

IEEE Guide for the Use of Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)

IEEE Standards Coordinating Committee 20

Sponsored by the
IEEE Standards Coordinating Committee 20 on
Test and Diagnosis for Electronic Systems

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1232.3™-2014

IEEE Guide for the Use of Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)

Sponsored by the

**IEEE Standards Coordinating Committee 20 on
Test and Diagnosis for Electronic Systems**

Approved 20 August 2014

IEEE-SA Standards Board

Abstract: Guidance to developers of IEEE Std 1232-conformant applications is provided in this guide. A simple doorbell is used as an example system under test to illustrate how the static model constructs of Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE) are used to form a diagnostic reasoner knowledge base. Each of AI-ESTATE's knowledge base types is discussed in conceptual terms, and how those concepts are represented in exchange files is shown. Also, some of the nuanced aspects of diagnostic knowledge bases in AI-ESTATE are clarified. An example reasoner session is provided to illustrate the use of AI-ESTATE services.

Keywords: AI-ESTATE, Bayesian Network, diagnosis, diagnostic inference, diagnostic model, diagnostic services, D-matrix, fault tree, IEEE 1232.3™, knowledge exchange, reasoner, system test

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2014 by The Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 10 October 2014. Printed in the United States of America.

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by The Institute of Electrical and Electronics Engineers, Incorporated.

PDF: ISBN 978-0-7381-9322-9 STD98797
Print: ISBN 978-0-7381-9323-6 STDPD98797

IEEE prohibits discrimination, harassment, and bullying.

For more information, visit <http://www.ieee.org/web/aboutus/whatis/policies/p9-26.html>.

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page, appear in all standards and may be found under the heading “Important Notice” or “Important Notices and Disclaimers Concerning IEEE Standards Documents.”

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (“IEEE-SA”) Standards Board. IEEE (“the Institute”) develops its standards through a consensus development process, approved by the American National Standards Institute (“ANSI”), which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied “AS IS” and “WITH ALL FAULTS.”

Use of an IEEE standard is wholly voluntary. The existence of an IEEE standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854 USA

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE-SA Website at <http://ieeexplore.ieee.org/xpl/standards.jsp> or contact IEEE at the address listed previously. For more information about the IEEE-SA or IEEE's standards development process, visit the IEEE-SA Website at <http://standards.ieee.org>.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE-SA Website at the following URL: <http://standards.ieee.org/findstds/errata/index.html>. Users are encouraged to check this URL for errata periodically.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at <http://standards.ieee.org/about/sasb/patcom/patents.html>. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this IEEE guide was completed, the AI-ESTATE Users Guide Working Group had the following membership:

Simon Jessop, *Chair*
Darryl Busch, *Co-Chair*

Michael Bodkin
Nathan Fortier

Mike Seavey
John Sheppard

Liessman Sturlaugson
Timothy Wilmering

The following members of the individual balloting committee voted on this guide. Balloters may have voted for approval, disapproval, or abstention.

Malcom Brown
Darryl Busch
Keith Chow
William Frank
Randall Groves
Werner Hoelzl
Noriyuki Ikeuchi
Anand Jain

Albert Livshitz
Teresa Lopes
Mukund Modi
Ion Neag
Alexandros Nikitas
Leslie Orlidge
Ulrich Pohl

Bartien Sayogo
Mike Seavey
John Sheppard
Joseph Stanco
Walter Struppler
Ronald Taylor
Timothy Wilmering
Daidi Zhong

When the IEEE-SA Standards Board approved this guide on 21 August 2014 it had the following membership:

John Kulick, *Chair*
Jon Walter Rosdahl, *Vice-chair*
Richard H. Hulett, *Past Chair*
Konstantinos Karachalios, *Secretary*

Peter Balma
Farooq Bari
Ted Burse
Clint Chaplain
Stephen Dukes
Jean-Philippe Faure
Gary Hoffman

Michael Janezic
Jeffrey Katz
Joseph L. Koepfinger*
David Law
Hung Ling
Oleg Logvinov
T. W. Olsen
Glenn Parsons

Ron Peterson
Adrian Stephens
Peter Sutherland
Yatin Trivedi
Phil Winston
Don Wright
Yu Yuan

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Richard DeBlasio, *DOE Representative*
Michael Janezic, *NIST Representative*

Don Messina
IEEE-SA Content Publishing

Patricia Gerdon
IEEE-SA Technical Community Programs

Introduction

This introduction is not part of IEEE Std 1232.3™-2014, IEEE Guide for the Use of Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE).

IEEE Std 1232™-2010, IEEE Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE) defines formal information models and software services specific to several different types of diagnostic reasoners to enable unambiguous access to and understanding of the information supporting system test and diagnosis. This guide supplements IEEE Std 1232-2010 and is part of the document set.

This guide has been prepared to help all users of IEEE Std 1232-2010. IEEE Std 1232-2010 may be used within any test discipline and with any carrier language, and the examples provided in this guide should be seen as typical.

This guide explains AI-ESTATE features and application using a consistent simple doorbell system example. It describes the static diagnostic model information contained in the Common Element Model (CEM) and four additional specialized diagnostic model types. The guide also shows how the standard supports the dynamic operation of the reasoner using standard services.

The need for a guide arose from the experience of users in the creation of diagnostic models and applications using IEEE Std 1232-2010. This experience showed that further information and examples on the implementation and application of the standard was required. The purpose of this guide is to provide guidance in the technique of implementation, application, and usage of the basic diagnostic models defined in IEEE Std 1232-2010 to create diagnostic models and reasoning applications. This is seen as particularly important in promoting the use of a relatively new and unambiguous method of describing diagnostic information. This guide is not intended to be used as an instruction manual for IEEE Std 1232-2010 nor as a substitute for formal training, but by its nature it should find some application in the training environment.

Clause 5 is intended as a brief introduction to the history and development of AI-ESTATE. Clause 7 concentrates on the description of the static diagnostic models used to describe system test and diagnosis information. The Common Element Model (CEM), Fault Tree Model (FTM), Bayesian Network Model (BNM), Dmatrix Inference Model (DIM), and Diagnostic Logic Model (DLM) are all covered. Clause 8 provides a description of the AI-ESTATE reasoner services and provides a diagnostic session sequence example.

IEEE Std 1232-2010 does not specify any particular software language, interface bindings, or client-reasoner communication framework, and this guide does not provide any advice on this topic.

Contents

1. Overview	1
1.1 Scope	1
1.2 Purpose	1
1.3 Word usage	2
1.4 IEEE download site	2
2. Normative references	2
3. Definitions, acronyms, and abbreviations	2
3.1 Definitions	2
3.2 Acronyms and abbreviations	2
4. Background	3
5. Use cases	5
5.1 Introduction	5
5.2 Static model exchange	5
5.3 Technology insertion	5
5.4 Session data export	5
6. Introduction to the doorbell example	5
7. Static models	7
7.1 Overview	7
7.2 Common Element Model	8
7.3 Fault Tree Model	39
7.4 Bayesian Network Model	44
7.5 Dmatrix Inference Model	56
7.6 Diagnostic Logic Model	67
8. Reasoner services	79
8.1 Introduction to services	79
8.2 Reasoner states	80
8.3 Reasoner status codes	82
8.4 Diagnostic session sequence examples	83
Annex A (informative) Bibliography	89
Annex B (normative) Information object registration	90

IEEE Guide for the Use of Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)

IMPORTANT NOTICE: *IEEE Standards documents are not intended to ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. Implementers of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.*

This IEEE document is made available for use subject to important notices and legal disclaimers. These notices and disclaimers appear in all publications containing this document and may be found under the heading “Important Notice” or “Important Notices and Disclaimers Concerning IEEE Documents.” They can also be obtained on request from IEEE or viewed at <http://standards.ieee.org/IPR/disclaimers.html>.

1. Overview

1.1 Scope

IEEE Std 1232.3™-2014 provides guidance to developers of IEEE Std 1232-conformant diagnostic applications that utilize the models, services, or exchange formats specified by IEEE Std 1232.

1.2 Purpose

IEEE Std 1232 addresses the complex domain of system test and diagnosis and specifies formal semantic models for information exchange and services to be provided by diagnostic reasoners in this domain. The formality of the semantic models necessitates an in-depth understanding of the associated concepts to ensure that interoperability and interchangeability requirements can be satisfied. IEEE Std 1232.3 provides this understanding by describing the history, motivation, and key concepts from AI-ESTATE to help developers extract maximum benefit from IEEE Std 1232. In addition, examples of instantiated models, exchange files, and service scenarios are provided.

1.3 Word usage

In this document, the word *shall* is used to indicate a mandatory requirement. The word *should* is used to indicate a recommendation. The word *may* is used to indicate a permissible action. The word *can* is used for statements of possibility and capability.

1.4 IEEE download site

The schemas and examples that accompany this guide are available on the Internet at <http://standards.ieee.org/downloads/1232>.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1232-2010, IEEE Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE).¹

3. Definitions, acronyms, and abbreviations

3.1 Definitions

For the purposes of this document, the terms and definitions from IEEE Std 1232-2010 apply. The *IEEE Standards Dictionary Online*² should be consulted for terms not defined in IEEE Std 1232-2010.

3.2 Acronyms and abbreviations

AI-ESTATE	Artificial Intelligence Exchange and Service Tie to All Test Environments
BIT	built-in-test
BNM	Bayesian Network Model
CEM	Common Element Model
DCM	Dynamic Context Model
DIM	Dmatrix Inference Model

¹ IEEE publications are available from The Institute of Electrical and Electronics Engineers (<http://standards.ieee.org/>).

² *IEEE Standards Dictionary Online* subscription is available at:
http://www.ieee.org/portal/innovate/products/standard/standards_dictionary.html.

DLM	Diagnostic Logic Model
FTM	Fault Tree Model
RSM	Reasoner Service Model
SCC20	Standards Coordinating Committee 20
SUT	system under test
XML	eXtensible Markup Language

4. Background

The AI-ESTATE standard provides formal models of diagnostic information to enable unambiguous access to and understanding of the information supporting system test and diagnosis. The standard defines formal information models and software services specific to several different types of diagnostic reasoners. The purpose is to provide semantically sound definitions of diagnostic knowledge and to specify software exchange and service interfaces that are consistent with the state of the practice in modern test and diagnostic systems (e.g., the use of eXtensible Markup Language [XML] and web services).

In 1976, the IEEE established the Standards Coordinating Committee 20 (SCC20) for purposes of standardizing on the Abbreviated Test Language for All Systems (ATLAS). Since then, SCC20 has expanded its scope to develop standards for larger system-level test and diagnostic related systems. In 1989, the IEEE approved a project authorization request (PAR) for SCC20 to develop a new standard focusing on diagnostic systems that use techniques from the maturing field of artificial intelligence—the Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE) standard under project P1232. In 1995, SCC20 approved and published the AI-ESTATE standard, IEEE Std 1232-1995. The name was changed to the current name for the publication of the 1995 standard. The entire AI-ESTATE standard encompassed IEEE Std 1232-1995 (Overview and Architecture), IEEE Std 1232.1-1997 (Data and Knowledge Specification), and IEEE Std 1232.2-1998 (Service Specification). The three standards were merged into a single standard for the 2002 version. The standard was updated in 2010.

Two of the primary motivations behind adopting consensus standards for diagnostic system development are to reduce cost by improving interoperability and to minimize repeated design of similar systems. The IEEE SCC20 standards focus on promoting information interoperability between components of a test or health monitoring system. The emphasis by government and commercial entities on acquisition reform based on commercial standards for diagnostic systems, combined with declining budgets, mandates the need for more affordable health management system development and operation.

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.

AI-ESTATE specifies models using the ISO 10303-11 [B1] EXPRESS information modeling language. Within EXPRESS, models are defined using a simple hierarchy partitioned along schemata, entities, and attributes. Furthermore, legal values of attributes are defined through constraints on those attributes. The scope of the language is to define the information to be used or generated by a system or process and is not intended to define database formats, file formats, or exchange formats. Such formats are specified in a related set of standards, including those used by AI-ESTATE for model exchange: ISO 10303-21 [B2] and [B3], and ISO 10303-28 [B5].

In addition to the lexical EXPRESS language, ISO 10303-11 provides for presenting EXPRESS models in a graphical form using EXPRESS-G. Figure 1 is an example of a portion of an information model in

EXPRESS-G. To read this figure, the solid rectangles correspond to entities while the dashed rectangles are defined types. The principal distinction between entities and defined types is that entities are complex types with attributes that relate them to other entities or types within the model. For example, the entity *Test has two attributes—subAction and allowedOutcomes. allowedOutcomes is of type TestOutcome. The attributes are shown as solid lines with small bubbles on the ends. The bubbles identify the type of the attribute. Defined types, on the other hand, associate meaningful labels with base types such as strings, integers, or Booleans. More information on EXPRESS-G can be found in Annex B of IEEE Std 1232-2010 or Schenck and Wilson [B7].

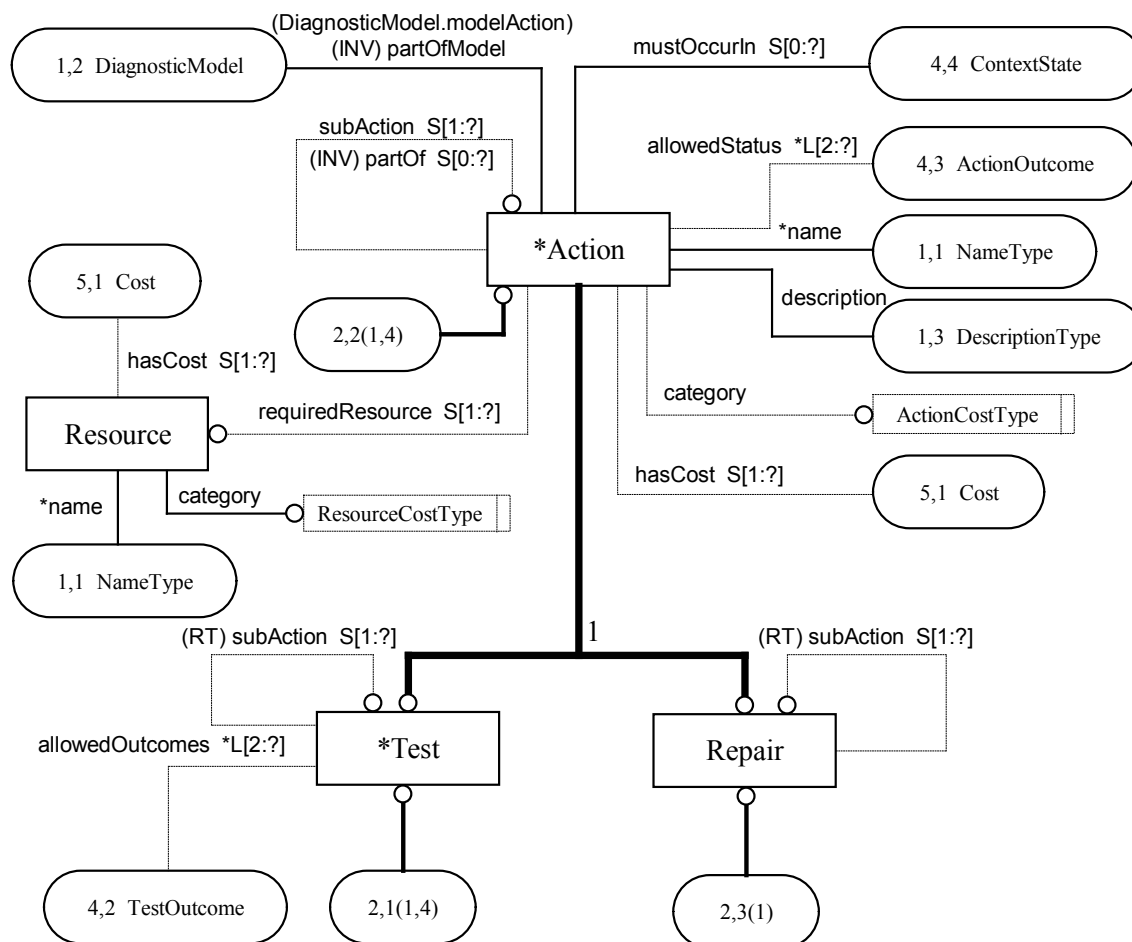


Figure 1—Example EXPRESS-G information model

The EXPRESS language incorporates a number of object-oriented features, such as encapsulation, abstraction, and inheritance. Encapsulation comes from the specification of schemata where concepts specific to a domain or subdomain are contained within a single schema. Concepts from other schemata can be used through the USE and REFERENCE interface. Abstraction is supported through the specification of entity or class hierarchies where specialization occurs through subtyping. Attributes and constraints defined for a supertype are inherited by all of the subtypes.

Perhaps the most significant feature of the EXPRESS language is in the definition of mathematical and logical constraints. EXPRESS supports defining constraints in two ways: through rules (which are applied globally in a schema) and through “where” clauses (which are applied to attributes of an entity). It is through these constraints that much of the computer-processable semantics are defined. By constraining

relationships and legal values, often in nontrivial ways, applications are able to discern if the information being received satisfies the intended meaning when it was generated and transmitted.

5. Use cases

5.1 Introduction

The following diagnostic use cases highlight the value of AI-ESTATE conformance at a system level. These cases illustrate the primary motivations behind adopting the standard: to reduce cost by improving interoperability and to minimize repeated design of similar systems.

5.2 Static model exchange

Utilizing the AI-ESTATE standard, two separately developed conformant diagnostic applications may exchange diagnostic model information without modification. Modifications to the model or the application are normally necessary in order to translate the model information into the new application format, resulting in added cost and complexity. Utilizing the standard, this model exchange is performed without any modifications thereby promoting open information exchange without exposing application details while reducing cost. The new application is able to immediately load and use the diagnostic information provided by the first application.

5.3 Technology insertion

Users have need to upgrade their reasoning applications, but doing so also requires migration of existing diagnostic knowledge to the new application format. AI-ESTATE defines the technology-independent interaction between the client diagnostic application and the reasoner application. This allows client application upgrades and changes while maintaining reasoner application functionality. Similarly, this allows a user to upgrade their reasoner independent of the client application, including changing the underlying reasoning technology.

5.4 Session data export

Detailed records of the diagnostic session provide valuable maintenance process information that can be used for continuous process analysis and improvement. The Dynamic Context Model (DCM) provides a structured record of each step performed during diagnosis. The consistent data model and format allows the development of automated management and analysis tools for process monitoring and process improvement.

6. Introduction to the doorbell example

This guide describes the features of IEEE Std 1232-2010 using a simple doorbell example. The doorbell circuit example, shown in Figure 2, is simple enough to understand, but also provides enough complexity to illustrate the important aspects of the standard. The circuit elements, proper functioning, expected failures, and test methods are described below.

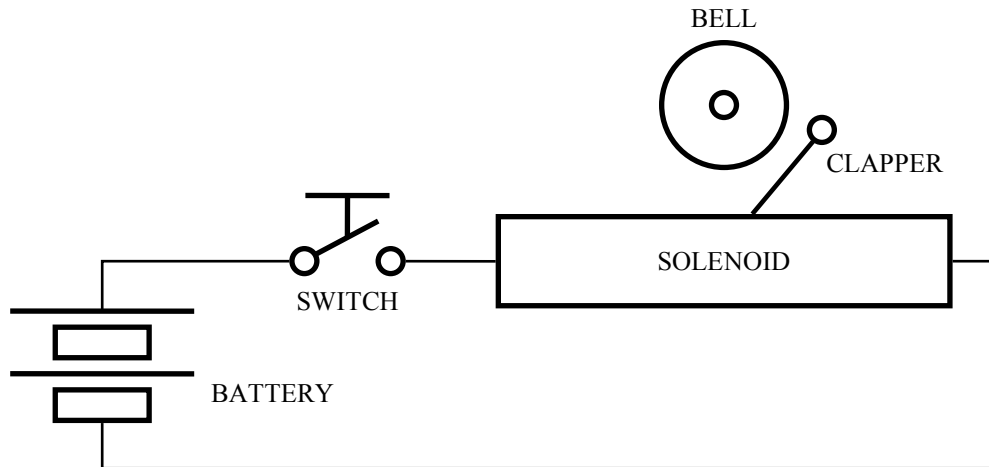


Figure 2—Doorbell circuit schematic

The circuit contains three main items: a clapper-type bell, a push button-type switch, and a battery. The circuit wiring is ignored for brevity. Proper operation consists of the following. Pressing the switch completes the circuit and allows the battery to energize the solenoid in the clapper bell. The solenoid drives the clapper to strike the bell. When the clapper moves to strike the bell, it breaks the circuit and cuts power to the solenoid and a spring pulls the clapper back. The solenoid reenergizes and the clapping sequence repeats for as long as the switch is pressed.

This circuit can fail in several ways. The battery can die, the switch can fail to make contact, the switch can jam closed, the clapper can be stuck in the open circuit position, the clapper can be stuck in the closed circuit position, or the solenoid can be open circuit. These are listed in Table 1. The Name column gives a unique name for each fault that is used in the remainder of this guide to refer to each fault. The Name and Description columns correspond to fields that are present in AI-ESTATE models and exchange files.

Table 1—Doorbell faults

Name	Description
<i>SW-C</i>	Switch sticks closed
<i>SW-O</i>	Switch stuck open
<i>Batt</i>	Battery dead or low
<i>Sol-O</i>	Solenoid open circuit
<i>Clap-O</i>	Clapper stuck open circuit
<i>Clap-C</i>	Clapper stuck closed circuit

The circuit can be tested in a number of ways. Table 2 provides a list of the doorbell tests. The Name column gives a unique name for each test that is used in the remainder of this guide to refer to each test. The Comment column provides more detail for each test, including a description of its pass/fail logic.

Table 2—Doorbell tests

Name	Description	Comment
<i>push</i>	Push the button on the bell circuit system and verify the bell sounds.	This test covers most of the doorbell system, excluding <i>SW-C</i> . Its pass/fail logic is: Fail if one or more of { <i>SW-O</i> , <i>Batt</i> , <i>Sol-O</i> , <i>Clap-O</i> , <i>Clap-C</i> } is present. Pass if all of { <i>SW-O</i> , <i>Batt</i> , <i>Sol-O</i> , <i>Clap-O</i> , <i>Clap-C</i> } are absent.
<i>push/push</i>	Push the button multiple times to check if it sticks closed.	This test covers the switch in isolation, and only tests for <i>SW-C</i> . It uses touch sensation to check if the switch tends to stick closed, without relying on sounding the doorbell. Its pass/fail logic is: Fail if <i>SW-C</i> is present. Pass if <i>SW-C</i> is absent.
<i>volt</i>	Check the battery with a battery tester.	This test covers the battery in isolation. Its pass/fail logic is: Fail if <i>Batt</i> is present. Pass if <i>Batt</i> is absent.
<i>bridge</i>	Apply a bridge across the button and verify the bell sounds.	By bridging the switch, this test checks operation of the rest of the doorbell system excluding the switch. Its pass/fail logic is: Fail if one or more of { <i>Batt</i> , <i>Sol-O</i> , <i>Clap-O</i> , <i>Clap-C</i> } is present. Pass if all of { <i>Batt</i> , <i>Sol-O</i> , <i>Clap-O</i> , <i>Clap-C</i> } are absent. This test covers the same set of faults as the <i>push</i> test, excluding <i>SW-O</i> , and thus allows one to isolate <i>SW-O</i> by process of elimination when <i>push</i> fails and <i>bridge</i> passes. It does not cover the switch in isolation in the sense that <i>volt</i> and <i>stim</i> cover the battery and clapper bell in isolation, respectively.
<i>stim</i>	Apply a stimulus across the bell and verify it sounds.	This test covers the clapper bell in isolation. Its pass/fail logic is: Fail if one or more of { <i>Sol-O</i> , <i>Clap-O</i> , <i>Clap-C</i> } is present. Pass if all of { <i>Sol-O</i> , <i>Clap-O</i> , <i>Clap-C</i> } are absent.

7. Static models

7.1 Overview

IEEE Std 1232-2010 defines four technology-specific “static” exchange model types for use during diagnostic reasoner sessions:

- Bayesian Network Model (BNM)
- Dmatrix Inference Model (DIM)
- Diagnostic Logic Model (DLM)
- Fault Tree Model (FTM)

The static model types are governed by their respective EXPRESS schemata. This guide refers to an exchange file that conforms to one of these static model types as a BNM instance, DIM instance, DLM instance, or FTM instance, respectively, or as a static model instance in general. A static model instance contains the knowledge base necessary for a reasoner to form diagnostic conclusions during a diagnostic session based on the test observations provided by the client application. These four exchange model types are called “static” because the knowledge base is not intended to change during a diagnostic session, and can in fact remain unchanged and be reused over many sessions and over a fleet of many individual systems. In contrast, a DCM instance is generated live during a session from the information accumulated during that session, and each diagnostic session generates another DCM instance. Users are free to update their static model instances over time. However the update process is outside of the scope of AI-ESTATE and updates are expected to occur outside of any diagnostic session.

The knowledge base in a static model instance is applicable to a particular SystemItem being the system under test (SUT). The same knowledge base can be used to diagnose all instances (i.e., all serial numbers) of the specified SystemItem.

To understand the role of a static model instance in a diagnostic session, it is helpful to understand what it contains and what information is held elsewhere. A static model instance contains such things as:

- The set of actions that could be performed during the session, particularly tests and repairs, and potentially also setup actions
- The set of possible diagnostic conclusions
- The relationships that allow a reasoner to make diagnostic conclusions from test evidence
- Parameters for determining a course of action to take, including restrictions and optimization parameters

External to the static model instance, the reasoner may have algorithms and/or meta-rules that allow it to compute actual conclusions using the static model instance and dynamic information provided by the client via reasoner services. While a static model instance is applicable to a specific SystemItem, the algorithms and meta-rules are generally applicable to all SystemItems, and may be hard-coded into the reasoner.

External to the reasoner are one or more applications that interact with the reasoner. These are referred to as the client-side. The client-side has its own static and dynamic information that it needs to fulfill its role in the diagnostic session, including the following:

- Perhaps the most important static client-side information is the detailed procedural knowledge necessary to perform any action that the reasoner might recommend and to compute the outcome for each action. This procedural knowledge may be very extensive, such as the tech manual for the system under test, or a test program set in the case of automated testing. At the other extreme, a human on the client-side might simply be able to perform each action intuitively based on the action name and description provided by the reasoner.
- The client knows the name (e.g., the make and model of car) and identifier (e.g., the vehicle identification number of the car) of the system under test, and provides that information to the reasoner at the start of the session. The client also provides either the name or the URL of the static model instance that the reasoner is to use during the session.
- If applicable, the client knows the discrepancies that were observed prior to the current session, which led to the need to diagnose the system under test in the current session. The client provides this information to the reasoner using the setDiscrepancies service.

7.2 Common Element Model

7.2.1 Overview

The AI-ESTATE Common Element Model (CEM) schema defines data types and relationships that are common to all static reasoner models within AI-ESTATE, including definitions for Tests, Functions, Faults, Failures, Costs, etc. The technology-specific schemata, BNM, DIM, DLM, and FTM, then import and extend the CEM by adding the specialized types and relationships that drive their particular methods of reasoning. AI-ESTATE does not define conformant exchange of data that conform only to the CEM schema. In particular, the CEM does not define the relationships necessary for any reasoning method to form diagnostic conclusions from test observations. Users can think of the CEM schema as defining an abstract supertype for reasoner knowledge bases, where “abstract” is used in the object-oriented sense that

the CEM is not intended to be instantiated, and where each technology-specific schema defines a subtype knowledge base that can be instantiated.

A second way to think of the CEM is as a library of reusable type definitions that can be used by other schemata. Some CEM types are used this way in the DCM and Reasoner Service Model (RSM) for instance. This treatment is appropriate for the simpler types defined in the CEM, but is impractical for many of the entity types because they have cardinality constraints on relationships with other CEM entities that hinder a schema developer's flexibility to reuse a particular CEM entity. For example, a schema developer wanting to reuse the CEM Test entity would also need to import DiagnosticModel, which in turn needs the import of Diagnosis, and SystemItem (RepairItem or FunctionItem). An instance of the developed schema would need to have instances of these other entities whether they are wanted or not.

Figure 3 shows the portion of the doorbell static model instance that is governed by the CEM schema. Each of the technology-specific models described in subsequent clauses—FTM, BNM, DIM, and DLM—have these entity instances with some extensions.

As can be seen in Figure 3, a static model instance can be thought of as a network of nodes where the nodes are entity instances (the blocks in the figure), and entity attributes that reference entities are the network arcs. The upper text in each block states the entity type, e.g., DiagnosticModel. The lower text in each block gives the value of the entity's name attribute if such an attribute exists for the entity. For the TestOutcome and DiagnosisOutcome entities, which do not have a name attribute, the lower text in each block gives the PASS/FAIL/GOOD/BAD outcome value. Other entity attributes are omitted for space considerations.

The CEM entities and relationships, exemplified in Figure 3, are discussed in subsequent subclauses within 7.2.

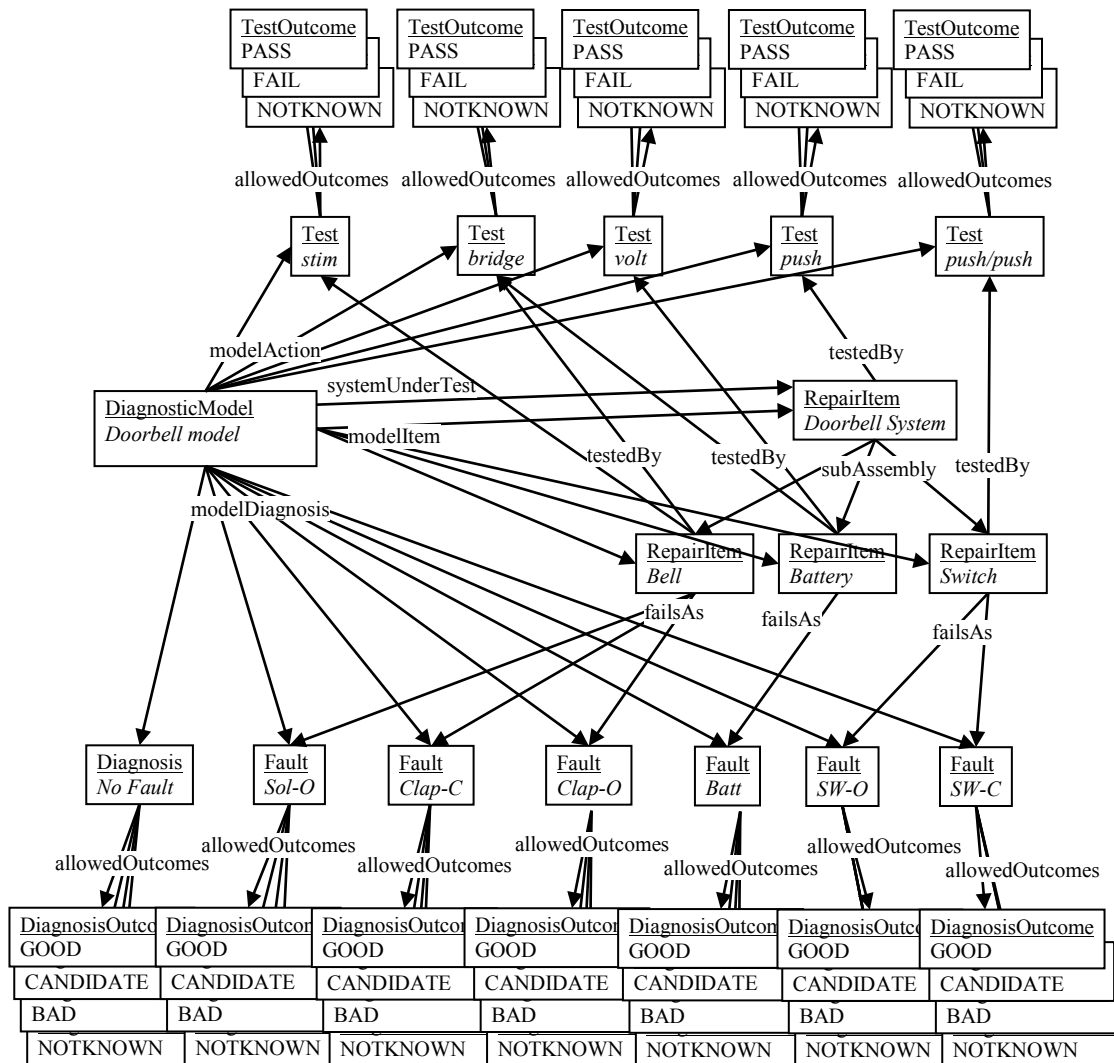


Figure 3—Doorbell static model content governed by the CEM

7.2.2 Names and descriptions

Many of the entity types defined in the CEM have a name attribute including: Action, Test, Repair, Diagnosis, Fault, Failure, SystemItem, RepairItem, FunctionItem, ContextState, ModeOfOperation, CostCategory, DiagnosticModel, FailureMechanism, Level, and Resource. These name attributes hold special significance within AI-ESTATE because they are identifiers that need to be recognized across the reasoner-client interface. For example, when a reasoner recommends actions to the client via the recommendActions service, it does so passing the names of those Actions, Tests, and Repairs. The client recognizes those names and knows how to perform the actions by name. When the client communicates to the reasoner that it has performed those action via the applyActions service, it does so by passing the names of the actions, which the reasoner recognizes. Other services use names in a similar way.

The CEM states that a name “is also human readable,” but this is not enforced by the CEM schema. There is no expectation that names be in common English or any other language. Names may have

implementation-specific internal structure, such as alpha-numeric codes for skill levels, maintenance level, breakdown of the system under test, versioning, etc. Such structured names can help facilitate development and management of static model instances and/or client-side activities during a session. However, reasoners are not required to recognize internal structures in names. Reasoner-side behavior that relies on structured information within name strings will need an agreement between parties.

The CEM definition for NameType allows an empty string. However, static model instances should use non-empty strings for all NameType attributes.

NOTE 1—Future versions of AI-ESTATE might require that names be non-empty strings.³

The primary role for description attributes is that they are part of the outputs of the reasoner to the client. For example, the reasoner service getDiagnosticResults outputs not only the names of Diagnoses, but also their corresponding descriptions. The reasoner service recommendActions outputs not only the names of Actions, but also their corresponding descriptions. The client could find these descriptions useful, perhaps by displaying them to a human reader, or perhaps by parsing them for computer-interpreted information. There can be applications where there is little or no client-side test program or technical manual, and all client-side activity is reliant on a human reading and acting on the description text that the reasoner outputs. In this scenario, the description strings may be quite lengthy and may even have structured text.

A secondary role for description attributes is that they can be used by model developers to document the entity instances in the model. The level of detail for the documentation is implementation specific. Consider the list of doorbell tests in Table 2 for example. The column labeled Description gives a brief description of each test, and these could be used as a minimalist description attribute of each Test in the static model instance. Model developers who want more complete documentation, however, might concatenate the contents of the Description and Comment cells and use that for the description attribute. Model developers might even choose to include process data in the description, such as developer names, versioning, and information about the stage in the development process.

Reasoners are not required to utilize description attributes in static models, except to pass description strings as-is to the client via services. Static model instances may even have empty description attributes. Reasoner behavior that relies on the contents of description attributes will need an agreement between parties. There is one possible exception to simply passing description strings as-is: a reasoner may need to convert description strings—which are legal strings in an ISO 10303-21 [B2] or ISO 10303-28 [B5] exchange file—into legal strings in the implementation-specific framework the reasoner uses for services. This may include converting any escape sequences in the description strings, for example.

NOTE 2—The same static model entities that have a name attribute generally also have a description attribute. The exceptions are the Purpose entity, which has a description but not a name, and the Resource entity, which has a name but not a description. Both of these are oversights that might be corrected in future versions of IEEE Std 1232.

7.2.3 DiagnosticModel

The DiagnosticModel instance represents a “container” for all the information in the static model instance. AI-ESTATE sometimes describes other entities as being “in the DiagnosticModel.” The prominent role of DiagnosticModel is evident Figure 3, where its attributes modelItem, modelAction, and modelDiagnosis signify containment by pointing explicitly to all instances of RepairItem, Test, Fault, FunctionItem, Action, Diagnosis, and Failure (the latter four are not used in the doorbell example). These entities in turn have attributes that point to other entity types such as TestOutcomes and DiagnosisOutcomes in Figure 3. These are also said to be contained by DiagnosticModel, and so on, for all entity instances in the network of relationships emanating from DiagnosticModel.

³ Notes in text, tables, and figures of a standard are given for information only and do not contain requirements needed to implement this standard.

NOTE—The ISO 10303-21 and ISO 10303-28 exchange formats use identifier references to represent relationships between entity instances. The entities contained by a DiagnosticModel are not literally nested within it.

Each static model instance, be it a BNM, FTM, DIM, or DLM, has exactly one instance of the entity DiagnosticModel. This constraint is enforced by a RULE constraint in each of the technology-specific schemata. The DiagnosticModel instance will be the specialized subtype of DiagnosticModel that is defined in the technology-specific schema, such as a BayesNetworkModel or a DmatrixInferenceModel. The CEM's definition of DiagnosticModel could be thought of as abstract as far as static model exchange is concerned. However DiagnosticModel is not declared an abstract type, and indeed DCM instances have DiagnosticModel instances that are not subtypes.

DiagnosticModel has a name attribute. The name of the DiagnosticModel only comes into play during a reasoner session when loading the static model instance. There are two alternative services. The client can use the loadDiagnosticModel service, which instructs the reasoner to load a static model instance based on the DiagnosticModel.name value provided by the client. The reasoner handles the details of how and where the instance is stored and accessed. The expectation is that the client will know the name of the DiagnosticModel that is appropriate to use in each situation. The client has the alternative of using the loadDiagnosticModelFromLocation service instead; in which case the service returns the value of the DiagnosticModel.name attribute. For this service, the expectation is that the client will know the URI of the static model instance that is appropriate to use in each situation.

DiagnosticModel has a required attribute systemUnderTest that indicates the intended SUT for the static model instance. During a diagnostic session, a reasoner uses the static model instance if the actual SUT for the session matches the systemUnderTest attribute. The systemUnderTest attribute indicates the intended SUT by pointing to a SystemItem entity (a RepairItem or a FunctionItem) in the static model instance, which this guide refers to as the SUT RepairItem or SUT FunctionItem. That SystemItem is required to have no hierarchical parents per the noParent constraint in the CEM schema. In Figure 3 the systemUnderTest attribute points to the RepairItem named *Doorbell System*, which represents the overall doorbell system.

In a typical static model instance the systemUnderTest attribute will point to a RepairItem, as opposed to a FunctionItem. This approach is appropriate when the static model instance is intended to be used to diagnose a physical system under test, such as a vehicle, engine, or electronic box. The same static model instance can be used in multiple diagnostic sessions that test a particular class of system (e.g., a particular make and model) and with various instances of that class (e.g., various serial numbers). The SUT RepairItem denotes the class of system under test, not the instance of the class.

The approach of having systemUnderTest point to a RepairItem can also be used for non-physical systems, such as software systems, that nevertheless fit into that mold.

The systemUnderTest attribute can alternately point to a FunctionItem in the static model instance. One use case is if the intended use is for testing or operational monitoring of a subset of functions of a plant, but not the entire plant. The plant functions are represented in the static model instance by the SUT FunctionItem, its descendant FunctionItems, and the associated Failures. This choice of systemUnderTest indicates to the reasoner that the focus of the diagnostic session is to test that the plant functions are GOOD or BAD, not to isolate any failures to a RepairItem.

A static model instance can contain both a RepairItem hierarchy and FunctionItem hierarchy representation of the system under test. However, the systemUnderTest attribute can only point to one RepairItem or one FunctionItem. When both hierarchies are included in a static model instance, the intention is that these two representations be thoroughly interconnected via the attribute and inverse attribute pair: RepairItem.includesFunction and FunctionItem.implementedBy, so that the reasoner can produce conclusions for both.

7.2.4 RepairItems

RepairItems represent a part of the system under test, or the entire system under test, that can be acted on by a Repair action. RepairItems often represent physical items, though they are also intended to represent non-physical systems under test, and non-physical parts of a physical system; a typical example being software. In contrast, FunctionItems represent the functional aspects of the system under test. The twin representations of the system, physical and functional, have a legacy in defense acquisitions. They also have a legacy in different approaches to diagnostic modeling.

RepairItems are intended to be organized into a RepairItem hierarchy in the static model instance, where the children form the parts of the parent. In particular, if the `systemUnderTest` attribute of `DiagnosticModel` points to a SUT RepairItem, then its hierarchical descendant RepairItems represent the parts breakdown of the SUT. In the doorbell content in Figure 3, the SUT RepairItem is the *Doorbell System*, which in turn is composed of three parts: a *Switch*, a *Battery* and a *Bell*. Wiring and connectors are left out of the model for simplicity.

The CEM imposes few restrictions on the RepairItem hierarchy other than it is acyclic. Recipients of static model instances should be prepared to handle the range of possibilities. There can be any depth to the hierarchy, and depth can vary across the hierarchy. There can be more than one root RepairItem in a static model instance. If there is more than one root, their respective hierarchies may be connected or disconnected. RepairItems can have zero, one, or more parents. A RepairItem can be a descendant of more than one root. There can be more than one path between a given pair of ancestor-descendant RepairItems. The depth of a RepairItem in the hierarchy can depend on the path taken from the root, and also on the root that was chosen. Of these possibilities, two that are expected to be relatively common in static model instances are multiple root RepairItems and hierarchies with varying depth. They are discussed later in this subclause.

It is at the model creator's discretion whether to model the system under test with RepairItems, FunctionItems, or both. When RepairItems are included, they can have varying degrees of impact on reasoner sessions depending on how interconnected the RepairItem hierarchy is with other important parts of the static model instance, as described in the following bulleted list:

- If the RepairItem hierarchy is disconnected from the rest of the static model instance, then it simply conveys the parts breakdown information and is not expected to impact a reasoner session.
- If Faults are associated with RepairItems via the attribute `Fault.failedItem`, and if reasoning is based on Test-Fault relationships, then reasoners that dynamically generate test strategy can seek to isolate down to a repairable unit. The identification of repairable units is described in the next bullet.
- If Repair actions are associated the RepairItem hierarchy via the attribute `RepairItem.repairedBy`, then the Repair actions identify which of the RepairItems are the repairable units. A Repair action is presumed to correct all faults/failures in the RepairItem(s) that point to it, and any descendant RepairItems. This allows creators of static model instances to define sophisticated albeit indirect relationships describing which Repair actions correct which Faults. Since multiple RepairItems can point to the same Repair action, it might be more accurate to say that reasoners seek to isolate to a single Repair action.
- Alternatively, the repairable units in the RepairItem hierarchy can be assigned to a particular Level via the attribute `RepairItem.atIndentureLevel`, with all other RepairItems assigned to some other Level(s). The client can convey that it is only interested in a certain Level when it calls the services `getDiagnosticResults` and `recommendActions`.
- Otherwise, each leaf RepairItem that is a descendant of the SUT RepairItem is interpreted to be a repairable unit.

NOTE 1—One drawback to relying on Level entities to denote repairable units is that the client conveys the Level(s) that it is interested in with each service call to `getDiagnosticResults` and `recommendActions`. A reasoner has no way of knowing if the client has only a temporary interest in certain Levels.

AI-ESTATE allows a static model instance to contain `RepairItems` that are not the SUT `RepairItem` or its descendants. These should be interpreted as external systems, such as test equipment or something in the install environment of the SUT. External system `RepairItems` are useful to include in a static model instance when the external systems impact the diagnostic session. A static model instances may represent “safe to turn on tests” for the test equipment using external system `RepairItems` with associated `Faults` and `Tests` that detect those faults. When a safe to turn on test returns `FAIL`, the reasoner interprets it as a fault in the external system `RepairItem` (i.e., the test equipment) rather than the SUT. In another example, external system `RepairItems` may represent the install environment of a SUT that is tested in situ. The install environment can cause some `Tests` in the static model instance to return `FAIL`; those being tests that are intended to test the SUT. Part of the job of a reasoner would be to isolate the failure to being within the SUT, or something external to the SUT.

The CEM also allows a `RepairItem` to be part of the SUT and part of an external system, since a `RepairItem` can point to multiple parents with its `partOf` attribute. This guide does not provide a use case for such a configuration.

A simple case is where the SUT has no serviceable parts, and can merely be tested for good or bad health. Then the SUT `RepairItem` would have no descendants in the static model instance, and no `Repair` actions associated with the SUT `RepairItem`.

Figure 4 illustrates a more elaborate case, where `Repair` actions are associated with the `RepairItem` hierarchy in order to convey which `Repair` action (or actions) corrects each `Fault`. Figure 4 builds upon the relevant portions of the doorbell static model content shown in Figure 3 by adding `Repair` actions, subassembly `RepairItems` under the *Bell* `RepairItem`, and another `Fault`. The `Repair` action named *replace bell assembly* is associated with the *Bell* `RepairItem` through its `repairedBy` attribute, which indicates to a reasoner that this `Repair` will correct all `Faults` that can occur in the *Bell* `RepairItem` and its hierarchical descendants. The `Repair` action named *lubricate clapper hinge* is another option for correcting one of these `Faults`, namely *Clap-Friction*. During a session, a reasoner might recommend the *lubricate clapper hinge* `Repair` action over the *replace bell assembly* `Repair`, depending on the current diagnosis and the optimization criteria, since lubricating typically is less costly than replacement. This last point would be evident to the reasoner by assigning costs to the `Repair` actions, or by the fact that the one `Repair` acts on the child `RepairItem` of the other `Repair`.

The model relationships linking `Faults` to their corrective `Repair` action(s) are indirect in that they go through the `RepairItem` hierarchy. These relationships are illustrated by dashed arcs in Figure 4. To form the desired `Fault-Repair` relationships, model creators may need to craft portions of their `RepairItem` accordingly. Figure 4 illustrates this point by adding subAssembly `RepairItems` to the *Bell* `RepairItem` from Figure 3. The subAssembly `RepairItem` named *Clapper Hinge* was added to single out the *Clap-Friction* `Fault` for association with the *lubricate clapper hinge* `Repair` that corrects it. The *Clapper Hinge* `RepairItem` was made a subAssembly of the *Bell* `RepairItem` to convey that the *Clap-Friction* `Fault` can also be corrected by the *replace bell assembly* `Repair`. The subAssembly `RepairItem` named *Bell Other* was added because the `Faults` *Sol-O*, *Clap-C*, and *Clap-O* can no longer be associated directly with the *Bell* `RepairItem` due to the `faultsAtRepairItemLeaf` constraint.

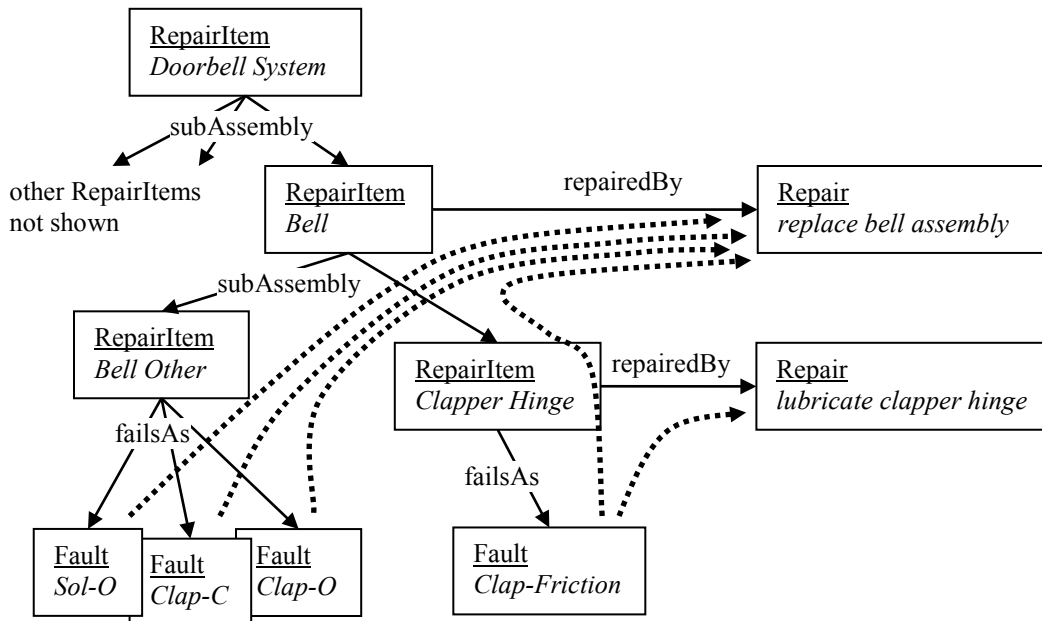


Figure 4—Example associating Faults with specific Repairs

RepairItems also have an attribute `testedBy` that is inherited from `SystemItem`. Reasoners are not required to use the attribute `RepairItem.testedBy` for reasoning, since each of the technology-specific model schemata defines its own relationships for reasoning from tests. From the technology-specific relationships a reasoner can deduce, if desired, which RepairItems are tested by which Tests. Model creators are free to leave the `RepairItem.testedBy` attribute empty. When this attribute is populated, it should be consistent with the relationships for reasoning from tests defined in each technology-specific model schema (i.e., BNM, FTM, etc.). AI-ESTATE does not state a requirement or recommendation for consistency. This guide suggests that the `testedBy` attribute of a RepairItem consist of those Tests that partially or wholly clear or indict the RepairItem, per the technology-specific inferences combined with chaining those conclusions to RepairItems as described in 7.2.9. When applying this definition of consistency, users would exclude any single fault assumption in their reasoning because it has the tendency to relate every Test with every RepairItem. For example, consider a DIM instance that has been processed as described in 7.5.1 and 7.5.4. Take any inference rule where a FAIL TestOutcome of a Test indicates a set of CANDIDATE DiagnosisOutcomes for Faults. The indicted RepairItems are found by chaining the conclusions from Faults to RepairItems via the attribute `Fault.failedItem`. To meet the definition of consistency, the `testedBy` attribute of each of these RepairItems would include the Test. Other inference rules in the DIM instance will similarly add their associated Tests to the `testedBy` attributes of these and other RepairItems.

NOTE 2—RepairItem has an attribute `failsAs` that identifies the Faults that the RepairItem can potentially have. AI-ESTATE erroneously defines the `failsAs` attribute so that it is required to be non-empty. This conflicts with the `faultAtLeaf` constraint on Fault entities that prevents them from being associated with non-leaf RepairItems. This might be corrected in future versions of AI-ESTATE.

7.2.5 Levels

The Level entity type is part of the hierarchy modeling facilities of AI-ESTATE and is intended to be used in conjunction with one or more of the hierarchical entity types: `SystemItem` and `Diagnosis`, and their subtypes `RepairItem`, `FunctionItem`, `Fault`, and `Failure`. A Level entity in a static model instance assigns a name and description to a hierarchical level, or a range of hierarchical levels. The association is made

through the `atIndentureLevel` attribute of `SystemItem` and `Diagnosis` (and their subtypes), which points to a `Level` entity.

NOTE 1—The `Action` entity type and its subtypes were intended to have an `atIndentureLevel` attribute, analogous to that of `SystemItem` and `Diagnosis`. It was unintentionally omitted and might be added in future revisions of AI-ESTATE. The `recommendActions` service has an input parameter `levelsOfInterest` that refers to the missing association between `Level` and `Action`.

A typical use case for `Level` entities would be to assign the levels in a `RepairItem` hierarchy to correspond to a system, subsystem, assembly, component, etc. Another use for `RepairItem` levels could be to distinguish between the `RepairItems` that are line replaceable units and those that are not. On the other hand, users might simply name their levels 1, 2, 3, etc. Note that reasoners are not required to interpret the name or description strings of the levels to perform any particular reasoning, such as isolating to `RepairItems` associated with a `Level` with a certain name. Reasoners that do so will need an agreement between parties.

During a session, the client can use levels to filter results returned by the reasoner. When the client calls the `getDiagnosticResults` or `recommendActions` services, it can optionally direct the reasoner to restrict results to those at a particular level(s) of interest. Although the `Levels` in a static model instance might have some deeper significance to the client and the instance creator, reasoners need only treat `Levels` as filter criteria and nothing more.

NOTE 2—Users might be tempted to use the `levelsOfInterest` parameter of the `getDiagnosticResults` service as a means to set the isolation goals of the reasoner. That is, to treat the `levelsOfInterest` as a directive to the reasoner to isolate to that particular level. That is beyond the intended semantics of the `levelsOfInterest` parameter. Moreover, there is a drawback to using `levelsOfInterest` in that way. The service intentionally gives the client freedom to change the value of `levelsOfInterest` with each call, which could be useful for some applications. Since a client's `levelsOfInterest` can change frequently within a session, it does not generally provide a sound basis for the reasoner to form an isolation strategy. See 7.2.4 for a more sound approach.

`Level` entities within a static model instance are subject to the following constraints. For the constraints that address how `Levels` relate to hierarchical entity types (`SystemItem`, `RepairItem`, `FunctionItem`, `Diagnosis`, `Fault`, and `Failure`), the description uses the term *hierarchical entity* for brevity.

The CEM schema permits a static model instance to have zero `Level` entity instances. Lack of `Levels` merely conveys a simpler hierarchical modeling, and precludes filtering by level in the services.

The `Level` instances in a static model instance can be thought of as forming one-dimensional “chains” via the attribute `Level.successor` and its inverse `Level.predecessor`. A static model instance can have zero or more chains of `Levels`. A chain can be as short as a single `Level`, and there is no upper limit on chain length. The cardinality of the successor and predecessor attributes dictate that a given `Level` instance belongs to exactly one chain. The `noRepeats` constraint dictates that a chain be acyclic.

The CEM schema permits a static model instance to have `Levels` that are not used in the `atIndentureLevel` attribute of any instance of hierarchical entity. This is evident from the lack of inverse attribute on `Level`. There is no intended interpretation for this situation, but recipients of static model instances should be prepared for its occurrence. Along the same lines, a hierarchy is not required to use the entire length of a chain of `Levels`; root hierarchical entities are not required to point to the start of a chain of `Levels`, and leaf hierarchical entities are not required to point to the end of a chain of `Levels`.

As an example, a static model instance might contain several chains of `Levels`, where the different chains are used by different entity type hierarchies. The `RepairItem` hierarchy could point to one chain, a `Diagnosis` and `Fault` hierarchy could point to a second chain, and `FunctionItem` and `Failure` hierarchies could point to a third chain.

Alternatively, a static model instance might contain a single chain of Levels, and each of the separate hierarchies might use that chain. Furthermore, the single chain of Levels might be used in a “synchronized” way across all the hierarchies. All root entities of the hierarchies could point to the first Level in the chain, all leaf entities of the hierarchies could point to the last Level in the chain, and the intermediate entities could point into the chain in some coordinated way.

Hierarchical entities have a `parentLevelConsistent` constraint that dictates several things. If a hierarchical entity has an `atIndentureLevel` attribute, then each of its child hierarchical entities (if any) is required to have one too. Conversely, if a hierarchical entity omits the `atIndentureLevel` attribute, then each of its child hierarchical entities (if any) is required to omit it too. Ultimately this dictates consistent inclusion or omission of the `atIndentureLevel` attribute within a connected hierarchy; all connected hierarchical entities are required to include it, or all are required to omit it. The term *connected* in this context means that a path exists between the hierarchical entities by following one or more `subAssembly/partOf` attributes in the case for `SystemItem` and its subtypes, or `subGroup/memberOf` attributes in the case for `Diagnosis` and its subtypes. The `parentLevelConsistent` constraint does not enforce consistency between disconnected hierarchies. For example, a static model instance could have a connected hierarchy of `RepairItems` to represent the SUT where each `RepairItem` has an `atIndentureLevel` attribute, and also have a separate set of `RepairItems` to represent external equipment with no `atIndentureLevel` attributes.

The `parentLevelConsistent` constraint on `SystemItem` also dictates that if `atIndentureLevel` attributes are present then the Levels used by a parent hierarchical entity and its children are required to be consistent with respect to the chain of Levels. The consistency rule is that a child is required to point to the same Level as its parent or to the successor Level. There is no requirement that all children of a hierarchical entity point to the same Level. Similarly, if a child hierarchical entity has multiple parents, there is no requirement that all parents point to the same Level.

7.2.6 ContextState, ModeOfOperation, and Purpose

IEEE Std 1232-2010 defines the entity types `ContextState`, `ModeOfOperation`, and `Purpose` to support use cases where external conditions are relevant to the diagnostic process, such as limiting what actions can be performed unless the proper conditions are met. Per AI-ESTATE, conditions include “system state, operational state, and the particular reason for performing diagnosis.” Conditions can be considered a more advanced feature, and these types will not necessarily be found in more basic static model instances.

The expectation is that users that use conditions would do so using multiple facets of condition. The `ModeOfOperation` entity type is one such facet built into the CEM, which represents the system state or operational state. An example for `ModeOfOperation` would be a military vehicle that has several mode settings such as *mission mode*, *maintenance mode*, *transport mode*, and *storage mode*. These values could be used as the name attributes for `ModeOfOperation` entities, with the description attributes providing further elaboration.

The `Purpose` entity type is another facet of condition built into the CEM. A `Purpose` entity represents the reason for performing the associated actions. `Purpose` does this mainly through its `hasRole` attribute, which is a set of one or more values of the Role-type enumeration: `TRAINING`, `VERIFICATION`, `SCHEDULED_MAINTENANCE`, `MAINTENANCE_ACTION`, `READY_FOR_ISSUE`, and `USER_DEFINED_ROLE`. The value `MAINTENANCE_ACTION` is intended to be used as the reason for Test and Repair actions that would be performed during the detection, isolation, and repair stages of a session (stages are described in subclause 7.3.6 of IEEE Std 1232-2010). The value `VERIFICATION` is intended to be used as the reason for Test actions to be performed during the verification stage of a session. The value `READY_FOR_ISSUE` is intended to be used for any actions to be performed to return the SUT to service. The value `USER_DEFINED_ROLE` allows users to define their own reasons for actions. The description attribute of `Purpose` may then be used to clarify the reason represented by each instance of `Purpose`, especially for those that use `USER_DEFINED_ROLE`, and any `Purpose` entities that have the same value of `hasRole` (`hasRole` is not constrained to be unique).

Other facets of condition beyond ModeOfOperation and Purpose are left to users to define, if desired, through extensions.

Each ContextState entity instance assigns a unique name and a description to a particular multi-faceted condition, thereby mapping the multiple facet dimensions to a single dimension of ContextStates enumerated by names. The names and descriptions of ContextStates are the only aspects of the conditions that are actually passed via the reasoner manipulation services. Users however cannot omit the underlying facets from their static model instances because they are required attributes of ContextStates.

During a reasoner session, the client conveys the current ContextState to the reasoner using the service setContext. The burden is on the client to know the current conditions throughout the session and translate them to a ContextState name. There is no information in a static model instance that would allow a reasoner to unambiguously derive the current ContextState by other means. If conditions change during the session, the burden is on the client to call setContext again to convey the new ContextState.

The reasoner uses the current ContextState to focus its diagnostic strategy on Actions that are available within the current ContextState. These are the Actions that include the current ContextState in their mustOccurIn attribute, and Actions that have an empty mustOccurIn attribute. The reasoner is not strictly limited to recommending only Actions that are available in the current ContextState. The reasoner can effectively suggest changing to a different ContextState by recommending other Actions. Furthermore, the output of the recommendActions service allows the reasoner to indicate the particular ContextStates for each recommended Action. The burden is on the client to perform whatever activities are necessary to change the conditions to the new ContextState if it follows the reasoner recommendation.

Figure 5 shows an example set of ContextState, ModeOfOperation, and Purpose entities in a static model instance that builds upon the military vehicle example mentioned above. The static model instance is intended to be used in the following situations:

- For the purpose of embedded diagnostics and operator alerts while the vehicle is performing a mission
- For the purpose of collecting so-called operator gripes post-mission while the vehicle is in maintenance mode
- For post-mission troubleshooting and repair while the vehicle is in maintenance mode

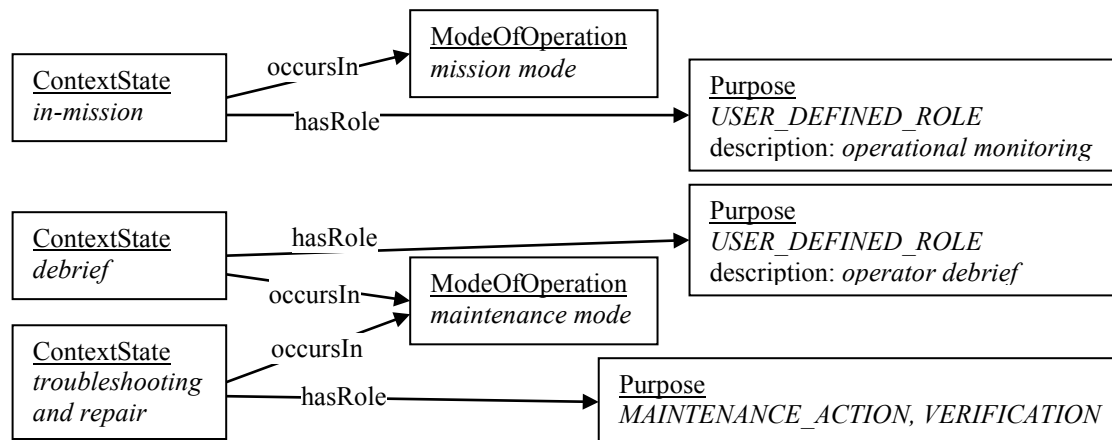


Figure 5—Example use of ContextState, ModeOfOperation, and Purpose

The reasoner and client are embedded on the vehicle for this example. There might also be aspects of the client off-vehicle, such as a handheld maintenance display, and telemetry of health data to a command center. At the start of a mission, the client would start a session and use the `setContextState` service to set the `ContextState` to *in-mission*. Then throughout the mission the client frequently uses the `applyActions` service to provide the reasoner with the latest continuous built-in-test (BIT) results. The `mustOccurIn` attribute of the continuous BITs in the static model instance associate them with the *in-mission* `ContextState`. The client would also use the `getDiagnosticResults` service to obtain updated diagnostic conclusions throughout the mission. The reasoner would return only conclusions of interest to the operator or command center, as determined by a combination of the service input parameters and by whether the `Diagnosis` entities are associated with the *in-mission* `ContextState` through the `Diagnosis.mustOccurIn` attribute. The client would also use the `recommendActions` service to query the reasoner for any initiated BIT to be run, and the reasoner could recommend initiated BITs that are associated with the *in-mission* `ContextState`. After the mission, the session would be kept open or resumed for entry of any so-called operator gripes, which would consist of `TestOutcomes` provided through the `applyActions` service, and for which the client would set the `ContextState` to *debrief*. After debrief, the session would be kept open or resumed for post-mission troubleshooting and repair of any problems identified by BIT or the operator. The client would switch to the *troubleshooting and repair* `ContextState`, and the activities would use the `Tests` (automated and human-in-the-loop) and `Diagnoses` that are associated with this `ContextState`.

A permutation of this example could be where mission BIT is simply recorded, and where the reasoner session starts post-mission. At the start of the post-mission session, the client would set the `ContextState` to something akin to *debrief* and provide the reasoner with a summary of the BIT outcomes using the `applyActions` service. The *in-mission* `ContextState` as defined would not be appropriate because it was intended to be used during the mission. From there, the session would proceed as described above.

The military vehicle example brings up the point that users sometimes need different reasoner behavior in different situations. In the example, the reasoning used during *in-mission* needs to continuously evaluate the health of the SUT as it changes in time, and there is lesser need to optimize test strategy because continuous BIT was always running. The reasoning used during *troubleshooting and repair* can assume that the SUT health is constant in time except for repairs, and that finding a near-optimal test and repair strategy is important. The reasoning used during debrief can assume that the reported observations occurred in the past and are potentially stale. Reasoners may use the `ContextState` to recognize different diagnostic situations and alter their behavior accordingly, particularly by keying off the enumerated value in the `Purpose.hasRole` attribute. The fact that the example needed to use `USER_DEFINED_ROLE` suggests that additional enumeration values such as `OPERATIONAL_MONITORING` and `OPERATOR_DEBRIEF` would be useful. Otherwise, the example as currently written using `USER_DEFINED_ROLE` would need a special agreement between parties for the reasoner to recognize the significance of these `ContextStates`.

7.2.7 Outcomes

IEEE Std 1232-2010 defines an abstract `Outcome` entity type with concrete subtypes: `DiagnosisOutcome`, `TestOutcome`, and `ActionOutcome`. An instance of `DiagnosisOutcome`, `TestOutcome`, or `ActionOutcome` in a static model instance represents a potential state-value (i.e., `GOOD`, `BAD`, `PASS`, `FAIL`, etc.) that can be taken on by its associated state-variable: `Diagnosis`, `Test`, or `Action`, respectively. A typical static model instance will contain many `DiagnosisOutcomes`; as much as one `DiagnosisOutcome` instance for each possible value (`GOOD`, `BAD`, etc.) of each modeled `Diagnosis` (including subtypes). The same applies to `TestOutcomes`, and to `ActionOutcomes` to a lesser extent. Figure 3 contains many `DiagnosisOutcomes` and `TestOutcomes`.

NOTE—IEEE Std 1232-2010 represents the potential state-values of `Diagnoses`, `Tests`, and `Actions` as EXPRESS entities so that they can be easily cross-referenced from other entities in the static model instance, and so that they can be easily checked during schema validation of the instance. The EXPRESS modeling language and exchange formats provide a natural way of doing this for entities.

In a static model instance, the cross references to *DiagnosisOutcome* and *TestOutcome* are typically used to express inference relationships. Examples can be found in the clauses on technology-specific static models. DCM instances also make use of cross-references to Outcomes. In a DCM instance, the cross references are typically used to represent that a particular Outcome state has been observed, inferred, or hypothesized.

7.2.8 Diagnosis, Fault, Failure, and DiagnosisOutcome

IEEE Std 1232-2010 provides three entity types for modeling diagnostic conclusions: Fault, Failure, and Diagnosis. Diagnosis as a supertype of Fault and Failure, but for purposes of this subclause the term *Diagnosis* refers to a Diagnosis that is neither a Fault nor a Failure.

- Faults are associated with *RepairItems*, where each Fault entity represents a particular physical flaw in its associated *RepairItem*. The term *physical* is not intended to exclude nonphysical systems such as software systems. A Fault can correspond to a complete breakage of a part, or an attribute of a part that is out of specification. Examples of potential Faults for electronic systems include: burned out components, detached component leads, bent or broken pins, bad contacts, shorts, out of specification resistance or capacitance, etc. Examples for mechanical systems include: broken shafts and linkages, broken attachments, blocked fluid lines, fluid leaks, jammed mechanisms, bent shafts, excessive looseness, worn out parts, cracked or broken gear teeth, etc.
- Failures are associated with *FunctionItems*, and are the functional manifestations of Faults. A Failure can be one of two things.
 - 1) If the Failure entity has no hierarchical children within its *subGroup* attribute, then it represents a particular way that its associated *FunctionItem* can be out of specification in some aspect, or completely lost. It is referred to as a “leaf Failure.” Potential examples of leaf Failures for an amplifier *FunctionItem* include: no output signal, stuck at power rails, gain or offset out of specification, excessive noise, excessive distortion, excessive nonlinearity, reduced bandwidth, reduced dynamic range, etc.
 - 2) If the Failure entity has a *subGroup* attribute (inherited from *Diagnosis*) with hierarchical children, then the Failure entity is a summary or roll-up conclusion of its child Failures. It is referred to as a “roll-up Failure.” For example, the more extreme leaf Failures in the previous bullet could be rolled up under a Failure named *No amplification* while the lesser leaf Failures could be rolled up under a Failure named *Degraded amplification*. Note that the Failure hierarchy relationships are not intended to represent dependency flow, though there are some similarities.
- A Diagnosis entity that is neither a Fault nor a Failure can be one of three things.
 - 1) If the name attribute of the Diagnosis is *No Fault*, then the Diagnosis represents the special conclusion that there is no fault in the system under test. It is referred to as the “NoFault Diagnosis.” Some static model types make use of this special conclusion. The NoFault Diagnosis also has a role in the DCM and reasoner services.
 - 2) If the Diagnosis has a *subGroup* attribute with hierarchical children, then the Diagnosis is a summary or roll-up conclusion of its child conclusions. It is referred to as a “roll-up Diagnosis.” The child conclusions of a roll-up Diagnosis are one or more Faults, Failures, and Diagnoses in any combination. Note that the Diagnosis hierarchy relationships are not intended to represent dependency flow, though there are some similarities.
 - 3) If the Diagnosis has no hierarchical children and is not named *No Fault*, then it is a diagnostic conclusion that has not been modeled to the level of detail of a Fault or leaf Failure. It is referred to as a “leaf Diagnosis.”

NOTE 1—The CEM allows a Failure’s hierarchical subGroup attribute to point to Diagnoses, Faults, and Failures. The intent was that any hierarchical child of a Failure would be constrained to being a Failure. This restriction may be included in future versions of the CEM.

NOTE 2—Subclause 6.1.25 of IEEE Std 1232-2010 describes the roll-up relationship between a Diagnosis and its children, but is incomplete and not entirely correct. The proper roll-up logic is as follows: If any of the children are BAD then the roll-up Diagnosis is BAD. If all of the children are GOOD then the roll-up Diagnosis is GOOD. If the children are a combination of one or more NOTKNOWN and zero or more GOOD then the roll-up Diagnosis is NOTKNOWN. If the children are combination of at least one CANDIDATE and zero or more GOOD and zero or more NOTKNOWN, then the roll-up Diagnosis is CANDIDATE, unless all the CANDIDATES in an ambiguity group are confined to the descendants, then the roll-up Diagnosis is BAD. If any of the children is user defined, then the roll-up Diagnosis is implementation specific.

The entities Diagnosis, Fault, Failure, RepairItem, and FunctionItem are interrelated in a number of ways. These are shown in the EXPRESS-G schema diagram in Figure 6 and described in the following paragraphs. Note that other attributes of Diagnosis, Fault, Failure, RepairItem, and FunctionItem are omitted from Figure 6 for clarity.

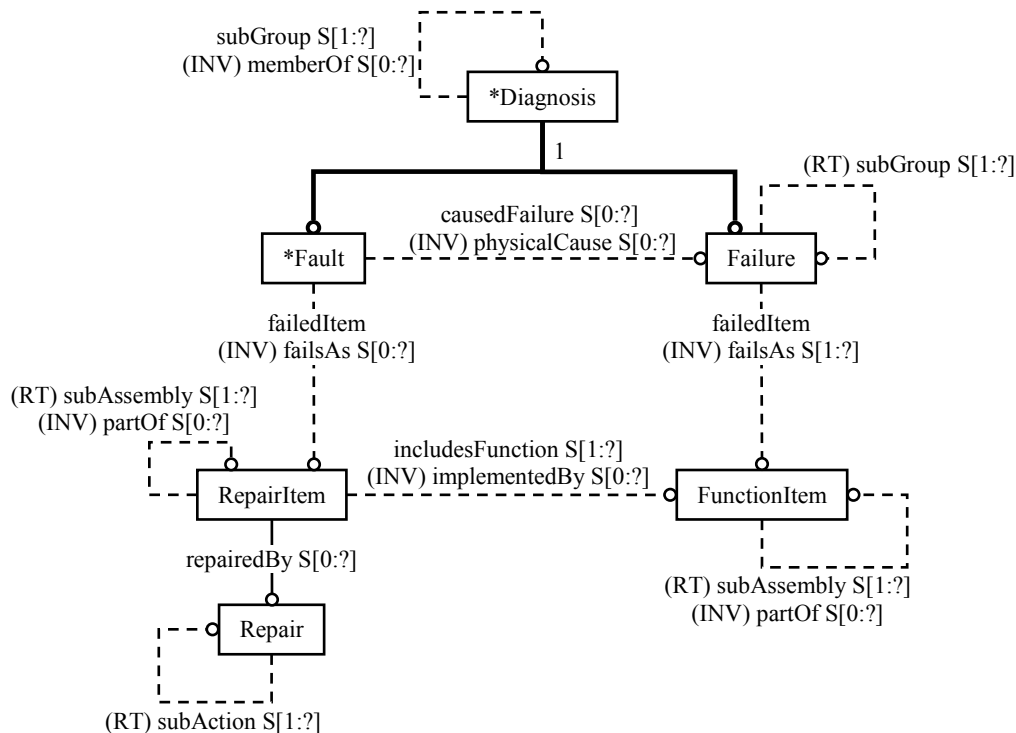


Figure 6—Relationships between diagnostic conclusions

Faults have a many-to-one relationship with RepairItems. The attribute Fault.failedItem points to a single RepairItem, which is the RepairItem that the Fault occurs in. Conversely, each leaf RepairItem can break in a variety of ways, represented by the inverse attribute RepairItem.failsAs that points to a disjoint set of multiple Faults. The attribute Fault.failedItem is declared optional by the CEM schema. Omitting the attribute Fault.failedItem represents that the relationship is unmodeled, or that the associated RepairItem is unmodeled. A static model instance should include the failedItem attribute of each Fault instance unless RepairItems are unmodeled, because useful reasoning capabilities are lost when the attribute is omitted. By omitting the attribute, it becomes ambiguous to a reasoner whether the Fault is within the system under test,