Standard Diagnostic Services for the ATS Framework

John W. Sheppard
Department of Computer Science
Montana State University
Bozeman, MT 59717
john.sheppard@cs.montana.edu

Stephyn G. W. Butcher, Patrick J. Donnelly
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
steve.butcher@jhu.edu, donnell@cs.jhu.edu

Abstract—**The US Navy has been supporting the demonstration of several IEEE standards with the intent of implementing these standards for future automatic test system procurement. In this paper, we discuss the second phase of a demonstration focusing on the IEEE P1232 AI-ESTATE standard. This standard specifies exchange formats and service interfaces for diagnostic reasoners. The first phase successfully demonstrated the ability to exchange diagnostic models through semantically enriched XML files. The second phase is focusing on the services and has been implemented using a web-based, service-oriented architecture. Here, we discuss implementation issues and preliminary results.**

Keywords–AI-ESTATE, ATS framework, diagnostics, IEEE standards, interoperability

I. Introduction

The Department of Defense (DoD) has established a partnership between government, industry, and academia to address architectural design and standardization issues for Automatic Test Systems (ATS). The DoD ATS Framework Working Group is focusing on defining an information framework and identifying standards for next-generation ATS. The principal requirement to be satisfied by the framework and associated standards is to provide an open architecture for ATS to reduce overall cost of development and ownership for resulting families of standards and the ATS which employ them. Based on work in the 1990s when the ATS Research and Development Integrated Product Team defined a set of "critical interfaces" for ATS, the current working group has been selecting, supporting the development of, and demonstrating commercial standards to be used in ATS with the intent of, ultimately, implementing these standards in future ATS procurement programs.

The Institute for Electrical and Electronics Engineers (IEEE), through its Standards Coordinating Committee 20 (SCC20), is developing interface standards focusing on several elements defined in the ATS Framework. One of these standards—IEEE P1232 Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)—has been chosen for demonstration prior to mandate. Previously, we presented the results of the first phase of the AI-ESTATE demonstration, focusing on semantic interoperability of diagnostic models to support the Diagnostic Data (DIAD) element of the framework. The results of that demonstration successfully showed the effectiveness of semantic modeling in information exchange. In addition, the engineering burden imposed by stronger semantic requirements was demonstrated to be manageable.

In the second phase, the focus was on the Diagnostic Services (DIAS) element, intended to support reasoner interoperability. The AI-ESTATE standard accomplishes this by specifying the implementation of semantically defined software services in a service-oriented architecture. Here, we present an overview of the semantic interoperability problem in the context of diagnostic reasoning and forecast the results of the second phase of the demonstration.

II. The AI-ESTATE Standard

The AI-ESTATE standard has been under development since the late 1980s and has undergone several significant modifications and advances over the intervening 20 years. AI-ESTATE is based upon a foundation of formal semantic information models developed in the EXPRESS modeling language [1]. Initially, the standard was published in three parts:

- Architecture: The first part focused on a conceptual view of a diagnostic reasoner in the context of an abstract test environment [2]. This first standard imposed requirements on the following two standards rather than on any particular AI-ESTATE implementation.
- Knowledge Exchange: The second part introduced the first information models for diagnosis [3] and incorporated a little-used member of the Standard for the Exchange of Product model data (STEP) family for file exchange—EXPRESS-I [4].
- Software Services: The third part used the syntax of the EXPRESS information modeling language to specify software services, focusing mostly on model access services [5]. No specific implementation strategy was either specified or assumed; however, a binding strategy illustrated with the C programming language was provided as guidance.

Subsequent to their publication, all three standards were upgraded to full-use, and the revision process began.

During the revision of the standard, the DMC determined that maintaining the standard would be simplified if the three "components" were combined into a single document. In addition, it was pointed out that the exchange format was not

The research described in this paper was supported by the US Navy, NAVAIR PMA-260D.

particularly useful since it was never intended for file exchange. Furthermore, the STEP community had already specified a standard specifically for file exchange, known as the "STEP physical file format" [6]. Thus, in 2002, a new version of the standard was approved making these changes [7].

The IEEE requires that all of their standards either be reaffirmed, revised, or withdrawn every five years. Shortly after publication of the 2002 version of AI-ESTATE, it was discovered that a significant error was introduced into the standard. Specifically, while the information models were being updated, the specification for failure rate had been deleted from the standard. At the time the error was discovered, the DMC believed a simple "corrigendum" could be prepared repairing the error.

As the process of preparing the corrigendum began, the DMC also decided to explore incorporating an XML-based file exchange into the standard. To accomplish this, the corrigendum changed into an amendment, and the DMC decided to develop a set of XML schemata for the models in the standard [8],[9]. As the amendment proceeded, the DMC determined that there were several places where the standard could be improved, and the XML schemata should follow the STEP specification for deriving XML from information models [10]. AI-ESTATE now started a full revision process.

In the spring of 2009, the newly revised AI-ESTATE standard was balloted [11]. That standard is now in the process of being revised based on over 700 comments received. The demonstration described in this paper has been developed in parallel and has provided several additional "rogue comments" for incorporation into the draft to be recirculated by the end of the year.

III. Semantic Interoperability

As mentioned in the previous section, the fundamental principle that underlies AI-ESTATE and the approach developed to write the standard is that of providing a sound semantic specification of the information to be exchanged with a diagnostic reasoner. Within software, generally two approaches exist to exchanging information: 1) through the exchange of static files, and 2) through a set of software services defined as an Application Program Interface (API). AI-ESTATE provides specifications both for file exchange and for software APIs. Although not stated explicitly, software-based access to model elements could also be done through the STEP Standard Data Access Interface (SDAI) [12].

Several previous papers have been published focusing on the role of semantic models in addressing data exchange within AI-ESTATE [13], supporting information integration [14], and facilitating data mining [15]. In this paper, we focus on the role of semantic interoperability in defining software services and on the demonstration of the AI-ESTATE services. Within AI-ESTATE, five information models were developed to support model exchange. In addition, a sixth information model has been developed—the *dynamic context model* (DCM)—that defines the semantics of much of the information required when performing diagnosis. In addition, a new information model is under development as a result of the P1232 ballot that defines a set of types for the information passed by way of the services. We will discuss each of these two models here.

A. *Dynamic Context Model*

The AI-ESTATE standard describes the DCM as a model used that "captures a record of the reasoning session that may be used in any diagnostic context by an adequate abstraction of widely used diagnostic principles. … The DCM data and knowledge are developed during a diagnostic session, unlike those of the [other AI-ESTATE models] (which consist of static diagnostic data and knowledge." As described, the DCM represents only a history of the diagnostic session; however, the semantics of the model do far more than that. First, the model defines the concept of a diagnostic *session*. A session is a container for everything that happens while a system is being tested or monitored and while reasoning about test/monitor information is going on. Thus, the session encapsulates the entire diagnostic process.

While the session is used as a "container" for the information captured during diagnosis, the key entity defined within the DCM corresponds to a *step* in the diagnostic process. The balloted version of AI-ESTATE defines a step as follows:

```
ENTITY Step;
  Name : OPTIONAL NameType;
  actionsPerformed : LIST OF ActiveAction;
  activeModels : SET [1:?] OF DiagnosticModel;
  optimizedByCost : SET OF CostCategory;
  optimizedByDistribution :
        ReliabilityDirected;
  optimizedByUser : HypothesisDirected;
  outcomesInferred : SET OF ActualOutcome;
  outcomesObserved : SET OF ActualOutcome;
  serviceLog : OPTIONAL LIST [1:?] OF
        ServiceState;
  stepContext : OPTIONAL ContextState;
  timeOccurred : OPTIONAL TimeStamp;
  userHypothesis : OPTIONAL SET [1:?] OF
        ActualOutcome;
  lifeCycleStatus : LIST [1:?] OF ActualUsage;
INVERSE
  owningSession : Session FOR trace;
WHERE
  modelsInSet :
        (SELF.activeModels <=
                SELF.owningSession.modelSet);
  hypothesisWithUserDirected :
        (NOT(SELF.optimizedByUser) OR
                EXISTS(SELF.userHypothesis));
END_ENTITY;
```

To interpret, the step entity defines the main pieces of information to be collected during diagnosis and includes concepts such as the actions that are performed (including any setup actions, adjustments or repairs, and especially tests), the results of these actions (especially test results/outcomes), and any inferences drawn as a result.

The step also provides the ability to specify how the diagnostic reasoner should choose tests to perform, assuming the reasoner has that capability. Specifically, the standard assumes the reasoner as a default test choice capability. The specifics of that capability are immaterial; it could follow a static fault tree, perform a brute-force end-to-end test, or optimize test selection based on information gain. Up to three additional criteria can also be selected for optimization:

1. Cost-based optimization: Incorporate cost factors as specified in an instance of the AI-ESTATE Common Element Model (CEM) such as the time required to perform a test or the skill level of the technician performing the test.
2. Probability-based optimization: Incorporate information on the prior probability of a diagnosis incorporated into an instance of the CEM. Usually, these priors are determined based on failure rate information; however, the specific way the reasoner determines priors is not specified.
3. Hypothesis-directed optimization: Incorporate knowledge of an expert, a technician, or any other "agent" that might be able to furnish a hypothesis to the reasoner. The reasoner then redirects its test choice process to focus on verifying or denying that hypothesis.

The ability to optimize by any or all of these approaches has been available to several diagnostic tools for at least 20 years, one of which was co-developed in 1988 by the lead author [16].

The AI-ESTATE standard does not specify how to use this model except to require that a reasoner be able to export a fully populated DCM using one of the two interchange formats. These exported files can then be used by applications to record the history of the session or support *post mortem* analysis of the session to support trending or maturation of the models. In addition, the DCM includes the ability to reference other DCMs to enable reasoners with the ability to incorporate historical information into the reasoning process to access that historical information. Because of the formal information model, these tools are able to use the exported information and understand exactly what the information means.

It is possible that a reasoner can also use the DCM to specify its internal representation of the state of diagnosis. While such a use is neither required nor necessarily even recommended, this use points out the richness of the model in that it is more than just historical data. In fact, it is this potential use that led to the word "dynamic" being incorporated into the name. As diagnosis proceeds, information integrity is maintained by mapping the information used to the DCM.

B. Service Interface Model

In addition to the DCM, the DMC is in the process of developing a new information model to specify semantics of the data being passed to and from the services. To explain this model, we also need to explain the service specification itself. In the balloted version of P1232, two classes of services were specified—model management services and reasoner manipulation services. The model management services were intended to permit a client to create, get, put, and delete elements in any of the AI-ESTATE models. The reasoner manipulation services were intended to provide the primary means of a client communicating with a diagnostic reasoner during a diagnostic session. As a result of the P1232 ballot, the model manipulation services are being deleted, leaving only services for communicating with the reasoner.

We will describe a complete use case in the next section; however, here we describe three of the services specified. Diagnosis is a process of obtaining observations (e.g., test results) and drawing conclusions about the target of diagnosis based on those observations. Thus, three key services include getting a recommendation for a test to perform (`recommendActions`), reporting the results of that test to the reasoner (`applyActions`), and asking that the reasoner update its belief in the state of the target system (`updateState`). These services are specified using the syntax of EXPRESS as follows:

```
FUNCTION recommendActions(
  maxNumber : OPTIONAL INTEGER;
  levelsOfInterest : OPTIONAL SET [1:?] OF
        NameType;) :
  LIST [1:?] OF ActionRecomendation;
END_FUNCTION;

PROCEDURE applyActions
  (actions : LIST [1:?] OF ActualAction);
END_PROCEDURE;

PROCEDURE update_state;
END_PROCEDURE;
```

The `recommendActions` service has two optional attributes that specify the maximum number of actions to return as the levels of indenture of the target system for the analysis. It returns a set of `ActionRecommendation`, which is a new entity to be included in the new information model.

```
ENTITY ActionRecomendation;
  actionNames : LIST [1:?] OF NameType;
  actionDescriptions : LIST OF [1:?]
        DescriptionType;
  sequenceDescription : OPTIONAL
        DescriptionType;
  costCategories : LIST [0:?] OF NameType;
  catDescriptions : LIST [0:?] OF
        DescriptionType;
  estimates: LIST [0:?] CostValue;
  uppers : LIST [0:?] CostValue;
  lowers : LIST [0:?] CostValue;
  units : LIST [0:?] STRING;
END_ENTITY;
```

The `applyActions` service passes a list of `ActualAction`, which is an entity also defined in the new model.

```
ENTITY ActualAction;
  actionName : NameType;
  statusValue : OPTIONAL AssignedValue;
  statusConfidence : OPTIONAL ConfidenceValue;
  costLabels : OPTIONAL LIST [0:?] OF NameType;
  costValues : OPTIONAL LIST [0:?] OF
        CostValue;
END_ENTITY;
```

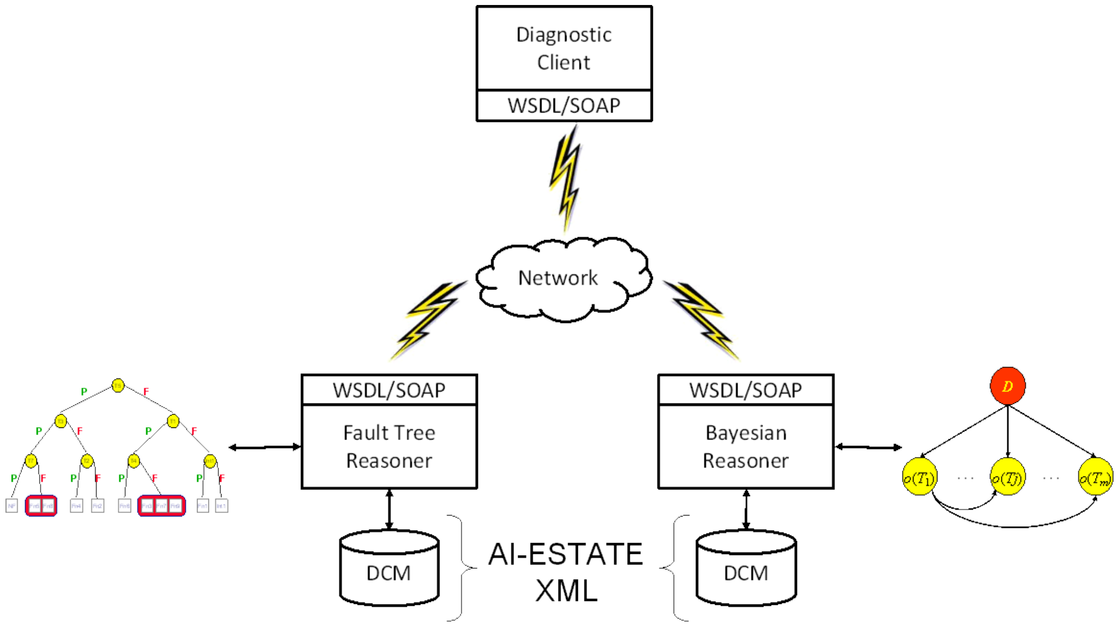

Figure 1. AI-ESTATE Phase II Demonstration Concept

Thus, by including semantic definitions of the parameters and return values, AI-ESTATE provides an additional layer of semantic interoperability in the service specification as well.

IV. Demonstrating AI-ESTATE Services

As discussed above, the revised AI-ESTATE standard contains a set of model management and reasoner manipulation services, intended to provide a well-defined, semantically valid software interface for performing fault diagnosis. In addition, the AI-ESTATE standard currently specifies these services using the Web Service Description Language (WSDL), thus enabling a web service-oriented implementation.[1] Note that using web services is only one of several ways the services might be implemented; however, for purposes of this demonstration, this is the approach we used.

The primary objective of this phase of the demonstration can be summarized as follows:

> *To demonstrate the ability of IEEE Std 1232 (AI-ESTATE) to ensure the interchangeability of diagnostic reasoners through the definition of encapsulated services.*

To this end, the focus of this demonstration was on the implementation of the services specified in the current draft of IEEE P1232. That clause specifies two classes of services—model management services and reasoner manipulation services. Since model management services are to be deleted from the standard these were not implemented in the demonstration. Instead, the focus was on reasoner manipulation services since it is in this area where reasoner interchangeability is realized.

[1] Recently, the BRC decided to remove the WSDL specification from the standard; however, the demonstration project proceeded with them since they permitted a level of "net-centricity" to be demonstrated.

The basic structure implemented in this phase of the demonstration is shown in Figure 1. Key to the demonstration is the abstraction created by specifying the reasoner manipulation services, thereby encapsulating the reasoner. In particular, by specifying these services and the associated communication protocol (in this case, the Simple Object Access Protocol—SOAP), it should no longer matter how the specific reasoner or reasoner type behind the interface has been implemented. This was demonstrated by using two different types of reasoners—a fault tree reasoner (to exemplify legacy diagnostic systems) and a Bayesian reasoner (to exemplify state-of-the-art diagnostic systems).

In a "confederation of services" architectural environment, exactly what is a client and what is a server can get a bit hazy when each actual application may be a provider of some services (and thus a server) and a consumer of other services (and thus a client). Even the diagnostic reasoner is a client of other servers when called on service methods that end with "`FromLocation`." These indicate that the reasoner should acquire the specified resource at the URL indicated.

However, the AI-ESTATE standard is only concerned with a central client-server relationship—the one between the application calling the reasoner manipulation service methods (client) and the application responding to those service calls (server). Note that the client need not be the ultimate consumer of the services nor the responding server the actual producer of the services. This ability to delegate and adapt permits legacy and otherwise mismatched applications to be integrated into the "confederation."

We chose to implement the services in SOAP because this is what the balloted standard suggested for web services. For the producing application (server), the starting point was the Bayesian reasoner from the Phase I demonstration. This application was largely implemented in Ruby, embedded in a web application container and responded to custom API calls via HTTP GET and POST. For Phase II, a standalone server was created that could respond to the standard service calls via SOAP and then delegate them to the reasoner into which it had been embedded.

For the consuming application (client), a standalone server was created from scratch that could both communicate via SOAP with the reasoner and via XML-RPC with the actual end client or GUI. This separation permitted information traveling from the reasoner to the end user to be decorated with additional information and for information traveling from the end user to the reasoner to be stripped of such decorations.

Finally, in order to avoid multiple copies of models being propagated throughout the system, a simple model server was created that all applications needing the models could access. A more complete implementation of the model server would integrate with tools like those developed in Phase I for creating and editing models.

V. A Use Case

To illustrate the application of the standard reasoner, imagine the diagnostic client resides on a test station at a maintenance depot but two diagnostic reasoners are located on possibly different servers at remote locations. The test station has access to the internet or a closed intranet and is able to communicate with these reasoners over the network. A unit under test (UUT) is attached to the test station, and the maintenance technician requests that a model be loaded for that UUT. Once the model is loaded, testing begins.

Alternatively, one can imagine using different models and reasoners depending on the type of testing going on. For example, a technician could use the Bayesian model for the initial fault diagnosis. Bayesian models provide an excellent way to account for uncertainty in the test process and give more robust information to a technician to determine the proper course of action. Once the fault is isolated and the repair completed, the unit must undergo a verification/recertification test. Due to constraints typically imposed on the recertification process, a "certified" fault tree could be used during recertification. From a process point of view, the diagnostic application would function the same in both cases, thus simplifying training and use in the maintenance process.

With either scenario, at each step of testing, the tester would request a test recommendation from the diagnostic server. Upon receiving the recommendation, the tester would run the corresponding test (if possible) and provide the results back to the server. Based on the results of the test (which might include an inability to perform that test), the diagnostic hypothesis would be updated and returned. The process would continue until either no more testing was required or the technician determined the hypothesis was sufficient to take action. At the end of the session, the technician would request that the session be exported and associated with a Maintenance Action Form (MAF) for the UUT.

Using this scenario, the following basic use case was developed to identify the key services to be implemented. As an interesting (and important) byproduct of the process of developing the use case, several services were identified as being "missing" from the standard. These services are in the process of being incorporated into the version to be recirculated for re-ballot. These missing services are discussed in the next section.

Step 1–Reasoner discovery: The client needs to determine what reasoners are available for diagnosis. This service has not been standardized in P1232, so we will assume the client knows the available reasoners at the start of the process.

Step 2–Model discovery: Similar to the first step, the client needs to know what models are available for use by the reasoner. Of particular importance is making sure the model type matches the reasoner type. For example, if the reasoner uses Bayesian inference, it would not make sense to send it a fault tree.

Step 3–Start diagnostic process: As suggested, this is the point at which a diagnostic session begins and the session entity would be "instantiated" in the DCM. Of course, the reasoner is free to instantiate the DCM at the end of a session, but this step indicates a session entity is required. The defined service is `initializeDiagnosticProcess`.

Step 4–Load model: Here a diagnostic model is loaded into the reasoner. P1232 permits models to be loaded in one of two ways—`loadModel` and `loadModelFromLocation`. The former assumes the reasoner knows where the model is, and the latter permits the client to direct the reasoner to the model location. We implemented the latter service. At this point, the identified model is "attached" to the session in the DCM.

Step 5–Activate model: Currently, P1232 permits multiple models to be associated with a session. It is likely this will be changed in the published standard; however, currently this feature requires the desired model to be activated for use by the reasoner. The associated service is `setActiveModel`, and this has the effect of creating a step in the DCM and attaching the activated model to that step.

Step 6–Set UUT Usage: For purposes of indexing into associated failure distributions, the reasoner may need to know how long it has been since the UUT was last serviced or since the UUT was put into operation. The `setUsage` service permits this information to be associated with the current step. It is likely this information will be moved to the session entity.

Step 7–Set optimization: One of the underlying assumptions the DCM made in writing the P1232 standard was the diagnostic reasoners often provide the capability of optimizing the test selection process. As discussed previously, three different types of optimization have been defined in the services. At this point, the desired optimization can be set using one or more of the services `optimizeByCostCriteria`, `optimizeByDistribution`, or `optimizeByUser-Hypothesis`. It is assumed the client either has a complete copy of the model or at least knows the important elements contained in the model (such as available cost criteria or the set of diagnoses).

Step 8–Set initial results: This step is not necessary but often occurs in practice. The idea is to present the reasoner with known information such as symptoms, Built-In Test (BIT) results, etc. so that diagnosis can take this prior information into account. At this point, two services would be applied in sequence—`applyActions` (where the prior information is supplied as a set of action or test results) and `updateState` (where the reasoner is asked to draw inferences from the supplied information). As a byproduct, inferences would be associated with the current step in the DCM and a new step would be appended to the end of the session trace.

Step 9–Enter main diagnosis loop: At this point, the following process is repeated until such time as either the reasoner or the client determines no further testing is required. Specifically, three services are applied in sequence—`recommendActions`–`applyActions`–`updateState`. The latter two have the same effect as in Step 8 above. The `recommendActions` service provides a list of one or more actions or tests to perform with associated figures of merit. The figures of merit permit the client to choose whether to perform the actions, which actions to perform, and whether or not to terminate.

Step 10–Provide results: Actually, this last step can be performed at any point in the above sequence; however, at a minimum it needs to be done at the end of the diagnostic session. Here, the current hypothesis produced by the reasoner is supplied to the client through `getDiagnosticResults`.

VI. Issues and Deficiencies

The DoD is placing more and more emphasis on net-centricity in future ATS and maintenance system procurements. Within the context of AI-ESTATE and diagnostics, net-centricity can be achieved through different approaches which can be broadly grouped into those that are data-centered and procedure-centered. Data-centered approaches generally fall under the category of RESTful interfaces although their adherence to the principles of representational state transfer (REST) can vary. REST centers on resources and Uniform Resource Locators (URL) to resources. In REST, the standard's `startDiagnosticProcess` would be mapped to `/myreasoner/session/create` where the parameters for the UUT and serial number of the `actualRepairItem` are supplied as XML or query parameters. The call would return a session identifier, which would be used in subsequent API calls. For example to call `recommendActions` using REST, one would access the recommended actions as a resource on the reasoner using a URL such as `/myreasoner/123456/actions/list`. The session id has become part of the URL which identifies the resource for this particular session. While the semantics of the standard could be achieved using REST, the standard has an explicit implementation bias towards procedure-centered approaches because the services must have the same names as those in the standard.

Procedure-centered approaches have a long history. The Common Object Request Broker Architecture (CORBA) is one of the oldest object-oriented remoting approaches. Remote Method Invocation (RMI) fulfills a similar role in Java-based solutions. For net-centric solutions, the oldest remote procedure call (RPC) type of approach is the standard HTTP GET/POST. This approach uses a URL and query parameters. Our `startDiagnosisProcess` could be implemented in HTTP GET/POST using a URL such as

```
/myreasoner/startDiagnosticProcess?uut=X&actual
             RepairItem=Y.
```

Unlike REST, HTTP GET/POST services can also be formally described in a WSDL. Another alternative is XML-RPC. In this case, an XML document is sent describing the method to be invoked and the parameters to that method on the remote server by POSTing an XML document to a service URL. XML-RPC eventually evolved into SOAP; however, XML-RPC is still used today for those who find defining a SOAP service through a WSDL to be "overkill." One final alternative is to use SOAP itself and one of its various encoding combinations including "rpc" or "document" with "literal" or "encoded." There is also a format recommendation called "wrapped document literal." All of these formats have their good and bad points.

Although the standard services definition precludes some implementations (REST), it lays the foundation for a simplified process of coordination and integration when parties must negotiate or use implementations. This process might be even further enhanced by the presence of reference implementations; however, SCC20 has historically taken a position against either providing reference implementations or conformance test suites.

Other than the various implementation issues and alternatives, the process by which the services have been implemented for demonstration served a very useful purpose. Several items were found to be missing in the balloted standard that would be of value (and even required), given common uses of diagnostic applications. Two areas were such deficiencies were found were with the management of resource availability and the specification of optimization criteria.

The balloted standard includes a service that permits a client to specify which resources (e.g., test instruments, power sources, etc.) were available: `setAvailableResources`. While the AI-ESTATE CEM includes information on resources, the DCM does not capture what resources have been identified as available. From the perspective of the service, the DCM does not need to store this information; however, given the DCM is supposed to capture a history of all information relevant to the diagnostic process, omitting this information was a serious oversight. As a result, the DCM has been modified as part of the demonstration to capture this information.

While the problem identified above illustrates a situation where the DCM does not support a service, the second problem illustrates the opposite situation—where no service exists to support information in the DCM. Specifically, the balloted version of the DCM defines four attributes of a step to support the optimization process:

- `Step.optimizedByCost`
- `Step.optimizedByUser`

- `Step.userHypothesis`
- `Step.optimizedByDistribution`

Unfortunately, no corresponding reasoner manipulation services were defined that enabled the client to set these attributes. Technically, the model management services could be used to set these values, except an approach was taken where model management services worked with entity ids and reasoner manipulation services worked with entity names. This mismatch made it impossible to use the model management services with the reasoner manipulation services to set the optimization information. As a result, three services have been added as a part of this demonstration to address this deficiency:

- `optimizeByCostCriteria`
- `optimizeByDistribution`
- `optimizeByUserHypothesis`

The purpose of the demonstration process was to provide a way of "testing" a standard being considered by the DoD for future mandate. The intent is to provide a proof-of-concept that the standard works and will satisfy the DoD's requirements. But there is a more important benefit to the demonstration process, especially when no *de facto* standard exists. The demonstration process serves as a valuable tool for identifying deficiencies/errors and testing alternatives, thus providing a mechanism for producing an even more effective standard.

VII. CONCLUSION

In this paper, we discussed the role of information modeling in the context of defining software services to ensure *semantic* interoperability of elements in a test environment. The focus of our discussion was the P1232 AI-ESTATE standard since this standard used information models for this very purpose. We also provided a high-level discussion of the second phase of a demonstration project for the Navy where the services specified in AI-ESTATE were implemented to show interoperability characteristics.

Key conclusions to be drawn, both from the process by which the standard was developed and from the demonstrations performed are that semantic interoperability remains a "tricky" issue. One of the main goals of the DMC was to define a standard that would be independent of implementation language and permit "plug-and-play" functionality of diagnostic reasoners. Unfortunately, this goal has not been realized fully, and there is considerable doubt whether such a goal will ever be realized. Therefore, the emphasis has shifted from being completely language independent and fully interchangeable to that of providing a specification whereby "contract negotiations" between parties developing these implementations is minimized.

In the revised standard, the use of WSDL and SOAP has been removed, and associated WSDL files are no longer available. In addition, the binding strategy section of the standard has been rewritten to say,

> It is beyond the scope of this standard to define bindings for each implementation language. However, in the interest of interoperability, the standard provides the following guidance for services passing and returning data:
>
> - Component implementations should use native messages.
> - Object-oriented implementations should use objects.
> - Procedural implementations should use structures.
> - Other implementations should use XML entities defined by Part 28 schemas.
>
> The application and diagnostic reasoner programs may be written in different languages as long as the translation is handled transparently to the two programs, i.e., in the binding layer or lower. When publishing the interface, it is recommended that documentation of traceability of the elements of the interface to the services specified in the standard be provided.

In spite of the weakening of expectation for AI-ESTATE implementations, the primary goals of the P1232 standard have been achieved—to incorporate domain specific terminology, to *facilitate* portability of diagnostic knowledge, and to *enable* consistent exchange and integration of diagnostic capabilities. Specifically, the information models themselves provide formal definitions of the domain specific terminology and promote semantically valid exchange of diagnostic knowledge. By using these models as the basis for defining the services, these models also enable exchange and integration of diagnostic capabilities providing a major starting point in negotiating the system interfaces for the diagnostic reasoners.

ACKNOWLEDGMENT

We thank the members of the ATS Framework Working Group and the members of the IEEE SCC20 DMC for their constant vigilance in the creation and demonstration of several IEEE standards. We appreciate their comments on the standards and their support through the completion of this demonstration. In particular, we thank Alicia Helton, Oscar Fandino, Darryl Busch, Tim Wilmering, Mike Malesich, and Jennifer Fetherman.

REFERENCES

[1] ISO 10303-11:1994, *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 11: Description Methods: The EXPRESS Language Reference Manual*, Geneva, Switzerland: The International Organization for Standardization, 1994

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

[3] IEEE Std 1232-1997, *IEEE 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, 1997.

[4] ISO/TR 10303-12:1997, *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 12: Description Methods: The EXPRESS-I Language Reference Manual*, Geneva, Switzerland: The International Organization for Standardization, 1997.

[5] IEEE Std 1232-1998, *IEEE Trial Use Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE): Service Specification*, Picataway, NJ: IEEE Standards Press, 1998.

[6] ISO 10303-21:1994, *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 21: Implementation Method: Clear Text Encoding of the Exchange Structure*, Geneva, Switzerland: The International Organization for Standardization, 1994.

[7] IEEE Std 1232-2002, *IEEE Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE): Overview and Architecture*, Piscataway, NJ: IEEE Standards Association Press, 1995.

[8] *eXtensible Markup Language (XML) Schema Part 1: Structures*, Second Edition. W3C Recommendation 28 October 2004. Available from World Wide Web.

[9] *XML Schema Part 2: Datatypes*, Second Edition. W3C Recommendation, 28 October 2004. Available from World Wide Web.

[10] ISO 10303-28:2007, *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 28: Implementation Methods: XML Representation of EXPRESS Schemas and Data Using XML Schemas*, Geneva, Switzerland: The International Organization for Standardization, 2007.

[11] IEEE P1232, *IEEE Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)*, Draft 4, Piscataway, NJ: IEEE Standards Association Press, 2009.

[12] ISO 10303-22:1998, *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 22: Implementation Methods: Standard Data Access Interface Specification*, Geneva, Switzerland: The International Organization for Standardization, 1998.

[13] John W. Sheppard, Stephyn G. W. Butcher, Patrick J. Donnelly, and Benjamin R. Mitchell, "Demonstrating Semantic Interoperability of Diagnostic Models via AI-ESTATE," *Proceedings of the IEEE IEEE Aerospace Conference*, Big Sky, MT, March 2009.

[14] Timothy J. Wilmering, "Semantic Requirements on Information Integration for Diagnostic Maturation," *IEEE AUTOTESTCON 2001 Conference Record*, Valley Forge, PA, September 2001.

[15] Timothy J. Wilmering and John W. Sheppard, "Ontologies for Data Mining and Knowledge Discovery to Support Diagnostic Maturation," *Proceedings of the 18th International Workshop on Principles of Diagnosis (DX-07)*, Nashville, TN, May 2007.

[16] William R. Simpson, John W. Sheppard, and C. Richard Unkle, "POINTER—An Intelligent Maintenance Assistant," *IEEE AUTOTESTCON '89 Conference Record*, Philadelphia, PA, September 1989.