IMPLEMENTING AN AI-ESTATE BASED DIAGNOSTIC ENGINE COMPONENT

Amanda Jane Giarla and William L. Simerly, Hamilton Software, Inc., 2270 Northpoint Parkway, Santa Rosa, CA 95047, (707) 542-2700, amanda@hamsoft.com, simerly@hamsoft.com.

Abstract – This paper focuses on the construction of a Diagnostic Engine Component (DEC) based on the IEEE 1232 Standard known as "AI-ESTATE" as part of a new approach to constructing and using Automatic Test System (ATS) software in the support of the modern digital avionics maintenance program. Traditional ATS have been constructed of single monolithic software programs that run on single machines. The design approach discussed in this paper is based on decomposing the monolithic software program into primary elements then re-engineering the elements into a new type of software component. While the construction of the DEC is detailed, the roles of other components are discussed along with deployment and how the test engineer develops the diagnostic models and the test program set.

Preface

This work is in response to the U. S. Department of Defense, Small Business Innovative Research (SBIR) Program Solicitation AF98-252 [1]. The U.S. Air Force, SA-ALC/LDKAB at KELLY AFB TEXAS, funds the project. The Technical Point of Contact is David Wade of the SA-ALC/LDAE-ADTIC office, Kelly AFB, Texas. Phase-I was completed in February 1999. Phase-II started in June of 1999 and will conclude in the first quarter of 2001.

Introduction

The ability to diagnose faults in complex electronic systems is a critical capability in the modern electronic maintenance program. This paper focuses on the design of a Diagnostic Engine Component (DEC) based on the IEEE 1232 diagnostic standard called AI-ESTATE (Artificial Intelligence Exchange and Service Tie to All Test Environments) [2] and the DEC's relationship to other components composing an Automatic Test System (ATS). Due to the characteristics of AI-ESTATE and the nature of our component based approach the diagnostic engine is operationally independent with regard to test languages, test equipment, devices and units under test (DUT's & UUT's) and signals. The DEC supports the diagnoses of faults in digital avionics systems and devices as well as other types of electronic systems that employ digital, analog or mixed signals and their respective fault types.

We begin with a general discussion of our Air Force SBIR funded approach to building a component-based ATS. Task oriented configurations of components are discussed to provide an operational context. Next the paper discusses the detailed design of the Diagnostic Engine Component which implements AI-ESTATE followed by a brief descriptions of the other three domain components of our Phase-II efforts – the Test System Component (TSC), the Application Executive Component (AEC) and the Model Editing Component (MEC). Discussion continues with a description of test program and diagnostic model development. We end with a discussion of the benefits followed by a summary.

Component-Based Approach

The original solicitation's [1] primary requirement, among many requirements, focused on de-coupling diagnostic reasoning from the rest of the elements in a typical ATS of monolithic construction. The principal investigator Amanda Jane Giarla proposed an approach [3] that decomposes any monolithic type of program into primary elements. The primary elements are then reengineered into a set of components that connect and inter-operate on a single platform or set of platforms across a network.

Differentiating Component Types

It should be noted that the types of components constructed as a result of reengineering ATS elements are different and more advanced than what is currently identified in the software engineering literature as "software components."

Software Components

Szyperski provides us with a clear and useful definition of software components. "Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system" [4]. It should be noted however that this definition of software component facilitates the software engineer. In fact software components are resources that software engineers utilize to construct programs.

Domain Components

The type of software components we have produced are not designed to be resources that software engineers compile together in the production of programs. They are instead intended to be resources for systems engineers and test engineers. The systems engineer will configure the components into a domain-oriented solution, in this case a digital avionics ATS. The test engineer will utilize the system of components as tools to construct test programs and diagnostic reasoning models as well as run tests and diagnostic sessions on systems, DUT's and UUT's. Hence the name domain components. Domain components fit Szyperski's definition of software component in that a domain component is a binary unit of independent production, acquisition and deployment. However, domain components contain characteristics that software components do not.

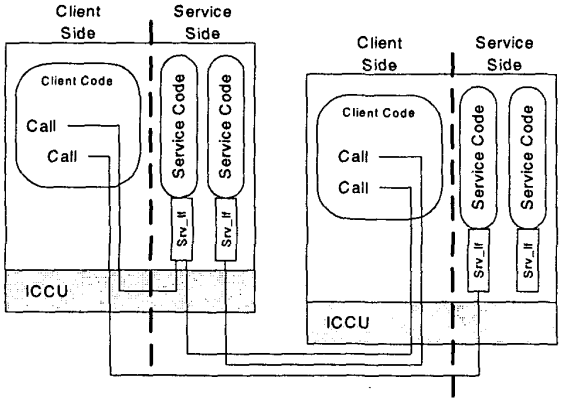

Figure 1 Components Plugged into Inter Component Communications Units (ICCU)

Domain Component Characteristics

The characteristics of our domain components include the following (refer to Figure 1).

- Components are constructed as programs with a main routine providing runtime component independence.
- Components contain both a client side and a service side.
- Components can be deployed all on a single platform or distributed across the Internet on dissimilar platforms with dissimilar operating systems.
- Components connect and interoperate through a "supporting component framework" [4].
- Component client and service code operate independently from the supporting component framework.
- Components contain immutable interface contracts to their services.

Supporting Component Framework

We call our supporting component framework an ***I***nteroperable, ***C***onnectivity ***E***nabling ***F***ramework (ICEF). ICEF is constructed of a set of ICCU's whose implementation is based on any of the common types of systems integration enabling technologies. For example the following provide connectivity and interoperabilty.

- **COM/DCOM** (Component Object Model/Distributed COM).
- **CORBA** (Common Object Request Broker Architecture).
- **DCE/RPC** (Distributed Computing Environment Remote Procedure Call).
- **RMI** (Remote Method Invocation).

It should be noted that it is possible to construct an ATS where the supporting framework contains mixed integration enabling technologies. In our SBIR Phase-I contract we successfully constructed and experimented with a framework that contained both COM/DCOM and CORBA and a *"technology-bridge."* [3]

Component Based Task Configurations

A component-based task configuration includes the usage of only the required components

to accomplish the task. The following are example tasks and possible associated configurations of components, refer to Figure 2.

- Test program library development might utilize the AEC and one of the TSC's.
- Diagnostic model development might use the AEC, and MEC and optionally the manual use of one of the TSC's.
- An on-board diagnostic run might use the AEC, DEC, the flight line TSC and the dbC.
- Historical data analysis might use the AEC along with access to the DBMS's analytic software through the dbC.
- A training session might use the AEC, DEC, dbC and a simulation.

Phase-II System Architecture

The system architecture is composed of a set of Domain Components that plug into a set of Inter Component Communications Units (ICCU), (refer to Figure 1 and 3). The domain components are a composite of both client and service code, (refer to Figure 1). The ICCUs are composed of DCOM-based interfaces that provide for component distribution.

Independence

The domain components and the ICEF component framework are designed to function as independent elements. It is expected that the ICCU's will evolve over time with regard to the enabling technologies and transport infrastructure. For example we are currently considering constructing the ICCU's out of more advanced enabling technologies such as COM+ or EJB (Enterprise Java Beans). Each component is designed to be computationally independent; that is to say each component is a program that can run in stand-alone mode. It is the intention, however, that the components work together to provide a solution to a domain problem such as those outlined in the task configuration section discussed above.

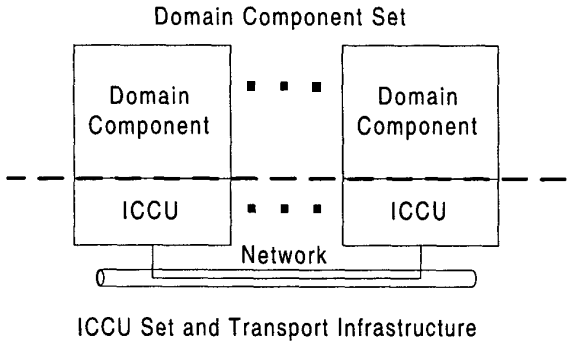

Figure 2 Overview of Systems Architecture

Component Architectures

In the conduct of Phase-II we are building four components AEC, DEC, TSC and MEC. This section will discuss the architecture of each of the components.

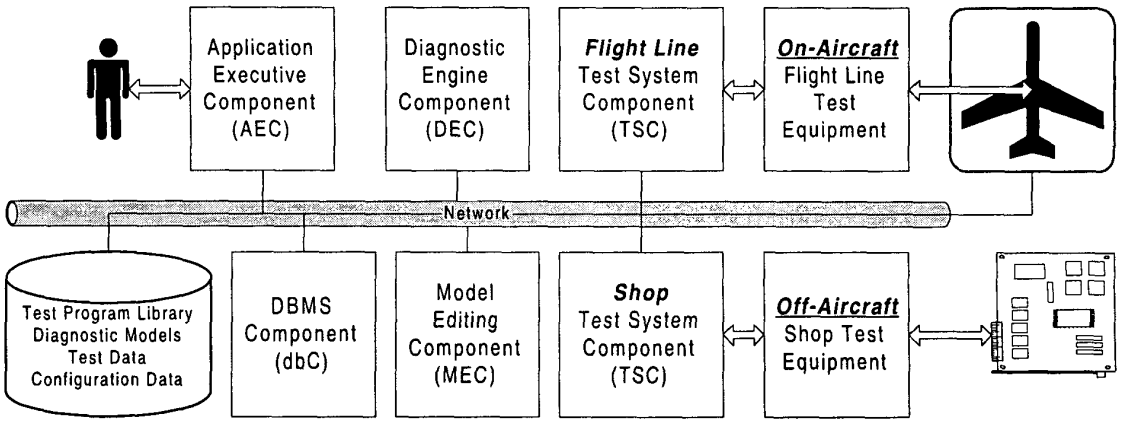

Figure 3 Operational Layout of Digital Avionics Oriented ATS

Diagnostic Engine Component

The Diagnostic Engine Component (DEC) is a resource for the test engineer to utilize in conducting diagnostics. The DEC is thus a domain component. It should be noted however, that the DEC is a composition of software components that a software engineer utilized to construct the domain component. The most important software component in the DEC is the diagnostic model management component (MMC) (refer to Figure 4).

Diagnostic Reasoner

It should be noted that the MMC becomes a fault tree type diagnostic reasoner when the standardized fault tree data and knowledge is loaded and the information entities are converted into entity objects. This of course is one of the advantages of model-based diagnostic reasoning. The modeled intelligence of diagnostic reasoning can be changed without a recompilation of the engine software making it a powerful tool for the test engineer.

Model Management Component

The MMC is an object-oriented implementation written in C++ that loads, manages and saves AI-ESTATE compliant diagnostic information models and implements the AI-ESTATE services [5].

The behavior of the MMC is as follows:

- On startup, waits for service calls instructing it to perform actions.
- Reads knowledge bases and diagnostic models from exchange files that are compliant with IEEE 1232 (AI-ESTATE) [6] and ISO 10303 Part-21 (Clear Text Encoding, also known as "SPFF" or STEP Physical File Format) [7].
- Converts diagnostic information model entity instances encoded in the exchange file to objects in local memory.
- Builds an internal set of objects that map information entity IDs to the memory addresses of the corresponding entity objects.
- Provides AI-ESTATE services for the loaded model(s).

Figure 4 shows the major elements of the MMC's architecture. COM interfaces provide access to the AI-ESTATE services within the component. There is a one-to-one correspondence between the interfaces and the services provided. The COM interfaces are exposed using DCOM making the MMC a distributable component.

The model loader/saver loads knowledge bases and diagnostic information models from exchange files. It uses the entity factory to create objects corresponding to the information entity instances, (which model the relational reasoning information) in the exchange files. The entity set is a container object that manages access to the entity objects. Services request access to entity objects (or in-core reasoning information) from the entity set by passing the internal entity ID number. The entity set returns a pointer to the entity's object in memory.

Interfaces

There are 6 major groupings of AI-ESTATE services, one corresponding to each of the 5 AI-ESTATE supported information models [6], and one for the control and reporting services [5]. The information models are:

- Common Element Model (CEM)
- Fault Tree Model (FTM)
- Diagnostic Inference Model (DIM)
- Enhanced Diagnostic Inference Model (EDIM)
- Dynamic Context Model (DCM)

The primitive AI-ESTATE services are designed to provide standardized access to the information entities within these models. The interfaces in our design are organized by entity type; that is, there is a COM class for each entity type in the models. There is also a COM class for the control and reporting services. Services are implemented as methods in these classes.

Classes

The following subsections specify the major C++ classes of the MMC.

CModelLoaderSaver

The ModelLoader/Saver object in Figure 4 is an instantiation of the CModelLoaderSaver class. In response to a call to either of the AI-ESTATE services *load_model* or *load_knowledge*, the COM object that implements the service creates a model loader object and calls the *load* method. *load* accepts the pathname of the exchange file. Once

the file has been opened, the model loader uses an internal parser to extract the encoded entity instance definitions, which it passes one at a time, to the entity factory to create internal objects. The entity factory returns a pointer to the new entity object along with its internal ID. *load* then uses these to tell the entity set object to add an entry for the entity. In response to a call to either of the AI-ESTATE services *save_model* or *save_knowledge*, the COM object steps through the Entity Objects and writes each object out in the information model form encoded as prescribed by ISO 10303-21. [7]

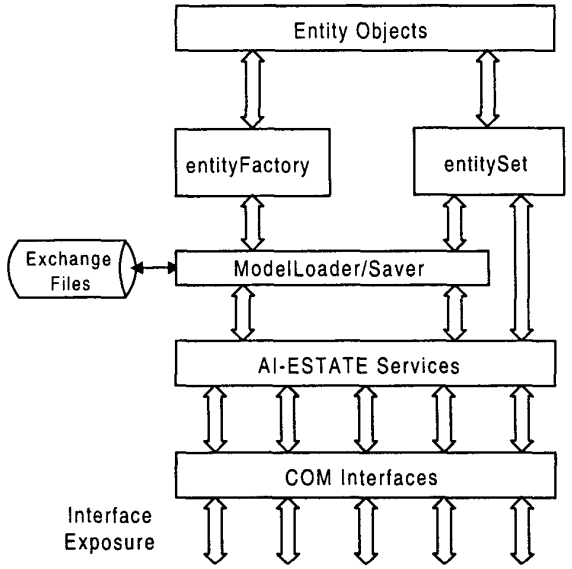

Figure 4 Architectural Overview of Model Management Component

CEntityFactory

The entityFactory object in Figure 4 is an instantiation of the CEntityFactory class. It is based on the *factory method* object-oriented design pattern [8]. It accepts a SPFF-encoded ISO 10303-21 [7] AI-ESTATE entity instance definition and creates the appropriate information entity object.

CEntityFactory's *create* method does all the work. It accepts a string containing the encoded entity definition and returns a pointer to the new entity object, or NULL if the object could not be created successfully. It also returns the internal ID of the new object.

The purpose of standardizing the exchange file is to facilitate the exchange of data and knowledge between diagnostic reasoners of different manufacture.

As specified in section 7.3.4 of [7], information entity instance names are encoded in the exchange file as a number sign, "#", followed by an unsigned decimal integer. This means that several models may contain entities with duplicate ID's. Our MMC handles the duplicate ID problem in the following way. The ID number is used as the internal entity's ID unless that ID value is already in use in which case it will be assigned a new ID. All references to the entity are also changed to the new value.

Following the entity instance name in the encoded entity instance definition is a keyword that identifies the type of the entity. This is what the entity factory uses to determine which type of object to instantiate. If the entity type is unrecognized, *create* will return a NULL pointer to indicate failure. The class that defines an entity object contains an overloaded constructor that accepts the parameter list of the encoded entity instance definition. It uses the constructor to initialize its own member variables.

CEntitySet

The entitySet object in Figure 4 is an instantiation of the CEntitySet class. It manages a set of objects that map entity IDs to the memory addresses of the associated entity objects. An entry is added to the set by calling the *addEntry* method, which accepts the ID and a pointer to the entity object. Method *removeEntry* accepts the ID of an entry object to be removed. Method *getEntity* accepts an ID and returns a pointer to the corresponding object.

Member variable *m_pEntitySet* is a pointer to the instantiation of the set itself. It is created using the standard template library (STL) **map** template [9]. There is only one entity set object. It is global. All AI-ESTATE service implementations refer to it for access to entities.

AI-ESTATE Entity Classes

The objects represented by the "Entity Objects" block of Figure 4 are instantiated from a class hierarchy that matches that of the EXPRESS entity definitions specified in [6]. The only difference between the two hierarchies is that the entity object classes of the MMC are all derived from a root abstract base class called CEntity. This is so we can manipulate the entity objects polymorphically.

AI-ESTATE Services COM Classes

There is a COM class associated with each entity type specified in [6]. The methods in those classes implement the AI-ESTATE Services [5]. The "AI-ESTATE Services" block in Figure 4 represents them. Only those services required for demonstration have been fully implemented. All others are "stubs" meaning that they exist and may be called but they don't do anything.

Extending the MMC for Optimization

The information model as prescribed by the IEEE 1232 standard does not consider the issues of inference optimization. That is, the information model contains all of the relevant *"information,"* but the formation of the model is organized around the relationships of *"information"* rather that the computational speed of inference. We are in the process of extending the DEC's capabilities to convert the Entity Objects into forms that support optimizations such as entropy directed searches. [10] To maintain compliance with the 1232 standard the DEC must contain an internal service switch that will be signaled to switch through DEC services (refer to Figure 5).

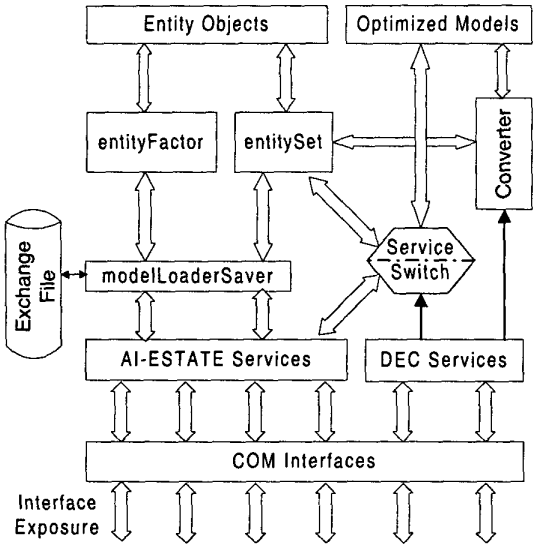

Figure 5 Optimization Extensions

Test System Component

The Test System Component (TSC) is a domain component written in C++ that provides standardized access to real world test instrumentation. The test instrumentation could be a single stand-alone instrument, a rack of VXI instrumentation, or some aggregation and configuration of both. Our Phase-II TSC inter-operates with LabVIEW, which orchestrates the activity of the instruments.

Application Executive Component

The Application Executive Component (AEC) is an application program written in Java that performs two primary functions:

- provides a graphical user interface (GUI) through which human users interact with the ATS and
- test sequence control.

Model Editing Component

The Model Editing Component (MEC) is an application program written in C++ that utilizes the OpenGL graphics library to perform two primary functions:

- provide the user with a 4GL-based diagnostic modeling tool and
- generate an exchange file ready for the Diagnostic Engine Component.

Test and Model Development

Test program development is a major concern for test engineers. There is a distinct difference between developing test programs and diagnostic models in monolithic ATS and componentized ATS.

Test Program Language

Since our set of domain components are operationally independent with regard to test languages the developer may choose the target test development language. If a Test System Component does not exist for the target language then one must be developed. This is a task similar in nature to developing an instrument driver. Our Phase-II system uses LabVIEW as the test language development package. Future releases will contain TSC's that use HP-VEE and ATLAS.

Diagnostic Development

Diagnostic development or "Test Program Set (TPS) development" methods are major concerns for test engineers. In the monolithic system the test engineer developed a sequence of interwoven calls to tests, diagnostic decisions, GUI actions and data base actions. In our set of domain components the test engineer will be developing a library of tests and a separate library of diagnostic models.

Development Cycle

The following list suggests the elements of a diagnostic development cycle based on our set of domain components. This list is far from complete. It does not include data base actions and other capabilities available in other available domain components.

- Analyze the system or device to test and diagnose.
- Create plan for development of required hardware, tests and diagnostic decisions.
- Construct any required hardware.
- Use the test language and a subset of an existing test library *OR* construct a new library of tests.
- Use the Model Editing Component to update or modify an existing diagnostic model *OR* construct a new diagnostic model.
- Verify developed tests and models.

Verification

You will notice this paper does not discuss in detail the issues of how to verify tests and models. These issues were not ready for publication at the time the paper was written. The reader will note, however, these are issues of deep concern and hold a high priority position in the project.

Test Library Development

There are three issues to discuss regarding test library development: developing tests in the selected test language, listing the tests available in the library and translating between diagnostic model information and test library test information.

Test Development

The issue of developing tests is the subject of the test language and so is considered outside the scope of this paper.

Listing Tests in the Library

As tests are developed the test engineer enters a mapping pair in the *tests map file*. The mapping pairs list the test names in the AI-ESTATE information model and their associated test library names. The TSC then uses the *tests map file* to map test names in the information model to test names in the test library.

Translating Information

A test or measurement in the test library may require a complex set of parameters as input data and may return a complex set of data forms as output. The diagnostic model does not encode parametric data that tests may require as input data and the model only requires a simple outcome evaluation of the test to make a diagnostic inference. Hence we must translate between the diagnostic model information and the test information.

The *tests map file* provides the first step in translation. There should be a one-to-one correspondence between a test in the AI-ESTATE diagnostic model and the high-level test names in the test library. When the diagnostic model selects the next test to perform the TSC maps the model name to the test language name and initiates a high-level test without parameters. Once the high-level test is initiated, the parameters are then applied to a lower-level test. Individual high-level tests in the library may in fact be a composite of low-level tests which, when run together, perform the logical test of the model. This is an issue of test library design.

The diagnostic model expects a return of pass, fail, test not known, test not available or some user defined values. These test outcome values are specified in the AI-ESTATE standard [6]. The second step in the translation occurs when the high-level test that the TSC initiated evaluates the results of the low-level test as pass, fail or user-defined. The TSC will return the evaluation of the low-level test or return a test not known or test not available.

Diagnostic Model Development

Diagnostic models are developed utilizing our 4GL Model Editing Component (MEC) (refer to Figure 2). Although we provide a graphical modeling editor any model editor provided by other vendors will work if the editor is able to save the

model in AI-ESTATE compliant exchange file format.

Benefits

A component-based approach to automatic test systems has several benefits over the monolithic program approach to automatic test systems. The following list of benefits provides for both advanced operation as well as lowering the total cost of a maintenance program.

- Tests stored in test program libraries may be re-used on numerous systems and devices with various diagnostic models.
- Diagnostic model development and editing are conducted without recompilation of the diagnostic system or any component within the set.
- Diagnostic models can be exchanged between IEEE 1232 (AI-ESTATE) compliant diagnostic reasoners.
- Models can be easily upgraded to match system, device and test equipment upgrades.
- Vendors can narrow their product development focus to domain components within their core capabilities.
- Competition among component vendors will result in high quality components at a lower price per component.
- The set of components can be operated over the Internet.
- Components are easily replaced without recompilation.
- The component-based approach to ATS is a ***true*** open systems architecture.

Summary

We started this paper with a discussion of our component-based approach to automatic test systems. The types of components we are building are system independent, which provides for the basis of a true open system. The open system of components can thus support the maintenance of digital avionics as well as any other types of maintenance programs. We showed how different configurations of components support variations of maintenance task operations. We showed how the construction of the Diagnostic Engine Component forms an IEEE 1232 (AI-ESTATE) compliant diagnostic reasoner. The standards-based reasoner supports the exchange of standardized data and knowledge. We briefly described the other three domain components of our system. We showed how a test program development engineer would develop both test programs and diagnostic models. And finally we discussed the benefits of using a component based automatic test system where the diagnostic reasoner is AI-ESTATE compliant.

References

[1] Dean, J. "AI-ESTATE Implementation," DoD SBIR Program, Vol. 98.1, Sol. No. AF98-252, 1997,

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

[3] Giarla, A., "Implementing AI-ESTATE in a Component Based Architecture, Phase-I," Proceedings of Systems Readiness Technology Conference, AUTOTESTCON 1999, IEEE 1999, ISBN 0-7803-5432-X.

[4] Szyperski, C., 1997. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley

[5] IEEE Standard 1232.2-1998, *IEEE Trial-Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Service Specification.*

[6] IEEE Standard 1232.1-1997, *IEEE Trial-Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Data and Knowledge Specification.*

[7] ISO standard 10303-21, *Industrial automation systems and integration – Product data representation and exchange – Part 21: Implementation methods: Clear text encoding of the exchange structure.*

[8] Erich Gamma ... [et al.]. *Design Patterns – Elements of Reusable Object-Oriented Software*, pages 107-116. Addison-Wesley, Reading, MA, December 1995.

[9] Alexander Stepanov and Meng Lee. *The Standard Template Library*, pages 36-37. Hewlett-Packard Co., October 31, 1995.

[10] Simpson, W., Sheppard, J., *System Test and Diagnostics*, 1994, Kluwer Academic Publishers.