, an embedded test process, an automatic or semi-automatic test system, a manual test system, or a field maintenance environment.

The purpose of this paper is to discuss several potential application scenarios for test systems that might use the AI-ESTATE set of services. It is recognized that the AI-ESTATE Service Specification is in the formative stages; therefore, services indicated in this document may differ significantly from the services currently found in P1232.2 [2]. It is hoped that these differences will be illuminating in that they may point out requirements for additional services or requirements on the final architecture of AI-ESTATE and its supporting documents. Finally, several issues that arose from generating these scenarios are discussed, pointing out possible deficiencies in the current draft of the standard.

II. USAGE SCENARIOS

To facilitate the discussion on AI-ESTATE usage, we will use the set of state diagrams that were presented and

discussed at an earlier meeting of the AI-ESTATE subcommittee. These diagrams are intended to illustrate the ways in which an AI-ESTATE conformant reasoner might be used in a test environment. Since these diagrams were developed prior to any drafts of the P1232.2 Service Specification, direct correlation to the services defined in that document should not be expected. We will provide more direct mapping to the services as we expand the discussion in this paper.

We will focus on four state diagrams corresponding to four different scenarios. These scenarios correspond to the following:

1. A generic diagnostic process with an intelligent diagnostic controller
2. Diagnosis with an operator overriding a reasoner test choice
3. Reasoner justification of a test choice
4. Determination of whether a conflict has occurred in the inference process

For each scenario, we will present the state diagram and explain the diagram in the context of an AI-ESTATE conformant reasoner. We will also attempt to map the transitions in the diagram to the “current” set of services as defined in P1232.2 Draft 2.0 [2]. Note that several of the services defined in the draft may have been modified or deleted, and several new services may have been added.

III. A GENERAL DIAGNOSTIC PROCESS

To begin the discussion of using the P1232.2 Service Specification, consider the state diagram in Figure 1. 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 prudent for a user of the reasoner to also maintain knowledge of the current state. This will define the set of services available to the user at that point in the diagnostic process.

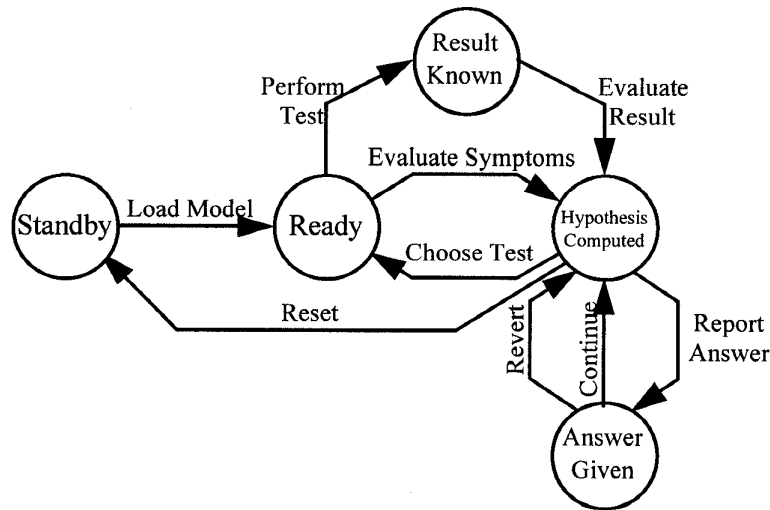


Figure 1. State diagram for diagnostic process.

The first thing to note is that there are two states in Figure 1 that correspond to the mode settings of P1232.2 [2]. In P1232.2, the reasoner is assumed to be in one of three states: NULL, INACTIVE, or ACTIVE. In Figure 1, STANDBY would correspond to NULL, and READY would cover both INACTIVE and ACTIVE. In effect, 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 P1232.2 and, in some sense, may indicate a particular implementation. It is unclear, however, how to discuss usage scenarios without assuming some kind of implementation.

The process illustrated in Figure 1 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 1, 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, it is possible the service, *get_most_likely_diagnoses* could be used to generate a hypothesis of sorts. In general, this is probably true; however, all of the models currently specified in P1232.1 [3] provide the ability to compute a hypothesis. Further, with several services to be added in the next draft of the specification, we can generate a hypothesis in a straightforward way. To accomplish this, note the following services which are to be added.

- *get_number_steps(model_id)*—returns number of steps taken
- *get_test(model_id, step_no)*—returns test performed at *step_no*
- *get_outcome(model_id, step_no)*—returns outcome of test performed at *step_no*

- *get_diagnosis(model_id, diagnosis_id, step_no)*—returns confidence of *diagnosis*
- *get_anomaly(model_id,diagnosis)*—returns anomaly associated with *diagnosis*
- *get_repair_action(model_id,anomaly)*—returns repair action associated *anomaly*

Given these services¹, one method of computing a hypothesis is as follows:

```

hypothesis_set = ∅
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 ≥ θ,
        hypothesis = hypothesis ∪ {diagnosis}
od

```

This method assumes some threshold 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 of the READY state to compute the hypothesis. This leads naturally to the service *report_answer* which 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 a decision made by the user of the information, the reasoner may then be requested 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 optimizer that selects tests to be performed to minimize some cost

¹ Currently, these services are not defined. Therefore, liberties will be taken in their definition to illustrate concepts. It is likely that the actual form of the service call will change from that presented in this paper.

function. A possible sequence of 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. 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 = ∅
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)) ≠ 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

```

od

IV. OVERRIDING THE REASONER'S TEST CHOICE

In both of the previous scenarios, it was assumed the reasoner chose the test to be performed by the test system, either by applying some optimization process or by traversing a predefined test sequence or tree. In some test environments, the technician wants control over which tests are performed and when they are performed. In the context of a dynamic reasoner, this requires services to support a test choice override function as illustrated in Figure 2.

As before, this diagram assumes the presence of at least two mode states (STANDBY and READY). The normal procedure would be for the reasoner to choose a test and present it to the reasoner's client for consideration. This is shown on the path, READY — *evaluate_symptoms* — HYPOTHESIS-COMPUTED — *choose_test* — READY — *perform_test* — RESULT-KNOWN — *evaluate_result* — HYPOTHESIS-COMPUTED — *choose_test* — READY. Once back in the READY state, the client is in a position to determine whether or not to accept the test or override the test choice and submit a new test to the test system to process. This alternative is illustrated on the READY — *operator_choose_test* — READY cycle.

Two alternatives are possible to handle the test override process. The first alternative permits the client to select any test and submit that test to the test system. This would be implemented with the following service calls.

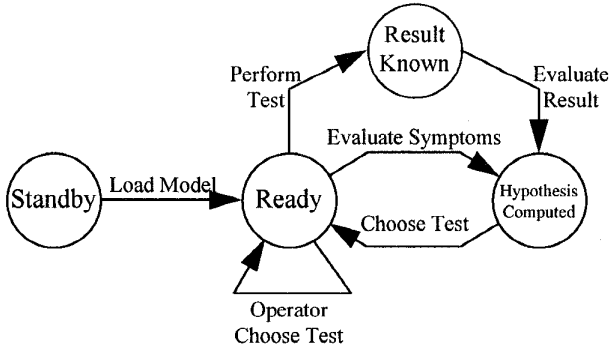


Figure 2. Test choice with operator override.

```

tests = get_all_tests(model_id)
test_choice = user_select_test(user_id,tests)
get_test_outcome(test_choice,&outcome,&confidence)
status_code =
    apply_test_outcome(model_id,outcome,confidence)

```

The second alternative constrains the available set of tests from which the user may choose. This results in a slightly modified version of the service requests listed above.

```

tests = get_all_tests(model_id)
for all test_id in tests do
    if NOT(test_available(test_id)) then
        tests = tests ∩ COMP({test_id})
    od
test_choice = user_select_test(user_id,tests)
get_test_outcome(test_choice,&outcome,&confidence)
status_code =
    apply_test_outcome(model_id,outcome,confidence)

```

where NOT(\bullet) is the negation of the logical expression designated by ' \bullet ' and COMP(\bullet) the set complement of the set designated by ' \bullet '.

The derived service, *test_available*, returns a Boolean value based on whether or not the test is available for selection. It is unlikely that the criteria for determining availability can be standardized as this is highly implementation dependent. However, it is possible that a service that provides this kind of service, independent of the underlying selection criteria, may be useful for the standard.

V. JUSTIFICATION OF TEST CHOICES

Explanation facilities rely heavily on current context and on the underlying approach to diagnostic reasoning. As such,

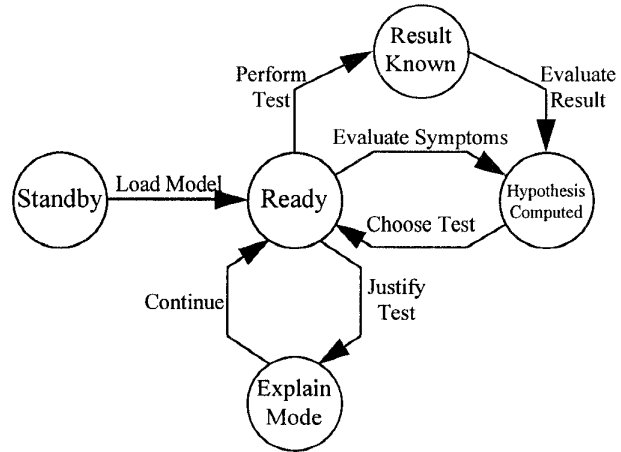


Figure 3. Justification for test choice.

it is very difficult to define a set of standard explanation services. The approach taken by the P1232.2 Service Specification is to provide several low-level services that permit querying of the current state in the reasoning process. With these low-level services, a particular reasoner should be able to construct a set of explanations to satisfy a wide variety of questions posed by users. In this section and the next, we will illustrate how two different explanation services might be constructed. The low-level explanation services were provided in the discussion on the diagnostic process.

The first explanation facility we discuss provides a justification for the current test choice. Since it is less clear how such an explanation would be constructed for a fault tree, we will focus on the case where the reasoner is processing an EDIM. The state diagram illustrating the process is given in Figure 3. In the next section, we will describe a process for justifying a set of inferences from the current set of test inferences. It will be noted that the two state diagrams are essentially identical since the focus will be on the state labeled EXPLAIN-MODE. While the reasoner sits in the READY state, the client of the reasoner may request that the current test choice be justified. At that point, the reasoner transitions to EXPLAIN-MODE and services such as the following are executed.

```

attributes = get_active_cost_attributes(model_id,TEST)
cost_attributes = get_test_costs(model_id,test_id)
for all attrib in attributes ∩ cost_attributes do
    present(get_value(model_id,test_id,attrib))
od
for all outcome in get_test_outcomes(test_id) do
    status_code =
        apply_test_outcome(model_id,outcome,1.0)
    present(get_current_hypothesis(model_id))
    revert(1)
od

```

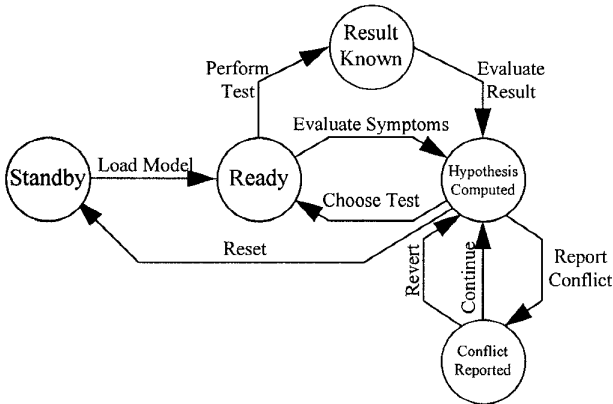


Figure 4. Handling conflict in test results.

This service provides the list of costs associated with the current test and displays the resulting hypothesis for each of the outcomes of the test. This permits the user to determine how the current hypothesis is refined based on the indicated test which is probably one of the better ways of evaluating the value of a test choice. Obviously, other methods for explaining a test choice may also be possible.

VI. HANDLING CONFLICT

When conflict arises in the inference process, the reasoner either needs to be able to detect that the conflict has occurred (and then report the conflict to the client) or be able to process that conflict and continue in the reasoning process. In the latter case, no special services are required. Figure 4 shows a state diagram which indicates that conflict might be detected as a result of processing a test result and computing the new hypothesis. We can assume a straightforward service for handling the conflict (once the conflict is detected) in which the conflict is presented to the client and an option is provided to either continue or revert. This service can be constructed as follows:

```

conflict = get_current_hypothesis(model_id)
if conflict = TRUE then
  choice = choose(continue,revert)
  if choice = revert then
    revert(1)
  fi
fi
  
```

To complete the definition of this service, we need to add services to *get_current_hypothesis* (or create a new Boolean service) capable of detecting that the conflict has occurred. There are many types of conflict to be considered including invalidation of the hypothesis, contradiction of a previous test inference, or support for an inconsistent diagnosis. For the sake of illustration, we will only consider the case where test outcomes disagree. If we assume the defined service, *get_result*, returns the inferred value for a test (since we have a separate service, *get_outcome*, to be used in generating a test history trace), we can use two services directly to make this determination.

```

conflict = FALSE
step_no = get_number_steps(model_id)
test_id = get_test(model_id,step_no)
if get_result(model_id,test_id) ≠ NULL and
  get_outcome(model_id,step_no) ≠
  get_result(model_id,test_id) then
  conflict = TRUE
fi
  
```

VII. CONCLUSIONS

The process of defining application services for an agent to perform must be implementation independent but must not make implementation difficult. The advantage to prototyping a standard as it is being developed (even manually) aids significantly in determining the ability of the proposed standard to meet the requirements of the users of that standard. This paper has presented several application scenarios and described methods for using the current set of services defined in P1232.2 [2] to satisfy requirements set forth by those scenarios. The process of applying these services was very enlightening and demonstrated both the power of the current set of services and the complexity of using those services to define a reasoner.

REFERENCES

- [1] IEEE Std 1232-1995, *Trial Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Overview and Architecture*, Piscataway, New Jersey: IEEE Standards Press, 1995.
- [2] IEEE P1232.2, *Trial Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Service Specification*, Draft 2.0, 1996.
- [3] IEEE P1232.1, *Trial Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Data and Knowledge Specification*, Draft 4.5, 1996.