INTERNATIONAL STANDARD

IEC 62243

First edition
2005-07

IEEE 1232™

Artificial intelligence exchange and service tie to all test environments (AI-ESTATE)

Reference number
IEC 62243(E):2005
IEEE Std. 1232(E):2002

Publication numbering

As from 1 January 1997 all IEC publications are issued with a designation in the 60000 series. For example, IEC 34-1 is now referred to as IEC 60034-1.

Consolidated editions

The IEC is now publishing consolidated versions of its publications. For example, edition numbers 1.0, 1.1 and 1.2 refer, respectively, to the base publication, the base publication incorporating amendment 1 and the base publication incorporating amendments 1 and 2.

Further information on IEC publications

The technical content of IEC publications is kept under constant review by the IEC, thus ensuring that the content reflects current technology. Information relating to this publication, including its validity, is available in the IEC Catalogue of publications (see below) in addition to new editions, amendments and corrigenda. Information on the subjects under consideration and work in progress undertaken by the technical committee which has prepared this publication, as well as the list of publications issued, is also available from the following:

- **IEC Web Site (www.iec.ch)**

- **Catalogue of IEC publications**

  The on-line catalogue on the IEC web site (www.iec.ch/searchpub) enables you to search by a variety of criteria including text searches, technical committees and date of publication. On-line information is also available on recently issued publications, withdrawn and replaced publications, as well as corrigenda.

- **IEC Just Published**

  This summary of recently issued publications (www.iec.ch/online_news/ justpub) is also available by email. Please contact the Customer Service Centre (see below) for further information.

- **Customer Service Centre**

  If you have any questions regarding this publication or need further assistance, please contact the Customer Service Centre:

  Email: custserv@iec.ch
  Tel: +41 22 919 02 11
  Fax: +41 22 919 03 00

INTERNATIONAL STANDARD

IEC 62243

First edition
2005-07

IEEE 1232™

Artificial intelligence exchange and service tie to all test environments (AI-ESTATE)

International Electrotechnical Commission, 3, rue de Varembé, PO Box 131, CH-1211 Geneva 20, Switzerland
Telephone: +41 22 919 02 11 Telefax: +41 22 919 03 00 E-mail: inmail@iec.ch Web: www.iec.ch

The Institute of Electrical and Electronics Engineers, Inc, 3 Park Avenue, New York, NY 10016-5997, USA
Telephone: +1 732 562 3800 Telefax: +1 732 562 1571 E-mail: stds-info@ieee.org Web: www.standards.ieee.org

Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

ARTIFICIAL INTELLIGENCE EXCHANGE AND SERVICE TIE TO ALL TEST ENVIRONMENTS (AI-ESTATE)

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with an IEC Publication.
6) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC/IEEE 62243 has been processed through IEC technical committee 93: Design automation.

The text of this standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1232 (2002)	93/214/FDIS	93/220/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives.

The committee has decided that the contents of this publication will remain unchanged until 2007.

IEC/IEEE Dual Logo International Standards

This Dual Logo International Standard is the result of an agreement between the IEC and the Institute of Electrical and Electronics Engineers, Inc. (IEEE). The original IEEE Standard was submitted to the IEC for consideration under the agreement, and the resulting IEC/IEEE Dual Logo International Standard has been published in accordance with the ISO/IEC Directives.

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

Use of an IEC/IEEE Dual Logo International Standard is wholly voluntary. The IEC and IEEE disclaim liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEC or IEEE Standard document.

The IEC and IEEE do not warrant or represent the accuracy or content of the material contained herein, and expressly disclaim any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEC/IEEE Dual Logo International Standards documents are supplied “AS IS”.

The existence of an IEC/IEEE Dual Logo International 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 IEC/IEEE Dual Logo International 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.

Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, 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 publishing and making this document available, the IEC and IEEE are not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Neither the IEC nor IEEE is undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEC/IEEE Dual Logo International Standards or IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations – Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received 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 interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of IEC/IEEE Dual Logo International Standards are welcome from any interested party, regardless of membership affiliation with the IEC or IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA and/or General Secretary, IEC, 3, rue de Varembé, PO Box 131, 1211 Geneva 20, Switzerland.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, 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.

NOTE – 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 with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

IEEE Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATES)

Sponsor

IEEE Standards Coordinating Committee 20

Approved 13 November 2002

American National Standards Institute

Approved 13 June 2002

IEEE-SA Standards Board

Abstract: AI-ESTATE is a set of specifications for data interchange and for standard services for the test and diagnostic environment. The purpose of AI-ESTATE is to standardize interfaces between functional elements of an intelligent diagnostic reasoner and representations of diagnostic knowledge and data for use by such diagnostic reasoners. Formal information models are defined to form the basis for a format to facilitate exchange of persistent diagnostic information between two reasoners, and also to provide a formal typing system for diagnostic services. This standard then defines the services to manipulate diagnostic information and to control a diagnostic reasoner.
Keywords: AI-ESTATE, diagnosis, diagnostic interference, diagnostic model, diagnostic services, dynamic content, fault tree, knowledge exchange, system test

IEEE Introduction

This AI-ESTATE standard provides a formal framework for exchanging diagnostic knowledge and constructing diagnostic reasoners. The intent is to provide a standard framework for identifying required information for diagnosis and defining the diagnostic information in a machine-processable way. In addition, software interfaces are defined whereby diagnostic tools can be developed to process the diagnostic information in a consistent and reliable way.

Artificial intelligence exchange and service tie to all test environments (AI-ESTATE)

1. Overview

The Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE) standard was developed by the Diagnostic and Maintenance Control (D & MC) Subcommittee of the IEEE Standards Coordinating Committee 20 (SCC 20) on Test and Diagnosis for Electronic Systems to serve as a standard for the application of artificial intelligence to system test and diagnosis. This AI-ESTATE standard defines interfaces among reasoners and reasoning system users, test information knowledge bases, and more conventional databases. In addition to interface standards, the AI-ESTATE standard includes a set of formal data specifications to facilitate the exchange of system under test related diagnostic information.

This standard describes a set of formal data and knowledge specifications consisting of the logical representation of devices, their constituents, the failure modes of those constituents, and tests of those constituents. The data and knowledge specification provides a standard representation of the common data elements required for system test and diagnosis. This will facilitate portability of test-related knowledge bases for intelligent system test and diagnosis.

The goals of this standard are summarized as follows:

— Incorporate domain specific terminology
— Facilitate portability of diagnostic knowledge
— Permit extensibility of diagnostic knowledge
— Enable the consistent exchange and integration of diagnostic capabilities

This standard provides a controlled extension mechanism to allow inclusion of new diagnostic technology outside the scope of the AI-ESTATE specification.

One of the purposes of this standard is to define information models for knowledge bases to be used in the context of test and diagnosis and, from these models, to derive a data interchange format. The specifications in this standard shall support fully portable diagnostic knowledge. No host computer dependence is contained in the AI-ESTATE standard.

AI-ESTATE defines key data and knowledge specification formats. Implementations that use only these specification formats will be portable. This does not preclude use of AI-ESTATE interfaces with nonconformant specification formats; however, such implementations may not be portable. As shown in Figure 1, a diagnostic model can be moved from one AI-ESTATE implementation to another by translating it into the interchange format. Another AI-ESTATE implementation can then utilize this information as a complete package by translating the data and knowledge from the interchange format to its own internal form.

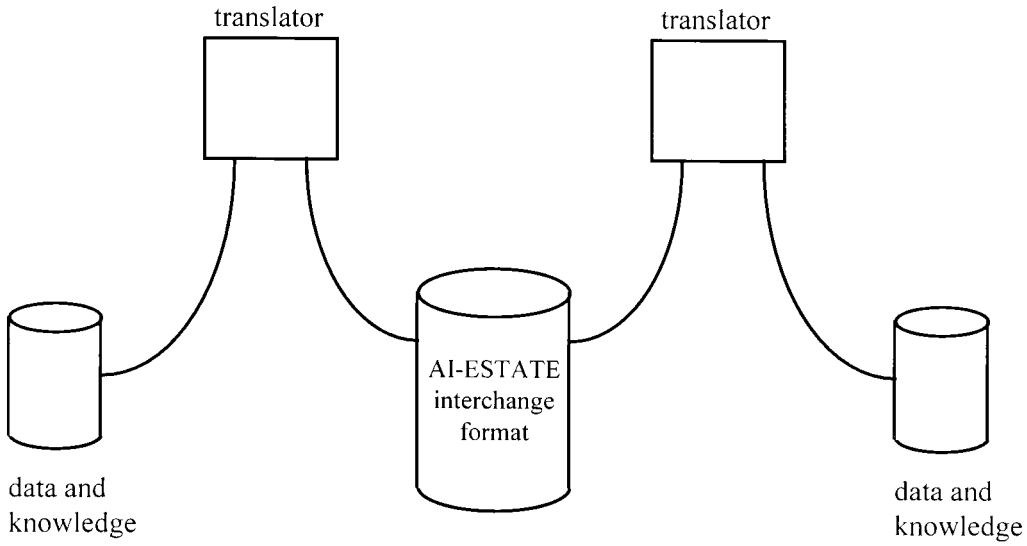


Figure 1—An example of AI-ESTATE's portability mechanism for data and knowledge

The translation step is not a requirement; an AI-ESTATE implementation may use the interchange format for its own internal form.

Software specifications defined in this standard will ensure the interchangeability of diagnostic reasoners through the definition of encapsulated services. This will allow diagnostic reasoners to be interchanged within an AI-ESTATE conformant system with no effect on the other elements of the system.

1.1 Scope

The AI-ESTATE standard defines formal specifications for supporting system diagnosis. These specifications support the exchange and processing of diagnostic information and the control of diagnostic processes. Diagnostic processes include, but are not limited to, testability analysis, diagnosability assessment, diagnostic reasoning, maintenance support, and diagnostic maturation.

1.2 Purpose

The AI-ESTATE standard provides formal models of diagnostic information to ensure unambiguous access to an understanding of the information supporting system test and diagnosis. The standard unifies and expands on the specifications published in IEEE Std 1232™-1995 [B3],[1] IEEE Std 1232.1™-1997 [B4], and IEEE Std 1232.2™-1998 [B5].

[1]The numbers in brackets correspond to those of the bibliography in Annex A.

1.3 Conventions used in this standard

This standard specifies information models using the EXPRESS language and uses the following conventions in their presentation:

All specifications in the EXPRESS language are given in the `Courier` type font. This includes references to entity and attribute names in the supporting text. The EXPRESS models found in Clause 5 include comment delimiters "(*" and "*)," thus allowing extraction of the models from an electronic version of the standard for direct use.

Each entity of each EXPRESS schema is presented in a separate subclause. Within a schema, subclauses are listed in alphabetical order by constants, types, enumerated types, select types, entities, and then functions. The subclause structure begins with the actual EXPRESS specification, then each attribute of the entity is described below the attribute definition heading. If any constraints have been specified, these are described below the formal propositions heading.

This standard uses the vocabulary and definitions of relevant IEEE standards. In case of conflict of definitions, the following precedence shall be observed: 1) AI-ESTATE definitions (Clause 3); 2) SCC20 documentation and standards; and 3) IEEE 100™, *The Authoritative Dictionary of IEEE Standards Terms*, Seventh Edition [B2].

Clause 6 of this standard presents the formal specification of the encapsulated services of this standard. EXPRESS is used to represent the interface of each individual service defining the semantics and type of the required value to be returned.

2. References

This standard shall be used in conjunction with the following publications.

ISO 10303-11:1994, Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 11: Description Methods: The EXPRESS Language Reference Manual.[2]

ISO 10303-21:2002, Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 21: Implementation Methods: Clear Text Encoding of the Exchange Structure.

3. Definitions and acronyms

For the purposes of this standard, the following terms and definitions apply. IEEE 100, *The Authoritative Dictionary of IEEE Standards Terms*, Seventh Edition [B2], should be referenced for terms not defined in this clause.

3.1 Definitions

3.1.1 ambiguity: In fault isolation, the inability to localize to a single diagnosis for a repair level, given a set of test results, observations, or other information.

3.1.2 ambiguity group: The collection of all diagnoses that are in ambiguity.

[2]ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genéve 20, Switzerland/Suisse (http://www.iso.ch/). ISO publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA (http://www.ansi.org/).

3.1.3 application executive: A software component (or the role of a software component) that administers, and coordinates services to other components within a test system.

3.1.4 diagnosis: The conclusion(s) inferred from tests, observations, or other information.

3.1.5 diagnostic procedure: A structured sequence of tests, observations, and other information used to localize a fault or faults.

3.1.6 element: The smallest entity of an Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE) model. For example, in a particular model, the smallest test, the smallest diagnosis, and the no-fault conclusion are all elements.

3.1.7 failure: The loss of ability of a repair item, equipment, or system to perform a required function. The manifestation of a fault. Within AI-ESTATE models, a manifestation is given by the outcome of a test.

3.1.8 false alarm: An indicated fault where no fault exists.

3.1.9 fault: A defect or flaw in a hardware or software component.

3.1.10 fault isolation: The process of reducing the set of diagnoses in ambiguity to a degree sufficient to undertake an appropriate corrective action.

3.1.11 fault localization: The reduction of ambiguity by the application of tests, observations, or other information.

3.1.12 interoperability: The ability of two or more systems or elements to exchange information and to use the information that has been exchanged.

3.1.13 knowledge base: A combination of structure, data, and function used by reasoning systems.

3.1.14 level of maintenance: A level at which test, diagnosis, and repair operates (e.g., maintenance depot, factory, in the field).

3.1.15 portability: The capability of being moved between systems.

3.1.16 protocol: A set of conventions or rules that govern the interactions of processes or applications within a computer system or network.

3.1.17 reasoning system: A system that can combine elements of information and knowledge to draw conclusions.

3.1.18 replaceable unit: A collection of one or more parts considered as a single part for the purposes of replacement and repair due.

3.1.19 resource: Any capability that is to be scheduled, assigned, or controlled by the underlying implementation to assure nonconflicting usage by processes.

3.1.20 service: A software interface providing a means for communicating information between two applications. An action or response initiated by a process (i.e., server) at the request of some other process (i.e., client).

3.1.21 system: 1) A collection of interacting, interrelated, or interdependent elements forming a collective, functioning entity; 2) a set of objects or phenomena grouped together for classification or analysis; 3) a collection of hardware or software components necessary for performing a high-level function.

3.1.22 test: A set of stimuli, either applied or known, combined with a set of observed responses and criteria for comparing these responses to a known standard.

3.1.23 test strategy: An approach taken to combine factors including constraints, goals, and other considerations to be applied to the testing of a system under test.

3.2 Acronyms

AI-ESTATE	Artificial Intelligence Exchange and Service Tie to All Test Environments
BIT	Built-In Test
CEM	Common Element Model
DCM	Dynamic Context Model
DIM	Diagnostic Inference Model
EDIM	Enhanced Diagnostic Inference Model
FTM	Fault Tree Model
SNMP	Simple Network Management Protocol
UTC	Coordinated Universal Time

4. Description of AI-ESTATE

4.1 AI-ESTATE architecture

This standard provides the following:

— Overview of the AI-ESTATE architecture
— Formal definition of diagnostic models for systems under test
— Formal definition of encapsulated software services for diagnostic reasoners

AI-ESTATE focuses on two distinct aspects of the stated purpose. The first aspect concerns the need to exchange data and knowledge between conformant systems. Two approaches can be taken to address this need: providing interchangeable files and providing services for retrieving the required data or knowledge through an information management system. AI-ESTATE is structured such that either approach can be used. The second aspect concerns the need for functional elements of an AI-ESTATE conformant system to interact and interoperate. The AI-ESTATE architectural concept provides for the functional elements to communicate with one another via a ‘‘communication pathway’’ as depicted in Figure 2. Essentially, this pathway is an abstraction of the services provided by the functional elements to one another. Thus, the implemented services provide a communication pathway between the reasoner and the rest of the test system.

Services are provided by reasoners to the other functional elements of an AI-ESTATE conformant system. Reasoners can include (but are not necessarily limited to) diagnostic systems, test sequencers, maintenance data feedback analyzers, intelligent user interfaces, and intelligent test programs. AI-ESTATE will not specify services between functional elements that do not incorporate artificial intelligence capabilities. Thus, services are provided by a reasoner to the test system, the human presentation system, a maintenance data/knowledge collection system, and possibly the system under test. The reasoner shall use services provided by these other systems as required. Note that these services shall not be specified by the AI-ESTATE standard.

Data interchange formats are specified to provide a means for exchanging knowledge bases between conformant systems without the need to apply an information management system. Recognizing that

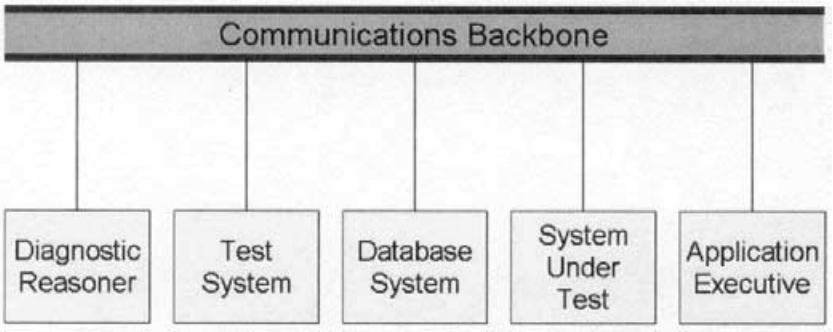


Figure 2—AI-ESTATE architectural concept

some applications may provide services for extracting required data or knowledge, services are specified to permit an application to query the diagnostic system for this purpose.

This standard facilitates the use of standard representations of diagnostic data and knowledge within the context of an AI-ESTATE implementation. In specifying data and knowledge for these domains, a structure has been constructed, as shown in Figure 3. At the top level is the Common Element Model that specifies elements common to the AI-ESTATE domain of equipment test and diagnosis in its entirety. Examples of common element constructs are `diagnosis` (diagnostic conclusions about the system under test), `repair_item` (the physical entity being repaired), `resource`, and `test`. These constructs are characterized by attributes such as costs and failure rates, which are also specified in the Common Element Model.

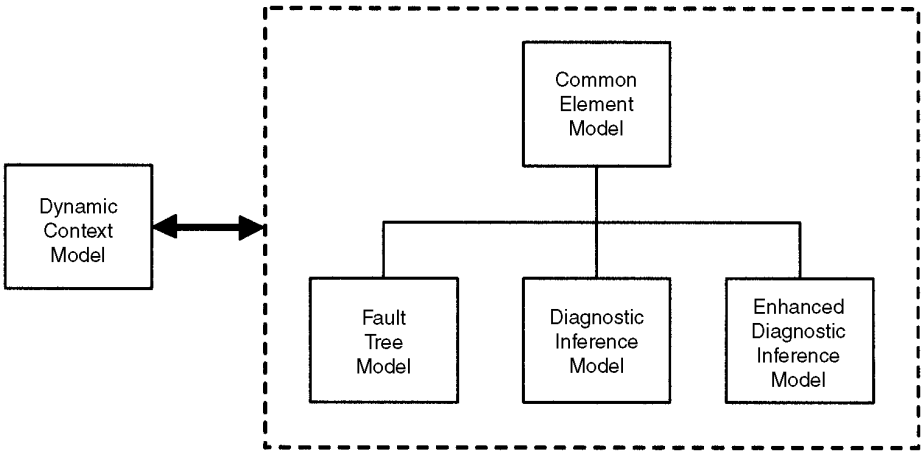


Figure 3—Hierarchical structure of AI-ESTATE models

Below this top layer is a layer of application-specific data and knowledge formats (i.e., the Diagnostic Inference Model, the Enhanced Diagnostic Inference Model, and the Fault Tree Model). These models take advantage of the constructs in the Common Element Model and tailor the constructs to the application's particular reasoning requirements.

Other data and knowledge specification formats are envisioned and will be included in future revisions of this standard. Examples include a constraint-based model, a Bayesian network model, and a neural network model. The Common Element Model has been specified such that other data and knowledge specification formats can also utilize its constructs as base elements that are tailored to the particular application's needs.

This standard also defines the encapsulated services to be provided by a diagnostic reasoner in an AI-ESTATE conformant implementation. All of the basic services are defined relative to the entities and attributes of the information models. These services can be thought of as reasoner responses to client requests from the other components of the system architecture, which include the human presentation (or user interface) system, the maintenance data/knowledge collection (or information management) system, other reasoners, and test controller(s), as shown in Figure 4. As can be seen in Figure 4, each of the elements that interface with the reasoner will provide their own set of encapsulated services to their respective clients, but those service definitions are beyond the scope of this document.

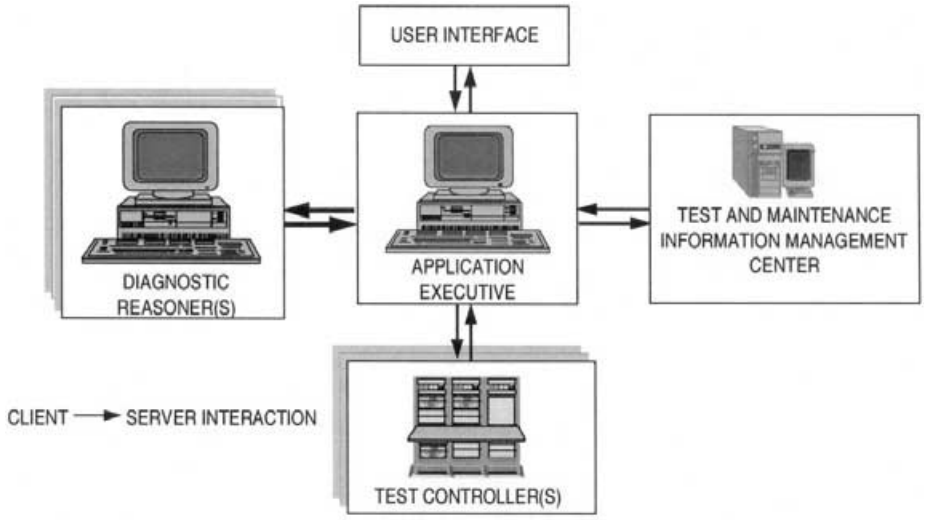


Figure 4—AI-ESTATE client server view

The definition of these services was performed by first creating accessor services for the static information models. As the specification of these services became clearer, it became apparent that to define useful services adequately for an active diagnostic reasoner new entities would have to defined. These entities represent the context within which reasoning is performed and thereby maintain the state of the diagnostic process. Thus evolved the Dynamic Context Model as specified in 5.3 of this standard. Model management and reasoner manipulation services are defined in Clause 6 to provide a more manageable interface for interacting with the reasoner.

The definition of encapsulated services provides a means for hiding the details of any particular reasoner implementation. Such services encompass an *abstraction* of that behavior which is common to all diagnostic reasoners, regardless of implementation details. Therefore, it is the mechanism of encapsulation that provides for the interchangeability of diagnostic reasoners within an AI-ESTATE conformant system.

It should be noted that to be conformant, a reasoner implementation should provide, at the least, a status indicator (see 4.5) as a response to any service request defined by this specification. The diagnostic reasoner shown in Figure 5 becomes AI-ESTATE conformant when it provides the services specified by this standard to the application executive client.

In defining the behavior of the diagnostic reasoner, it is reasonable to expect applications to be developed whereby multiple models are used by a single reasoner or multiple reasoners interact. For the latter, it is assumed that the application executive will be responsible for managing these reasoners and reconciling information provided by the reasoners. For the single reasoner, it is assumed that the reasoner will manage the models and that this management will be based on definition of a consistent

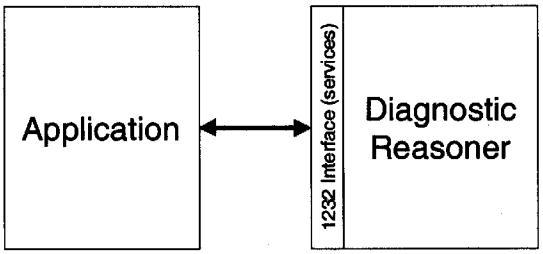


Figure 5—IEEE Std 1232-2002 interface layer

name-space with proper scoping defined by the models' contexts. For reasoners, responsible for merging information in multiple models (or for editing tools that merge models), it is assumed that such a merger will be conducted either through low-level model management services or through some proprietary mechanism. This standard does not define how model merging is to be accomplished, and no specific merger service is provided.

4.2 Interchange format

AI-ESTATE models are intended to facilitate data interchange in the context of test and diagnosis. The data interchange format will permit exchange of diagnostic models, using a neutral standard format, thus providing portability of diagnostic knowledge across applications. The interchange format used for the models defined in this standard is derived from ISO 10303-21:2002. Given this exchange structure, it is assumed that one and only one model will be contained in any given file.

The version of this standard used shall be noted in the header of the exchange file, e.g., IEEE Std 1232-2002.

4.3 Binding strategy

The intent of the binding strategy is to guide software developers in the creation of a binding layer that will expose an interface that matches the interface of the AI-ESTATE services as they are specified in this standard. The binding layer will thus insulate the application and the diagnostic reasoner from any non-AI-ESTATE details such as connectivity technology, memory management, etc.

An AI-ESTATE software system will consist of at least two components—the application and a diagnostic reasoner. The diagnostic reasoner will present an interface conformant to IEEE Std 1232-2002; the application will use AI-ESTATE services as needed by calls to this interface.

For each AI-ESTATE service there will be a corresponding function in the binding layer that will be written in the implementation language. The interfaces provided by the functions should correspond exactly to the interfaces of the services they implement (or as closely as possible given the constraints of the implementation language). All other details should be hidden from the client. This implies that the binding layer provides data-type definitions that correspond to the types specified in this standard.

The application and diagnostic reasoner programs can be written in different languages as long as the translation is handled transparently to the two programs, i.e., in the binding layer or lower. When publishing the interface, it is recommended that documentation of traceability of the elements of the interface to the services specified in the standard be provided.

For example, consider the get_name service as specified in EXPRESS:

```
FUNCTION get_name
  (ID_entity : entity_id):
   name_type;
END_FUNCTION;
```

It has the name get_name, accepts one argument of entity_id, and returns a name_type. The declaration of a corresponding binding function written in C would be

```
name_type get_name (entity_id ID_entity);
```

This might exist in a C header file and would provide the client code with an interface corresponding exactly to that of the EXPRESS form. For example,

```
name_type get_name (entity_id ID_entity)
{
  name_type name;
⋮
  name = ...; /* Could be a call to a function in the */
              /* server-side binding layer.*/
⋮
  return name;
}
```

The following C data types could be defined to correspond to the AI-ESTATE types:

```
typedef char*    name_type;
typedef void*    entity_id;
```

For pure object-oriented languages such as Java™, the interface shall be presented as methods in objects. It is suggested that the information model be used to start building the class hierarchy.

4.4 Extensibility

4.4.1 Extending information models

The data and knowledge specification should be sufficiently flexible to allow for differences in the common tools that might use it, yet also remain formal and rigid enough to support porting of an AI-ESTATE exchange file to any AI-ESTATE conformant application. This flexibility is achieved by using the EXTEND schema. In general, an extended schema will not be usable by other AI-ESTATE conformant applications. In applying the EXTEND schema, the following rules shall apply:

a) An IEEE Std 1232-2002 information model can be extended for any single model, hereafter referred to as the *extended model*, by inclusion of one or more schemata that define all entities used in the extended model. The schemata shall be legal EXPRESS and shall be syntactically and semantically correct. Further, extensions shall not be used to rename or redefine data in the standard schemata.
b) Data contained in a model definition with an EXTEND schema shall include only entity types that are declared in IEEE Std 1232-2002, or are legal, previously defined extension entities, as defined in subsequent requirements.
c) An EXTEND schema entity shall be declared to be a subtype of an entity defined in IEEE Std 1232-2002 or a previously defined EXTEND schema entity.

d) A model defined with EXTEND schemata shall not cause an application to reach an invalid conclusion when processing the model with the extended entities removed. In other words, any conclusion drawn by a conformant reasoner using an extended model without the extended entities will be consistent with the information contained in the nonextended model. Further, consistency cannot be achieved through extension.
e) All schema, entity, type, function, and procedure identifiers declared within the EXTEND schema shall begin with the prefix "extend_."
f) The EXTEND schema shall use the EXPRESS reference interface specification, "REFERENCE FROM," to allow visibility of the entities within the EXTEND schema.

The following is an example of EXTEND schema:

```
SCHEMA extend_schema_a;

REFERENCE FROM common_element_model (outcome, cost);

TYPE extend_attribute_type1 = STRING;
END_TYPE;

TYPE extend_attribute_type2 = REAL;
END_TYPE;

ENTITY extend_ent_a
    SUBTYPE OF (cost);
    att1 :    extend_attribute_type1;
    att2 :    extend_attribute_type2;
END_ENTITY;

ENTITY extend_ent_b
    SUPERTYPE OF (extend_ent_c);
    SUBTYPE OF (outcome);
    att3 :    extend_attribute_type1;
END_ENTITY;

ENTITY extend_ent_c
    SUBTYPE OF (extend_ent_b);
    att4 :    extend_attribute_type1;
    att5 :    SET OF [0:?] cost;
END_ENTITY;

END_SCHEMA;
```

4.4.2 Extending services

Any application can provide services beyond those defined in this standard; however, such services will not be recognized as conforming to the standard.

4.5 Status codes

An AI-ESTATE diagnostic reasoner shall provide a means for its clients to determine the success or failure of service requests. This shall be implemented by means of the exception

entity within the Dynamic Context Model. This entity will assume one of the following mnemonic values:

```
Operation_completed_successfully
Nonexistent_data_element_requested
Missing_or_invalid_argument
Operation_out_of_sequence
Invalid_model_schema
Service_not_available
Unknown_exception_raised
```

The actual mapping will be provided by the implementation.

4.6 Conformance

This subclause defines the requirements for conformance with IEEE Std 1232-2002. It defines the minimum capabilities that are required and what subsets and extensions are allowed. This standard specifies minimal conformance for IEEE Std 1232-2002 conformant model development systems and IEEE Std 1232-2002 conformant application runtime systems. It is expected that actual implementations will conform at levels beyond the minimal levels.

Extensions to the information models are only allowed through the facility of the EXTEND schema. An EXTEND schema shall conform to the formal EXPRESS syntax and semantics found in ISO 10303-11:1994. These extensions shall be clearly identified as specified in 4.4 and should be submitted to the IEEE D & MC subcommittee. An implementation may extend the data and knowledge specification defined in this and subsequent clauses provided that the implementation clearly identifies what extensions have been implemented.

For an application to be IEEE Std 1232-2002 conformant, the specific elements of the standard to which the application conforms shall be specified and published using the matrix in Table 1. Minimal requirements for filling out this matrix are provided in 4.6.1 and 4.6.2.

Core model elements are those elements in the information model that exist as required attributes of some other required entity within the information model. Enhanced model elements are those elements in the information model that exist as optional attributes of required entities or as any kind of attribute of optional entities within the information model. Core model management services are those services that manipulate core entities in the information models. Enhanced model management services are those services that manipulate enhanced entities in the information model. Core reasoner

Table 1—Conformance matrix format

		CEM	FTM	DIM	EDIM	DCM
Model exchange	Read core model elements					
	Write core model elements					
	Read enhanced model elements					
	Write enhanced model elements					
Application services	Core model management services					
	Enhanced model management services					
	Core reasoner manipulation services					
	Enhanced reasoner manipulation services					

services are those reasoner manipulation services listed within the standard as ''required,'' and enhanced reasoner services are those reasoner manipulation services listed within this standard as ''optional.''

To claim conformance to core model element exchange, at a minimum the core models shall include at least one of the FTM, DIM, EDIM, or DCM *in addition to* the CEM.

To claim conformance to core model management services, at a minimum the core services shall include at least those for one of the FTM, DIM, EDIM, or DCM *in addition to* those for the CEM.

To claim conformance to enhanced model element exchange, the corresponding core model exchange conformance shall also be claimed. When claiming conformance to enhanced model exchange, the specific enhanced elements shall be listed.

To claim conformance to reasoner manipulation services, one indicates conformance under the DCM.

To claim conformance to the enhanced services, the corresponding core service conformance shall also be claimed. When claiming conformance to enhanced services, the specific services shall be listed.

Note that certain services that operate on elements of the DCM depend on the reasoner being in the proper state and may depend on other services having been performed first. State dependence tables are provided in 4.7 (Table 4 and Table 5).

4.6.1 Minimal model development conformance

To claim minimal conformance to IEEE Std 1232-2002 for model development, a conformance matrix containing at least the following shall be provided, as shown in Table 2.

Table 2—Sample conformance model matrix

	CEM	FTM	DIM	EDIM
Read/write core model elements	All model elements	All model elements for at least one model shall be specified		
Read/write enhanced model elements	Specify which.model elements	N/A[a]		

[a]Not applicable.

4.6.2 Minimal runtime environment conformance

To claim minimal conformance to IEEE Std 1232-2002 for the application runtime environment, a conformance matrix containing at least the items listed in Table 3 shall be provided.

4.7 Service order dependence

The basic execution model for an AI-ESTATE conformant diagnostic reasoner assumes the ability to manipulate the Dynamic Context Model (DCM) to maintain reasoner state and follows the high-level state diagram shown in Figure 6.

Table 3—Sample runtime conformance matrix format

	CEM	FTM	DIM	EDIM	DCM
Core model management services	All services	All services for at least one model shall be specified			All services
Enhanced model management services	Specify which services	N/A[a]	Specify which services	Specify which services	Specify which services
Core reasoner manipulation services	N/A[a]				All services
Enhanced reasoner manipulation services	N/A[a]				Specify which services

[a]Not applicable.

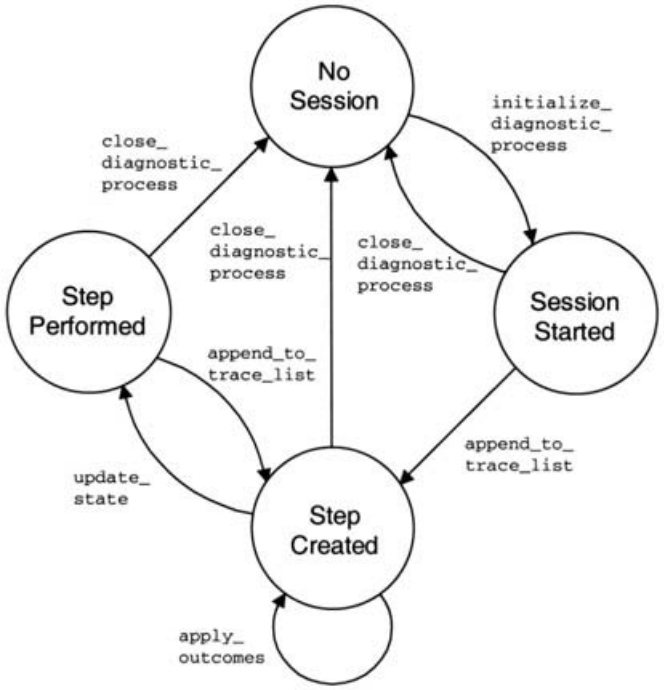


Figure 6—Execution model for an AI-ESTATE conformant diagnostic reasoner

Initially, the reasoner will be in the state indicated by "No Session." From this state, a process is started wherein a session within the DCM is created. Upon initiation of the process, the state transitions to "Session Started." At this point, for reasoning to occur, the reasoner creates a step within the DCM and attaches that step to the session. The state then transitions to "Step Created." From this state, tests are selected, sent to the test system or monitored from a test system, and test information is captured. Once the test information is captured, the resulting test outcomes are applied and the state of the DCM is updated to reflect any inferences drawn. At the same time, the state transitions to "Step Performed." If additional testing is required, another step is created, and the state transitions back to "Step Created." Otherwise, the session and diagnostic process can be closed, and the state returns to "No Session." Note that at any point in the process, the session can be closed with the state transitioning directly to "No Session."

Throughout a diagnostic session, the current step in the session shall be identified as the last instantiated step in that session. All services related to the DCM are executed within the context of the associated session. For a given reasoner, one and only one DCM can be instantiated at a time.

The availability of services is dependent on the reasoner state at any given time. For instance, while reasoning one cannot alter models other than the DCM. The definition of legal states and order dependence of the services defined within this standard are given in the service dependence matrix, shown in Table 4 and Table 5.

Table 4—Core model (CEM, FTM, DIM, EDIM) management services

Service	State	Subclause	Note
create_⟨entitytype⟩	no-session	6.1.1	
delete (ID_⟨entitytype⟩ : entity_id)	no-session	6.1.2	
get_⟨attributename⟩	all	6.1.3	
put_⟨attributename⟩	no-session	6.1.4	
get_⟨attribute_name⟩_count (ID_entity : entity_id)	no-session	6.1.5	
does_⟨attribute_name⟩_exist (ID_entity : entity_id)	no-session	6.1.6	
get_nth_⟨attribute_name⟩	all	6.1.7.1	
get_last_⟨attribute_name⟩	all	6.1.7.2	
put_nth_⟨attribute_name⟩	no-session	6.1.7.3	
put_last_⟨attribute_name⟩	no-session	6.1.7.4	
insert_in_⟨attributename⟩_list	∼no-session[a]	6.1.8.1	
append_to_⟨attributename⟩_list	∼no-session[a]	6.1.8.2	
remove_from_⟨attributename⟩_list	∼no-session[a]	6.1.8.3	
Add_to_⟨attributename⟩_set	no-session	6.1.9.1	
remove_from_⟨attributename⟩_set	no-session	6.1.9.2	
get_type ⟨attributename ⟩	all	6.1.10	

[a]The designation "∼no-session" means all states other than "no-session".

Table 5—Dynamic core model management services. A state dependence column is omitted since all services are available in all states

Service	Subclause
create_⟨entitytype⟩	6.1.1
delete (ID_⟨entitytype⟩ : entity_id)	6.1.2
get_⟨attributename⟩	6.1.3
put_⟨attributename⟩	6.1.4
get_⟨attribute_name⟩_count (ID_entity : entity_id)	6.1.5
does_⟨attribute_name⟩_exist (ID_entity : entity_id)	6.1.6
get_nth_⟨attribute_name⟩	6.1.7.1
get_last_⟨attribute_name⟩	6.1.7.2
put_nth_⟨attribute_name⟩	6.1.7.3
put_last_⟨attribute_name⟩	6.1.7.4
insert_in_⟨attributename⟩_list	6.1.8.1
append_to_⟨attributename⟩_list	6.1.8.2
remove_from_⟨attributename⟩_list	6.1.8.3
add_to_⟨attributename⟩_set	6.1.9.1
remove_from_⟨attributename⟩_set	6.1.9.2
get_type ⟨attributename⟩	6.1.10

5. Models

This clause contains the specifications for all of the information models used within an AI-ESTATE framework. Each of the models is defined using EXPRESS. A brief overview of EXPRESS and EXPRESS-G can be found in Annex B of this standard.

```
(* EXPRESS Specification starts here. *)

(*
```

5.1 Common Element Model

The Common Element Model permits the definition of the form and relationships of systems under test and tests at their most basic level. The diagnosis entity corresponds to a diagnostic conclusion about the system under test. Tests are represented as test entities within the model.

EXPRESS specification:

```
*)
SCHEMA AI_ESTATE_COMMON_ELEMENT_MODEL;
(*
```

5.1.1 Constant

EXPRESS specification:

```
*)
  CONSTANT
    no_fault    :name_type:= 'No Fault';
  END_CONSTANT;
(*
```

5.1.2 bound

This defines a type for specifying upper and lower bounds for numeric-based entities.

EXPRESS specification:

```
*)
  TYPE bound = REAL;
  END_TYPE;
(*
```

5.1.3 confidence_value

This defines a type for specifying "default" confidence values. For most uncertainty-based systems, default confidence is typically taken to be some number near 1.0.

EXPRESS specification:

```
*)
  TYPE confidence_value = REAL;
  WHERE
    range      : (0.0 <= SELF) AND (SELF <= 1.0);
  END_TYPE;
(*
```

Formal propositions:

range : The range of legal values for confidence is restricted, and the actual value is restricted to lie within this legal range.

5.1.4 cost_value

This defines a type for associating predicted values with a numeric attribute.

EXPRESS specification:

```
*)
  TYPE cost_value = REAL;
  END_TYPE;
(*
```

5.1.5 description_type

This type defines means for associating descriptive text with an entity within the model.

EXPRESS specification:

```
*)
  TYPE description_type = STRING;
  END_TYPE;
(*
```

5.1.6 name_type

This type defines a means for associating an identifying name with an entity in the model.

EXPRESS specification:

```
*)
  TYPE name_type = STRING;
  END_TYPE;
(*
```

5.1.7 non_negative_integer

EXPRESS specification:

```
*)
  TYPE non_negative_integer = INTEGER;
  WHERE
    non_negative:  (SELF >= 0);
  END_TYPE;
(*
```

Formal propositions:

non_negative : Ensures that the value is not less than zero.

5.1.8 cost_type

This defines a type for categorizing a cost attribute used to optimize the diagnostic process.

EXPRESS specification:

```
*)
  TYPE cost_type = ENUMERATION OF
    (USER_DEFINED_COST,
    PERFORMANCE,
    SETUP,
    ACCESS,
    REENTRY);
  END_TYPE;
(*
```

5.1.9 diagnosis_value

This defines a type that enables specification of the type of outcome associated with a diagnosis. For a particular diagnosis, the set of outcomes shall include, at a minimum, GOOD and CANDIDATE. Further, no more than one of each is permitted.

EXPRESS specification:

```
*)
  TYPE diagnosis_value = ENUMERATION OF
    (GOOD,
    CANDIDATE,
    USER_DEFINED_DIAGNOSIS_OUTCOME,
    DIAGNOSIS_NOT_KNOWN);
  END_TYPE;
(*
```

5.1.10 goal

Type goal defines an enumeration of possible optimization criteria used to guide diagnosis. Example criteria include minimum cost, minimum expected time, and minimum number of tests.

EXPRESS specification:

```
*)
  TYPE goal = ENUMERATION OF
    (MINIMUM_TESTS,
    MINIMUM_NON_TIME_COST,
    MINIMUM_TIME_COST,
    MINIMUM_EXPECTED_NON_TIME_COST,
    MINIMUM_EXPECTED_TIME_COST,
    USER_DEFINED_OPTIMIZATION);
  END_TYPE;
(*
```

5.1.11 non_time_unit

This type defines legal units for non-time-related costs. Non-time costs are typically incurred in terms of monetary amounts; however, the user-defined value allows for other types of non-time costs to be included.

EXPRESS specification:

```
*)
  TYPE non_time_unit = ENUMERATION OF
    (USER_DEFINED_NON_TIME,
    U_S_DOLLAR,
    POUND_STERLING,
    YEN,
    EURO,
    COUNT);
  END_TYPE;
(*
```

5.1.12 role

This type defines the role associated with a purpose in the required context of a hierarchical element.

EXPRESS specification:

```
*)
  TYPE role = ENUMERATION OF
    (TRAINING,
    VERIFICATION,
    SCHEDULED_MAINTENANCE,
    MAINTENANCE_ACTION,
    READY_FOR_ISSUE,
    USER_DEFINED_ROLE);
  END_TYPE;
(*
```

5.1.13 severity_category

Enumerated type severity_category is used to assign one of four standard values to the severity attribute. Values assigned are one of CATASTROPHIC, CRITICAL, MARGINAL, or MINOR (in decreasing order of severity). Note that this can be used in conjunction with failure probability information (derived from failure rate) to determine the criticality of a diagnosis.

EXPRESS specification:

```
*)
  TYPE severity_category = ENUMERATION OF
    (CATASTROPHIC,
    CRITICAL,
    MARGINAL,
    MINOR);
  END_TYPE;
(*
```

5.1.14 test_value

Type test_value is used to characterize outcomes that are associated with a test.

EXPRESS specification:

```
*)
  TYPE test_value = ENUMERATION OF
    (PASS,
    FAIL,
    TEST_NOT_KNOWN,
    TEST_NOT_AVAILABLE,
    USER_DEFINED_TEST_OUTCOME);
  END_TYPE;
(*
```

5.1.15 time_basis

This type permits specification of the basis for recording time-related information. Specifically, it permits specification of operating time, clock time, or some user-defined version of time.

EXPRESS specification:

```
*)
  TYPE time_basis = ENUMERATION OF
    (NOT_KNOWN,
    OPERATING_TIME,
    CLOCK_TIME,
    USER_DEFINED_BASIS);
  END_TYPE;
(*
```

5.1.16 time_unit

This type defines legal units for time-related costs. The most common units of time (as derived from IEEE Std 716™-1995 [B9], which is referred to as the ATLAS standard) are specified; however, the user-defined value allows for other types of time costs to be included.

EXPRESS specification:

```
*)
  TYPE time_unit = ENUMERATION OF
    (USER_DEFINED_TIME,
    HOURS,
    MINUTES,
    SECONDS,
    MSEC,
    USEC,
    NSEC,
    PSEC);
  END_TYPE;
(*
```

5.1.17 action

Entity action is a specific action taken either to test or repair an item. Actions are the primary entities against which costs are associated. Ultimately, the diagnosis and repair processes are concerned with optimizing the sequence of actions required to return a unit to service.

EXPRESS specification:

```
*)
  ENTITY action;
    has_name              : name_type;
    has_description       : description_type;
    has_cost              : OPTIONAL SET [1:?] OF cost;
    required_resource     : OPTIONAL SET [1:?] OF resource;
    action_frequency      : OPTIONAL frequency;
    order_constraint      : OPTIONAL order_operator;
  UNIQUE
    one_name  : has_name;
  END_ENTITY;
(*
```

Attribute definitions:

has_name	: Provides a unique name for the action.
has_description	: Provides a description of the action.
has_cost	: A set of costs associated with the action represented by this entity; the set is optional but, if specified, shall include at least one cost attribute.
required_resource	: Identifies the resources required to perform a given action. This attribute is optional with a minimum cardinality of one, should it exist.

action_frequency : Specifies, for periodic actions, how frequently a given action needs to be performed.
order_constraint : Attribute order_constraint identifies an ordering precondition that must be satisfied for a particular action to be performed.

5.1.18 atom

Entity atom is a primitive element within the first-order logic. An atom can be either a constant (e.g., TRUE, FALSE, or some number) or a variable that takes on some value. This is a subtype of logic_element and a supertype of one of variable or const.

EXPRESS specification:

```
*)
  ENTITY atom
    ABSTRACT SUPERTYPE OF (ONEOF(variable, const))
    SUBTYPE OF(logic_element);
  END_ENTITY;
(*
```

5.1.19 const

A constant within the first-order logic. This is a subtype of atom and has an associated value.

EXPRESS specification:

```
*)
  ENTITY const
    SUBTYPE OF(atom);
    constant_value      : name_type;
  END_ENTITY;
(*
```

Attribute definitions:

constant_value : Specifies the particular value to be taken on by a constant.

5.1.20 cost

Costs are categorized by the type of cost to which they relate. One dimension of the cost set identifies whether the cost is a measure of time in seconds (or some other unit), or if it is a calculated cost. A time-related cost is a measure of the time it takes to perform a task. A non-time-related cost is an expense that is computed, perhaps in financial terms or by an objective function. The second dimension to the cost group is based on the task to which the cost pertains: performance, setup, access, re-entry.

Each cost has an associated unit to enable consistent processing of included values. Two types of units are defined as enumerated types and include time_unit and non_time_unit. Time_unit is an enumerated type that defines the time units to be associated with a time-related cost. non_time_unit

is an enumerated type covering predominant international currencies and defines types of cost units to be associated with non_time_related_costs.

EXPRESS specification:

```
*)
  ENTITY cost
    ABSTRACT SUPERTYPE OF (ONEOF(time_cost, non_time_cost));
    cost_element                    : OPTIONAL cost_type;
    upper                           : OPTIONAL bound;
    lower                           : OPTIONAL bound;
    predicted_value                 : OPTIONAL cost_value;
    criterion                       : SET [1:?] OF cost_category;
  WHERE
    valid_bound : NOT(EXISTS(upper))                                    OR
              NOT(EXISTS(lower))                                        OR
                 (lower <= upper);
    valid_upper_bound : NOT(EXISTS(predicted_value))                    OR
              NOT(EXISTS(upper))                                        OR
                 (predicted_value <= upper);
    valid_lower_bound : NOT(EXISTS(predicted_value))                    OR
              NOT(EXISTS(lower))                                        OR
                 (lower <= predicted_value);
  END_ENTITY;
(*
```

Attribute definitions:

cost_element : An attribute that specifies what type of cost is being defined. The type is an enumeration consisting of user_defined, performance, setup, access, and re-entry. One of these cost types shall be specified.

upper : An optional attribute providing the nominal upper bound for the value of the cost entity.

lower : An optional attribute providing the nominal lower bound for the value of the cost entity.

predicted_value : An attribute providing the nominal, expected, or some other predicted value for the cost entity.

criterion : Attribute criterion identifies the cost criterion that will be used to classify the cost parameter. For example, criterion can identify skill level as a non-time cost. This attribute is a set because a particular cost item can represent multiple criteria.

Formal propositions:

valid_bound : When they exist, the lower and upper attributes are constrained to values such that the value of the lower attribute is less than or equal to the upper attribute. This constraint permits only one of the bounds to exist.

valid_upper_bound : When they exist, the expected_value and upper attributes are constrained to values such that the value of the expected_value attribute is less than or equal to the upper attribute.

valid_lower_bound : When they exist, the expected_value and lower attributes are constrained to values such that the value of the lower attribute is less than or equal to the expected_value attribute.

5.1.21 cost_category

EXPRESS specification:

```
*)
  ENTITY cost_category;
    description     : description_type;
    name            : name_type;
  UNIQUE
    one_name        : name;
  END_ENTITY;
(*
```

Attribute definitions:

`description` : Attribute `description` is used to provide an elaborated description of what the cost category is.
`name` : Attribute `name` provides a means for identifying the cost category.

5.1.22 cost_rate

In addition to providing basic time and cost units, it may be necessary to associate a `cost_rate` with an entity (e.g., resource). The `cost_rate` attribute defines a ratio of non-time cost to some unit of time and can be used, e.g., to specify labor rates or rental rates.

EXPRESS specification:

```
*)
  ENTITY cost_rate;
    time_cost_unit            : time_unit;
    denominator               : cost_value;
  WHERE
    denominator_is_non_zero   : SELF.denominator <> 0;
  END_ENTITY;
(*
```

Attribute definitions:

`time_cost_unit` : An attribute to provide the time units for computing the cost rate. In other words, this attribute will provide the time part of cost per unit time.
`denominator` : An attribute that provides a divisor for the `cost_rate`, thus permitting values other than unity.

Formal propositions:

`denominator_is_non_zero` : Since the `cost_rate` entity defines a ratio that permits specification of the denominator, this rule constrains the denominator to be nonzero, thus preventing dividing by zero.

5.1.23 diagnosis

Diagnosis can be used for the representation of a lattice of diagnostic conclusions. Typically, a diagnosis consists of failures or fault. The diagnoses can be interconnected in a lattice to classify constituent units, e.g., by function. This construct provides a generalized grouping mechanism for the constituents of the unit under test and their failure modes. The diagnosis lattice resulting from using a structure of these entities shall not be cyclic, in the sense that no diagnosis be an ancestor of itself.

Example: A diagnosis might be of system under test #4 (flight computer), consisting of a diagnosis of BOARD #1 (A/D converter), BOARD #2 (CPU), BOARD #3 (Display), etc. A diagnosis of BOARD #3 may consist of diagnoses of CHIP #1, CHIP #2, CHIP #3, and all may be at different levels.

EXPRESS specification:

```
*)
  ENTITY diagnosis
    SUPERTYPE OF (ONEOF(failure, fault))
    SUBTYPE OF(hierarchical_element);
    has_outcome                : OPTIONAL SET [2:?] OF diagnosis_outcome;
    has_rate                   : OPTIONAL failure_rate;
    mechanism                  : OPTIONAL SET [1:?] OF failure_mode;
    severity                   : OPTIONAL severity_category;
  WHERE
    outcomes_required_for_atomic_diagnosis  :  (SIZEOF(SELF.members)    >
             0) OR EXISTS(has_outcome);
    minimal_outcomes  :  (NOT(EXISTS(has_outcome)))                    XOR
             ((SIZEOF(QUERY(tmp          <*       has_outcome          |
                tmp.standard_diagnosis_value = GOOD)) = 1)             AND
             (SIZEOF(QUERY (tmp    <*    has_outcome                    |
                tmp. standard_diagnosis_value  =  CANDIDATE))  =  1));
    mechanism_at_leaves  : NOT(EXISTS(mechanism)) OR (SIZEOF(members) = 0);
    faults_at_leaves : (NOT('AI_ESTATE_COMMON_ELEMENT_MODEL.fault'      IN
             TYPEOF(SELF)))                                            OR
             (SIZEOF(members) = 0);
    children_same_type  :  QUERY(tmp  <*  SELF.members                 |
               NOT('AI_ESTATE_COMMON_ELEMENT_MODEL.diagnosis'
                    IN TYPEOF(tmp))) = [];
  END_ENTITY;
(*
```

Attribute definitions:

has_outcome : A set of two or more outcomes that are the expected outcomes of the diagnosis. An outcome is a characterization of the logical value for the corresponding truth or falsity of the diagnosis applying to the current state. This attribute is shown to be optional; however, it is constrained such that it is required if the diagnosis has no member diagnoses. Since diagnosis outcomes are the basis for determining the final conclusions in AI-ESTATE, outcomes shall be available.

NOTE—Diagnoses in the data model are not restricted to two outcomes; any number of outcomes are permitted for a test, but at least two outcomes are required.

`has_rate` : A value indicating the measure of the expected failure rate of the diagnosis, i.e., the number of failures per time span. This value can be converted to a probability of failure.

`mechanism` : Identifies the failure mode or modes associated with this diagnosis.

`severity` : Attribute `severity` associates a level of severity for the given diagnosis. This attribute is optional since not all models need to use this information. Criticality can be derived by selecting all of the diagnoses at a particular severity level and then ranking them by their failure probabilities, which are derived from the failure rates.

Formal propositions:

`outcomes_required_for_atomic_diagnosis` : Determines whether or not outcomes are associated with a diagnosis and requires that an atomic diagnosis (i.e., a diagnosis for which there are no sub-diagnoses) have outcomes. The cardinality on the `diagnosis_outcome` set ensures that, should outcomes exist, there are at least two of them. Note that nonatomic diagnoses are permitted, but not required, to have diagnostic outcomes.

`minimal_outcomes` : Requires that either the set of outcomes not be defined, or that the set of outcomes include, at a minimum, exactly one outcome of value GOOD and exactly one outcome of value CANDIDATE.

`mechanism_at_leaves` : This rule ensures that a `mechanism` (i.e., a cause) for a given diagnosis to be true only occurs at the leaf of the diagnosis lattice. Note that the leaf is not required to provide the mechanism.

`faults_at_leaves` : This rule ensures that a diagnosis of type `fault` can only exist at the leaf of the diagnosis lattice. Note that the leaf of the lattice is not required to be of type `fault`.

`children_same_type` : Ensures that all children (i.e., members) of a diagnosis are also of type diagnosis.

5.1.24 diagnosis_outcome

Entity `diagnosis_outcome` is a set of outcomes associated with any diagnosis in a model. Diagnosis outcomes associate discrete values to ranges of states in the diagnosis definition and form the basis for the diagnostic process. For any given diagnosis in an AI-ESTATE model, at least two outcomes shall be defined. Thus, at a minimum, outcomes corresponding, e.g., to good/candidate should be provided with a given specification. At times more than two outcomes may be appropriate, e.g., good/suspect/candidate/bad.

EXPRESS specification:

```
*)
  ENTITY diagnosis_outcome
    SUBTYPE OF(outcome);
    standard_diagnosis_value    : diagnosis_value;
  INVERSE
    for_diagnosis               : diagnosis FOR has_outcome;
  END_ENTITY;
(*
```

Attribute definitions:

standard_diagnosis_value : This attribute identifies whether the outcome is a standard value from the perspective of traditional diagnostic outcomes. In most model-based systems, the standard values are GOOD and CANDIDATE. A third value, USER_DEFINED_DIAGNOSIS_OUTCOME, is used to indicate whether additional outcomes have been defined.

for_diagnosis : Links the outcome to the specific diagnosis with that outcome.

5.1.25 diagnostic_model

Diagnostic reasoning involves drawing conclusions from test outcomes. The relationships between test outcomes and diagnostic conclusions are defined in diagnostic models. A diagnostic model is a collection of entities that provide information about the system under test. A diagnostic_model refers to some AI-ESTATE conformant model as specified in this standard. Entities from the Common Element Model are referenced as required by those diagnostic models.

EXPRESS specification:

```
*)
  ENTITY diagnostic_model;
    name                 : name_type;
    description          : description_type;
    model_element        : SET [3:?] OF hierarchical_element;
    orders_steps         : OPTIONAL SET [1:?] OF order_operator;
  UNIQUE
    one_name  : name;
  WHERE
    one_diagnosis_is_no_fault  : (SIZEOF(QUERY(tmp <* model_element |
                    tmp.name = 'no_fault')) = 1);
    element_is_rollup  :  element_rollup(SELF,SELF.model_element);
  END_ENTITY;
(*
```

Attribute definitions:

name : An attribute used to identify the entity uniquely.
description : Attribute used to provide an elaborated explanation of the model.
model_element : An attribute to identify the various model elements in a diagnostic model. Model elements are any of the hierarchical elements as defined by the information model (i.e., tests, diagnoses, repairs, resources, functions).
orders_steps : This attribute identifies an optional set of ordering operators to override any optimization process that may be applied for diagnosis.

Formal propositions:

one_diagnosis_is_no_fault : This rule verifies that one of the diagnosis entities defined in the model has a name that corresponds to the constant no_fault. This rule ensures that any AI-ESTATE conformant model explicitly defines a special diagnostic conclusion indicating no fault found.
element_is_rollup : This rule verifies that all of the hierarchical elements referenced at this level are defined as being part of this model. There is a companion rule for the hierarchical elements to ensure that they are listed in the rollup at this level.

5.1.26 failure

A failure is a manifestation of a fault within a system. When considering a functional model, it is the failure of the system under test to perform some intended function.

EXPRESS specification:

```
*)
  ENTITY failure
    SUBTYPE OF(diagnosis);
    failed_item    : OPTIONAL func;
  END_ENTITY;
(*
```

Attribute definitions:

failed_item : Identifies the specific element (i.e., function) that has failed. This is a direct tie to the functional part of the system associated with a diagnosis.

5.1.27 failure_mode

A failure mode is a specific mode or means by which a unit or system can fail. The specific failure mode is associated, first with the diagnosis, then (by way of the appropriate subtype) to the item that has failed.

EXPRESS specification:

```
*)
  ENTITY failure_mode;
    name                    : name_type;
    description             : description_type;
  UNIQUE
    one_name  : name;
  END_ENTITY;
(*
```

Attribute definitions:

`name` : Used to identify the failure mode uniquely.
`description` : Attribute used to provide an elaborated explanation of what the failure mode is.

5.1.28 failure_rate

This entity indicates the rate of occurrence of a diagnosis (including its subtypes fault and failure). The `failure_rate` can either be an actual observed value or an estimate. To specify `failure_rate`, a `time_basis` shall be defined. `Time_basis` is used in the context of determining the computational basis for failure rate statistics.

EXPRESS specification:

```
*)
  ENTITY failure_rate;
    description   : description_type;
    basis         : time_basis;
  END_ENTITY;
(*
```

Attribute definitions:

`description` : Attribute used to provide an elaborated explanation of the failure rate.
`basis` : An attribute to define the basis for computing the failure rate. In particular, this attribute identifies if the `failure_rate` is computed based on operation time or clock time.

5.1.29 fault

A fault is the physical cause of anomalous behavior within a system. A fault typically corresponds to some physical breakdown in the system under test.

EXPRESS specification:

```
*)
  ENTITY fault
    SUBTYPE OF(diagnosis);
    failed_item   : OPTIONAL repair_item;
  END_ENTITY;
(*
```

Attribute definitions:

failed_item : Identifies the specific element (i.e., repair item) that has failed. This is a direct tie to the physical part of the system associated with a diagnosis.

5.1.30 frequency

Frequency is the rate (i.e., the number of times something occurs per unit time).

EXPRESS specification:

```
*)
  ENTITY frequency;
    time_span      : time_cost;
    action_count   : non_negative_integer;
  END_ENTITY;
(*
```

Attribute definitions:

time_span : Attribute time_span identifies the span of time between periodic actions. While this attribute is of type time_cost note that the semantics of this type are identical to a time-related entity.

action_count : Attribute action_count identifies the number of times in a time span the action is performed.

5.1.31 func

A function corresponds to any "functional behavior" represented within a system. The func entity is intended to provide a placeholder for functional information in the event the diagnostic or testability model is function-oriented. Note the type of entity cannot be "function" since this is a reserved word in EXPRESS.

EXPRESS specification:

```
*)
  ENTITY func
    SUBTYPE OF(hierarchical_element);

  INVERSE
    implemented_by       : SET [1:?] OF repair_item FOR includes_function;
  WHERE
    children_same_type   : QUERY(tmp  <*  SELF.members                    |
                           NOT('AI_ESTATE_COMMON_ELEMENT_MODEL.func'
                           IN TYPEOF(tmp)))=[];
  END_ENTITY;
(*
```

Attribute definitions:

implemented_by : This attribute identifies the repair item or items used to implement a particular function within a system.

Formal propositions:

children_same_type : Ensures any children (i.e., members) of func are of type func.

5.1.32 hierarchical_element

A hierarchical element is an abstract entity containing a name, description, and associated lattice structure. It is intended to roll up common attributes of entities such as test, diagnosis, repair_item, func, and resource.

EXPRESS specification:

```
*)
  ENTITY hierarchical_element
    ABSTRACT SUPERTYPE OF (ONEOF(diagnosis, repair, resource, test, func,
                           repair_item));
    name                 : name_type;
    description          : description_type;
    members              : SET OF hierarchical_element;
    at_indenture_level   : OPTIONAL level;
    must_occur_in        : SET OF required_context;
  INVERSE
    part_of_model        : SET [1:?] OF diagnostic_model FOR model_element;
    parents              : SET OF hierarchical_element FOR members;
  UNIQUE
    one_name  : name;
  WHERE
    graph_is_acyclic     : element_dag(members,parents);
    parent_level_consistent : ((NOT(EXISTS(at_indenture_level)))       AND
              (SIZEOF(QUERY(tmp <* parents | EXISTS(at_indenture_level))) =
              0))                                                       XOR
              (SIZEOF(QUERY (tmp  <*  parents                             |
                  (SELF.at_indenture_level = tmp.at_indenture_level)
                    OR
                  (SELF. at_indenture_level.predecessor                   =
              tmp.at_ indenture_level)))                                  =
                  SIZEOF(SELF.parents));
  END_ENTITY;
(*
```

Attribute definitions:

name	: An attribute used to identify uniquely the particular hierarchical element.
description	: Attribute used to provide an elaborated explanation of what the hierarchical element is.
members	: The set of constituent elements of which this entity is composed. This attribute is required but the set can be empty (e.g., if the element is at the lowest level of indenture).
at_indenture_level	: Identifies the specific level of indenture for a particular hierarchical element.
must_occur_in	: Attribute must_occur_in provides a list of context items required for a hierarchical element to be considered. This attribute

is defined to be a set. If the set is empty, then the corresponding hierarchical element is available in all contexts.

part_of_model : A pointer to a set of models that reference a given hierarchical element. This attribute provides a mechanism for linking models based on hierarchical elements identified during the diagnostic process.

parents : The set of elements that have the current element as a member. This attribute is required but the set can be empty (e.g., if the element is at the highest level of indenture).

Formal propositions:

graph_is_acyclic : The structure of hierarchical_element is referred to as a *lattice* and corresponds to a directed acyclic graph. In other words, traversing the member relationships from a hierarchical_element should not result in returning to the same hierarchical_element. This constraint is verified by using the function element_dag, which traverses the hierarchical_element structure.

parent_level_consistent : This proposition ensures that either no indenture levels exist from the hierarchy or that the indenture levels are consistent. Specifically, the constraint is imposed that the level of the parent elements are the same as the current element or are predecessors to the current element.

5.1.33 level

This entity refers to the level of indenture for a particular hierarchical element. This defines a common level for purposes of constraining diagnosis or analysis for testability and diagnosability assessment. A total order of levels is defined for a particular model/system.

EXPRESS specification:

```
*)
  ENTITY level;
    name               : name_type;
    description        : description_type;
    successor          : OPTIONAL level;
  INVERSE
    predecessor        : level FOR successor;
  UNIQUE
    one_name  : name;
  WHERE
    levels_distinct: (NOT(EXISTS(SELF.successor))                    XOR
                     (SELF  <>  SELF.successor))                     AND
                     (NOT(EXISTS(SELF.predecessor))                  XOR
                     (SELF  <>  SELF.predecessor));
    consistent_level: (NOT(EXISTS(SELF.predecessor))                 XOR
                     (SELF  =  SELF.predecessor.successor))          AND
                     (NOT(EXISTS(SELF.successor))                    XOR
                     (SELF  =  SELF.successor.predecessor));
    no_repeats:  levels_acyclic(SELF);
  END_ENTITY;
(*
```

Attribute definitions:

name : Specifies a unique name for the level of the hierarchical element. This attribute is intended to provide a means of identifying levels for purposes of determining applicability of tests, diagnoses, etc., within the model.
description : Attribute used to provide an elaborated explanation of what the level is.
successor : Identifies the "next" level in the total order of levels. This is optional in that the final level will not have a successor.
predecessor : Identifies the "previous" level in the total order of levels when the previous level exists.

Formal propositions:

levels_distinct : Ensures that a particular level is distinct from its successor or predecessor level.
consistent_level : Ensures that the predecessor level points to the current level as successor and that the successor level points to the current level as predecessor.
no_repeats : Tests to see if the current entity has the same level appearing in the successor chain. If so, it creates a cycle in the levels, which is illegal.

5.1.34 logic_element

The entity logic_element identifies a particular element to be used in a logical expression. This is an abstract supertype of atom (a primitive element within logic) and function (an n-ary term that returns a single value).

EXPRESS specification:

```
*)
  ENTITY logic_element
    ABSTRACT SUPERTYPE OF (ONEOF(atom, logic_function));
    label:                name_type;
  UNIQUE
    one_label: label;
  END_ENTITY;
(*
```

Attribute definitions:

label : Associates an identifying label to the logical element. This label corresponds to, e.g., variable names, function names, and predicate names.

5.1.35 logic_function

The entity logic_function is a function used in first-order logic. This is a subtype of logic_element and a supertype of predicate.

EXPRESS specification:

```
*)
  ENTITY logic_function
    SUBTYPE OF(logic_element);
    argument                  : LIST OF logic_element;
  END_ENTITY;
(*
```

Attribute definitions:

argument : Specifies a list of logic elements as arguments to the given logic_function. Note that the argument list can be empty; however, an argument list is required. The cardinality of set of arguments defines the arity of the function.

5.1.36 mode_of_operation

Entity mode_of_operation specifies a particular mode or state of operation for a unit or system. For example, for an aircraft, modes of operation may correspond to weight-on-wheels or in-flight.

EXPRESS specification:

```
*)
  ENTITY mode_of_operation;
    name             : name_type;
    description      : description_type;
  UNIQUE
    one_mode         : name;
  END_ENTITY;
(*
```

Attribute definitions:

name : Attribute name is a unique attribute used to identify the mode of operation.
description : Attribute description provides a textual description of the specific mode of operation within which the system is being diagnosed.

5.1.37 non_time_cost

The non_time_cost construct represents cost attributes that are not time related and are computed. Financial costs as well as costs computed from an objective function fall into this category. Objective functions typically combine functions of weighted cost parameters that can be temporal, financial, or a quantification of some other expense of the test. Units for objective function-based costs would typically be defined as user_defined. The non_time_cost entity inherits all attributes from cost.

EXPRESS specification:

```
*)
  ENTITY non_time_cost
    SUBTYPE OF(cost);
    non_time_cost_unit  : non_time_unit;
```

```
    rate                  : OPTIONAL cost_rate;
  END_ENTITY;
(*
```

Attribute definitions:

non_time_cost_unit : An attribute to define the non-time units associated with the non-time-related cost.

rate : An optional attribute to provide a specific non-time cost to be associated with the rate. In other words, this attribute will provide the cost part of cost per unit time.

5.1.38 order_operator

Logically equivalent to the STRIPS operator first formalized by Fikes and Nilsson [B1], this defines an operator by which actions are taken within the test and diagnostic environment. The order operator (as defined for STRIPS) includes a name, a set of preconditions (i.e., logical conditions that must be satisfied for the operator to be applied), and a set of effects (i.e., modifications to the truth state of the problem at hand). The intent is to use these operators in constructing a plan to perform a test and diagnostic task. Order operators also correspond to actions in this model.

EXPRESS specification:

```
*)
  ENTITY order_operator;
    name                  : name_type;
    description           : description_type;
    precondition          : SET OF predicate;
    effect                : SET OF predicate;
  INVERSE
    implements_action     : action FOR order_constraint;
  UNIQUE
    one_name              : name;
  END_ENTITY;
(*
```

Attribute definitions:

name : Identifies the name of the ordering operator.

description : Provides a textual description of the ordering operator.

precondition : Identifies a set of logical conditions that must be satisfied for the operator to be applied. The set of preconditions corresponds to a conjunction of the individual predicates.

effect : Identifies a set of logical effects on the truth state of the planning problem (i.e., diagnostic problem). The effects are a conjuncted set of predicates now assumed to be true.

implements_action : Associates the ordering operator with a specific action within the model. Since actions have costs associated with them, this relationship also facilitates optimal planning. Note that an action can exist independent of an order operator (as indicated by the OPTIONAL modifier); however, an order operator CANNOT exist without an associated action.

5.1.39 outcome

Associated with any test or diagnosis in a model is a set of outcomes. Outcomes associate discrete values to ranges of measurements in the definition and form the basis for the diagnostic process. For any given test or diagnosis in an AI-ESTATE model, at least two outcomes shall be defined. Thus, at a minimum, outcomes corresponding, e.g., to pass/fail or good/candidate should be provided with a given specification. At times, more than two outcomes may be appropriate, e.g., pass/fail-low/fail-high or good/suspect/candidate/bad.

EXPRESS specification:

```
*)
  ENTITY outcome
    ABSTRACT SUPERTYPE OF (ONEOF(diagnosis_outcome, test_outcome));
    name             : name_type;
    description      : description_type;
    confidence       : OPTIONAL confidence_value;
  UNIQUE
    one_name         : name;
  END_ENTITY;
(*
```

Attribute definitions:

name	: An attribute used to identify the outcome uniquely.
description	: Attribute used to provide an elaborated explanation of what the outcome is.
confidence	: A measure of the confidence in the test outcome based on characteristics of performing the test. This measure is used for providing confidence measures in the diagnoses.

5.1.40 predicate

A predicate is a special kind of function in which the assigned values are limited to either TRUE or FALSE. This is one of the most common constructs used in sentences within the first-order logic. A predicate is a subtype of logic_function.

EXPRESS specification:

```
*)
  ENTITY predicate
    SUBTYPE OF(logic_function);
  END_ENTITY;
(*
```

5.1.41 purpose

Entity purpose specifies a given purpose for the context. In other words, it states the purpose for the associated action (i.e., test, diagnosis, or repair).

EXPRESS specification:

```
*)
  ENTITY purpose;
    has_role         : SET [1:?] OF role;
    optimizer        : SET [1:?] OF goal;
    description      : description_type;
  END_ENTITY;
(*
```

Attribute definitions:

has_role : This attribute identifies the associated role of the context within which test/diagnosis is occurring. For example, the role can be a maintenance test or a verification test.

optimizer : Attribute optimizer identifies the specific optimization criterion or criteria used to drive the diagnostic process.

description : Attribute description provides a textual description of the purpose of the context.

5.1.42 repair

When a repair item has been identified, some repair shall be made to return the unit to service. This entity associates a repair (with its various repair actions) to a repair item.

EXPRESS specification:

```
*)
  ENTITY repair
    SUBTYPE OF(hierarchical_element);
    repair_action:                    LIST [1:?] OF action;
  WHERE
    children_same_type: QUERY(tmp  <*  SELF.members                          |
                        NOT('AI_ESTATE_COMMON_ ELEMENT_MODEL.repair'
                          IN TYPEOF(tmp))) = [];
  END_ENTITY;
(*
```

Attribute definitions:

repair_action : Identifies the sequence of actions required to repair the repair item. Note that if preconditions need to be satisfied for execution of the repair, then an order operator must be associated with at least the first action in the repair action list containing the necessary preconditions.

Formal propositions:

children_same_type : Ensures that any children (i.e., members) of a repair are of type "repair."

5.1.43 repair_item

A diagnosis shall point to some part of the system to be adjusted, calibrated, repaired, replaced, etc. In short, a diagnosis shall lead the maintainer to the object of the maintenance action. The "physical"

entity in the system under test corresponding to the diagnosis (thus the entity diagnosis) is the entity defined by repair item and identifies a failure mode of the system under test.

EXPRESS specification:

```
*)
  ENTITY repair_item
    SUBTYPE OF(hierarchical_element);
    includes_function   : OPTIONAL SET [1:?] OF func;
    tested_by           : SET OF test;
    repaired_by         : SET OF repair;
  WHERE
    children_same_type : QUERY(tmp  <*  SELF.members                    |
                        NOT('AI_ESTATE_COMMON_ELEMENT_MODEL.repair_item'
                        IN TYPEOF(tmp))) = [];
  END_ENTITY;
(*
```

Attribute definitions:

includes_function	: This attribute identifies the set of functions that are performed by a given repair item.
tested_by	: This attribute identifies a set (possibly empty) of tests available to test a particular repair item.
repaired_by	: This attribute identifies a (possibly empty) set of repairs available to repair a given repair item.

Formal propositions:

children_same_type	: Ensures any children (i.e., members) of a repair_items are of type repair_item.

5.1.44 required_context

The entity required_context specifies the context required for a diagnostic problem (or portion of a problem) to be valid. Diagnostic models are generally created with a particular context (or contexts) in mind. This entity is a supertype of required_test_context AND required_diagnosis_context AND required_repair_context.

EXPRESS specification:

```
*)
  ENTITY required_context
    SUPERTYPE OF (required_test_context AND required_diagnosis_context AND
                 required_repair_context);
    name                  : name_type;
    description           : description_type;
    occurs_in             : SET OF mode_of_operation;
    has_purpose           : purpose;
  UNIQUE
    one_name: name;
  END_ENTITY;
(*
```

Attribute definitions:

name	: Attribute name provides a unique identifier for a particular required context within the model.
description	: Attribute description provides a means of associating descriptive text with the required context.
occurs_in	: Identifies the relevant operational mode within which the particular maintenance actions are being performed.
has_purpose	: Attribute has_purpose identifies the purpose of actions performed in the given context.

5.1.45 required_diagnosis_context

Entity required_diagnosis_context is a subtype of required_context and specifies the diagnosis context required for a particular diagnosis (or group of diagnoses) within a model to be valid.

EXPRESS specification:

```
*)
  ENTITY required_diagnosis_context
    SUBTYPE OF(required_context);
  END_ENTITY;
(*
```

5.1.46 required_repair_context

Entity required_repair_context is a subtype of required_context and specifies the repair context required for a particular repair item (or group of repair items) within a model to be repaired.

EXPRESS specification:

```
*)
  ENTITY required_repair_context
    SUBTYPE OF(required_context);
  END_ENTITY;
(*
```

5.1.47 required_test_context

Entity required_test_context is a subtype of required_context and specifies the test context required for a particular test (or group of tests) within a model to be performed.

EXPRESS specification:

```
*)
  ENTITY required_test_context
    SUBTYPE OF(required_context);
  END_ENTITY;
(*
```

5.1.48 resource

Any given action in the maintenance process is likely to require some external resource to perform that action. For example, a test could require test equipment and expendable resources, and a repair action could also require instrumentation, tools, and expendable resources. For any given action corresponding to test, a set of resources could be required.

EXPRESS specification:

```
*)
  ENTITY resource
    SUBTYPE OF(hierarchical_element);
    has_cost:                      OPTIONAL SET [1:?] OF cost;
  WHERE
    children_same_type : QUERY(tmp <* SELF.members                          |
                         NOT('AI_ESTATE_COMMON_ELEMENT_MODEL.resource'
                         IN TYPEOF(tmp))) = [];
  END_ENTITY;
(*
```

Attribute definitions:

has_cost : This attribute (optionally) associates a set of costs with this particular resource.

Formal propositions:

children_same_type : Ensures that any children (i.e., members) of a resource are of type resource.

5.1.49 test

A test typically involves a stimulus that is either supplied or known at the start of the test, and a set of observed responses. The interpretation of the responses is provided in the form of a test outcome. The test entity provides a means of defining the logical structure of testing in a diagnostic model and defines a part–whole hierarchy. A given test can be a member of a collection of tests at another level. Sequencing constraints for performing these tests are included in a separate constraint specification or are accessible elsewhere within the AI-ESTATE implementation. Tests are identified symbolically within the specification.

Details concerning the execution of a specific test are contained in other specifications that can be referenced symbolically but are not otherwise included in this specification.

EXPRESS specification:

```
*)
  ENTITY test
    SUBTYPE OF(hierarchical_element);
    has_outcome                    : OPTIONAL SET [2:?] OF test_outcome;
    test_action                    : LIST [1:?] OF action;
```

```
  WHERE
   outcomes_required_for_atomic_test:  (SIZEOF(SELF.members) > 0) OR
                                       EXISTS(has_outcome);
   minimal_outcomes: (NOT(EXISTS(has_outcome)))                          XOR
                     ((SIZEOF(QUERY(tmp <* has_outcome                     |
                       tmp.standard_test_value = PASS)) = 1)               AND
                     (SIZEOF (QUERY(tmp <* has_outcome                     |
                       tmp.standard_test_value = FAIL)) = 1));
   children_same_type:  QUERY(tmp  <*  SELF.members                        |
                        NOT('AI_ESTATE_COMMON_ELEMENT_MODEL.test'
                        IN TYPEOF(tmp))) = [];
  END_ENTITY;
(*
```

Attribute definitions:

`has_outcome` : A set of two or more outcomes that are the expected outcomes of the test. An outcome is a characterization of the observed response to the stimulus of a test. This attribute is shown to be optional; however, it is constrained such that it is required if the test has no member tests. Since test outcomes are the basis for diagnostic reasoning in AI-ESTATE, outcomes shall be available.

NOTE—Tests in the data model are not restricted to two outcomes; any number of outcomes are permitted for a test, but at least two outcomes are required.

Example: All tests of a system might be two-output tests with outcomes pass and fail, and have standard confidences for each outcome across all tests. In a model of such a system, two instances of the outcome entity would exist—one for pass and one for fail. All tests within the model would reference the same set of two outcomes in the `test_outcome` attribute.

`test_action` : Identifies the sequence of actions required to perform the associated test. Note that if preconditions need to be satisfied for execution of the test, then an order operator must be associated with at least the first action in the test action list containing the necessary preconditions.

Formal propositions:

`outcomes_required_for_atomic_test` : Determines whether outcomes are associated with a test and requires that an atomic test (i.e., a test for which there are no subtests) have outcomes. The cardinality on the `test_outcome` set ensures that, should outcomes exist, there are at least two of them. Note that nonatomic test are permitted, but not required, to have test outcomes.

`minimal_outcomes` : Requires either that the set of outcomes not be defined, or that the set of outcomes include, at a minimum, exactly one of outcome of value PASS and exactly one outcome of value FAIL.

`children_same_type` : Ensures that any children (i.e., members) of a test are of type "test."

5.1.50 test_outcome

Associated with any test in a model is a set of test outcomes. Test outcomes associate discrete values to ranges of measurements in the test definition and form the basis for the diagnostic process. For any given test in an AI-ESTATE model, at least two outcomes shall be defined. Thus, at a minimum, outcomes corresponding, e.g. to pass/fail should be provided with a given specification. At times more than two outcomes may be appropriate, e.g., pass/fail-low/ fail-high.

EXPRESS specification:

```
*)
  ENTITY test_outcome
    SUBTYPE OF(outcome);
    standard_test_value    : test_value;
  INVERSE
    for_test               : test FOR has_outcome;
  END_ENTITY;
(*
```

Attribute definitions:

standard_test_value : This attribute identifies whether or not the outcome is a standard value from the perspective of traditional test outcomes. In most model-based systems, the standard values are PASS and FAIL. A third value, USER_DEFINED_TEST_OUTCOME, is used to indicate whether additional outcomes have been defined.

for_test : Links the outcome to the specific test with that outcome.

5.1.51 time_cost

The time_cost construct represents cost attributes that are time-related. The time_cost entity inherits all attributes from cost.

EXPRESS specification:

```
*)
  ENTITY time_cost
    SUBTYPE OF(cost);
    time_cost_unit     : time_unit;
  END_ENTITY;
(*
```

Attribute definitions:

time_cost_unit : An attribute to define the time units associated with the time-related cost.

5.1.52 variable

An atom within first-order logic that has the feature of being able to have a value assigned to it. It is a subtype of atom.

EXPRESS specification:

```
*)
  ENTITY variable
    SUBTYPE OF(atom);
  END_ENTITY;
(*
```

5.1.53 element_dag

EXPRESS specification:

This function examines the set of hierarchical elements in a model and ensures that no child element is also a parent. In other words, when traversing the child relations, it ensures that no cycles exist in the model.

```
*)
  FUNCTION element_dag
    (members:SET [0:?] OF hierarchical_element;
    parents:SET [0:?] OF hierarchical_element):LOGICAL;
    LOCAL
      grand_parents : SET [0:?] OF hierarchical_element := [];
    END_LOCAL;
   IF (SIZEOF(parents) = 0) THEN
      RETURN(TRUE);
    END_IF;
    REPEAT i:= LOINDEX(members) TO HIINDEX(members);
      IF (members[i] IN parents) THEN
        RETURN(FALSE);
      ELSE
        REPEAT j:= LOINDEX(parents) TO HIINDEX(parents);
          grand_parents:= grand_parents + parents[j].parents;
        END_REPEAT;
        RETURN(element_dag(members,grand_parents));
      END_IF;
    END_REPEAT;
  END_FUNCTION;
(*
```

5.1.54 element_rollup

EXPRESS specification:

This function examines the set of hierarchical elements in a model and ensures that, when considering each of the "parts" (i.e., children) of a hierarchical element, these children are represented in the

top-level set. Thus, this function ensures that the set of hierarchical elements listed with a model is a "rollup" of all of the hierarchical elements in the model.

```
*)
  FUNCTION element_rollup
   (mdl:diagnostic_model;elem:SET [1:?] OF hierarchical_element):BOOLEAN;
   LOCAL
        check : BOOLEAN;
   END_LOCAL;

   REPEAT i := LOINDEX(elem) TO HIINDEX(elem);
       check : = FALSE;
       REPEAT j:= LOINDEX(elem[i].part_of_model) TO
                    HIINDEX(elem[i].part_of_model);
                 IF (elem[i].part_of_model[j] = mdl) THEN
                    check : = TRUE;
                 END_IF;
       END_REPEAT;
       IF (check = FALSE) THEN
         RETURN(FALSE);
       END_IF;
   END_REPEAT;
   RETURN(TRUE);
  END_FUNCTION;
(*
```

5.1.55 levels_acyclic

EXPRESS specification:

This function ensures that, for a particular entity occurring at a level, the chain of levels does not cycle back on itself.

```
*)
  FUNCTION levels_acyclic
    (lvl:level):BOOLEAN;
    LOCAL
      tst: BOOLEAN: = TRUE;
      target: level: = lvl;
    END_LOCAL;

    REPEAT WHILE (EXISTS(lvl.successor));
      IF (target = lvl.successor) THEN
        tst: = FALSE;
      END_IF;
      lvl: = lvl.successor;
    END_REPEAT;
    RETURN(tst);
  END_FUNCTION;
END_SCHEMA;
(*
```

5.1.56 Common Element Model EXPRESS-G diagrams

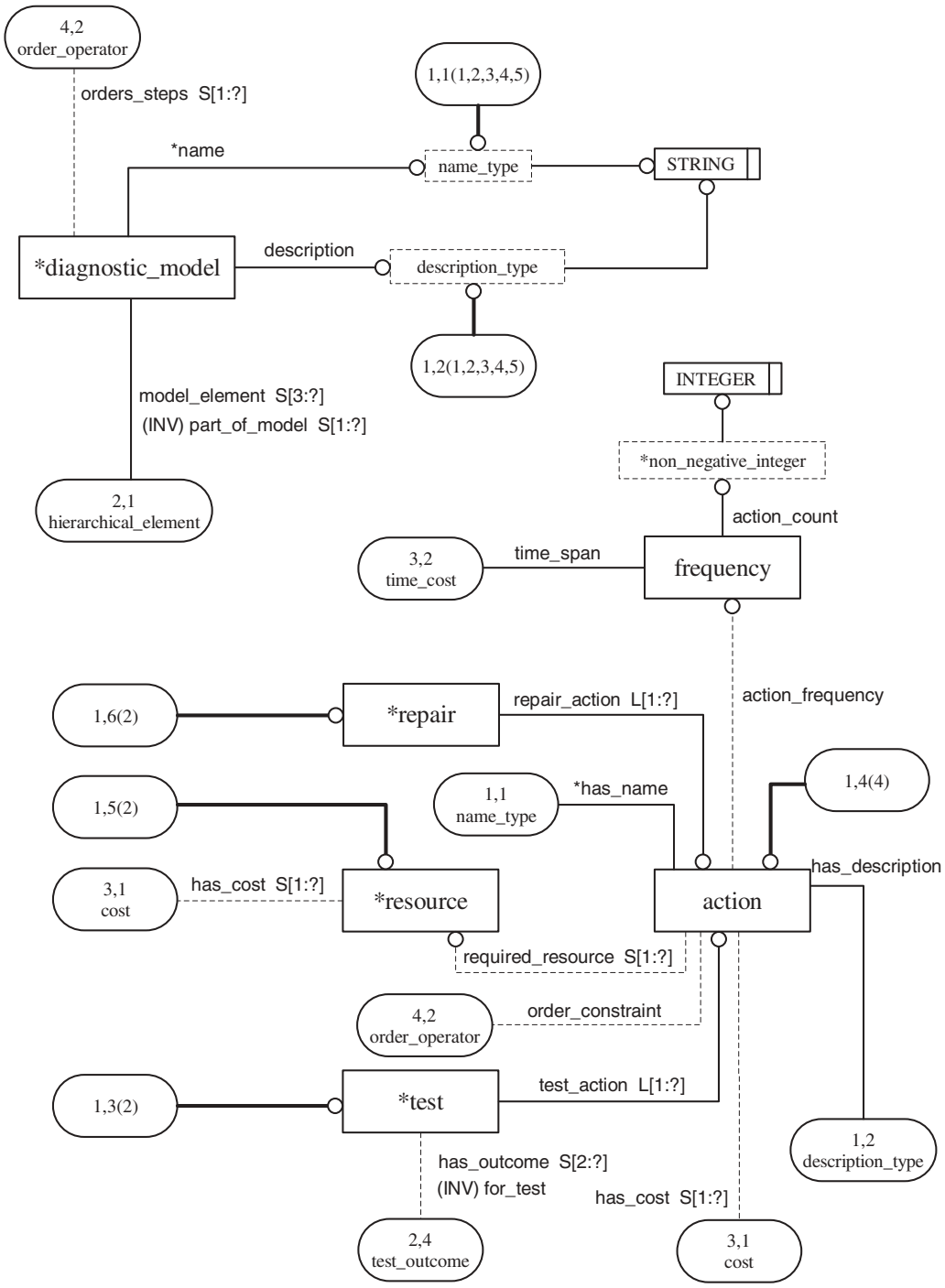


Figure 7—Common Element Model EXPRESS-G: Diagram 1 of 5

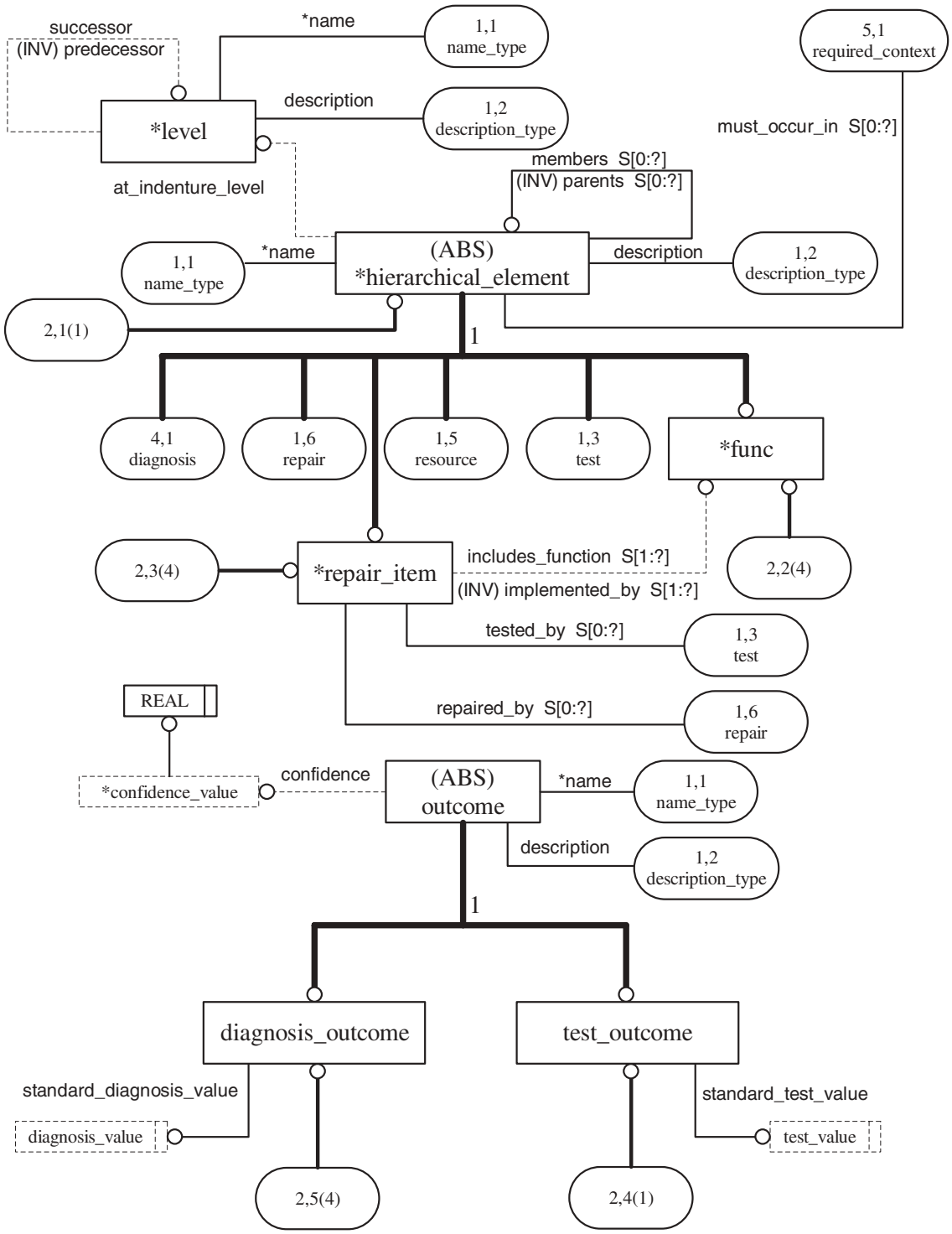


Figure 8—Common Element Model EXPRESS-G: Diagram 2 of 5

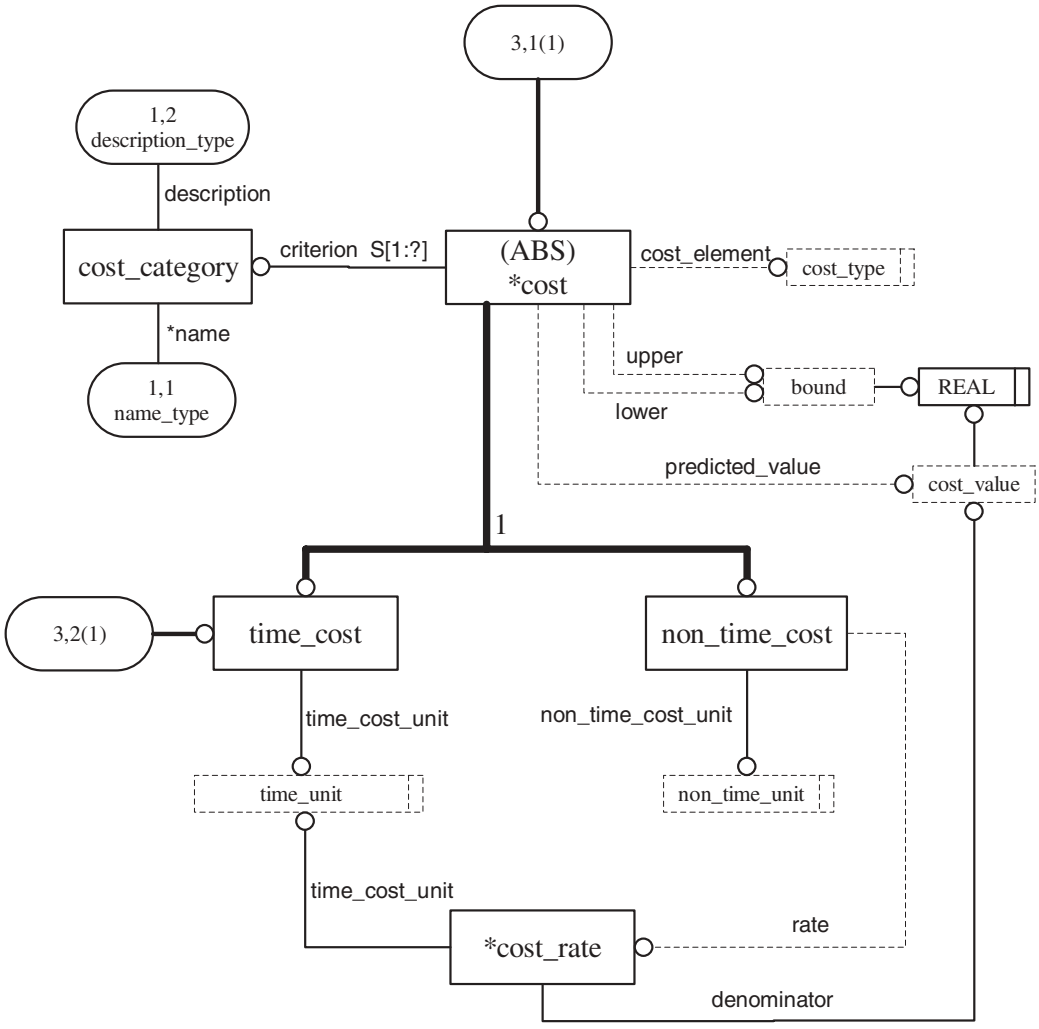


Figure 9—Common Element Model EXPRESS-G: Diagram 3 of 5

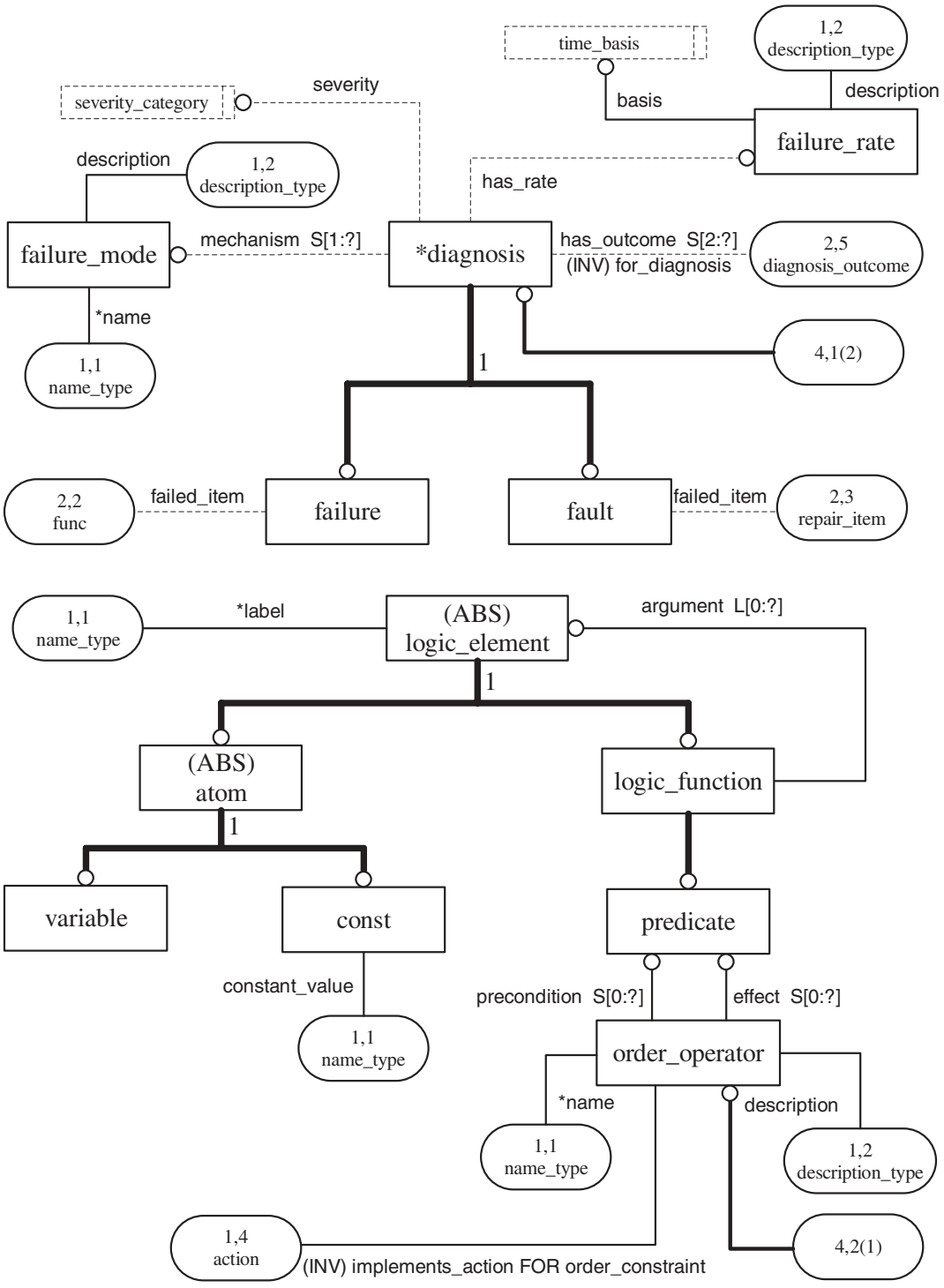


Figure 10—Common Element Model EXPRESS-G: Diagram 4 of 5

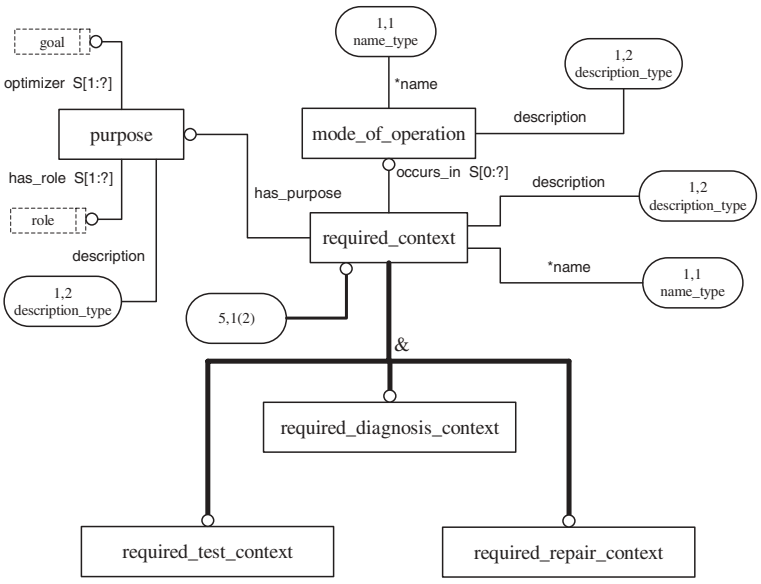


Figure 11—Common Element Model EXPRESS-G: Diagram 5 of 5

*)
(*

5.2 Diagnostic Inference Model

The AI-ESTATE Diagnostic Inference Model (DIM) schema is defined below. The constructs of this model were originally derived from the approach to system test and diagnosis known as *information flow modeling* or *dependency modeling*. The model utilizes many of the constructs defined in the AI-ESTATE Common Element Model.

An inference is a logical relationship between two tests or between a test and a diagnosis. Given a particular test outcome, inferences about other tests and/or diagnoses of the system can be made. Tests offer a view of the associated fault, function, or diagnosis.

In the AI-ESTATE DIM, test outcomes are limited to pass or fail outcomes. Inferences are identified between a particular test outcome and other test outcomes and asserted conditions of diagnostic units.

A particular test outcome that is inferred from another outcome is represented as a `test_inference` within the DIM. Analogously, a `diagnostic_inference` within the DIM asserts a condition of "good" or "candidate" on a particular diagnostic unit. These two inference elements are the terms that compose the inference associated with a test outcome. Both `test_inference` and `diagnostic_inference` are SUBTYPES of inference.

The set of inferences associated with a particular test outcome are limited either to a list of conjuncted inferences or a list of disjuncted inferences (not both). This representation limits the scope of the model to facilitate mapping to existing model-based reasoning tools.

EXPRESS specification:

```
*)
  SCHEMA AI_ESTATE_DIAGNOSTIC_INFERENCE_MODEL;
    REFERENCE FROM AI_ESTATE_COMMON_ELEMENT_MODEL
      (diagnostic_model,
      test_outcome,
      confidence_value,
      diagnosis_outcome);
(*
```

5.2.1 diagnostic_inference

Entity diagnostic_inference represents an inference of a diagnosis as good or still a candidate from a particular test outcome (as identified by outcome_inference). Uncertainty associated with this inference can be specified in the confidence attribute that has been inherited from the inference entity.

EXPRESS specification:

```
*)
  ENTITY diagnostic_inference
    SUBTYPE OF(inference);
    diagnostic_assertion  : diagnosis_outcome;
  WHERE
    no_user_defined  : diagnostic_assertion.standard_diagnosis_value    <>
                       USER_DEFINED_DIAGNOSIS_OUTCOME;
  END_ENTITY;
(*
```

Attribute definitions:

diagnostic_assertion : The condition of the diagnosis to which the inference applies. The condition of the unit is asserted to be either good or still a candidate. Since this is a diagnosis outcome, additional values can be assigned as well.

Formal propositions:

no_user_defined : Requires that the diagnostic outcome be one of the basic standard values as defined in the Common Element Model.

5.2.2 diagnostic_inference_model

This entity represents the constituents of the DIM. This construct also identifies the diagnoses, tests, and, optionally, required resources for the system under test being modeled.

EXPRESS specification:

```
*)
  ENTITY diagnostic_inference_model
    SUBTYPE OF(diagnostic_model);
    inference                    : SET [2:?] OF outcome_inference;
  END_ENTITY;
(*
```

Attribute definitions:

inference : This attribute identifies the set of outcome_inference that comprises the model of the system under test. To be useful, a model shall consist of at least two inferences, corresponding to inferences from the minimum number of outcomes for the minimum number of tests in the model.

5.2.3 inference

This entity is a supertype of the entities diagnostic_inference and test_inference. Inference is either a test inference or a diagnosis inference, but not both.

EXPRESS specification:

```
*)
  ENTITY inference
    ABSTRACT SUPERTYPE OF (ONEOF(diagnostic_inference, test_inference));
    confidence                : OPTIONAL confidence_value;
  END_ENTITY;
(*
```

Attribute definitions:

confidence : An entity that identifies the statistical confidence in the inference association from the outcome in outcome_inference to the following:

(a) The particular outcome identified in this entity if this entity is a test_inference.
(b) The particular diagnosis identified in this entity, in the condition identified in diagnostic_assertion, if this entity is a diagnostic_inference.

5.2.4 outcome_inference

This construct pairs a particular outcome of a particular test with a set of inferences represented in a conjunct/disjunct form. Each inference entity of the set is a single inference of type test_inference or diagnostic_inference. Since inference information is specific to the Diagnostic Inference Model (DIM), it is necessary to identify the test outcomes from the Common Element Model (CEM) with which the inferences are associated. Hence, the constructs for representing inferences are found in the DIM, along with a pairing of these inference constructs with test outcomes from the CEM.

EXPRESS specification:

```
*)
  ENTITY outcome_inference;
    disjuncts                    : OPTIONAL SET OF inference;
    conjuncts                    : OPTIONAL SET OF inference;
    associated_test_outcome      : test_outcome;
  UNIQUE
    one_outcome  : associated_test_outcome;
  WHERE
    conjunct_or_disjunct   : NOT(EXISTS(conjuncts)) OR
                 NOT(EXISTS(disjuncts));
    no_user_defined:  associated_test_outcome.standard_test_value       <>
      USER_DEFINED_TEST_OUTCOME
        ;
  END_ENTITY
(*
```

Attribute definitions:

disjuncts : This attribute is required, but the set can be empty. If the set is empty, then no inference can be drawn from the outcome. The set is interpreted as an ORed set of inferences to be drawn.

conjuncts : This attribute is required, but the set can be empty. If the set is empty, then no inference can be drawn from the outcome. The set is interpreted as an ANDed set of inferences to be drawn.

associated_test_outcome : This attribute identifies a particular outcome of the value of the test_outcome.for_test attribute to which the value of the outcome_inference attribute applies, where the test_outcome.for_test attribute identifies a particular test.

Formal propositions:

conjunct_or_disjunct : Constrains the outcome inference list to be a set of conjuncts or a set of disjuncts, but not both. The inference list can also be empty.

no_user_defined : Requires that the test outcome be one of the basic standard values as defined in the Common Element Model.

5.2.5 test_inference

Entity test_inference represents an inference about a test made from a test outcome. In other words, a test outcome that references this entity can depend on the particular outcome, identified in the outcome_inference, of the test identified in outcome_inference.for_test. Uncertainty associated with this inference can be specified in the confidence attribute that has been inherited from the inference entity.

EXPRESS specification:

```
*)
  ENTITY test_inference
    SUBTYPE OF(inference);
    outcome_inference   : test_outcome;
  WHERE
    no_user_defined     : outcome_inference.standard_test_value        <>
                          USER_DEFINED_TEST_OUTCOME;
   END_ENTITY;
(*
```

Attribute definitions:

outcome_inference : This attribute identifies the test and associated outcome to be inferred. The test is identified by the for_test attribute of the test outcome.

Formal propositions:

no_user_defined : Requires that the test outcome be one of the basic standard values as defined in the Common Element Model.

```
*)
END_SCHEMA;
(*
```

5.2.6 Diagnostic Inference Model EXPRESS-G diagram

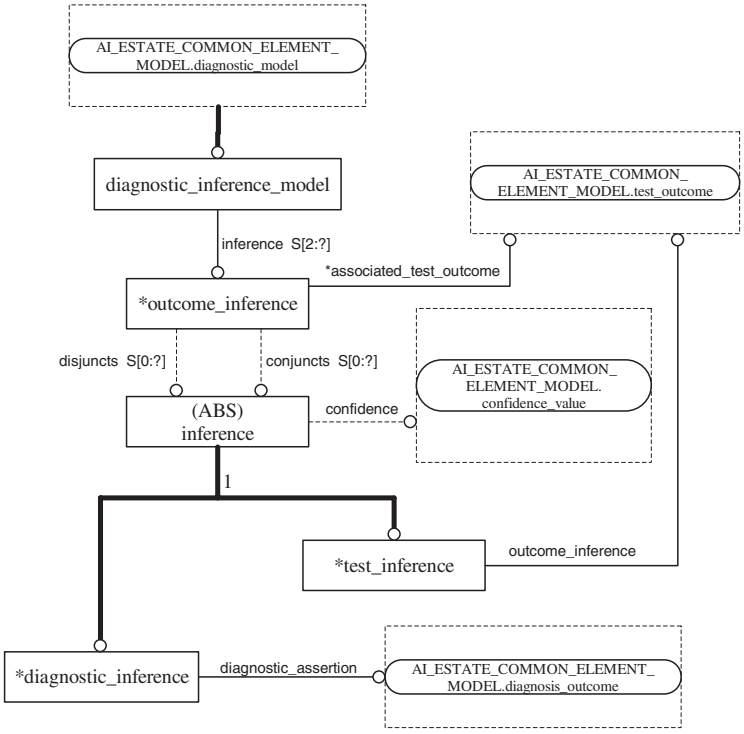


Figure 12—Diagnostic Inference Model EXPRESS-G: Diagram 1 of 1

*)
(*

5.3 Dynamic Context Model

The following information model captures the relevant dynamic information that may be used in any diagnostic context by an adequate abstraction of widely used diagnostic principles. The Dynamic Context Model (DCM) enables several important functions of an AI-ESTATE conformant reasoner. The functions relate to the state of the reasoning process at each step of a diagnostic session. The DCM data and knowledge are developed during a diagnostic session, unlike those of the Common Element Model (CEM), Fault Tree Model (FTM), Diagnostic Inference Model (DIM), and Enhanced Diagnostic Inference Model (EDIM) (which consist of static diagnostic data and knowledge).

A diagnostic session is initiated by identifying the model or models to be used for determining the existence of a fault in the unit undergoing test, and for isolating to a sufficient level to effect a maintenance action that will restore the system to a known functioning condition. The session is performed in a series of steps. At each step, one or more tests are performed. The DCM is used to record the state existing prior to performing any test at each of the following steps:

1) The model(s) currently in use
2) The status of all resources associated with the active models
3) The status of all tests associated with the active models
4) The status of all diagnoses associated with the active models
5) Optionally, the current fault hypothesis

The DCM is also used to record the tests performed at each step and any associated actual test outcome and outcome confidence values. The reasoner makes diagnostic and test inferences from the actual test outcome and associated confidence values.

EXPRESS specification:

```
*)
  SCHEMA AI_ESTATE_DYNAMIC_CONTEXT_MODEL;
    USE FROM AI_ESTATE_COMMON_ELEMENT_MODEL
      (name_type,
      diagnostic_model,
      diagnosis,
      test,
      confidence_value,
      test_outcome,
      resource,
      cost_value,
      time_cost,
      non_time_cost,
      diagnosis_outcome,
      logic_element,
      description_type,
      repair_item,
      action,
      cost_category,
      required_context,
      non_negative_integer);
(*
```

5.3.1 availability_type

Type availability_type defines a type that enables the availability of a test, model, or resource to be set. This is a logical type that indicates the associated entity is available if the value is set to TRUE. If the value is set to FALSE, then the entity is not available. An UNKNOWN indicates that the availability of the entity could not be determined.

EXPRESS specification:

```
*)
  TYPE availability_type = BOOLEAN;
  END_TYPE;
(*
```

5.3.2 calendar_year

Type calendar_year captures the calendar year in four-digit format.

EXPRESS specification:

```
*)
  TYPE calendar_year = STRING (4) FIXED;
  END_TYPE;
(*
```

5.3.3 day_stamp

Captures the day of the week.

EXPRESS specification:

```
*)
  TYPE day_stamp = non_negative_integer;
  END_TYPE;
(*
```

5.3.4 degree

This defines a type for a numeric value capturing the amount of degradation associated with the diagnosis. A higher value indicates an amount of degradation less than that of a lower value.

EXPRESS specification:

```
*)
  TYPE degree = REAL;
  WHERE
    standard_value  :  (0 <= SELF) AND (SELF <= 1);
  END_TYPE;
(*
```

Formal propositions:

standard_value : Constraint standard_value forces a degree to take on a value between zero (0) and one (1).

5.3.5 fraction_of_second

The fraction of a second to a precision of hundreths of a second.

EXPRESS specification:

```
*)
  TYPE fraction_of_second = non_negative_integer;
  WHERE
    hundreths: SELF <= 99;
  END_TYPE;
(*
```

Formal Propositions:

hundredths : Ensures that the fraction of a second is kept to a precision of hundredths of a second.

5.3.6 hour

Captures in coordinated universal time (UCT) the hour of the day.

EXPRESS specification:

```
*)
  TYPE hour = non_negative_integer;
  WHERE
    twenty_four_hour : SELF <= 23;
  END_TYPE;
(*
```

Formal propositions:

twenty_four_hour : Ensures that the legal hours for the day are between 0 and 23, in accordance with UCT.

5.3.7 minute

EXPRESS specification:

```
*)
  TYPE minute = non_negative_integer;
  WHERE
    sixty_minute_hour : SELF <= 59;
  END_TYPE;
(*
```

Formal propositions:

sixty_minute_hour : Ensures that the minute stamp is between 0 and 59.

5.3.8 month

Identifies the month.

EXPRESS specification:

```
*)
  TYPE month = legal_month;
  END_TYPE;
(*
```

5.3.9 optimal_failure_rate

Type optimal_failure_rate defines a type by which a step can determine whether or not the optimization process is dependent on failure rates.

EXPRESS specification:

```
*)
  TYPE optimal_failure_rate = BOOLEAN;
  END_TYPE;
(*
```

5.3.10 second

EXPRESS specification:

```
*)
  TYPE second = non_negative_integer;
  WHERE
    sixty_second_minute : SELF <= 59;
  END_TYPE;
(*
```

Formal propositions:

sixty_second_minute : Ensures that the number of seconds lies between 0 and 59.

5.3.11 hypothesis_direction

Enumerated type hypothesis_direction differentiates between the reasoner and the user in terms of relationship to a hypothesis.

EXPRESS specification:

```
*)
  TYPE hypothesis_direction = ENUMERATION OF
    (REASONER_DIRECTED,
    USER_DIRECTED);
  END_TYPE;
(*
```

5.3.12 legal_month

Identifies the enumerated set of legal months.

EXPRESS specification:

```
*)
  TYPE legal_month = ENUMERATION OF
    (JANUARY,
    FEBRUARY,
    MARCH,
    APRIL,
    MAY,
    JUNE,
    JULY,
    AUGUST,
    SEPTEMBER,
    OCTOBER,
    NOVEMBER,
    DECEMBER);
  END_TYPE;
(*
```

5.3.13 status_code

Enumerated type status_code identifies legal status codes to be returned by a service specified by this standard.

EXPRESS specification:

```
*)
  TYPE status_code = ENUMERATION OF
    (OPERATION_COMPLETED_SUCCESSFULLY,
    NONEXISTENT_DATA_ELEMENT_REQUESTED,
    MISSING_OR_INVALID_ARGUMENT,
    OPERATION_OUT_OF_SEQUENCE,
    INVALID_MODEL_SCHEMA,
    SERVICE_NOT_AVAILABLE,
    UNKNOWN_EXCEPTION_RAISED);
  END_TYPE;
(*
```

5.3.14 assigned_element

Type assigned_element is a SELECT type that selects between active_test and active_diagnosis.

EXPRESS specification:

```
*)
  TYPE assigned_element = SELECT
    (active_test,
    active_diagnosis);
  END_TYPE;
(*
```

5.3.15 cause

Select type cause captures information on root cause events that spawn a particular diagnostic session. As a select type, it results in selection of one of the associated entities for the actual cause (or causes). It is expected that the cause shall be used when performing postmortem analysis for the purpose of generating and using diagnostic histories or for computing field-based diagnosability metrics.

EXPRESS specification:

```
*)
  TYPE cause = SELECT
    (trigger,
    alarm);
  END_TYPE;
(*
```

5.3.16 active_action

Entity active_action corresponds to actions being taken in the test and diagnosis of a system. Active actions have actual costs that are tied back to the associated tests and resources.

EXPRESS specification:

```
*)
  ENTITY active_action;
    corresponds_to                : action;
    time_incurred                 : SET OF active_time_cost;
    cost_incurred                 : SET OF active_cost;
    resources_used                : OPTIONAL SET OF active_resource;
  WHERE
    resources_valid               : resources_available(resources_used);
    active_resource_is_resource   : set_of_resource(resources_used) <=
                                    corresponds_to.required_resource;
  END_ENTITY;
(*
```

Attribute definitions:

corresponds_to : Attribute corresponds_to identifies the action that is being instantiated by this entity.

time_incurred : Attribute time_incurred records the actual time associated with taking this action in the current step of the session trace.

cost_incurred : Attribute cost_incurred records the actual costs associated with taking this action in the current step of the session trace.

resources_used : Attribute resources_used lists the resources used in performing this action. Since the cardinality of the set begins at zero, this set can be empty. It is also an optional attribute indicating whether the reasoner cares if resources are used or not.

Formal propositions:

resources_valid : Constraint resources_valid uses function resources_available. This constraint prevents unavailable resources being listed as having been used.

active_resource_is_resource : Constraint active_resource_is_resource ensures that the set of resources_used is a subset of the required_resource attribute set of this entity's corresponds_to action attribute.

5.3.17 active_cost

Entity active_cost references a non_time_cost entity being used by some test or resource in the current diagnostic session. Several attributes are obtained by examining the referenced entity (e.g., units) and that entity's abstract supertype cost.

EXPRESS specification:

```
*)
  ENTITY active_cost;
    actual_value     : cost_value;
    corresponds_to   : non_time_cost;
```

```
  INVERSE
    action_cost  : SET OF active_action FOR cost_incurred;
  WHERE
    value_is_valid: (NOT(EXISTS(corresponds_to.lower))                AND
                     NOT(EXISTS(corresponds_to.upper)))               XOR
                    ((corresponds_to.lower <= actual_value)           AND
                    (actual_value <= corresponds_to.upper));
  END_ENTITY;
(*
```

Attribute definitions:

actual_value	: Attribute actual_value is used to store the actual cost incurred rather than the expected cost.
corresponds_to	: Attribute corresponds_to identifies the non_time_cost that is being "instantiated" by this entity.
action_cost	: Inverse attribute action_cost indicates the specific active_action to which this active_cost belongs.

Formal propositions:

value_is_valid	: Constraint value_is_valid ensures that actual_value lies between legal bounds, given that bounds have been defined.

5.3.18 active_diagnosis

Entity active_diagnosis corresponds to a diagnosis used by the reasoner. It is assumed that the outcome of the diagnosis can only take on a single value representing the current value. Since the session trace instantiates the diagnoses at each step, multiple values for the same diagnosis can be determined by traversing the trace.

EXPRESS specification:

```
*)
  ENTITY active_diagnosis;
    corresponds_to             : diagnosis;
    actual_outcome             : OPTIONAL diagnosis_outcome;
    actual_confidence          : OPTIONAL confidence_value;
  WHERE
    confidence_with_outcome    : (NOT(EXISTS(actual_confidence))      AND
                                  NOT(EXISTS(actual_outcome)))         OR
                                  (EXISTS(actual_confidence)           AND
                                  EXISTS(actual_outcome));
  END_ENTITY;
(*
```

Attribute definitions:

corresponds_to	: Attribute corresponds_to identifies the diagnosis in the diagnostic model that is actively evaluated or inferred in the session.
actual_outcome	: Attribute actual_outcome provides the current actual outcome associated with this diagnosis. This attribute is optional since the outcome may be tied to a diagnosis group.

actual_confidence : Attribute actual_confidence provides the current actual confidence value associated with the actual_outcome of the diagnosis. This attribute is optional since it is tied to the outcome, which is also optional.

Formal propositions:

confidence_with_outcome : Constraint confidence_with_outcome ensures that either an actual_outcome is paired with an actual_confidence or the active_diagnosis has neither an active_outcome nor an active_confidence. In other words, both active_outcome and active_confidence must be present simultaneously, if they exist at all.

5.3.19 active_model

Entity active_model references one of the models used in the current knowledge base. The name of the entity is obtained by examining the referenced model.

EXPRESS specification:

```
*)
  ENTITY active_model;
    availability         : availability_type;
    corresponds_to       : diagnostic_model;
    pathname             : STRING;
  END_ENTITY;
(*
```

Attribute definitions:

availability : Attribute availability is a logical attribute indicating whether this model is available for the current session.

corresponds_to : Attribute corresponds_to identifies the diagnostic model that is actively loaded and processed in the session.

pathname : Attribute pathname is a string providing the complete path to the file containing the model.

5.3.20 active_resource

Entity active_resource corresponds to a resource used as part of a test. Throughout the model, sets of active_resources are nonoptional but have initial cardinality of zero.

EXPRESS specification:

```
*)
  ENTITY active_resource;
    availability           : availability_type;
    corresponds_to         : resource;
    resource_action        : LIST [1:?] OF active_action;
```

```
  INVERSE
    associated_step       : step FOR resource_status;
  END_ENTITY;
(*
```

Attribute definitions:

availability : Attribute availability is a logical attribute indicating whether this resource is available for the current step. In this case, entities need to be instantiated in each step of the session trace.

corresponds_to : Attribute corresponds_to identifies the resource in the diagnostic model that is actively used in the session.

resource_action : Attribute resource_action identifies the action or actions required to use the given resource. Cost information from the resource can be derived by examining the costs of the associated actions.

associated_step : Inverse attribute associated_step provides a pointer back to the step referring to this active_resource entity.

5.3.21 active_test

Entity active_test corresponds to a test used by the reasoner at a given step. The outcome of a test can only take on a single value representing the current value. Since the session trace instantiates the tests at each step, multiple values for the same test (e.g., if a test is repeated) can be determined by traversing the trace.

EXPRESS specification:

```
*)
  ENTITY active_test;
    actual_confidence    : OPTIONAL confidence_value;
    actual_outcome       : OPTIONAL test_outcome;
    corresponds_to       : test;
    test_action          : LIST [1:?] OF active_action;
  WHERE
    confidence_with_outcome:   (NOT(EXISTS(actual_confidence))          AND
                                NOT(EXISTS(actual_outcome)))            OR
                               (EXISTS(actual_confidence)               AND
                               EXISTS(actual_outcome));
  END_ENTITY;
(*
```

Attribute definitions:

actual_confidence : Attribute actual_confidence provides the current actual confidence value associated with the actual_outcome of the test. This attribute is optional since it is tied to the outcome, which is also optional.

actual_outcome : Attribute actual_outcome provides the current actual outcome associated with this test. This attribute is optional since the outcome could be tied to a test group.

corresponds_to : Attribute corresponds_to identifies the test in the diagnostic model that is actively executed or inferred in the session.

test_action : Attribute test_action identifies the action or actions required to perform the given test. Cost information from the test can be derived by examining the costs of the associated actions.

Formal propositions:

confidence_with_outcome : Constraint confidence_with_outcome ensures that either an actual_outcome is paired with an actual_confidence or the active_test has neither an active_outcome nor an active_confidence. In other words, both active_outcome and active_confidence must be present simultaneously, if they exist at all.

5.3.22 active_time_cost

Entity active_time_cost references a time_cost entity being used by some test or resource in the current diagnostic session. Several attributes are obtained by examining the referenced entity (e.g., units) and that entity's abstract supertype cost.

EXPRESS specification:

```
*)
  ENTITY active_time_cost;
    actual_value        : cost_value;
    corresponds_to      : time_cost;
  INVERSE
    action_time         : SET OF active_action FOR time_incurred;
  WHERE
    value_is_valid      : (NOT(EXISTS(corresponds_to.lower))          AND
                           NOT(EXISTS(corresponds_to.upper)))         XOR
                           ((corresponds_to.lower <= actual_value)    AND
                           (actual_value  <=  corresponds_to.upper));
  END_ENTITY;
(*
```

Attribute definitions:

actual_value : Attribute actual_value is used to store the actual cost incurred rather than the expected cost.

corresponds_to : Attribute corresponds_to identifies the time_cost that is being instantiated by this entity.

action_time : Inverse attribute action_time indicates the specific active_action to which this active_time_cost belongs.

Formal Propositions:

value_is_valid : Constraint value_is_valid ensures that actual_value lies between legal bounds, given that bounds have been defined.

5.3.23 alarm

Entity `alarm` captures information on automatic notifications of a problem within a system that results in the initiation of a diagnostic session. Alarms could, e.g., result from visual or audible alarms presented to a user, built-in test (BIT) indications, or network management traps [as from simple network management protocol (SNMP)].

EXPRESS specification:

```
*)
  ENTITY alarm
    SUBTYPE OF(active_test);
  END_ENTITY;
(*
```

5.3.24 diagnosis_context

Entity `diagnosis_context` is a subtype of `step_context` and specifies the diagnosis context for a particular diagnosis (or group of diagnoses) within a model.

EXPRESS specification:

```
*)
  ENTITY diagnosis_context
    SUBTYPE OF(step_context);
    time_to_test              : OPTIONAL active_time_cost;
   END_ENTITY;
(*
```

Attribute definitions:

`time_to_test` : Attribute `time_to_test` specifies the amount of time available within a diagnostic session to complete testing and return a diagnosis. This is an optional attribute since time constraints may not be specified in all cases.

5.3.25 exception

Entity `exception` captures information on status codes and exceptions that could be raised with a particular step in the diagnostic session.

EXPRESS specification:

```
*)
  ENTITY exception;
    status              : status_code;
    description         : description_type;
    associated_service : description_type;
  END_ENTITY;
(*
```

Attribute definitions:

status : Attribute status identifies the status code associated with a given exception raised at the current step.

description: : Attribute description provides a textual description for the exception raised and its associated step in the session.

associated_service: : Attribute associated_service identifies the service called that resulted in the raised exception.

5.3.26 gripe

Entity gripe captures information on user complaints about the performance of the system. Typically, such complaints spawn some sort of diagnostic session to isolate the cause of the gripe.

EXPRESS specification:

```
*)
  ENTITY gripe
    SUBTYPE OF(trigger);
  END_ENTITY;
(*
```

5.3.27 history

Entity history captures maintenance history information on the unit or system under test. This history is in terms of previous test/diagnosis/repair sessions.

EXPRESS specification:

```
*)
  ENTITY history;
    past_session : LIST OF session;
  END_ENTITY;
(*
```

Attribute definitions:

past_session : Attribute past_session identifies a list of diagnostic sessions capturing historical maintenance information. A list is used to indicate a chronological sequence of events.

5.3.28 inferred_diagnosis

Entity inferred_diagnosis corresponds to a diagnosis that can be inferred by the reasoner. The diagnosis can only take on a single value representing the current value. Since the session trace instantiates the diagnoses at each step, the set of diagnoses is maintained at each step. Therefore, value changes for a diagnosis can be determined by traversing the trace.

EXPRESS specification:

```
*)
  ENTITY inferred_diagnosis;
    corresponds_to          : SET [1:?] OF diagnosis;
    inferred_confidence     : confidence_value;
    inferred_outcome        : diagnosis_outcome;
    grade                   : OPTIONAL degree;
    availability            : availability_type;
  INVERSE
    associated_step         : step FOR diagnoses;
  WHERE
    consistent_outcome      :check_diagnosis_out(corresponds_to,
                             inferred_outcome);
  END_ENTITY;
(*
```

Attribute definitions:

corresponds_to : Attribute corresponds_to identifies the set of diagnoses in the diagnostic model that is available to be inferred in the session. If the cardinality of the set is greater than one, then it is expected that this will be treated as a multiple fault.

inferred_confidence : Attribute inferred_confidence indicates the current confidence in the value associated with this diagnosis.

inferred_outcome : Attribute inferred_outcome indicates the current value for the diagnosis. The type of this attribute is referenced from the EDIM. Further, if diagnostic outcomes are provided and the cardinality of the set is greater than one, then the outcomes correspond with the associated members of the diagnosis set.

grade : Attribute grade identifies the level of degradation for the given diagnosis, inferred from the test information received so far. This attribute is optional since current AI-ESTATE models provide no direct basis for performing the needed inference.

availability : Attribute availability indicates whether or not this diagnosis is available for the current step.

associated_step : Inverse attribute associated_step provides a pointer back to the step referring to this inferred_diagnosis entity.

Formal propositions:

consistent_outcome : Constraint consistent_outcome ensures that the inferred outcomes exist in the set of legal outcomes associated with the diagnoses.

5.3.29 inferred_test

Entity inferred_test corresponds to a test whose outcome is inferred by the reasoner. The outcome of a test can only take on a single value representing the current value. Since the session trace instantiates the tests at each step, multiple values for the same test (e.g., if a test is repeated) can be determined by traversing the trace.

EXPRESS specification:

```
*)
  ENTITY inferred_test;
    corresponds_to       : SET [1:?] OF test;
    availability         : availability_type;
    inferred_confidence  : confidence_value;
    inferred_outcome     : test_outcome;
  INVERSE
    associated_step      : step FOR test_status;
  WHERE
    consistent_outcome   : check_test_out(corresponds_to,
                           inferred_outcome);
  END_ENTITY;
(*
```

Attribute definitions:

corresponds_to : Attribute corresponds_to identifies the set of tests in the diagnostic model that is available to be inferred in the session. If the cardinality of the set is greater than one, then it is expected that this will be treated as a conjunction of tests.

availability : Attribute availability indicates whether this test is available for the current step.

inferred_confidence : Attribute inferred_confidence provides the current confidence value associated with the inferred_outcome of the test. It is required that this attribute be the same size as the corresponds_to attribute, and that there exists a one-to-one correspondence between confidences and tests.

inferred_outcome : Attribute inferred_outcome provides the current outcome associated with this test. It is required that this attribute be the same size as the corresponds_to attribute, and that there exists a one-to-one correspondence between outcomes and tests.

associated_step : Inverse attribute associated_step provides a pointer back to the step referring to this inferred_test entity.

Formal propositions:

consistent_outcome : Constraint consistent_outcome ensures that the inferred outcomes exist in the set of legal outcomes associated with the tests.

5.3.30 logic_term

Entity logic_term captures the current logical state for purposes of ordering constraints. Since not all diagnostic reasoners are capable of doing partial-order planning, this information is optional within the Dynamic Context Model.

EXPRESS specification:

```
*)
  ENTITY logic_term;
    corresponds_to    : logic_element;
  END_ENTITY;
(*
```

Attribute definitions:

corresponds_to : Attribute corresponds_to identifies the specific logical element within the Common Element Model to which this term corresponds.

5.3.31 notice

Entity notice captures details on manufacturer or maintainer notices to a user that some sort of maintenance or diagnostics needs to be done on the system.

EXPRESS specification:

```
*)
  ENTITY notice
    SUBTYPE OF(trigger);
  END_ENTITY;
(*
```

5.3.32 repair_context

Entity repair_context is a subtype of step_context and specifies the repair context for a particular repair item (or group of repair items) within a model.

EXPRESS specification:

```
*)
  ENTITY repair_context
    SUBTYPE OF(step_context);
    available_repair_item     : SET OF repair_item;
   END_ENTITY;
(*
```

Attribute definitions:

available_repair_item : Attribute available_repair_items specifies those repair items that can be repaired in the current context.

5.3.33 session

Entity session provides a session trace for the current diagnostic session.

EXPRESS specification:

```
*)
  ENTITY session;
    trace              : LIST OF step;
    caused_by          : OPTIONAL SET [1:?] OF cause;
```

```
    updates_history    : OPTIONAL history;
  WHERE
    no_tests_last      : NOT(EXISTS(
                          trace[SIZEOF(trace)].outcomes_collected));
  END_ENTITY;
(*
```

Attribute definitions:

trace	: Attribute trace provides an ordered list of states (represented as steps) through which the diagnostic reasoner traverses during a session. It is expected that actions such as backing up would be recorded as a new step in the trace rather than having a step deleted from the trace.
caused_by	: Attribute caused_by identifies the event or events that caused the diagnostic session to occur.
updates_history	: Attribute updates_history identifies the specific history for the unit or system diagnosed by this session.

Formal propositions:

no_tests_last	: Ensures that the last step in the traces has no associated outcomes or tests that have just been collected. To have collected this information indicates that the state must be updated, and a new step must be created.

5.3.34 step

Entity step instantiates a step in the session trace and includes all information related to a specific step in the test process, including the current state of the reasoner. This state information reflects the status of all the attributes at the beginning of the step (creation of step).

EXPRESS specification:

```
*)
  ENTITY step;
    model_status                : SET [1:?] OF active_model;
    reasoner_hypothesis         : OPTIONAL SET OF inferred_diagnosis;
    test_status                 : SET [1:?] OF inferred_test;
    diagnoses                   : SET [2:?] OF inferred_diagnosis;
    resource_status             : SET OF active_resource;
    logic_state                 : OPTIONAL SET OF logic_term;
    optimized_by_failure_rate   : optimal_failure_rate;
    optimized_by_cost           : SET OF cost_category;
    user_hypothesis             : OPTIONAL SET [1:?] OF diagnosis;
    reason_by                   : hypothesis_direction;
    occurs_within               : step_context;
    service_result              : OPTIONAL LIST [1:?] OF exception;
    time_occurred               : OPTIONAL time_stamp;
    outcomes_collected          : OPTIONAL SET [1:?] OF assigned_element;
  INVERSE
    part_of                     : session FOR trace;
```

```
  WHERE
    test_active: NOT(EXISTS(outcomes_collected))                    XOR
      active_tests_available(outcomes_collected,test_status);
    resource_active: (SIZEOF(QUERY(temp <* resource_status           |
      NOT(temp.availability = FALSE)))=0)                           AND
      (set_of_resource(resource_status)                             <=
      resource_set_union(model_status));
    diagnosis_in_state: (reasoner_hypothesis <= diagnoses);
    all_tests: (set_of_test(test_status)                             =
      test_set_union(model_status));
    all_diagnoses: (set_of_diagnosis(diagnoses)                      =
      diagnosis_set_union(model_status));
    diagnosis_active: NOT(EXISTS(outcomes_collected))               XOR
      active_diagnoses_available(outcomes_collected,diagnoses);
  END_ENTITY;
(*
```

Attribute definitions:

model_status	: Attribute model_status lists all of the models associated with the current session and indicates their status at the current step.
reasoner_hypothesis	: Attribute hypothesis is an optional attribute consisting of the set of inferred_diagnosis making up the current hypothesis. This attribute is optional since some reasoners might not generate a hypothesis in mid-process; however, it is expected that the hypothesis will be determined before the active tests are performed and will be used to report the final diagnosis.
test_status	: Attribute test_status lists all of the tests used by the available models and is used to capture their status prior to running the active tests. For the first step in the session, the test_status corresponds to the set of symptoms used to start the diagnostic process.
diagnoses	: Attribute diagnoses lists all of the diagnoses used by the available models and is used to capture their status prior to running the active tests.
resource_status	: Attribute resource_status lists all of the resources used by the available models and is used to capture their status prior to running the active tests.
logic_state	: Attribute logic_state provides the current list of facts within the logic state of the diagnostic session. This state is compared to a set of preconditions in the Common Element Model to assist in determining appropriate ordering of actions.
optimized_by_failure_rate	: Attribute optimized_by_failure_rate specifies that test selection is dependent on failure rate when this Boolean attribute is TRUE.
optimized_by_cost	: Attribute optimized_by_cost identifies the set of cost criteria (if any) used to optimize test selection at a given step in the diagnostic process.
user_hypothesis	: Attribute user_hypothesis identifies a set (possibly empty) of diagnoses in the diagnostic model to be used as a hypothesis. This hypothesis would be used to guide test

selection to verify/deny that hypothesis. Note this is an optional attribute since a user might not have a hypothesis at all steps in the process.

`reason_by` : Attribute `reason_by` identifies whether or not reasoning is to be centered at a given step on a user-based hypothesis or a reasoner-based hypothesis.

`occurs_within` : Attribute `occurs_within` identifies the specific context for a given step during a diagnostic session. An initial context shall be used to start the diagnostic process for a particular session. Context can change as diagnosis proceeds, and this change in context is captured via the remainder of the dynamic context model. Once a context is set for a given step, that context will persist through subsequent steps in a session until changed.

`service_result` : Attribute `service_result` indicates the list of exceptions (if any) raised as a result of executing a service.

`time_occurred` : Attribute `time_occurred` records a time step at which the step began.

`outcomes_collected` : Attribute `outcomes_collected` indicates the set of active tests and diagnoses evaluated at this step in the process.

`part_of` : Inverse attribute `part_of` indicates the specific session trace to which this step belongs.

Formal propositions:

`test_active` : Constraint `test_active` ensures that the active tests recorded in the step are members of the set of all tests and are available to be performed.

`resource_active` : Constraint `resource_active` evaluates two predicates to ensure that the elements of the `resource_status` set are valid. The first predicate verifies all of the resources are available, and the second predicate verifies all of the resources are defined by the available models.

`diagnosis_in_state` : Constraint `diagnosis_in_state` ensures that the diagnoses in the hypothesis have been specified by the set of diagnoses given by the current state.

`all_tests` : Constraint `all_tests` uses function `test_set_union`. This constraint verifies that the union of the test sets from the available models equals the set listed in `test_status`.

`all_diagnoses` : Constraint `all_diagnoses` uses function `diagnosis_set_union`. This constraint verifies that the union of the diagnosis sets from the available models equals the set listed in diagnoses.

`diagnosis_active` : Constraint `diagnosis_active` ensures that the active diagnoses recorded in the step are members of the set of all diagnoses and are available to be evaluated.

5.3.35 step_context

Entity `step_context` specifies the current context for a diagnostic problem. Diagnostic models are generally created with a particular context (or contexts) in mind. This entity is a supertype of `test_context` AND `diagnosis_context` AND `repair_context`.

EXPRESS specification:

```
*)
  ENTITY step_context
    SUPERTYPE OF (test_context AND diagnosis_context AND repair_context);
    available_resources : SET OF resource;
    description         : description_type;
    name                : name_type;
    corresponds_to      : required_context;
  UNIQUE
    one_context : name;
  END_ENTITY;
(*
```

Attribute definitions:

`available_resources` : Attribute `available_resources` identifies the test and repair resources available for use during the diagnostic session.

`description` : Attribute `description` provides a textual description of contextual information related to the current step.

`name` : Attribute `name` provides a unique identifier to provide a means to reason over contextual information.

`corresponds_to` : Attribute `corresponds_to` identifies the required context against which an actual context is compared to determine if the context is consistent.

5.3.36 test_context

Entity `test_context` is a subtype of `step_context` and specifies the test context for a particular test (or group of tests) within a model.

EXPRESS specification:

```
*)
  ENTITY test_context
    SUBTYPE OF(step_context);
    available_tests          : SET OF active_test;
  END_ENTITY;
(*
```

Attribute definitions:

`available_tests` : Attribute `available_tests` identifies the tests that are available for use in diagnosis at the beginning of the diagnostic session.

5.3.37 time_stamp

Entity `time_stamp` provides a means of associating a time stamp to elements within the DCM. The time basis for capturing the time element within the time stamp shall be coordinated universal time (UCT).

EXPRESS specification:

```
*)
  ENTITY time_stamp;
    year_stamp         : calendar_year;
    month_stamp        : month;
    day_stamp          : day_stamp;
    hour_stamp         : hour;
    minute_stamp       : minute;
    second_stamp       : second;
    fraction_stamp     : fraction_of_second;
  END_ENTITY;
(*
```

Attribute definitions:

year_stamp	: Attribute year_stamp captures the four-digit year for the time stamp.
month_stamp	: Attribute month_stamp captures the month for the time stamp.
day_stamp	: Attribute day_stamp identifies the particular date within a month for the time stamp.
hour_stamp	: Attribute hour_stamp captures the hour of the day in UCT for the time stamp.
minute_stamp	: Attribute minute_stamp captures the minute of the hour in UCT according to UCT.
second_stamp	: Attribute second_stamp captures the number of seconds in accordance with UCT.
fraction_stamp	: Attribute fraction_stamp captures the fraction of a second to the hundreths of a second in accordance with UCT.

5.3.38 trigger

Entity trigger defines a type of event corresponding to a report of some kind that results in a diagnostic session being initiated. Typically, triggers correspond to human-generated or organization-generated events such as manufacturing notices/recalls or user "gripes." As such, a trigger is defined to be an abstract supertype of these two types of entities.

EXPRESS specification:

```
*)
  ENTITY trigger
    ABSTRACT SUPERTYPE OF (ONEOF(notice, gripe));
    description                : description_type;
    name                       : name_type;
  UNIQUE
    one_name       : name;
  END_ENTITY;
(*
```

Attribute definitions:

description	: Attribute description provides a means for associating descriptive text to characterize the trigger.
name	: Attribute name provides a unique name for identifying the trigger.

5.3.39 active_diagnoses_available

EXPRESS specification:

Function active_diagnoses_available determines whether each active diagnosis is a member of all diagnoses and whether the active_diagnoses are available by checking the corresponding inferred_diagnosis availability attributes.

```
*)
  FUNCTION active_diagnoses_available
    (active: SET [1:?] OF active_diagnosis;
    inferred: SET [1:?] OF inferred_diagnosis): LOGICAL;

    LOCAL
      member: LOGICAL;
      result: LOGICAL:= TRUE;
    END_LOCAL;

    REPEAT i:= LOINDEX(active) TO HIINDEX(active);
      member:= FALSE;
      REPEAT j: = LOINDEX(inferred) TO HIINDEX(inferred);
        IF (active[i].corresponds_to IN
          inferred[j].corresponds_to) THEN
          member:= TRUE;
          IF (inferred[j].availability = FALSE) THEN
            result:= FALSE;
          END_IF;
        END_IF;
      END_REPEAT;
      IF (member = FALSE) THEN
        result:= FALSE;
      END_IF;
    END_REPEAT;
    RETURN (result);
  END_FUNCTION;
(*
```

5.3.40 active_tests_available

EXPRESS specification:

Function active_tests_available determines whether each active test is a member of all tests and whether the active_tests are available by checking the corresponding inferred_test availability attributes.

```
*)
  FUNCTION active_tests_available
    (active: SET [1:?] OF active_test;
    inferred: SET [1:?] OF inferred_test) : LOGICAL;

    LOCAL
      member: LOGICAL;
      result: LOGICAL:= TRUE;
    END_LOCAL;
```

```
    REPEAT i:= LOINDEX(active) TO HIINDEX(active);
      member:= FALSE;
      REPEAT j:= LOINDEX(inferred) TO HIINDEX(inferred);
        IF (active[i].corresponds_to IN
          inferred[j].corresponds_to) THEN
          member:= TRUE;
          IF (inferred[j].availability = FALSE) THEN
            result:= FALSE;
          END_IF;
        END_IF;
      END_REPEAT;
      IF (member = FALSE) THEN
        result:= FALSE;
      END_IF;
    END_REPEAT;
    RETURN (result);
  END_FUNCTION;
(*
```

5.3.41 check_diagnosis_out

EXPRESS specification:

Function check_diagnosis_out checks the set of diagnostic outcomes against the set of legal outcomes of the diagnosis passed in to ensure that they are consistent. This is done by checking to see that the set of inferred outcomes are members of the set of available outcomes.

```
*)
  FUNCTION check_diagnosis_out
    (diag:SET [1:?] OF diagnosis; dout:diagnosis_outcome):BOOLEAN;
    LOCAL
      flag: BOOLEAN:= TRUE;
    END_LOCAL;
    REPEAT i:= LOINDEX(diag) TO HIINDEX(diag);
      IF (NOT(dout IN diag[i].has_outcome)) THEN
        flag:= FALSE;
      END_IF;
    END_REPEAT;
    RETURN(flag);
  END_FUNCTION;
(*
```

5.3.42 check_test_out

EXPRESS specification:

Function check_test_out checks the set of test outcomes against the set of legal outcomes of the test passed in to ensure that they are consistent. This is done by checking to see that the set of inferred outcomes are members of the set of available outcomes.

```
*)
  FUNCTION check_test_out
    (tst:SET [1:?] OF test; tout:test_outcome):BOOLEAN;
```

```
    LOCAL
      flag: BOOLEAN := TRUE;
    END_LOCAL;
    REPEAT i:= LOINDEX(tst) TO HIINDEX(tst);
      IF (NOT(tout IN tst[i].has_outcome)) THEN
        flag:= FALSE;
      END_IF;
    END_REPEAT;
    RETURN(flag);
  END_FUNCTION;
(*
```

5.3.43 diagnosis_set_union

EXPRESS specification:

Function diagnosis_set_union constructs a set of inferred_diagnosis by taking the union of all of the inferred_diagnosis entities associated with each member of a set of active_model.

```
*)
  FUNCTION diagnosis_set_union
    (mdl: SET [1:?] OF active_model): SET [1:?] OF diagnosis;

    LOCAL
      diagnosis_set : SET [1:?] OF diagnosis:= [];
    END_LOCAL;

    REPEAT i := LOINDEX(mdl) TO HIINDEX(mdl);
      diagnosis_set := diagnosis_set +
        QUERY(tmp <* mdl[i].corresponds_to.model_element|
        'AI_ESTATE_COMMON_ELEMENT_MODEL.diagnosis' IN TYPEOF(tmp));
    END_REPEAT;
    RETURN(diagnosis_set);
  END_FUNCTION;
(*
```

5.3.44 resource_set_union

EXPRESS specification:

Function resource_set_union constructs a set of active_resource by taking the union of all of the active_resource entities associated with each member of a set of active_model.

```
*)
  FUNCTION resource_set_union
    (mdl: SET [1:?] OF active_model): SET [1:?] OF resource;

    LOCAL
      resource_set: SET [1:?] OF resource:= [];
    END_LOCAL;

    REPEAT i:= LOINDEX(mdl) TO HIINDEX(mdl);
```

```
      resource_set:= resource_set +
      QUERY(tmp <* mdl[i].corresponds_to.model_element |
      'AI_ESTATE_COMMON_ELEMENT_MODEL.resource' IN TYPEOF(tmp));
    END_REPEAT;
    RETURN(resource_set);
  END_FUNCTION;
(*
```

5.3.45 resources_available

EXPRESS specification:

Function resources_available samples a set of resources to determine whether all of the resources in that set are currently available.

```
*)
  FUNCTION resources_available
    (resources: SET [0:?] OF active_resource): BOOLEAN;

    LOCAL
      result: BOOLEAN:= TRUE;
    END_LOCAL;

    IF SIZEOF(resources) > 0 THEN
      REPEAT i := LOINDEX(resources) TO HIINDEX(resources);
        IF resources[i].availability = FALSE THEN
          result:= FALSE;
        END_IF;
      END_REPEAT;
    END_IF;
    RETURN(result);
  END_FUNCTION;
(*
```

5.3.46 set_of_diagnosis

EXPRESS specification:

Function set_of_diagnosis constructs a set of diagnoses as defined in the CEM when given a set of inferred_diagnosis from the DCM.

```
*)
  FUNCTION set_of_diagnosis
    (diag: SET [0:?] OF inferred_diagnosis): SET [0:?] OF diagnosis;

    LOCAL
      diagnosis_set : SET [0:?] OF diagnosis:= [];
    END_LOCAL;

    REPEAT i:= LOINDEX(diag) TO HIINDEX(diag);
      diagnosis_set:= diagnosis_set + diag[i].corresponds_to;
    END_REPEAT;
    RETURN(diagnosis_set);
  END_FUNCTION;
(*
```

5.3.47 set_of_resource

EXPRESS specification:

Function set_of_resource constructs a set of resources as defined in the CEM when given a set of active_resources from the DCM.

```
*)
  FUNCTION set_of_resource
    (rsrc: SET [0:?] OF active_resource): SET [0:?] OF resource;

    LOCAL
      resource_set: SET [0:?] OF resource: = [];
    END_LOCAL;

    REPEAT i: = LOINDEX(rsrc) TO HIINDEX(rsrc);
      resource_set: = resource_set + rsrc[i].corresponds_to;
    END_REPEAT;
    RETURN(resource_set);
  END_FUNCTION;
(*
```

5.3.48 set_of_test

EXPRESS specification:

Function set_of_test constructs a set of tests as defined in the CEM when given a set of inferred_tests from the DCM.

```
*)
  FUNCTION set_of_test
    (tst: SET [0:?] OF inferred_test): SET [0:?] OF test;

    LOCAL
      test_set: SET [0:?] OF test: = [];
    END_LOCAL;

    REPEAT i: = LOINDEX(tst) TO HIINDEX(tst);
      test_set: = test_set + tst[i].corresponds_to;
    END_REPEAT;
    RETURN(test_set);
  END_FUNCTION;
(*
```

5.3.49 test_set_union

EXPRESS specification:

Function test_set_union constructs a set of tests by taking the union of all of the test entities associated with each member of a set of active_model.

```
*)
  FUNCTION test_set_union
    (mdl: SET [1:?] OF active_model): SET [1:?] OF test;
```

```
    LOCAL
      test_set: SET [1:?] OF test:= [];
    END_LOCAL;

    REPEAT i:= LOINDEX(mdl) TO HIINDEX(mdl);
      test_set:= test_set +
        QUERY(tmp <* mdl[i].corresponds_to.model_element |
        'AI_ESTATE_COMMON_ELEMENT_MODEL.test' IN TYPEOF(tmp));
      END_REPEAT;
      RETURN(test_set);
    END_FUNCTION;
  END_SCHEMA;
(*
```

5.3.50 Dynamic Context Model EXPRESS-G diagrams

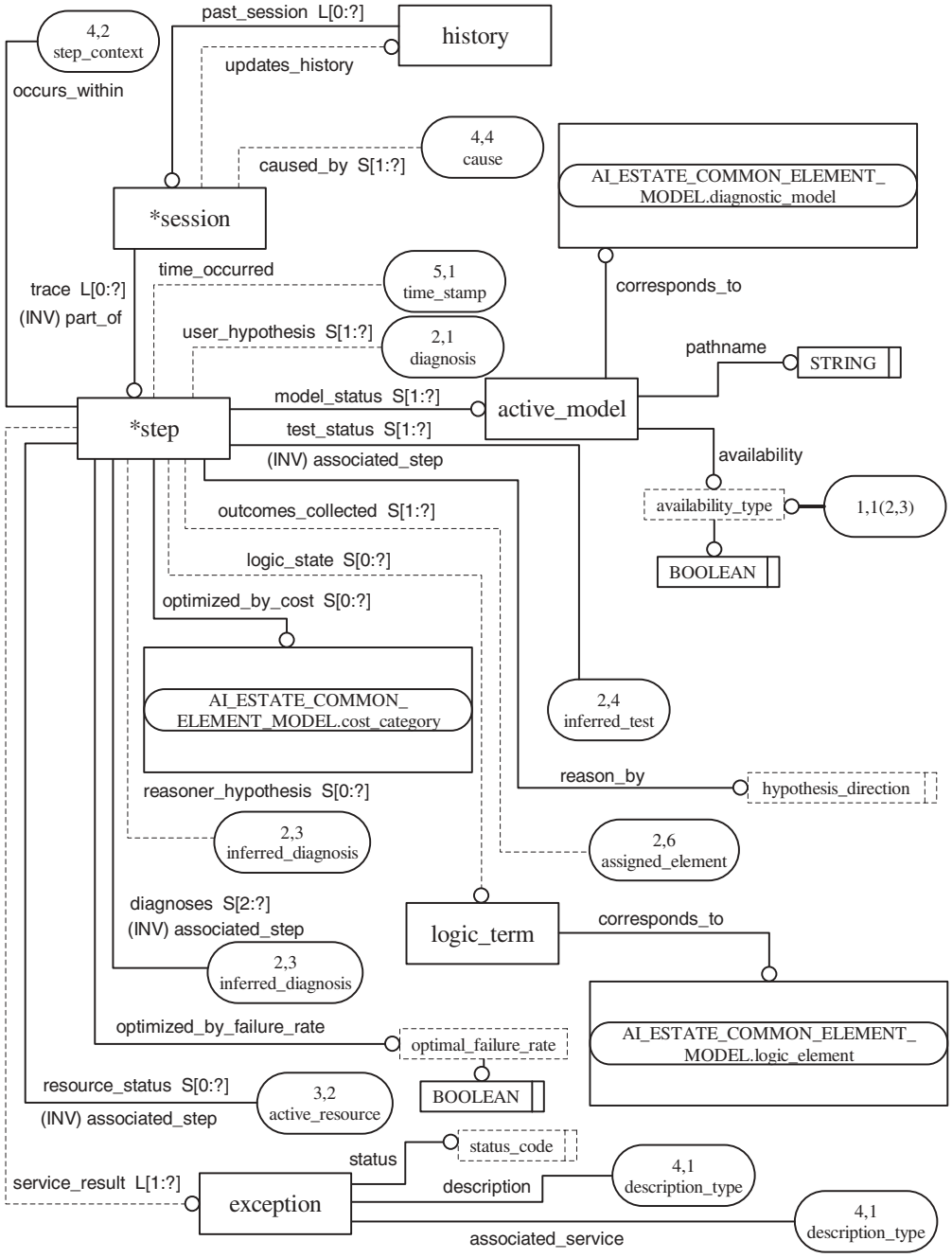


Figure 13—Dynamic Context Model EXPRESS-G: Diagram 1 of 5

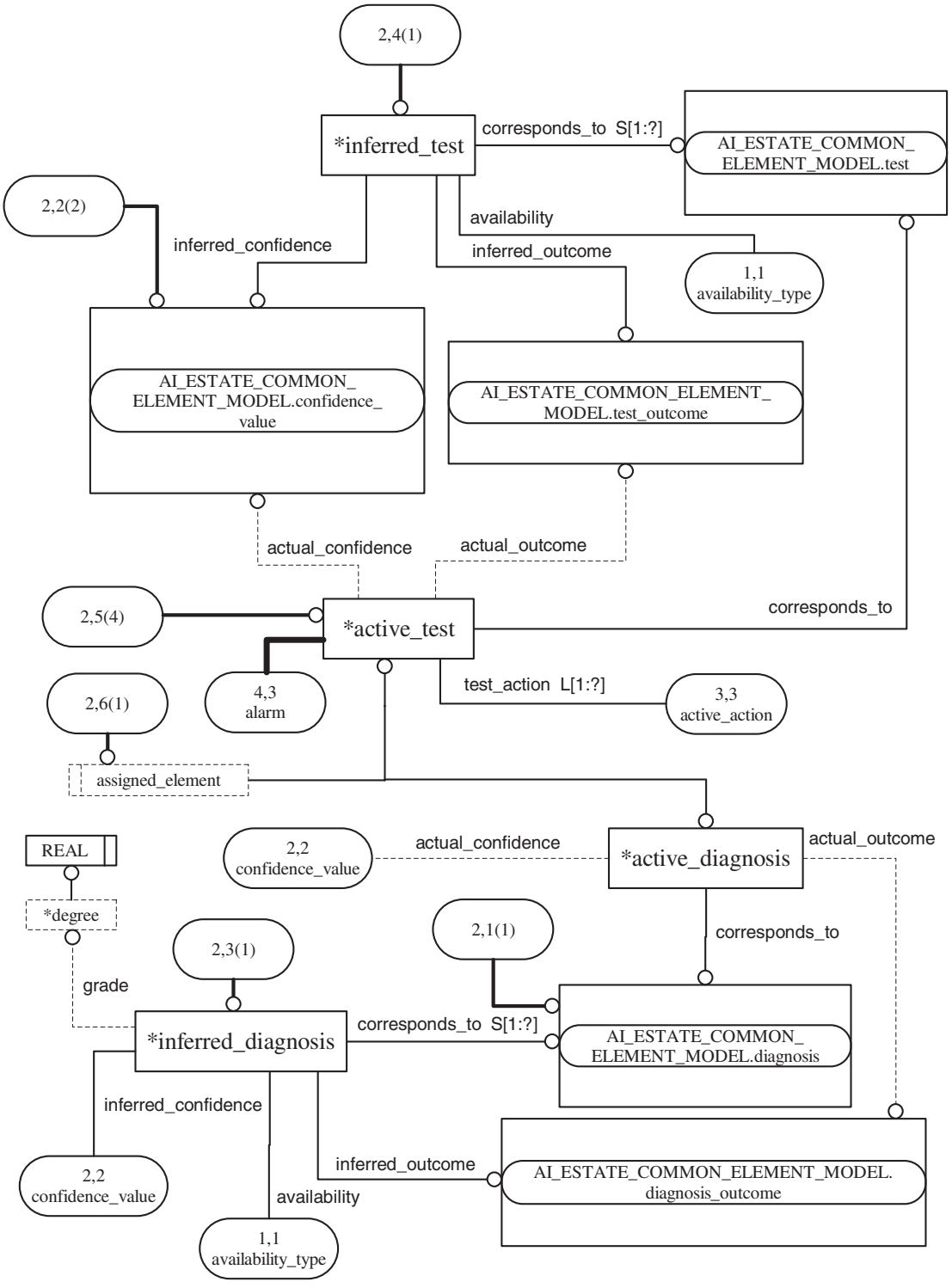


Figure 14—Dynamic Context Model EXPRESS-G: Diagram 2 of 5

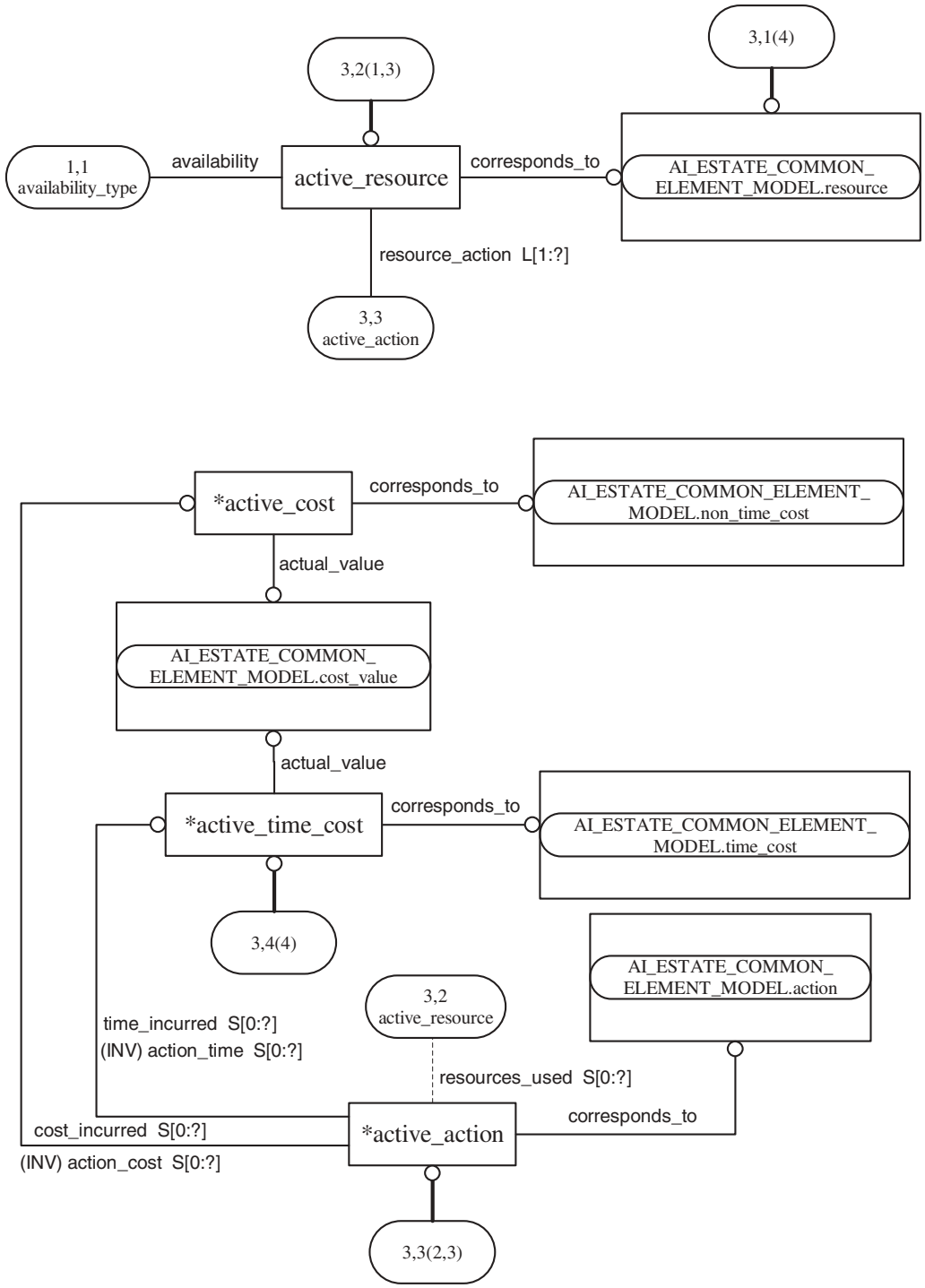


Figure 15—Dynamic Context Model EXPRESS-G: Diagram 3 of 5

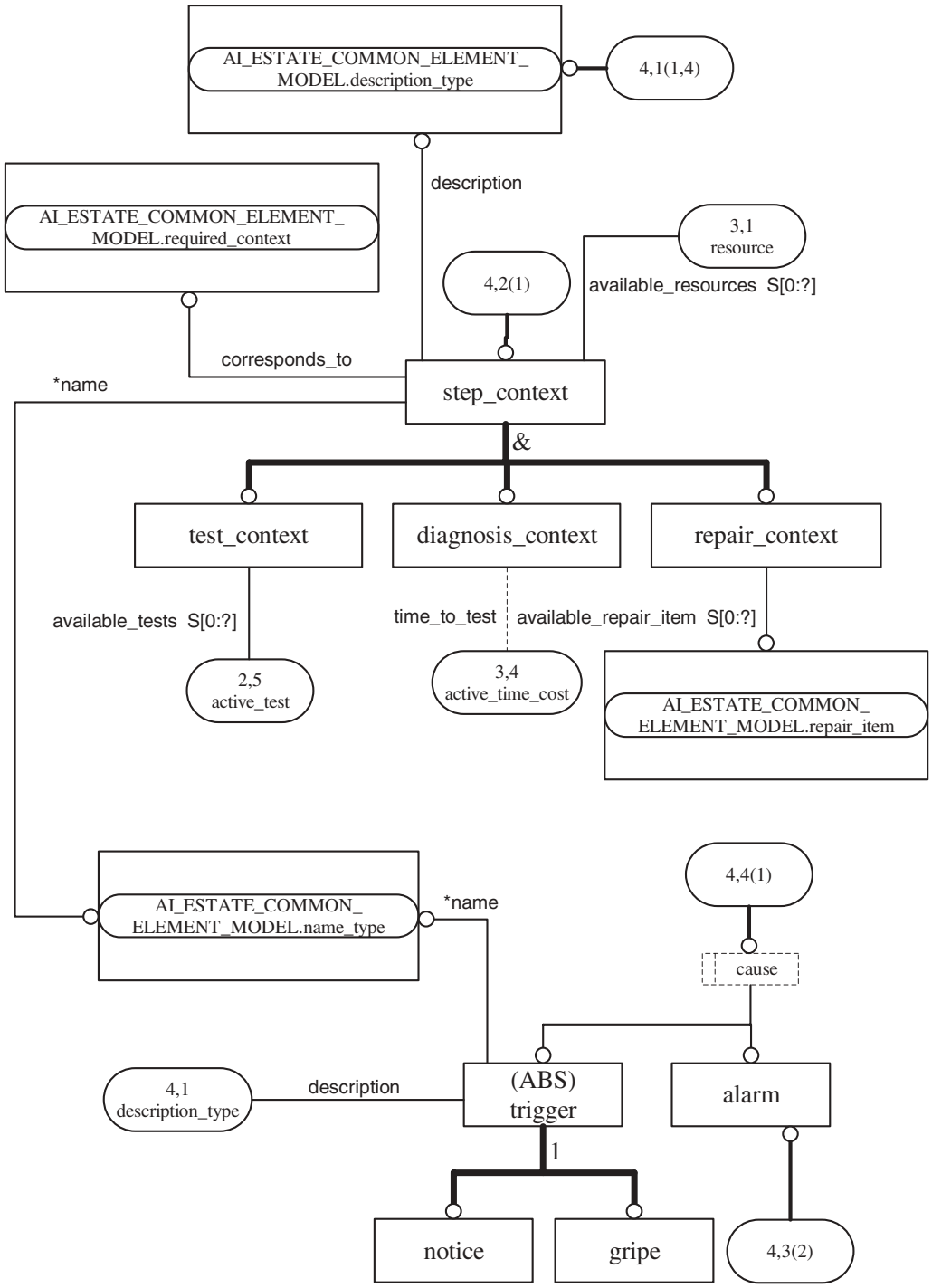


Figure 16—Dynamic Context Model EXPRESS-G: Diagram 4 of 5

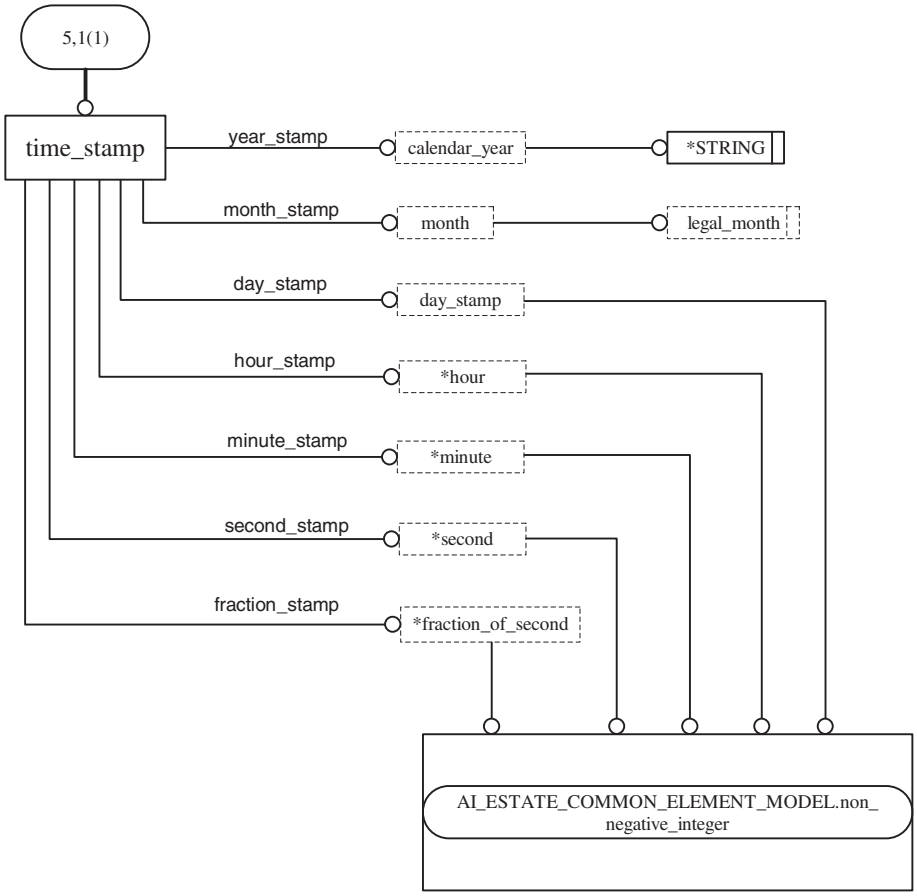


Figure 17—Dynamic Context Model EXPRESS-G: Diagram 5 of 5

*)
(*

5.4 Enhanced Diagnostic Inference Model

The AI-ESTATE Enhanced Diagnostic Inference Model (EDIM) schema is defined below. The constructs of this model were originally derived from the approach to system test and diagnosis known as *information flow modeling* or *dependency modeling* (Simpson and Sheppard [B6, B7, B8]). The model utilizes many of the constructs defined in the AI-ESTATE Common Element Model.

An inference is a logical relationship between two tests or between a test and a diagnosis. Given a particular test outcome, inferences about other tests and/or diagnoses of the system can be made. Tests offer a view of the associated fault, function, or diagnosis.

In the AI-ESTATE EDIM, test outcomes have been generalized beyond pass or fail outcomes to multiple outcomes for a test. Inferences are therefore identified between a particular test outcome and other test outcomes and asserted conditions of diagnostic units.

A particular test outcome that is inferred from another outcome is represented as a `test_inference` within the EDIM. Analogously, a `diagnostic_inference` within the EDIM asserts a condition of good, candidate, or some user-defined diagnosis outcome on a particular diagnostic unit. These two inference elements are the terms that compose the inference associated with a test outcome. Both `test_inference` and `diagnostic_inference` are subtypes of inference.

The set of inferences associated with a particular test outcome are represented in sum of products form (a disjunction of conjunctive terms). This representation provides flexibility and consistency in the logical expression of the inferences.

EXPRESS specification:

```
*)
SCHEMA ENHANCED DIAGNOSTIC INFERENCE MODEL;
  REFERENCE FROM AI_ESTATE_COMMON_ELEMENT_MODEL
    (diagnostic_model,
    test_outcome,
    confidence_value,
    diagnosis_outcome);
(*
```

5.4.1 inference_type

This type specifies whether or not an inference has been negated. This is intended to serve as a NOT operator. If an inference is considered NEGATIVE, it has been negated. If it is POSITIVE, it has not.

EXPRESS specification:

```
*)
  TYPE inference_type = ENUMERATION OF
    (POSITIVE,
    NEGATIVE);
  END_TYPE;
(*
```

5.4.2 diagnostic_inference

Entity `diagnostic_inference` represents an inference of a diagnosis as "good" or still a "candidate" from a particular test outcome (as identified by `outcome_inference`). Uncertainty associated with this inference can be specified in the confidence attribute that has been inherited from the inference entity.

EXPRESS specification:

```
*)
  ENTITY diagnostic_inference
    SUBTYPE OF(inference);
    diagnostic_assertion    : diagnosis_outcome;
  END_ENTITY;
(*
```

Attribute definitions:

`diagnostic_assertion` : The condition of the diagnosis to which the inference applies. The condition of the unit is asserted to be either "good" or still a "candidate." Since this is a diagnosis outcome, additional values can be assigned as well.

5.4.3 enhanced_diagnostic_inference_model

This entity represents the constituents of the EDIM. This construct also identifies the diagnoses, tests, and, optionally, required resources for the system under test being modeled.

EXPRESS specification:

```
*)
  ENTITY enhanced_diagnostic_inference_model
    SUBTYPE OF(diagnostic_model);
    inference    : SET [2:?] OF outcome_inference;
  END_ENTITY;
(*
```

Attribute definitions:

`inference` : This attribute identifies the set of `outcome_inference` that comprises the model of the system under test. To be useful, a model shall consist of at least two inferences corresponding to inferences from the minimum number of outcomes for the minimum number of tests in the model.

5.4.4 inference

This entity is a supertype of the entities `diagnostic_inference` and `test_inference`. Inference is either a test inference or a diagnosis inference, but not both.

EXPRESS specification:

```
*)
  ENTITY inference
    ABSTRACT SUPERTYPE OF (ONEOF(diagnostic_inference, test_inference));
    pos_neg      : inference_type;
    confidence   : OPTIONAL confidence_value;
  END_ENTITY;
(*
```

Attribute definitions:

`pos_neg` : A particular inference can either be a positive inference or negative inference. A positive inference is one that is TRUE. A negative inference is an inference that is FALSE. Thus, the negative inference is a positive inference with a NOT in front of it. In the EDIM, all inferences are treated as if they are asymmetric, meaning that nothing can be assumed about the inference from a particular outcome of a test because of another outcome of the same test. Thus, all `outcome_inferences` shall be explicitly defined.

`confidence` : A `confidence` entity that identifies the statistical confidence in the inference association from the outcome in `outcome_inference` to the following:

(a) The particular outcome identified in this entity if this entity is a `test_inference`.

(b) The particular diagnosis identified in this entity, in the condition identified in `diagnostic_assertion`, if this entity is a `diagnostic_inference`.

5.4.5 outcome_inference

This construct pairs a particular outcome of a particular test with a set of inferences represented in a conjunct/disjunct form. Each inference entity of the set is a single inference of type `test_inference` or `diagnostic_inference`. Since inference information is specific to the Enhanced Diagnostic Inference Model (EDIM), it is necessary to identify the test outcomes from the Common Element Model (CEM) with which the inferences are associated. Hence, the constructs for representing inferences are found in the EDIM, along with a pairing of these inference constructs with test outcomes from the CEM.

EXPRESS specification:

```
*)
  ENTITY outcome_inference;
    disjuncts                  : SET OF inference;
    conjuncts                  : SET OF inference;
    associated_test_outcome    : test_outcome;
  UNIQUE
    one_outcome    : associated_test_outcome;
  END_ENTITY;
(*
```

Attribute definitions:

`disjuncts` : This attribute is required, but the set can be empty. If the set is empty, then no inference can be drawn from the outcome. The set is interpreted as an ORed set of inferences to be drawn.

`conjuncts` : Each inference entity of the set is a single inference of type `diagnostic_inference` or `test_inference`. This attribute is required, but the set can be empty. If the set is empty, then no inference can be drawn from the outcome. The set is interpreted as an ANDed set of inferences to be drawn.

`associated_test_outcome` : This attribute identifies a particular outcome of the value of the `test_outcome.for_test` attribute to which the value of the `outcome_inference` attribute applies, where the `test_outcome.for_test` attribute identifies a particular test.

5.4.6 test_inference

Entity `test_inference` represents an inference about a test made from a test outcome. In other words, a test outcome that references this entity can depend on the particular outcome, identified in the `outcome_inference`, of the test identified in `outcome_inference.for_test`. Uncertainty associated with this inference can be specified in the confidence attribute that has been inherited from the inference entity.

EXPRESS specification:

```
*)
  ENTITY test_inference
    SUBTYPE OF(inference);
    outcome_inference     : test_outcome;
  END_ENTITY;
(*
```

Attribute definitions:

outcome_inference : This attribute identifies the test and associated outcome to be inferred. The test is identified by the for_test attribute of the test outcome.

*)
END_SCHEMA;
(*

5.4.7 Enhanced Diagnostic Inference Model EXPRESS-G diagram

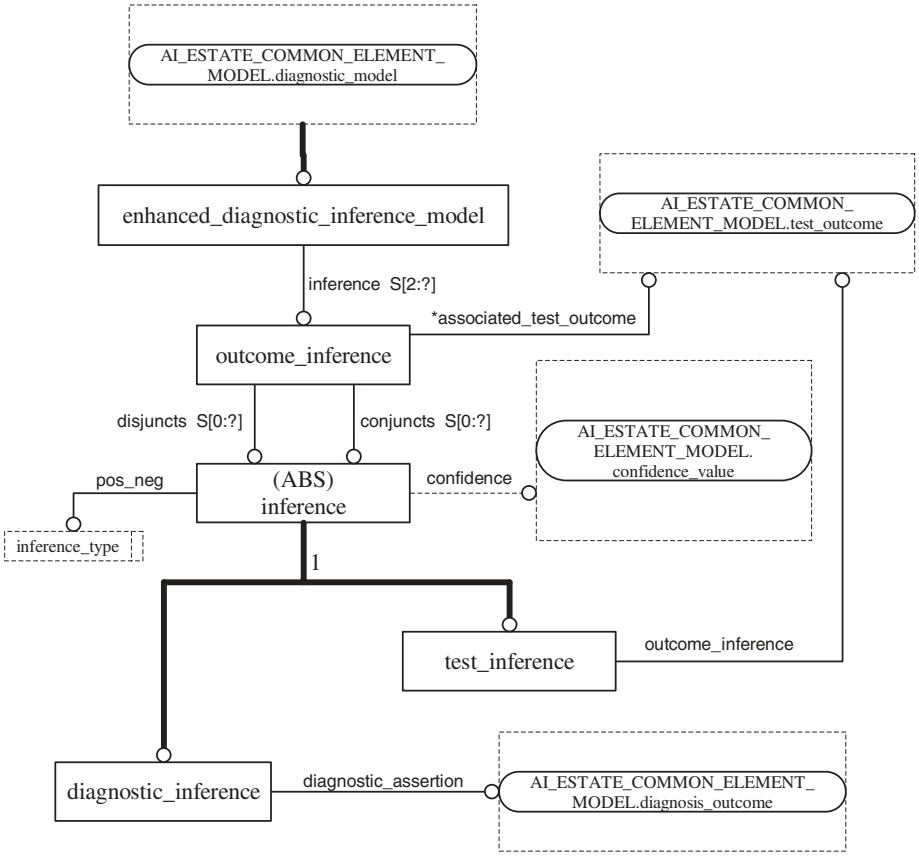


Figure 18—Enhanced Diagnostic Inference Model EXPRESS-G: Diagram 1 of 1

*)
(*

5.5 Fault Tree Model

This clause defines the AI-ESTATE Fault Tree specification. The constructs defined here are specific to the fault tree approach to system test and diagnosis. In this diagnostic method, a decision tree with fixed fault isolation strategies is constructed a priori and remains static during the diagnosis. The fault tree provides a test strategy that can be used without the aid of a reasoning system. This specification is included in the AI-ESTATE standard, since it is frequently used as the primary diagnostic strategy or in conjunction with other test generation strategies.

The structure of a fault tree can be viewed as a decision tree or table. The rows of the table correspond to the different tests to be run during the fault isolation procedure. Each column of a particular row corresponds to one of the possible outcomes for that test. The contents of a particular row and column

in the table identify the test result to be taken when the outcome identified occurs as the result of executing the test. The test result can identify the next row of the table to which to proceed, signal the diagnostic conclusion that is faulty, or simply provide information on the status of the fault isolation.

The Fault Tree Model draws on elements of the AI-ESTATE Common Element Model. The test entity corresponds to a row in the table such as that described previously. Each column position of a particular row in the table corresponds to the `test_result` entity in the model. The fault tree is processed by starting at the first step, executing the test associated with that test, and proceeding with the actions prescribed for the outcome that results. When another step of the fault tree appears in the column entry for the resulting test outcome, execution of the fault tree proceeds to that fault tree step. Eventually, the column entry for the resulting test outcome should identify the diagnosis and at this point (when no other fault tree steps appear in the column entry), processing of the fault tree is complete. As an option, running lists of suspected diagnoses can be included at any or all steps of the fault tree.

EXPRESS specification:

```
*)
SCHEMA AI_ESTATE_FAULT_TREE_MODEL;
  REFERENCE FROM AI_ESTATE_COMMON_ELEMENT_MODEL
    (diagnostic_model,
    test,
    test_outcome,
    diagnosis_outcome);
(*
```

5.5.1 fault_tree_model

The `fault_tree_model` entity represents the fault tree at the highest level of abstraction. Thus, it defines an entry point into the fault tree by identifying the first step (i.e., the root) of the tree. Multiple entry points can be defined but, to maintain acceptable form, they should be treated as separate models.

EXPRESS specification:

```
*)
  ENTITY fault_tree_model
    SUBTYPE OF(diagnostic_model);
    entry_point                    : fault_tree_step;
  END_ENTITY;
(*
```

Attribute definitions:

`entry_point` : Attribute `entry_point` identifies the first `fault_tree_step` in the fault tree. This is normally the starting point of the fault tree, but can represent any entry point into the tree.

5.5.2 fault_tree_step

This construct represents a row of the fault tree table. Its entries identify the test to be run at this step in the fault tree and which test result/action pairs to follow. The `test_step` attribute uses the test entity of the AI-ESTATE Common Element Model.

EXPRESS specification:

```
*)
  ENTITY fault_tree_step;
    result                 : SET [2:?] OF test_result;
    test_step              : test;
  INVERSE
    previous_result        : test_result FOR next_step;
  WHERE
    outcomes_are_valid     : result_outcomes(result) =
                             test_step.has_outcome;
  END_ENTITY;
(*
```

Attribute definitions:

result	: A set of at least two test_result entities that comprise the outcome/action pairs for the test identified in the test_step attribute. There should be a test_result entity in the result set for each possible outcome of the test in test_step.
test_step	: Identifies the test that is to be run for this step of the fault tree.
previous_result	: Identifies the result in the fault tree that leads to the new step in the fault tree.

Formal propositions:

outcomes_are_valid	: This rule verifies that there exists a legal outcome for the test for every test_result specified at this step in the fault tree. The rule is satisfied when the set of outcomes equals the set returned by the function result_outcomes.

5.5.3 test_result

Entity test_result provides the outcome associated with a test. That outcome is then paired with the corresponding test result, indicating the next step in the tree by pointing to that step. If appropriate, the next step of the fault tree to which execution should proceed is identified in the next_step attribute.

EXPRESS specification:

```
*)
  ENTITY test_result;
    next_step                    : OPTIONAL fault_tree_step;
    test_out                     : test_outcome;
    current_diagnosis_outcome    : SET OF diagnosis_outcome;
  INVERSE
    associated_step              : fault_tree_step FOR result;
  WHERE
    leaves_have_diagnoses  : (EXISTS(next_step))                          OR
                 (SIZEOF(current_diagnosis_ outcome) > 0);
  END_ENTITY;
(*
```

Attribute definitions:

next_step	: Identifies which fault_tree_step to execute next when the outcome in test_outcome results from the execution of the test of this fault_tree_step. This attribute is optional. When no fault_tree_step is identified, the test_result entity is a leaf of the fault tree.
test_out	: Identifies the outcome of the test of this fault_tree_step to which this construct applies.
current_diagnosis_outcome	: Identifies the diagnosis elements in the model that are indicted (i.e., accused) by the sequence of tests leading up to this point in the fault tree. This attribute is used to report the diagnosis resulting from traversing the tree.
associated_step	: Identifies the step with which the current result is associated. Since this is not a set, it enforces the tree structure of the fault tree (i.e., it is not a decision graph).

Formal propositions:

leaves_have_diagnoses	: This rule constrains the current_diagnosis_outcomes attribute that is a required attribute such that the associated list can be empty if associated with an internal node of the tree, but if the node is a leaf (i.e., a terminal step in the tree), then the current_diagnosis_outcomes list cannot be empty.

5.5.4 result_outcomes

EXPRESS specification:

Function result_outcomes takes a set of test results and returns the corresponding set of test outcomes to ensure that the outcomes listed correspond to the outcomes available at the step in the tree.

```
*)
  FUNCTION result_outcomes
    (results:SET [0:?] OF test_result) : SET [0:?] OF test_outcome;

    LOCAL
       t_out: SET [0:?] OF test_outcome := [];
    END_LOCAL;

    REPEAT i := LOINDEX(results) TO HIINDEX(results);
      t_out := t_out + results[i].test_out;
    END_REPEAT;
    RETURN(t_out);
  END_FUNCTION;
END_SCHEMA;
(*
```

5.5.5 Fault Tree Model EXPRESS-G diagram

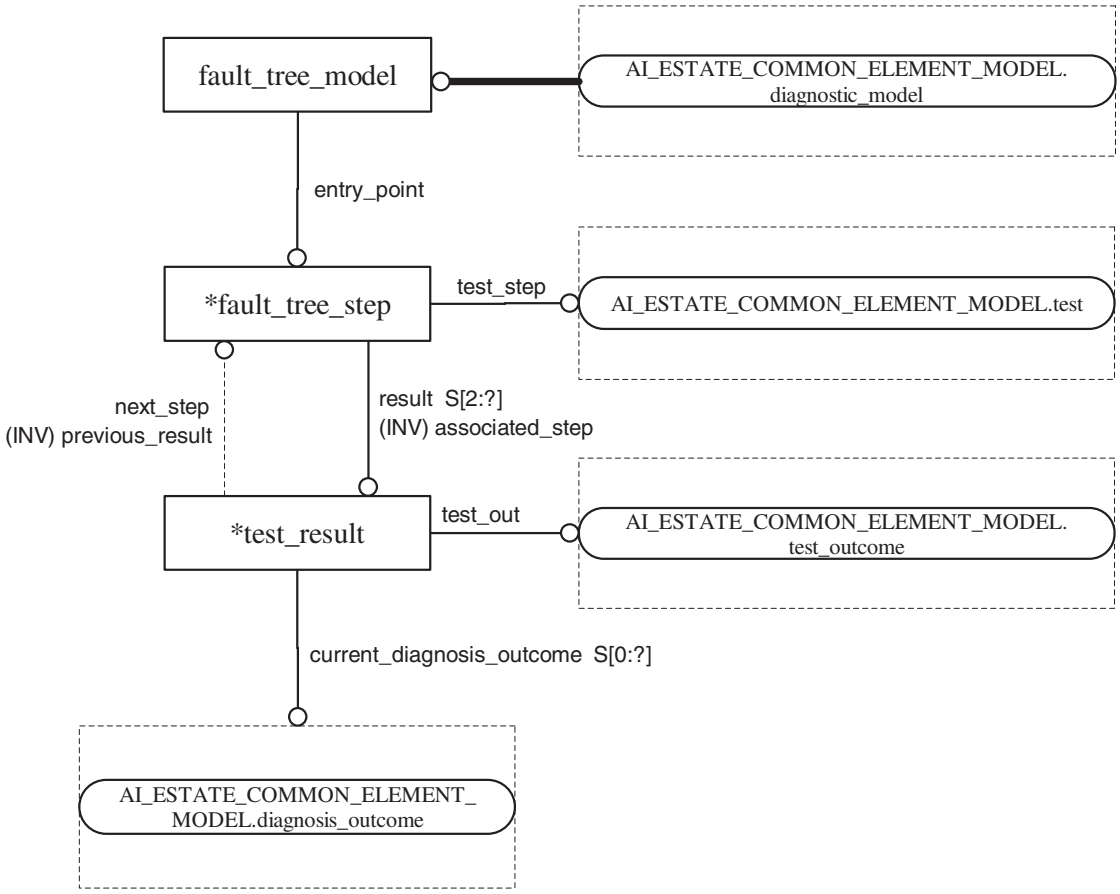


Figure 19—Fault Tree Model EXPRESS-G: Diagram 1 of 1

*)

6. Services

This clause specifies a set of services for model management and reasoner manipulation, The services are specified using EXPRESS notation and reference entities in the information models specified in Clause 5. The identification of an entity is provided within a service call via an ID parameter.

The name of the ID parameter in the service call will identify the entity type. The standard specifies a format for the ID parameter name; for example,

```
ID_⟨entitytype⟩ : entity_id
```

where ⟨entitytype⟩ is the type of the entity that the ID references. For the service definitions that follow, ⟨attributetype⟩ indicates the type of an attribute in the model and ⟨attributename⟩ indicates the specific attribute identifier from the model.

NOTE—entity_id uniquely identifies an entity of type ⟨entitytype⟩. Further, a particular entity_id is unique to the state of the reasoner and will persist through the corresponding reasoning session.

6.1 Model management services

Model management services are defined to facilitate management and manipulation of entities in diagnostic and Dynamic Context Models. The basic services, create, delete, get, and put, are intended to manipulate and manage model entities whether during model construction or diagnostic reasoning. In addition, several "utility" services are defined for more complex attributes corresponding to counting services, existence services, indexing services, list attribute services, and set attribute services. All of the basic services are defined relative to the entities and attributes of the information models defined in Clause 5.

Model management services are responsible for managing or manipulating model entities. Particular sequences of services can leave the model inconsistent according to the constraints of the model specification. It is up to the application managing the models to ensure consistency prior to using or exchanging information.

6.1.1 Create services

Primitive entity create services shall be defined according to the following specification. Create services return an `entity_id` pointing to the created entity.

```
FUNCTION create_⟨entitytype⟩ : entity_id;
END_FUNCTION;
```

Example:

```
FUNCTION create_diagnostic_model : entity_id;
END_FUNCTION;
```

6.1.2 Delete services

Primitive entity delete services shall be defined according to the following specification. If the entity referenced by `ID_entity` does not contain an attribute named ⟨`attributename`⟩, the get service will fail and `status_code` will be returned with the value `NONEXISTENT_DATA_ELEMENT_REQUESTED`.

```
PROCEDURE delete (ID_⟨entitytype⟩ : entity_id);
END_PROCEDURE;
```

Example:

```
PROCEDURE delete
    (ID_diagnostic_model : entity_id);
END_PROCEDURE;
```

6.1.3 Get services

Primitive attribute get services shall be defined according to the following specification. If the entity referenced by `ID_entity` does not contain an attribute named ⟨`attributename`⟩, the get service

will fail and status_code will be returned with the value NONEXISTENT_DATA_ELEMENT_REQUESTED.

```
FUNCTION get_⟨attributename⟩ (ID_entity : entity_id) :
    ⟨attributetype⟩;
END_FUNCTION;
```

Example:

```
FUNCTION get_name (ID_entity : entity_id) : name_type;
END_FUNCTION;
```

6.1.4 Put services

Primitive attribute put services shall be defined according to the following specification. If the entity referenced by ID_entity does not contain an attribute named ⟨attributename⟩, the put service will fail and status_code will be returned with the value NONEXISTENT_DATA_ELEMENT_REQUESTED.

```
PROCEDURE put_⟨attributename⟩
    (ID_entity : entity_id,
     attribute_value : ⟨attributetype⟩);
END_PROCEDURE;
```

Example:

```
PROCEDURE put_name
    (ID_entity : entity_id,
     attribute_value : name_type);
END_PROCEDURE;
```

6.1.5 Counting services

Count functions are provided with an entity identifier and return an integer value indicating the number of elements in an aggregate that are associated with the entity attribute identified by the function name. If the entity referenced by ID_entity does not contain an attribute named ⟨attributename⟩, the get service will fail and status_code will be returned with the value NONEXISTENT_DATA_ELEMENT_REQUESTED. The functions are defined according to the following specification:

```
FUNCTION get_⟨attribute_name⟩_count (ID_entity : entity_id) :
    non_negative_integer;
END_FUNCTION;
```

Example:

```
FUNCTION get_resource_count (ID_action : entity_id) :
    non_negative_integer;
END_FUNCTION;
```

6.1.6 Existence services

Existence functions are provided with an entity instance identifier and return a BOOLEAN value indicating the existence of the entity optional attribute identified by the function name. If the optional attribute does exist, then TRUE is returned. If the optional attribute does not exist, then FALSE is returned. If the entity referenced by `ID_entity` does not contain an attribute named ⟨`attributename`⟩, the service will fail and `status_code` will be returned with the value `NONEXISTENT_DATA_ELEMENT_REQUESTED`. The functions are defined according to the following specification:

```
FUNCTION does_⟨attribute_name⟩_exist (ID_entity : entity_id) :
    BOOLEAN;
END_FUNCTION;
```

Example:

```
FUNCTION does_resource_exist (ID_action : entity_id) :
    BOOLEAN;
END_FUNCTION;
```

6.1.7 Indexing services

Indexing functions are provided to get and put the value associated with an element in an aggregate given the entity instance identifier and an integer value indicating the ordinal number of the desired element (and, in the case of a put, the value to be "put"). Putting a value is destructive in that it replaces the value currently in the specified position. If the entity referenced by `ID_entity` does not contain an attribute named ⟨`attributename`⟩, the service will fail and `status_code` will be returned with the value `NONEXISTENT_DATA_ELEMENT_REQUESTED`. The functions are defined according to the following specification:

6.1.7.1 Get nth

```
FUNCTION get_nth_⟨attribute_name⟩
    (ID_entity : entity_id; n : integer) :
       ⟨attributetype⟩;
END_FUNCTION;
```

6.1.7.2 Get last

```
FUNCTION get_last_⟨attribute_name⟩
    (ID_entity : entity_id) :
       ⟨attributetype⟩;
END_FUNCTION;
```

6.1.7.3 Put nth

```
PROCEDURE put_nth_⟨attribute_name⟩
    (ID_entity  : entity_id;
     n : integer) :
        attribute_value : ⟨attributetype⟩;
END_PROCEDURE;
```

6.1.7.4 Put last

```
PROCEDURE put_last_⟨attribute_name⟩
    (ID_entity          : entity_id;
     attribute_value    : ⟨attributetype⟩) ;
END_PROCEDURE;
```

Examples:

```
FUNCTION get_nth_repair_resource
    (rpr_item              : entity_id;
     n : integer)          : repair_resource;
END_FUNCTION;

PROCEDURE put_nth_repair_resource
    (rpr_item              : entity_id;
     n                     : integer;
     attribute_value       : repair_resource) ;
END_PROCEDURE;

FUNCTION get_last_repair_resource
    (rpr_item : entity_id) : repair_resource;
END_FUNCTION;

PROCEDURE put_last_repair_resource
    (rpr_item              : entity_id;
     attribute_value       : repair_resource) ;
END_PROCEDURE;
```

6.1.8 List attribute services

6.1.8.1 Insert-in-list services

The item is inserted in the list at the location indicated by `index` and the remainder of the list is shifted to the right. An index value of zero references the first item in the list. If the entity referenced by `entity_id` does not contain an attribute named ⟨`attributename`⟩, the service will fail with a `status_code` of `NONEXISTENT_DATA_ELEMENT_REQUESTED`. If the `list_item` is not valid for the target attribute named ⟨`attributename`⟩ or the `index` value is not valid for the target list (i.e., out of range), the service will fail with a `status_code` of `MISSING_OR_INVALID_ARGUMENT`.

Services to insert an item into an attribute list are defined according to the following specification:

```
PROCEDURE insert_in_⟨attributename⟩_list
    (ID_entity     : entity_id;
     list_item     : ⟨attributetype⟩;
     index         : INTEGER);
END_PROCEDURE;
```

Example:

```
PROCEDURE insert_in_trace_list
    (ID_entity     : entity_id;
     list_item     : entity_id;
     index         : INTEGER);
END_PROCEDURE;
```

6.1.8.2 Append-to-list services

These services insert the specified entity at the end of the given list. To insert an entity into an empty list, the "append to" service must be used. If the entity referenced by `entity_id` does not contain an attribute named ⟨`attributename`⟩, the service will fail with a `status_code` of `NONEXISTENT_DATA_ELEMENT_REQUESTED`. If the `list_item` is not valid for the target attribute named ⟨`attributename`⟩, the service will fail with a `status_code` of `MISSING_OR_INVALID_ARGUMENT`.

Services to append an item to an attribute list are defined according to the following specification:

```
PROCEDURE append_to_⟨attributename⟩_list
    (ID_entity     : entity_id,
     list_item     : ⟨attributetype⟩);
END_PROCEDURE;
```

Example:

```
PROCEDURE append_to_trace_list
    (ID_entity     : entity_id,
     list_item     : entity_id);
END_PROCEDURE;
```

6.1.8.3 Remove-from-list services

The item at `index` is removed from the list and the remainder of the list is shifted left. An index value of zero references the first item in the list. If the entity referenced by `entity_id` does not contain an attribute named ⟨`attributename`⟩, the service will fail with a `status_code` of `NONEXISTENT_DATA_ELEMENT_REQUESTED`. If the index value is not valid for the target attribute list (i.e. out of range), the service will fail with a `status_code` of `MISSING_OR_INVALID_ARGUMENT`.

Services to remove an item from an attribute list are defined according to the following specification:

```
PROCEDURE remove_from_⟨attributename⟩_list
    (ID_entity     : entity_id;
     index         : INTEGER);
END_PROCEDURE;
```

Example:

```
PROCEDURE remove_from_trace_list
    (ID_entity     : entity_id;
     index         : INTEGER);
END_PROCEDURE;
```

6.1.9 Set attribute services

6.1.9.1 Add-to-set services

Attempting to add an item to a set that is already a member of the set is an error. If the entity referenced by `entity_id` does not contain an attribute named ⟨`attributename`⟩, the service will fail with a `status_code` of `NONEXISTENT_DATA_ELEMENT_REQUESTED`. If the `set_item` is not valid for the target attribute named ⟨`attributename`⟩, the service will fail with a `status_code` of `MISSING_OR_INVALID_ARGUMENT`.

Services to add an item to a set attribute are defined according to the following specification:

```
PROCEDURE add_to_⟨attributename⟩_set
    (ID_entity     : entity_id;
     set_item);    : ⟨attributetype⟩;
END_PROCEDURE;
```

Example:

```
PROCEDURE add_to_has_role_set
    (ID_entity     : entity_id;
     set_item      : role);
END_PROCEDURE;
```

6.1.9.2 Remove-from-set services

If the entity referenced by `entity_id` does not contain an attribute named ⟨`attributename`⟩, or if set_item is not a member of the set, the service will fail with `status_code NONEXISTENT_DATA_ELEMENT_REQUESTED`.

Services to remove an item from an attribute set are defined according to the following specification:

```
PROCEDURE remove_from_⟨attributename⟩_set
    (ID_entity     : entity_id;
     set_item      : ⟨attributetype⟩);
END_PROCEDURE;
```

Example:

```
PROCEDURE remove_from_has_role_set
    (ID_entity     : entity_id;
     set_item      : role);
END_PROCEDURE;
```

6.1.10 Typing service

Function `get_type` returns a string providing the type of the entity specified by `entity_id`. The type shall correspond to that returned by the TYPEOF function in EXPRESS when applied to an entity or attribute in an EXPRESS model (i.e., ⟨`model_name`⟩.⟨`element_type`⟩). If an `entity_id` is provided that does not exist in the model, then the exception `NONEXISTENT_DATA_ELEMENT_REQUESTED` shall be raised.

```
FUNCTION get_type
    (ID_entity : entity_id)   : STRING;
END_FUNCTION;
```

6.2 Reasoner manipulation services

In addition to managing model elements, a diagnostic reasoner is responsible for processing diagnostic information for the purpose of detection, localization, and isolation of faults within a system.

Reasoner manipulation services are derived from combinations of model management services, other reasoner manipulation services, and various processing algorithms. The reasoner manipulation services are subdivided into control services, analysis services, state accessor services, explanation services, and inference services.

6.2.1 Control services

6.2.1.1 attach_model

It is an error to attempt to attach a model that is already attached, and the exception OPERATION_OUT_OF_SEQUENCE is raised. The service will attach the diagnostic_model entity referenced by ID_diagnostic_model to the step entity referenced by ID_step. It does this by creating an active_model entity corresponding to the diagnostic model and adding the active_model to the step's model_status attribute.

A service to attach a diagnostic model to a step in the current reasoning session is defined according to the following specification:

```
PROCEDURE attach_model
    (ID_step                : entity_id,
     ID_diagnostic_model    : entity_id);
END_PROCEDURE;
```

6.2.1.2 detach_model

Procedure detach_model will detach the specified model from the current reasoning session. This service will delete the associated instance of active_model within the DCM, remove the reference to the attached diagnostic_model in the corresponds_to attribute of active_model, then remove the reference to the active_model in the model_status attribute of the step entity referenced in the parameter list. Detaching a model that is not currently attached shall raise the exception NONEXISTENT_DATA_ELEMENT_REQUESTED.

```
PROCEDURE detach_model
    (ID_step                : entity_id,
     ID_diagnostic_model    : entity_id);
END_PROCEDURE;
```

6.2.1.3 save_model

Procedure save_model will write the specified diagnostic model to secondary storage. If an attempt is made to save a nonexistent model or a model to an invalid path, then the exception MISSING_OR_INVALID_ARGUMENT is raised.

```
PROCEDURE save_model
    (ID_diagnostic_model  : entity_id ; path : STRING);
END_PROCEDURE;
```

6.2.1.4 load_model

Function load_model will load the specified diagnostic model from secondary storage. If an attempt to load a model that is already loaded is made, an error will occur and the exception

OPERATION_OUT_OF_SEQUENCE is raised. If an attempt is made to load a model using an invalid path, then the exception MISSING_OR_INVALID_ARGUMENT is raised.

```
FUNCTION load_model
    (path : STRING)    : entity_id;
END_FUNCTION;
```

6.2.1.5 save_exchange_model

Procedure save_exchange_model will write the specified model (either diagnostic model or DCM) to secondary storage using the exchange format specified in 4.2. For a diagnostic model, the entity_id for that model entity is provided. For a DCM, the entity_id for the session entity is provided. If an attempt is made to save a nonexistent model or a model to an invalid path, then the exception MISSING_OR_INVALID_ARGUMENT is raised. If an attempt is made to save an invalid exchange model (i.e., a model that does not satisfy all of the constraints specified in this standard), then the exception INVALID_MODEL_SCHEMA is raised.

```
PROCEDURE save_exchange_model
    (ID_diagnostic_model : entity_id ; path : STRING);
END_PROCEDURE;
```

6.2.1.6 load_exchange_model

Function load_exchange_model will load the specified exchange (as defined in 4.2) model (either diagnostic model or DCM) from secondary storage. For a diagnostic model, the entity_id for that model entity is returned. For a DCM, the entity_id for the session entity is returned. If an attempt to load a model that is already loaded is made, an error will occur and the exception OPERATION_OUT_OF_SEQUENCE is raised. If an attempt is made to load a model using an invalid path, then the exception MISSING_OR_INVALID_ARGUMENT is raised. If an attempt is made to load an invalid exchange model (i.e., a model that does not satisfy all of the constraints specified in this standard), then the exception INVALID_MODEL_SCHEMA is raised.

```
FUNCTION load_exchange_model
    (path : STRING)   : entity_id;
END_FUNCTION;
```

6.2.1.7 backtrack

Function backtrack will take an argument (n) specifying the number of steps to back up the state of the (active) DCM in a diagnostic session. This function returns the identifier of the step backed up to. If n is greater than the number of steps, then exception MISSING_OR_INVALID_ARGUMENT shall be raised. This service is destructive in that steps following the backed-up step are deleted from the session trace and are no longer accessible.

```
FUNCTION backtrack
    (n : integer)   : entity_id;
END_FUNCTION;
```

6.2.1.8 set_checkpoint

The intent of this service is to provide a means to save reasoner state and history so that it can be restored at some point in the future. Procedure set_checkpoint is implementation dependent and

saves the current contents of the DCM to secondary storage. If an attempt is made to save to an invalid path then the exception MISSING_OR_INVALID_ARGUMENT is raised.

```
PROCEDURE set_checkpoint
    (path : STRING)
END_PROCEDURE;
```

6.2.1.9 restore_checkpoint

Function restore_checkpoint is implementation dependent and restores the former content of the DCM from the identified saved checkpoint. The restored DCM completely replaces the current DCM. The service shall return an entity_id of type session. To obtain the last step executed apply, e.g., "id_curr_step = get_last_trace(id_session)." If this function is invoked and no checkpoint is set, then the exception NONEXISTENT_DATA_ELEMENT_REQUESTED is raised. If an attempt is made to restore from an invalid path, then the exception MISSING_OR_INVALID_ARGUMENT is raised.

```
FUNCTION restore_checkpoint
    (path : STRING)   : entity_id;
END_FUNCTION;
```

6.2.1.10 initialize_diagnostic_process

Function initialize_dianostic_process is implementation dependent—its behavior is to bring the reasoner (and DCM) to an initial state from which the diagnostic process can begin and implementations can perform their initialization routines. The models other than the DCM cannot be modified during an active diagnostic process. At this point, the behavior of the create, delete, put, and list attribute services for models other than the DCM become undefined. This service returns the identifier of the session entity in the DCM.

```
FUNCTION initialize_diagnostic_process   : entity_id;
END_FUNCTION;
```

6.2.1.11 close_diagnostic_process

Procedure close_dianostic_process is implementation dependent—its behavior is to terminate the diagnostic process and prohibit further modification of the DCM. The models other than the DCM can be modified after the diagnostic process is closed; however, the DCM cannot be modified; At this point, the behavior of all create, delete, put, and list attribute services for models other than the DCM become defined as specified, and the create, delete, put, and list attribute services for the DCM become undefined.

```
PROCEDURE close_diagnostic_process;
END_PROCEDURE;
```

6.2.1.12 assign_tests

The application executive may issue an assign_tests request in the interval between initialization of a new step and the execution of a specified test in order to pre-empt or override reasoner test selection. The entire set of test_performed attached to the identified step is replaced by the specified set.

NOTE—Assigning a set of tests makes no presumptions about necessary preconditions associated with those tests having been satisfied. Specifying a test that is not resident in the associated CEM shall raise exception NONEXISTENT_DATA_ELEMENT_REQUESTED.

```
PROCEDURE assign_tests
    (tst : SET [1:?] OF entity_id);
END_PROCEDURE;
```

6.2.1.13 select_test

Function select_test will return a set of identifiers of the next best recommended tests to execute based on current reasoner state information and its operative algorithm. This set will be attached to the current step attribute, test_performed.

```
FUNCTION select_test :
    SET [1:?] OF entity_id;
END_FUNCTION;
```

6.2.2 Analysis services

6.2.2.1 estimated_time_cost_to_isolate

Function estimated_time_cost_to_isolate returns the expected value of time costs over all paths leading from the current step to terminating diagnosis, using the current optimization criterion if it exists. Note that termination of diagnosis is implementation dependent.

```
FUNCTION estimated_time_cost_to_isolate
    (ID_time_cost : entity_id)    : cost_value;
END_FUNCTION;
```

6.2.2.2 estimated_non_time_cost_to_isolate

Function estimated_non_time_cost_to_isolate returns the expected value of non-time costs over all paths leading from the current step to terminating diagnosis, using the current optimization criterion if it exists. Note that termination of diagnosis is implementation dependent.

```
FUNCTION estimated_non_time_cost_to_isolate :
    (ID_non_time_cost : entity_id) : cost_value;
END_FUNCTION;
```

6.2.2.3 estimated_time_cost_to_repair

Function estimated_time_cost_to_repair returns the expected value of time costs over all paths leading from the current step to terminating diagnosis and performing subsequent repair, using the current optimization criterion if it exists. Note that termination of diagnosis is implementation dependent.

```
FUNCTION estimated_time_cost_to_repair :
    (ID_time_cost : entity_id) : cost_value;
END_FUNCTION;
```

6.2.2.4 estimated_non_time_cost_to_repair

Function `estimated_non_time_cost_to_repair` returns the expected value of non-time costs over all paths leading from the current step to terminating diagnosis and performing subsequent repair, using the current optimization criterion if it exists. Note that termination of diagnosis is implementation dependent.

```
FUNCTION estimated_non_time_cost_to_repair :
    (ID_non_time_cost : entity_id) : cost_value;
END_FUNCTION;
```

6.2.2.5 estimated_resources_to_isolate

Function `estimated_resources_to_isolate` will return the set of `entity_ids` for the required resources needed to isolate, ranked by probability across all paths leading from the specified step to terminating diagnosis given the current optimization criterion if it exists. Note that termination of diagnosis is implementation dependent.

```
FUNCTION estimated_resources_to_isolate(ID_step : entity_id) :
    SET [0:?] OF entity_id;
END_FUNCTION;
```

6.2.2.6 estimated_resources_to_repair

Function `estimated_resources_to_repair` will return the set of `entity_ids` for the required resources needed to repair, ranked by probability across all paths leading from the specified step to terminating diagnosis given the current optimization criterion if it exists. Note that termination of diagnosis is implementation dependent.

```
FUNCTION estimated_resources_to_repair(ID_step : entity_id) :
    SET [0:?] OF entity_id;
END_FUNCTION;
```

6.2.2.7 request_hypothesis_confidence

Function `request_hypothesis_confidence` returns the confidence of the supplied hypothesis given the current state of the reasoner. A hypothesis is given by a set of diagnoses that might be true.

```
FUNCTION request_hypothesis_confidence (hyp: SET [1:?] OF entity_id) :
    confidence_value;
END_FUNCTION;
```

6.2.2.8 get_test_outcomes_from_diagnosis

Given a set of tests, a diagnosis, and the associated `diagnosis_outcome`, function `get_test_outcomes_from_diagnosis` returns the set of test outcomes that would lead to that `diagnosis_outcome` being drawn.

```
FUNCTION get_test_outcomes_from_diagnosis
    (diag : entity_id;
     diag_out: diagnosis_outcome;
     tst : SET [1:?] OF entity_id) :
       SET [1:?] OF entity_id;
END_FUNCTION;
```

6.2.3 Reasoner state accessor services

6.2.3.1 request_test_resources_needed

Function request_test_resources_needed returns the required resources given a set of entity_ids of type test.

```
FUNCTION request_test_resources_needed
    (test_list : SET [1:?] OF entity_id) :
      SET [0:?] OF entity_id;
END_FUNCTION;
```

6.2.3.2 request_repair_resources_needed

Function request_repair_resources_needed returns the required resources given a set of entity_ids of type repair.

```
FUNCTION request_repair_resources_needed
    (repair_list : SET [1:?] OF entity_id) :
      SET [0:?] OF entity_id;
END_FUNCTION;
```

6.2.4 Explain services

6.2.4.1 show_session_trace

Function show_session_trace returns a formatted string that gives a step-by-step accounting of the inference process followed in the current diagnostic session. Note that actual information returned is implementation dependent.

```
FUNCTION show_session_trace :
    STRING;
END_FUNCTION
```

6.2.4.2 describe_reasoner

Function describe_reasoner returns a string that contains reasoner identification and configuration information. The information provided by the reasoner is reasoner specific and is not defined by this standard.

```
FUNCTION describe_reasoner :
    STRING;
END_FUNCTION;
```

6.2.5 Inference services

6.2.5.1 apply_ outcomes

Function apply_outcomes can be built from "put_actual_outcome," "put_actual_confidence," and the service "update_state." A one-to-one correspondence shall exist across the three sets. Upon completion of this service, a new step is appended to the end of the trace. This service returns the entity_id for the newly created step.

```
FUNCTION apply_outcomes
    (tst : SET [1:?] OF entity_id;
    out  : SET [1:?] OF entity_id;
    cf   : SET [1:?] OF confidence_value;
    ID_step : entity_id) :
      entity_id;
END_FUNCTION;
```

6.2.5.2 update_state

Procedure `update_state` triggers the reasoner's inference process whereby the inferred diagnostic and test outcomes are determined and a hypothesis (if applicable) is computed. The end result is that `current_step` is fully instantiated.

```
PROCEDURE update_state
    (ID_step : entity_id);
END_PROCEDURE;
```

6.2.5.3 get_most_likely_diagnoses

Function `get_most_likely_diagnoses` returns a ranked list of the `n` most likely inferred diagnoses. The order of the ranking shall be implementation specific. If `n` is greater than the size of the set of diagnoses, then exception `MISSING_OR_INVALID_ARGUMENT` is raised.

```
FUNCTION get_most_likely_diagnoses (n : INTEGER) :
    SET [1:?] OF entity_id;
END_FUNCTION;
```

Annex A

(informative)

Bibliography

[B1] Fikes, R.E. and Nilsson, N.J., "STRIPS: A new approach to the application of theorem proving to artificial intelligence," *Artificial Intelligence*, vol. 2, nos. 3–4, pp. 189–208, 1971.

[B2] IEEE 100, *The Authoritative Dictionary of IEEE Standards Terms*, Seventh Edition.

[B3] IEEE Std 1232-1995, IEEE Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments: Overview and Architecture.

[B4] IEEE Std 1232.1-1997, IEEE Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments: Data and Knowledge Specification.

[B5] IEEE Std 1232.2-1998, IEEE Trial-Use Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments: Service Specification.

[B6] Sheppard, John W. and Simpson, William R., "A mathematical model for integrated diagnostics," *IEEE Design and Test of Computers*, vol. 8, no. 4, pp. 25–38, Dec. 1991.

[B7] Sheppard, John W. and Simpson, William R., *Research Perspectives and Case Studies in System Test and Diagnosis*, Norwell, MA: Kluwer Academic, 1998.

[B8] Simpson, William R. and Sheppard, John W., *System Test and Diagnosis*, Norwell, MA: Kluwer Academic, 1994.

[B9] IEEE Std 716-1995, IEEE Standard Test Language for All Systems—Common/Abbreviated Test Language for All Systems (C/ATLAS).

For further reading

[B10] Barr, Avron, Feigenbaum, Edward A., and Cohen, Paul, *The Handbook of Artificial Intelligence*, vols. 1–3. Stanford, CA: Heuristech Press, 1981–1982.

[B11] Barr, Avron, Cohen, Paul, and Feigenbaum, Edward A., *The Handbook of Artificial Intelligence*, vol. 4. Reading, MA: Addison-Wesley, 1989.

[B12] Cantone, R., Pipitone, F., Lander, W., and Marrone, M., "Model-based reasoning for electronics troubleshooting," *Eighth International Joint Conference on AI*, pp. 207–211, 1983.

[B13] Charles Stark Draper Laboratory, *Modular Avionics Handbook, Vol. 5. Test and Diagnostics*. Cambridge, MA: Draper Laboratories, 1989.

[B14] deKleer, Johan, and Williams, Brian C., "Reasoning about multiple faults," *Proceedings of the National Conference on Artificial Intelligence*, Philadelphia, PA, pp. 132–145, 1986.

[B15] DePaul, R. A., Jr., "Logic modeling as a tool for testability," *Proceedings of the IEEE AUTOTESTCON*, New York: IEEE Press, 1985, pp. 203–207.

[B16] ANSI/IEEE Std 260.1™-1993, American National Standard Letter Symbols for Units of Measurement (SI Units, Customary Inch-Pound Units, and Certain Other Units).

[B17] Keiner, W., ''A navy approach to integrated diagnostics,'' *Proceedings of the IEEE AUTOTESTCON*, New York: IEEE Press, 1990, pp. 129–132.

[B18] Laffey, Thomas J., Perkins, Watton A., and Nguyen, Tin A., ''Reasoning about fault diagnosis with LES,'' *IEEE Expert*, Spring 1986, pp. 13–20.

[B19] Maguire, R. J. and Sheppard, J. W., ''Application scenarios for AI-ESTATE services,'' *Proceedings of the IEEE AUTOTESTCON*, New York: IEEE Press, 1996, pp. 68–72.

[B20] Pattipati, K. and Alexandridis, M., ''Application of heuristic search and information theory to sequential fault diagnosis,'' *IEEE Transactions on System, Man and Cybernetics*, vol. 20, no. 4, pp. 872–887, 1990.

[B21] Peng, Y. and Reggia, J., *Abductive Inference Models for Diagnostic Problem Solving*, New York: Springer-Verlag, 1990.

[B22] Pipitone, DeJong K. and Spears, W., ''An artificial intelligence approach to analog systems diagnosis,'' *Testing and Diagnosis of Analog Circuits and Systems*, R. Liu, ed. New York: Van Nostrand Reinhold, pp. 187–216, 1991.

[B23] Reiter, R., ''A theory of diagnosis from first principles,'' *Artificial Intelligence*, vol. 32, pp. 57–95, 1987.

[B24] Russell, Stuart and Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Upper Saddle River, NJ: Prentice-Hall, 1995.

[B25] Schenk, Douglas A. and Wilson, Peter R., *Information Modeling: The EXPRESS Way*, New York: Oxford University Press, 1994.

[B26] Sheppard, J. W. and Orlidge, L. A., ''Artificial intelligence exchange and service tie to all test environments (AI-ESTATE)—a new standard for system diagnostics,'' *Proceedings of the 1997 International Test Conference*, Los Alamitos, CA: IEEE Computer Society Press, 1997, pp. 1020–1029.

[B27] Sheppard, J. W., Bartolini, A., and Orlidge, L. A., ''Standardizing diagnostic information using IEEE AI-ESTATE,'' *Proceedings of the IEEE AUTOTESTCON*, New York: IEEE Press, pp. 82–87.

[B28] Simpson, W. and Sheppard, J., ''Performing effective fault isolation in integrated diagnostics,'' *IEEE Design and Test of Computers*, vol. 10, no. 2, 78–90, Mar. 1993.

Annex B

(informative)

Overview of EXPRESS

The models defined in this standard use the ISO 10303-11: 1994 EXPRESS language and the supporting EXPRESS-G graphical notation for their specification. To quote from ISO 10303-11: 1994, "*EXPRESS* is the name of the formal information modeling language used to specify the information requirements of other parts of this International Standard. . .. The language focuses on the definition of *entities*, which are the things of interest. The definition of entities is in terms of data and behavior. Data represents the properties by which an entity is realized and behavior is represented by constraints."

Within EXPRESS, models are defined using a simple hierarchy partitioned along *schemata*, *entities*, and *attributes*. Further, 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. Further, EXPRESS is not intended to be used as a programming language, since it contains no facilities for input/output, exception handling, or information processing.

This standard is intended to define an exchange format for models used in diagnostic systems. The standard uses EXPRESS to define the models, but these models are not the exchange format. Since EXPRESS is not intended to define exchange formats, an alternative representation shall be used for the actual format. ISO 10303-21:2002 de.nes an ASCII format for instantiations of EXPRESS models and can be used as an exchange format. This format is directly derivable from the EXPRESS models; therefore, it provides a natural exchange format to be specified by this standard. Since the actual format is derived from the EXPRESS, only the EXPRESS needs to be specified in this standard.

In B.1 through B.8, the major elements of an EXPRESS model are described. When available, the corresponding representation of the element in EXPRESS-G is also provided.

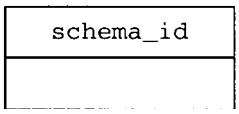
B.1 Schema

A *schema* is defined to be a collection of items forming part or all of a model. Within AI-ESTATE, a schema has been defined for the Common Element Model and for each model used in a particular approach to diagnosis (i.e., Fault Tree Model and Enhanced Diagnostic Inference Model).

The syntax of a schema definition consists of

```
SCHEMA schema_id ';' schema_body END_SCHEMA ';'
```

and a schema is represented in EXPRESS-G as



B.2 Entity

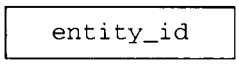
An *entity* is defined to be a type which represents information for processing purposes, based on explicit or implicit agreements about the meaning of the data. Within each schema of AI-ESTATE, the

primary data types are defined as entities, many of which have identifiers as indicated by `description` in the entity definition.

The syntax of an entity definition consists of

```
ENTITY entity_id [supertype] [subtype] ';' entity_body END_ENTITY ';'
```

and an entity is represented in EXPRESS-G as



B.3 Attribute

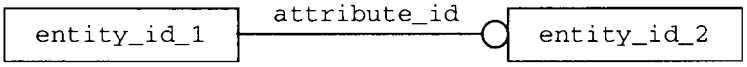
An *attribute* is defined to be a trait, quality, or property that is a characteristic of an entity. Attributes provide the primary elements of the definition of the entity body. Since entities are type definitions, attributes are frequently of types as defined by other entities.

Attributes can be optional or required. Frequently, attributes in AI-ESTATE are defined to be sets, and many of these sets have minimum cardinality of zero. An attribute that is optional is intended to be different from an attribute with cardinality of zero. For an optional attribute, an instantiated model may or may not include that attribute. If an attribute is required but may have a cardinality of zero, then a placeholder for that attribute shall be included in the instantiation even though no value is assigned.

The syntax of a simple attribute definition consists of

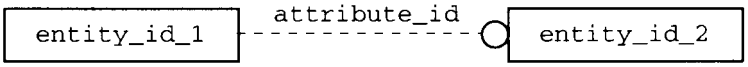
```
attribute_id ':' [OPTIONAL] base_type ';'
```

Required attributes are defined between entities or an entity and a type and are represented in EXPRESS-G as



In this case, `entity_id_1` has attribute `attribute_id`, which has type `entity_id_2`. The circle on the line can be treated as an "arrow-head," which determines the direction of the relationship between the two entities.

Similarly, optional attributes are defined between entities or an entity and a type and are represented in EXPRESS-G as



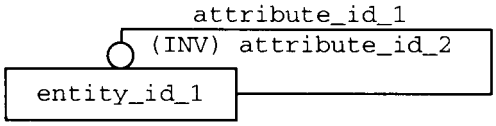
Finally, attributes can be defined to have an inverse relationship in which the named attribute (e.g., `entity_id_2` above) is associated with the declared entity (e.g., `entity_id_1` above). For example, within AI-ESTATE, several entities of the Common Element Model are defined to provide a lattice structure among entities of the same type. The attributes are defined to point to children in the lattice, and the parents are defined to be the inverse. Within the EXPRESS specification, an inverse

relationship is identified with the INVERSE keyword, and attributes following INVERSE define the inverse attribute.

The syntax for an inverse attribute consists of

```
attribute_id ':' [ SET [ '[' bound_1 ':' bound_2 ']' ] OF ] entity_id FOR
  attribute_id ';'
```

An example illustrating the use of inverse attributes can be represented in EXPRESS-G as



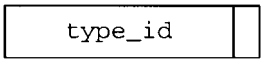
B.4 Type definition

A *type* is defined to be a representation of a domain of valid values. As discussed in B.2, entities are types corresponding to some collection of objects having common properties. At times, simpler types may need to be defined that are not included in the set of EXPRESS base types. Such types can be defined using existing base types or previously defined derived types.

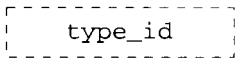
The syntax of a type definition consists of

```
TYPE type_id '=' defined_type ';'
```

A base type is represented in EXPRESS-G as



A defined data type is represented in EXPRESS-G as

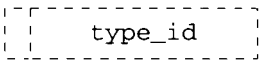


A select data type is a type consisting of a collection of other types in which an instantiation is of *one* of the listed types.

The syntax of a select type is

```
TYPE type_id '=' SELECT '(' defined_type { ',' defined_type } ')' ';'
```

and is represented in EXPRESS-G as

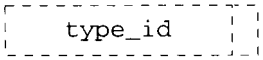


An enumeration data type is a type consisting of an ordered set of values represented by names. An instantiation of an enumeration type shall take on one of the specified values.

The syntax of an enumeration type is

```
TYPE type_id '=' ENUMERATION OF '(' enumeration_id { ','
  enumeration_id } ')' ';'
```

and is represented in EXPRESS-G as



B.5 Subtypes/supertypes

Within EXPRESS, subtypes and supertypes can be specified to define a classification structure for types. According to ISO 10303-11:1994, "A subtype is a more specific type than its supertype(s). A supertype is a more general type than its subtype(s). Since the subtype is a more specific kind of its supertype, every instance of a subtype is an instance of its supertype(s)." Because of this fact, all attributes of a supertype are inherited by its subtype. EXPRESS concepts of supertype and subtype, taken together, allow a type lattice to be constructed. A subtype/supertype relationship is typically called an "as-is" relationship in data modeling terms, i.e., a subtype is a "kind" of its supertype.

In defining supertype/subtype relationships between types, the subtype shall declare itself to be a subtype of some other type, but the supertype is not required to be declared as a supertype. Nevertheless, it is preferable for the supertype to be declared as such.

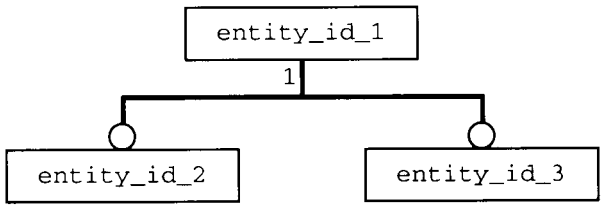
The syntax for a subtype consists of

```
subtype = SUBTYPE OF '(' entity_id { ',' entity_id } ')'
```

The syntax for a supertype consists of

```
supertype = [ABSTRACT] SUPERTYPE OF '(' supertype_expression ')'
supertype_expression = supertype_factor { (AND | ANDOR) supertype_factor }
supertype_factor = entity_id | one_of | '(' supertype_expression ')'
one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')'
```

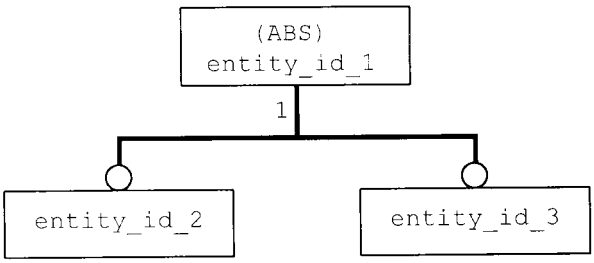
Subtypes and supertypes are defined between entities and are represented in EXPRESS-G as



In this example, the supertype is defined to be associated with one of the underlying subtypes. In other words, an instantiation of the supertype shall be either of subtype `entity_id_2` or of `entity_id_3` but not both.

A supertype can be defined to be "abstract" when instantiation of the supertype requires instantiation of a subtype as well. If the supertype is not abstract, it can be instantiated without any of its subtypes.

In EXPRESS-G, an abstract supertype is identified as



B.6 External schema references

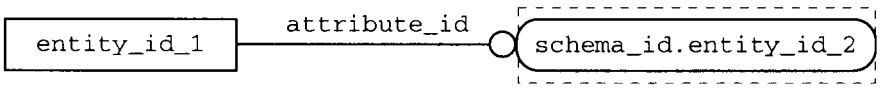
Entities declared in one schema can be referenced by another schema using schema interface specifications. Two types of interface specifications are possible—the use clause and the reference clause. The *use clause* identifies an entity in an external schema and treats that entity as if it is local to the current schema. The *reference clause*, on the other hand, identifies an entity in an external schema and treats the entity as an external entity in which remote access is allowed. Currently, the AI-ESTATE standard only uses reference clauses.

The purpose of defining the Common Element Model was to provide a means for specifying common data types across classes of diagnostic models. Each of these diagnostic models would then reference an instantiation of the Common Element Model to obtain access to the required elements of this model.

The syntax for the reference clause consists of

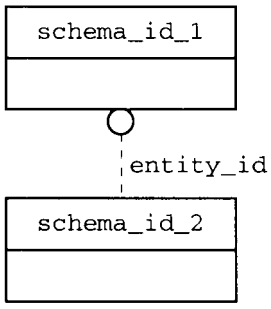
```
REFERENCE FROM schema_id [ '(' model_id { ',' model_id } ')' ] ';'
```

and is represented in EXPRESS-G as



where `attribute_id of entity_id_1` references `entity_id_2` within the external `schema schema_id`.

External references are also shown on a schema-level diagram as follows:



where `entity_id` is referenced from `schema_id_2` by `schema_id_1`.

B.7 Constraints and WHERE clauses

Attributes of an entity can be constrained to take on values within a defined domain. When such a situation occurs, the entity definitions shall include local rules to specify the appropriate constraint. Within AI-ESTATE, only uniqueness constraints and WHERE constraints are used.

A uniqueness constraint is identified in an entity definition with the keyword UNIQUE. Attribute identifiers listed following the UNIQUE keyword are constrained to take on unique values within an instantiation of the schema. In EXPRESS-G, uniqueness constraints are identified on lines corresponding to unique attributes with an asterisk ('*').

WHERE clauses specify the conditions that constrain the values of attributes for every instance of the entity. WHERE clauses are identified with the keyword WHERE, and the actual rules follow the keyword.

The syntax of a where clause consists of

```
[ label ':' ] expression ';'
```

The WHERE clause evaluates to true or false and only references attributes of the entity within which the WHERE clause appears. The constraints imposed on attributes of an entity within a WHERE clause shall hold for all instances of that entity. WHERE clauses cannot be represented in EXPRESS-G.

A few examples should help clarify the intent of the WHERE clause.

Example: In the `outcome` entity of the `common_element_model`, the following WHERE clause is defined:

```
WHERE
  range : (0.0 <= SELF.confidence <= 1.0);
```

This WHERE clause constrains the `confidence` attribute of the `outcome` entity to be in the range 0.0–1.0.

Example: In the `test_result` entity of the `fault_tree_model`, the following WHERE clause is defined:

```
WHERE
  leaves_have_diagnoses : (EXISTS(SELF.next_step)) OR
    (SIZEOF(SELF.current_diagnosis) > 0);
```

This rule determines whether or not another step in the fault tree is associated with this current `test_result`, as well as whether or not the set of `current_diagnosis` tied to this `test_result` is empty. The constraint permits `current_diagnosis` to be nonempty at a `test_result` that is in the interior of the tree, but *requires* `current_diagnosis` to be nonempty if `test_result` is at a leaf of the tree. If `(EXISTS(SELF.next_step))` is false, then the `test_result` is a leaf. If `(SIZEOF(SELF.current_diagnosis) > 0)` is false, then no diagnosis is associated with the tree. If both are false, then the constraint is violated.

B.8 Functions and procedures

As shown in B.7, EXPRESS provides a very rich language for defining constraints among entities in a model. One tool to assist in defining constraints is the function. In EXPRESS, functions and

procedures are as robust as most programming languages, except that they do not provide facilities for input/output or exception handling.

The syntax for a function definition consists of

```
function_block = function { statement } END_FUNCTION ';'
function = FUNCTION function_id ['(' parameter {';' parameter} ')']':'
parameter_type';'
function_id = simple_id
parameter = simple_id { ',' simple_id } ':' parameter_type
```

The syntax for a procedure definition consists of

```
procedure_block = procedure { statement } END_PROCEDURE ';'
procedure = PROCEDURE procedure_id ['(' parameter {';' parameter} ')' ] ';'
procedure_id = simple_id
parameter = simple_id { ',' simple_id } ':' parameter_type
```

A function or procedure can define a local set of variables (i.e., variables visible only within the scope of the function or procedure). The function shall return a value, and that value shall be of the type specified in the function header. Functions and procedures have no representation in EXPRESS-G.

Annex C

(informative)

List of Participants

At the time that this standard was completed, the Diagnostic and Maintenance Control (D & MC) Subcommittee had the following membership:

John W. Sheppard, *Co-Chair*

Mark A. Kaufman, *Co-Chair*

Timothy M. Bearse
Gregory P. Bowman
Antonius Bunsen
Bernard Dathy
Somnath Deb
Bernd Dinklage
David B. Droste
Sudipto Ghoshal
Amanda Jane Giarla
Eric S. Gould
Arnold M. Greenspan
David L. Kleinman
Franc Novak
Leslie A. Orlidge
Jean Pouilly
William C. Rodriguez
William R. Simpson
Philip T. Smith
Li Pi Su
Jack Taylor
Joerg H. Urban
J. Richard Weger
Timothy J. Wilmering

In addition, these individuals have contributed review and comments:

Antony Bartolini
Larry Howard
Michael Lynch
William Simerly
Lee A. Shombert

The following members of the balloting group voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Guy Adam
Gregory P. Bowman
Joseph F. Buch
Eric M. Bukata
Danny C. Davis
Somnath Deb
Guru Dutt Dhingra
William Brit Frank
Sudipto Ghoshal
Amanda Jane Giarla
Eric S. Gould
Osamu Karatsu
Mark A. Kaufman
Frank X. Kearns
Leslie A. Orlidge
Jean Pouilly
Narayanan Ramachandran
David E. Rolince
Eric Sacher
John W. Sheppard
Lee A. Shombert
William R. Simpson
Philip T. Smith
Joseph J. Stanco
William John Taylor
Kirk D. Thompson
Timothy J. Wilmering
Jehuda Ziegler

When the IEEE-SA Standards Board approved this standard on 13 June 2002, it had the following membership:

James T. Carlo, *Chair*

James H. Gurney, *Vice Chair*

Judith Gorman, *Secretary*

Sid Bennett	Arnold M. Greenspan	Peter H. Lips
H. Stephen Berger	James H. Gurney	Nader Mehravari
Clyde R. Camp	Raymond Hapeman	Daleep C. Mohla
Richard DeBlasio	Donald M. Heirman	William J. Moylan
Harold E. Epstein	Richard H. Hulett	Malcolm V. Thaden
Julian Forster*	Lowell G. Johnson	Geoffrey O. Thompson
Howard M. Frazier	Joseph L. Koepfinger*	Howard L. Wolfman
Toshio Fukuda		Don Wright

*Member Emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Alan H. Cookson, *NIST Representative*
Satish K. Aggarwal, *NRC Representative*

Savoula Amanatidis,
IEEE Standards Managing Editor

Standards Survey

The IEC would like to offer you the best quality standards possible. To make sure that we continue to meet your needs, your feedback is essential. Would you please take a minute to answer the questions overleaf and fax them to us at +41 22 919 03 00 or mail them to the address below. Thank you!

Customer Service Centre (CSC)

International Electrotechnical Commission
3, rue de Varembé
1211 Genève 20
Switzerland

or

Fax to: **IEC**/CSC at +41 22 919 03 00

Thank you for your contribution to the standards-making process.

Nicht frankieren
Ne pas affranchir

Non affrancare
No stamp required

A Prioritaire

RÉPONSE PAYÉE

SUISSE

Customer Service Centre (CSC)
International Electrotechnical Commission
3, rue de Varembé
1211 GENEVA 20
Switzerland

Q1 Please report on **ONE STANDARD** and **ONE STANDARD ONLY**. Enter the exact number of the standard: *(e.g. 60601-1-1)*

.....

Q2 Please tell us in what capacity(ies) you bought the standard *(tick all that apply).* I am the/a:

purchasing agent ❑
librarian ❑
researcher ❑
design engineer ❑
safety engineer ❑
testing engineer ❑
marketing specialist ❑
other.....

Q3 I work for/in/as a: *(tick all that apply)*

manufacturing ❑
consultant ❑
government ❑
test/certification facility ❑
public utility ❑
education ❑
military ❑
other.....

Q4 This standard will be used for: *(tick all that apply)*

general reference ❑
product research ❑
product design/development ❑
specifications ❑
tenders ❑
quality assessment ❑
certification ❑
technical documentation ❑
thesis ❑
manufacturing ❑
other.....

Q5 This standard meets my needs: *(tick one)*

not at all ❑
nearly ❑
fairly well ❑
exactly ❑

Q6 If you ticked NOT AT ALL in Question 5 the reason is: *(tick all that apply)*

standard is out of date ❑
standard is incomplete ❑
standard is too academic ❑
standard is too superficial ❑
title is misleading ❑
I made the wrong choice ❑
other .....

Q7 Please assess the standard in the following categories, using the numbers:
(1) unacceptable,
(2) below average,
(3) average,
(4) above average,
(5) exceptional,
(6) not applicable

timeliness.....
quality of writing.....
technical contents.....
logic of arrangement of contents .....
tables, charts, graphs, figures.....
other .....

Q8 I read/use the: *(tick one)*

French text only ❑
English text only ❑
both English and French texts ❑

Q9 Please share any comment on any aspect of the IEC that you would like us to know:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

ISBN 2-8318-8045-9

CODE PRIX XF

For price, see current catalogue

ICS 35.240, 35.160

Typeset and printed by the IEC Central Office
GENEVA, SWITZERLAND