

IEEE Std 1232.2-1998
(Issued February 1999 for trial use)

IEEE Std 1232.2-1998

IEEE Trial-Use Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE): Service Specification

IEEE Standards Coordinating Committee 20

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

19 February 1999

SH94678

IEEE Trial-Use Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE): Service Specification

Sponsor

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

Approved 16 September 1998

IEEE-SA Standards Board

Abstract: Formal software interfaces to system diagnosis tools and applications are defined. As part of the AI-ESTATE set of standards, this standard defines services to manipulate information models as defined in IEEE Std 1232.1-1997 and to control a diagnostic reasoner. This standard includes a new information model to manipulate dynamic information obtained during the process of system diagnosis. Service bindings to ANSI C and ANSI Ada are also provided.

Keywords: AI-ESTATE, artificial intelligence, diagnosis, dynamic context model, system test

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

Copyright © 1999 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 19 February 1999. Printed in the United States of America.

Print: ISBN 0-7381-1427-8 SH94678
PDF: ISBN 0-7381-1428-6 SS94678

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

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

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

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

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 all 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 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

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.

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; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

[This introduction is not part of IEEE Std 1232.2-1998, IEEE Trial-Use Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE); Service Specification.]

The set of AI-ESTATE (Artificial Intelligence Exchange and Service Tie to All Test Environments) standards is intended to provide 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 being defined whereby diagnostic tools can be developed to process the diagnostic information in a consistent and reliable way.

This component of the AI-ESTATE standards defines the formal software interface to a diagnostic reasoner. It is not the intent of this standard to define the implementation of the reasoner. Nor is it the intent to define the information needed for testing systems. Such standards exist (e.g., IEEE Std 716-1995 for ATLAS, IEEE Std 1226-1993 for ABBET, and IEEE Std 1149.1-1990 for boundary scan) and can be used in conjunction with this standard. This component uses the information defined in IEEE Std 1232.1-1997 to define key services provided by a diagnostic system.

Specifically, this standard includes service definitions for traversing and managing the models defined in IEEE Std 1232.1-1997. It also includes a new information model, defined in EXPRESS (ISO 10303-11-94), that is used to define the state of the reasoner to facilitate control of the dynamic aspects of diagnosis.

This standard is organized with a model in the body defining the dynamic context of diagnosis, followed by definitions of the software services in EXPRESS. Several normative annexes are included that define reasoning-specific services and bindings of the services to specific programming languages. Currently, normative annexes have been defined for a Fault Tree Model (FTM) (i.e., a fixed decision tree based on identified test outcomes) and the Enhanced Diagnostic Inference Model (EDIM) (i.e., a propositional model tying test outcomes to possible diagnoses). Both types of models are accepted in system diagnoses and, as a result, are in wide use in industry today. It is anticipated that additional models will be defined in the future, perhaps addressing concerns of diagnostic constraints, diagnostic rules, and diagnostic neural networks. In all cases, the underlying theme will be diagnosis.

Publication of this trial-use standard for comment and criticism has been approved by the Institute of Electrical and Electronics Engineers. Trial-use standards are effective for 24 months from the date of publication. Comments for revision will be accepted for 18 months after publication. Suggestions for revision should be directed to the Secretary, IEEE-SA Standards Board, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331 USA, and should be received no later than 19 August 2000. It is expected that following the 24-month period, this trial-use recommended practice, revised as necessary, shall be submitted to the IEEE-SA Standards Board for approval as a full-use recommended practice.

Participants

At the time that this standard was completed, the AI-ESTATE subcommittee has the following membership:

Timothy Bearse, *Service Working Group, Co-Chair*
Timothy Wilmering, *Service Working Group, Co-Chair*
Leslie Orlidge, *AI-ESTATE, Co-Chair*
John Sheppard, *AI-ESTATE, Co-Chair*

Guy Adam	Gary Hardenberg	Jean Pouilly
James D. Baker	Leonard Haynes	N. Ramachandran
Anthony Bartolini	John Heiser	William E. Reeves
Darrell Bartz	Shankar Hemmady	Bill Rodriguez
Michael Blair	Mark Kaufmann	David Roggendorf
Gregory Bowman	Richard Maguire	Howard Savage
Elaine Cashar	Gayle Matysek	Michael Seavey
Alexander Cheras	Robert McGarvey	Lee A. Shombert
Richard Churcher	Harry McGuckin	William R. Simpson
Bernard Dathy	Scott Miller	Butch Sneade
Danny C. Davis	James Mills	Joseph J. Stanco
Bernd Dinklage	Mukund Modi	William Taylor
Bernard Dugas	Dave Nichols	Jörg H. Urban
Arnold Greenspan	Hans Obermeier	Richard J. Weger
Wes Hand	Edward G. Perkins	Daniel Weiss
Arthur Hann		Will Young

Other individuals who have contributed review and comments are:

Somnath Deb	Sharon Goodall	Glenn Wright
	Robert Hayes	

The following persons were on the balloting committee:

Guy Adam	William Brit Frank	John W. Sheppard
James D. Baker	Arthur Hann	Lee A. Shombert
Timothy Bearse	Robert G. Hayes	William R. Simpson
Gregory Bowman	Ashley M. B. Hulme	Butch Sneade
Antonius Bunsen	Gayle Matysek	Joseph J. Stanco
Alexander Cheras	Leslie A. Orlidge	Kirk D. Thompson
Gregory W. Darnell	Edward G. Perkins	Jörg H. Urban
Bernard Dathy	N. Ramachandran	Richard J. Weger
Danny C. Davis	William E. Reeves	Timothy J. Wilmering
Somnath Deb	David Roggendorf	Glenn Wright
Bernd Dinklage	Eric Sacher	Jehuda Ziegler

When the IEEE-SA Standards Board approved this standard on 16 September 1998, it had the following membership:

Richard J. Holleman , <i>Chair</i>	Donald N. Heirman , <i>Vice Chair</i>	
	Judith Gorman , <i>Secretary</i>	
Satish K. Aggarwal	James H. Gurney	L. Bruce McClung
Clyde R. Camp	Jim D. Isaak	Louis-François Pau
James T. Carlo	Lowell G. Johnson	Ronald C. Petersen
Gary R. Engmann	Robert Kennelly	Gerald H. Peterson
Harold E. Epstein	E. G. "Al" Kiener	John B. Posey
Jay Forster*	Joseph L. Koepfinger*	Gary S. Robinson
Thomas F. Garrity	Stephen R. Lambert	Hans E. Weinrich
Ruben D. Garzon	Jim Logothetis	Donald W. Zipse
	Donald C. Loughry	

*Member Emeritus

Contents

1. Overview	1
1.1 Scope.....	2
1.2 Purpose.....	2
1.3 Application.....	2
1.4 Conventions used in this document	3
2. References.....	3
3. Definitions, acronyms, and abbreviations.....	4
3.1 Definitions.....	4
3.2 Acronyms and abbreviations.....	4
4. Description	5
4.1 Objectives	5
4.2 Encapsulation of services.....	5
4.3 Relationship to AI-ESTATE architecture	5
4.4 Extensibility	5
5. DCM specification	6
5.1 Overview	6
5.2 DCM	6
6. Services specification.....	31
6.1 Overview	31
6.2 Status codes.....	32
6.3 DCM EXPRESS service specification.....	33
6.4 CEM EXPRESS service specification	46
6.5 Control and reporting services	63
7. Conformance	64
Annex A (informative) Bibliography	65
Annex B (informative) Overview of EXPRESS	68
Annex C (normative) AI-ESTATE fault tree reasoner services	75
Annex D (normative) EDIM services.....	79
Annex E (normative) C language binding	85
Annex F (normative) Ada language binding	116

IEEE Trial-Use Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE): Service Specification

1. Overview

This standard defines a set of formal specifications for encapsulated services to be provided by diagnostic reasoners to their clients. This standard is one component in a set of AI-ESTATE standards. The service specifications will provide for interchangeability of AI-ESTATE– conformant reasoning systems within the AI-ESTATE architectural framework. Thus, services shall be provided by a reasoner to the human presentation system, the maintenance data/knowledge collection system, other reasoners, and the test controller(s). The reasoner will use services by these other functional elements as required. The services to be provided by these other functional elements will not be specified by the set of AI-ESTATE standards.

The goals of this component of the set of AI-ESTATE standards, formally defined in IEEE Std 1232-1995¹, are summarized as follows:

- Incorporate domain-specific terminology;
- Facilitate portability;
- Facilitate interchangeability.

This standard shall use the vocabulary and semantics of the test and diagnosis domain by incorporating terminology from Clause 3 of IEEE Std 1232-1995, IEEE Std 1232.1-1997, and other existing standards as applicable. Deviations shall be kept to a minimum and shall be explicitly defined to the user at first occurrence.

The specifications in this standard shall be fully portable. No host computer dependence shall be contained in the AI-ESTATE component standards. Host system dependence shall not be included in the AI-ESTATE service implementations.

¹Information on references can be found in Clause 2.

Specifications will ensure the interchangeability of diagnostic reasoners through the definition of encapsulated services. These services 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 scope of this standard is to define the services needed to support AI-ESTATE. It provides a formal definition of the encapsulated services that an AI-ESTATE-conformant diagnostic reasoner shall provide to the other elements of the architecture constituting an AI-ESTATE-conformant system. The services defined in this standard are applicable to the domain of system test and diagnosis, independent of the reasoning approach.

1.2 Purpose

The purpose of this standard is to provide the formal specifications of the services, transactions, and protocols necessary to insure the interoperability of AI-ESTATE-compliant systems. It provides a formal specification of the encapsulated services required of a diagnostic reasoner, which directs system test and diagnosis, and ensures the interchangeability of diagnostic reasoners within AI-ESTATE-conformant systems. This standard will specify the services required of a diagnostic reasoner by the other architectural components of an AI-ESTATE-conformant system.

1.3 Application

This standard defines the encapsulated services to be provided by a diagnostic reasoner in an AI-ESTATE-conformant implementation. 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). One view of a test environment is depicted in Figure 1. As can be seen in Figure 1, each of the elements that interface with the reasoner will interact through the application executive and will provide its own set of encapsulated services to its respective clients, but those service definitions are beyond the scope of this document.

The definition of these services was performed by first creating accessor services for the Common Element Model (CEM) as defined in IEEE Std 1232.1-1997. As the specification of these services became more clear, it became apparent that to adequately define useful services for an active diagnostic reasoner it would be necessary to define new entities that represent the context within which reasoning is performed. Thus evolved the Dynamic Context Model (DCM) as specified in Clause 5 of this document. Services for the DCM and CEM are then specified in Clause 6.

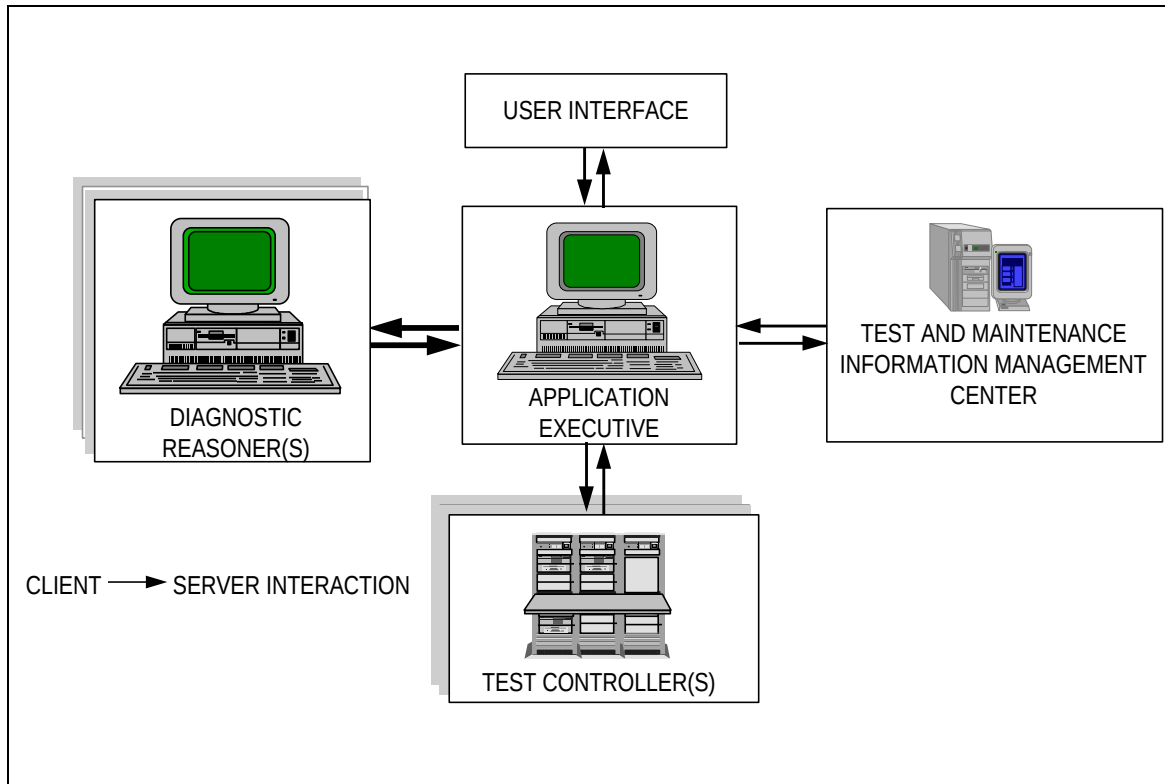


Figure 1—A client-server view of a test environment

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

1.4 Conventions used in this document

Clauses 5 and 6 and Annexes C and D present 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. Service typing conventions follow the typing conventions employed in IEEE 1232.1-1997 and Clause 5 of this document except where otherwise noted.

2. References

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

IEEE Std 100-1996, IEEE Standard Dictionary of Electrical and Electronics Terms.²

²IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (<http://standards.ieee.org>).

IEEE Std 1232-1995, IEEE Trial-Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Overview and Architecture.

IEEE Std 1232.1-1997, IEEE Trial-Use Standard for Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE): Data and Knowledge Specification.

ISO 10303-1:1994, Industrial automation systems and integration—Product data representation and exchange—Part 1: Overview and fundamental principles.³

ISO 10303-11:1994, Industrial automation systems and integration—Product data representation and exchange—Part 11: Description methods: The EXPRESS language reference manual.

ISO/IEC 8652:1995, Information technology—Programming languages—Ada.

ISO/IEC 9899:1990, Programming languages—C.

3. Definitions, acronyms, and abbreviations

3.1 Definitions

Clause 3 of IEEE Std 1232-1995 contains a complete glossary with definitions of all of the terms used in the set of AI-ESTATE standards and guides. A clear understanding of the following terms with respect to AI-ESTATE, together with the terms defined in IEEE Std 1232-1995 and IEEE Std 1232.1-1997, is particularly important in order to understand this document.

3.1.1 fault isolation: The process of reducing the number of anomalies that constitute a diagnosis; identification of an anomaly or anomalies to a degree sufficient to undertake an appropriate corrective action.

3.1.2 higher order service: A service that provides a complex behavior of a diagnostic reasoner, possibly defined using a combination of primitive services.

3.1.3 primitive service: A service that accesses or manipulates defined elements of AI-ESTATE—conformant models (e.g., get, put, create, delete).

3.1.4 service: An action or response initiated by a process (i.e., a server) at the request of some other process (i.e., a client).

3.2 Acronyms and abbreviations

The following acronyms and abbreviations are used in this standard:

AI-ESTATE	Artificial Intelligence Exchange and Service Tie to All Test Environments
CEM	Common Element Model
DCM	Dynamic Context Model
EDIM	Enhanced Diagnostic Inference Model
FTM	Fault Tree Model

³ISO and ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse. They 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.

4. Description

4.1 Objectives

The objectives of this standard, as stated, are to provide a formal specification of the encapsulated services required of a diagnostic reasoner, which directs system test and diagnosis, and to ensure the interchangeability of diagnostic reasoners within AI-ESTATE-conformant systems. Any diagnostic reasoner that implements the services defined in this standard will be encapsulated in the sense that the reasoner's specific implementation details will be hidden from the clients of the system that request its services.

4.2 Encapsulation of services

This component of the set of AI-ESTATE standards provides the encapsulated service definitions required by all other components of the AI-ESTATE standards. Services have been organized by entity for each type of diagnostic model that they address. Other services will likely be defined in the future as new paradigms for system test and diagnosis are introduced.

Encapsulation reduces the apparent complexity of systems. In the object-oriented paradigm, encapsulation dictates that the data structure that constitutes an object and the corresponding methods that act upon the object are both hidden from the client issuing the request for service. The only process apparent to the client is that of issuing a service request and receiving an understandable response. The approach taken in this standard is at a somewhat higher level of abstraction. It places no burden on the server about how the implementation accomplishes encapsulation. This specification dictates that the server will be given a request with any requisite data and will return an expected well-formed value. The only further requirement on the services is the strong typing dictated by the models of IEEE Std 1232.1-1997 and Clause 5 of this document.

The approach taken in this document is primarily one of providing primitive services to build a solid foundation based on first principles of diagnostic reasoning. These primitive services can be combined into higher order services within the client system. As experience in applying the standard is accumulated, more layers composed of higher order services will likely be added.

The services will together constitute a logical layer between a reasoner and the other elements of an AI-ESTATE-conformant system. This layer will make it possible for a reasoner to be interchanged with any other reasoner that is similarly encapsulated. The interchange of such reasoners will be transparent to the other elements of the system that request reasoner services.

4.3 Relationship to AI-ESTATE architecture

This standard defines the services that will be required by the other components of the AI-ESTATE architecture, which are defined in IEEE Std 1232-1995.

4.4 Extensibility

As it is the intention of this specification to define the services that together constitute the interface to a diagnostic reasoner, no extensions are permitted to this specification.

5. DCM specification

5.1 Overview

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 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 CEM, Fault Tree Model (FTM), 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 step:

- The model(s) currently in use;
- The status of all resources associated with the active models;
- The status of all tests associated with the active models;
- The status of all diagnoses associated with the active models;
- Optionally, the current fault hypothesis.

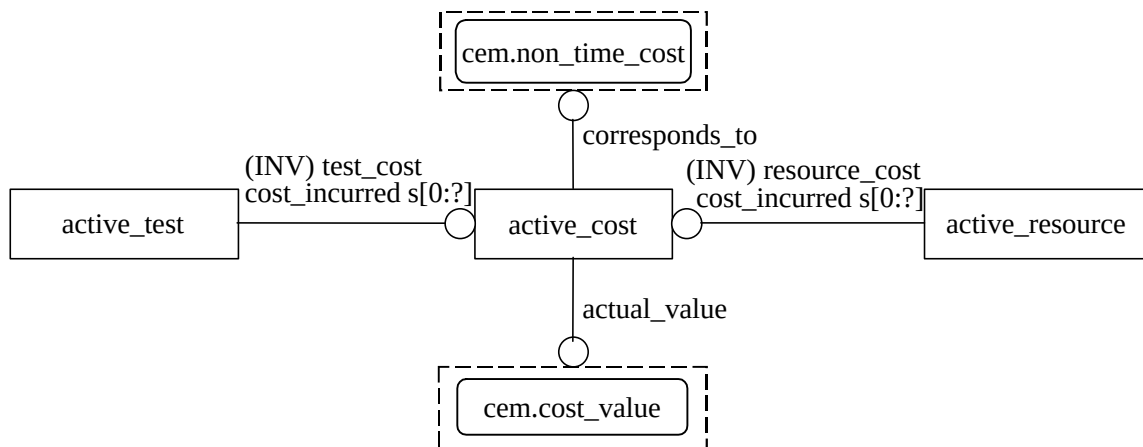
The DCM is also used to record the tests performed at each step and any associated actual test outcome and outcome confidence values. The reasoner makes diagnostic and test inferences from the actual test outcome and associated confidence values.

5.2 DCM

The entities of the AI-ESTATE DCM are defined in this subclause. For each entity of the model, the EXPRESS-G view is shown, followed by the entity's description, its EXPRESS specification, and attribute definitions. Following the entity subclauses, the model constraint functions are defined. The complete EXPRESS specification is then shown in its entirety. The EXPRESS-G representation of the DCM is shown below. The constructs of this model are the dynamic components of the AI-ESTATE domain of equipment test and diagnosis. They are constructed from the basic units of the static data and knowledge specification components of the AI-ESTATE architecture. References to external models, as shown in the EXPRESS-G, correspond to the AI-ESTATE CEM and the AI-ESTATE EDIM.

The diagram illustrates a Goal Structing Notation (GSN) for a diagnostic system. It features several entities (dashed boxes) and their relationships (solid lines with labels). Key entities include `session`, `step`, `active_resource`, `active_model`, `active_test`, `cem.diagnostic_knowledge`, `cem.name_type`, `cem.test_cost`, `cem.test`, `cem.assertion`, `cem.confidence_value`, `cem.outcome`, `cem.cost_value`, `cem.non_time_cost`, `cem.confidence_value`, and `cem.outcome`. Relationships include `knowledge_base`, `corresponds_to`, `pathname`, `trace`, `(INV) part_of`, `model_status`, `test_status`, `(INV) associated_step`, `diagnoses`, `hypothesis`, `state`, `confidence`, `inferred_confidence`, `inferred_outcome`, `availability`, `actual_value`, `resources_used`, `time_incurred`, `cost_incurred`, and `actual_confidence`. The diagram is organized into several interconnected components, with some entities having multiple instances or states indicated by brackets and subscripts.

EXPRESS-G view



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.

Attribute 'corresponds_to' identifies the non_time_cost that is being 'instantiated' by this entity.

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

Inverse attribute 'resource_cost' indicates the specific active_resource to which this active_cost belongs.

Inverse attribute 'test_cost' indicates the specific active_test to which this active_cost belongs.

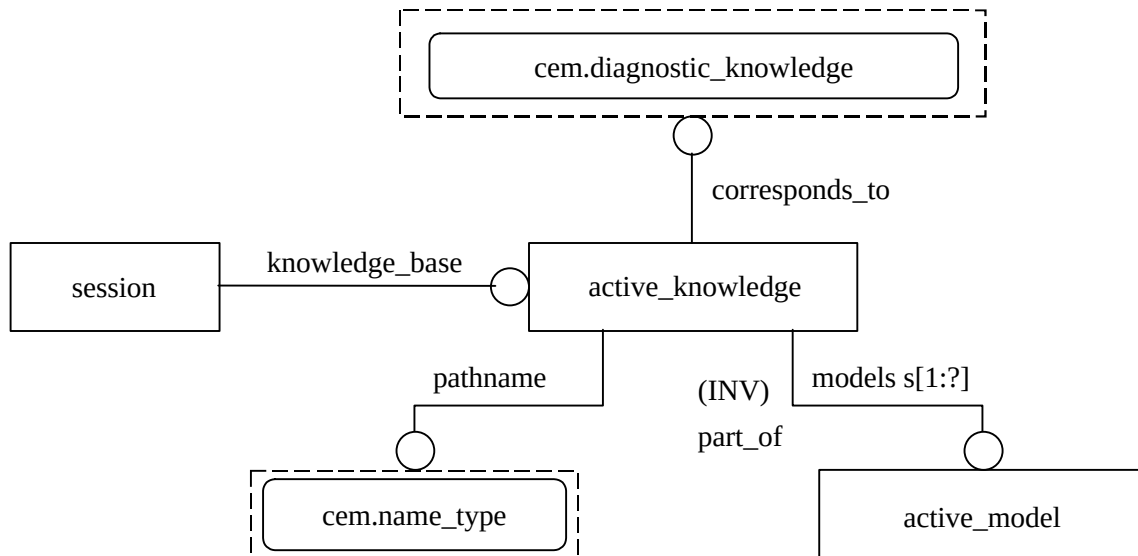
Constraint 'value_is_valid' ensures that actual_value lies between legal bounds, given bounds have been defined.

Constraint 'resource_or_test' ensures either a resource_cost inverse or a test_cost inverse exists.

```
ENTITY active_cost;
    corresponds_to      :      non_time_cost;
    actual_value        :      cost_value;
INVERSE
    resource_cost       :
        SET [0:1] OF active_resource FOR cost_incurred;
    test_cost           :
        SET [0:1] OF active_test 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));
    resource_or_test    :
        SIZEOF(resource_cost) + SIZEOF(test_cost) = 1;
END_ENTITY;
```

5.2.2 active_knowledge

EXPRESS-G view



Entity ‘`active_knowledge`’ references the knowledge base being used in the current diagnostic session. The name of the entity is obtained by examining the referenced knowledge base.

Attribute ‘`corresponds_to`’ identifies the diagnostic knowledge base that is actively loaded and processed in the session.

Attribute ‘`pathname`’ is a string (of type ‘`name_type`’) providing the complete path to the file containing the knowledge base.

Attribute ‘`models`’ lists all of the models referenced by the knowledge base.

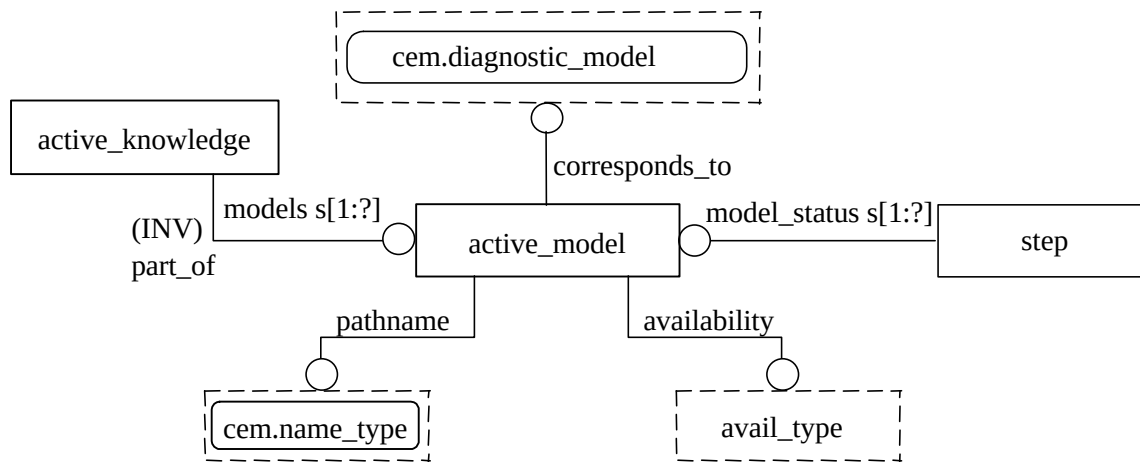
Constraint ‘`models_is_rollup`’ ensures the models attribute lists all models that reference this knowledge base. Thus, the list provides a rollup list of models for the knowledge base.

```

ENTITY active_knowledge;
  corresponds_to      :      diagnostic_knowledge;
  pathname            :      name_type;
  models              :      SET [1:?] OF active_model;
WHERE
  models_is_rollup    :
                      active_model_rollup(SELF, models);
END_ENTITY;
  
```

5.2.3 active_model

EXPRESS-G view



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.

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

Attribute ‘pathname’ is a string (of type ‘name_type’) providing the complete path to the file containing the model.

Attribute ‘availability’ is a LOGICAL attribute indicating whether this model is available for the current session.

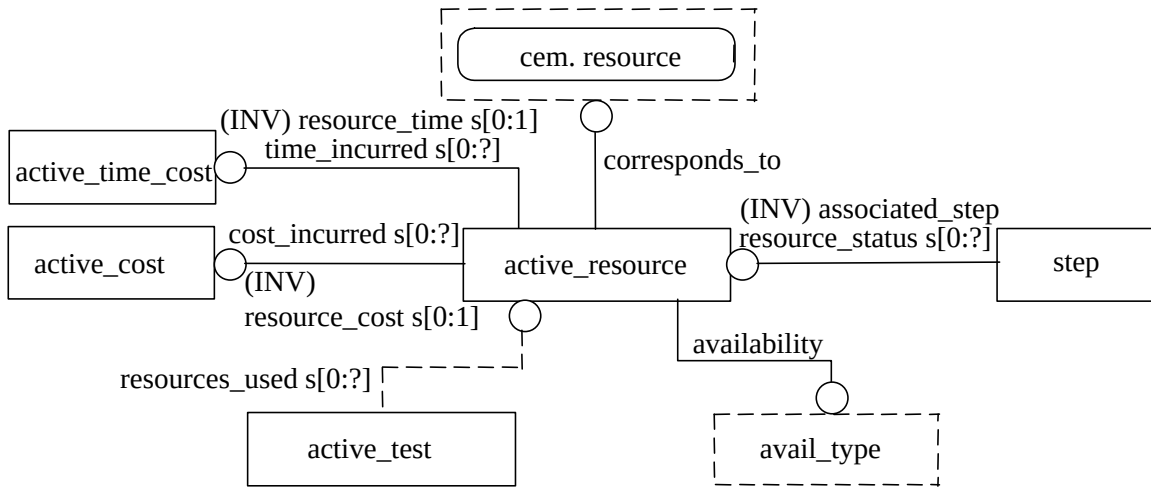
Inverse attribute ‘part_of’ provides a pointer back to the knowledge base referencing this model.

```

ENTITY active_model;
    corresponds_to      :      diagnostic_model;
    pathname            :      name_type;
    availability         :      avail_type;
INVERSE
    part_of             :      active_knowledge FOR models;
END_ENTITY;
  
```

5.2.4 active_resource

EXPRESS-G view



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 0.

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

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.

Attribute 'cost_incurred' records the actual costs associated with using this resource in the current step of the session trace.

Attribute 'time_incurred' records the actual times associated with using this resource in the current step of the session trace.

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

Constraint 'active_cost_is_cost' ensures the set of cost_incurred is a subset of the has_cost cost attribute set of this entity's corresponds_to resource attribute.

Constraint 'active_time_is_time' ensures the set of time_incurred is a subset of the has_cost cost attribute set of this entity's corresponds_to resource attribute.

```

ENTITY active_resource;
  corresponds_to      : resource;
  availability        : avail_type;
  cost_incurred       : SET [0:?] OF active_cost;
  time_incurred       : SET [0:?] OF active_time_cost;
INVERSE
  associated_step     : step FOR resource_status;

```

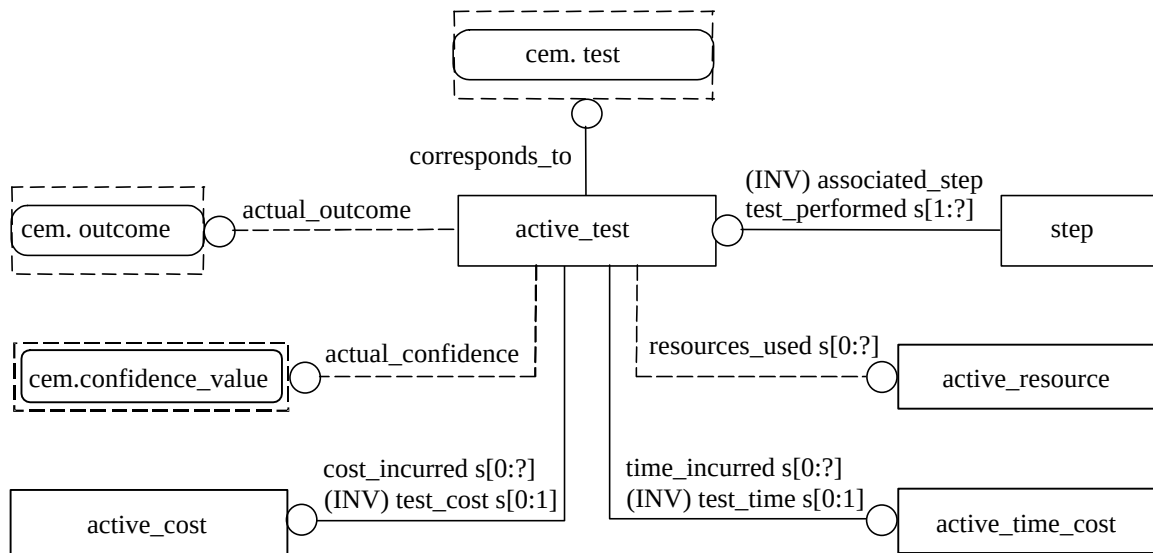
```

WHERE
    active_cost_is_cost      :
        set_of_cost(cost_incurred) <= corresponds_to.has_cost;
    active_time_is_time      :
        set_of_time(time_incurred) <= corresponds_to.has_cost;
END_ENTITY;

```

5.2.5 active_test

EXPRESS-G view



Entity ‘**active_test**’ corresponds to a test used by the reasoner. It is assumed 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.

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

Attribute ‘**actual_outcome**’ provides the current actual outcome associated with this test. This attribute is optional since the outcome may be tied to a test group.

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.

Attribute ‘**resources_used**’ lists the resources used in performing this test. Since the cardinality of the set begins at 0, this set can be empty. It is also an optional attribute indicating whether the reasoner cares if resources are used or not.

Attribute ‘**cost_incurred**’ lists the actual costs incurred in performing this test. If the test was inferred, the costs are likely to be empty. In addition, it is not expected that these costs would include costs of the resources since **active_resource** also includes an attribute for **cost_incurred**.

Attribute 'time_incurred' lists the actual times incurred in performing this test. If the test was inferred, the times are likely to be empty. In addition, it is not expected that these times would include times of the resources since active_resource also includes an attribute for time_incurred.

Inverse attribute 'associated_step' provides a pointer back to the step referring to this active_test entity.

Constraint 'resources_valid' uses function resources_available defined in 5.2.11.5. This constraint ensures that the resources listed as having been used were available at the time they were used.

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.

Constraint 'active_cost_is_cost' ensures the set of cost_incurred is a subset of the has_cost cost attribute set of this entity's corresponds_to test attribute.

Constraint 'active_time_is_time' ensures the set of time_incurred is a subset of the has_cost cost attribute set of this entity's corresponds_to test attribute.

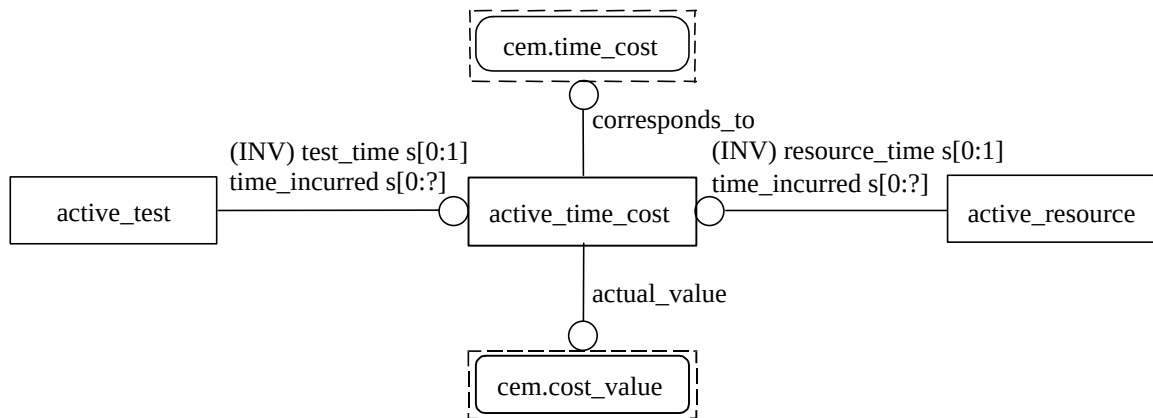
Constraint 'active_resource_is_resource' ensures the set of resources_used is a subset of the test_resource attribute set of this entity's corresponds_to test attribute.

```

ENTITY active_test;
    corresponds_to      :      test;
    actual_outcome       :      OPTIONAL outcome;
    actual_confidence    :      OPTIONAL confidence_value;
    resources_used       :      OPTIONAL SET [0:?] OF
                                active_resource;
    cost_incurred        :      SET [0:?] OF active_cost;
    time_incurred        :      SET [0:?] OF active_time_cost;
INVERSE associated_step:
    step FOR test_performed;
WHERE
    resources_valid      :
        resources_available(resources_used);
    confidence_with_outcome :
        (NOT(EXISTS(actual_confidence)) AND
         NOT(EXISTS(actual_outcome))) OR
        (EXISTS(actual_confidence) AND
         EXISTS(actual_outcome));
    active_cost_is_cost   :
        set_of_cost(cost_incurred) <= corresponds_to.has_cost;
    active_time_is_time   :
        set_of_time(time_incurred) <= corresponds_to.has_cost;
    active_resource_is_resource :
        set_of_resource(resources_used) <=
        corresponds_to.test_resource;
END_ENTITY;
```

5.2.6 active_time_cost

EXPRESS-G view



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**.

Attribute ‘corresponds_to’ identifies the **time_cost** that is being instantiated by this entity.

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

Inverse attribute ‘resource_time’ indicates the specific **active_resource** to which this **active_time_cost** belongs.

Inverse attribute ‘test_time’ indicates the specific **active_test** to which this **active_time_cost** belongs.

Constraint ‘value_is_valid’ ensures that **actual_value** lies between legal bounds, given bounds have been defined.

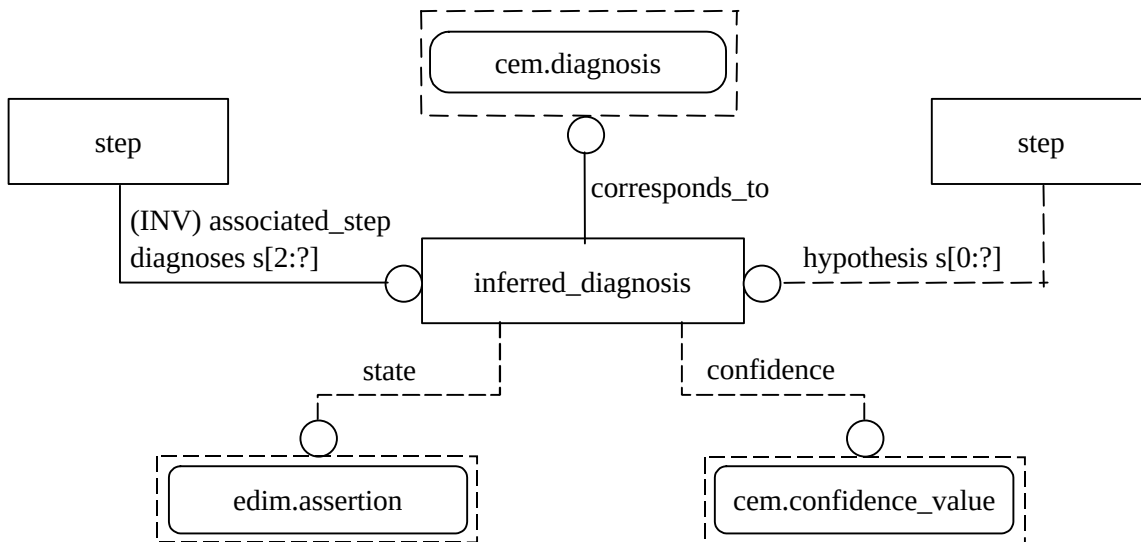
Constraint ‘resource_or_test’ ensures either a **resource_time** inverse or a **test_time** inverse exists.

```

ENTITY active_time_cost;
    corresponds_to      :      time_cost;
    actual_value        :      cost_value;
INVERSE
    resource_time       :
        SET [0:1] OF active_resource FOR time_incurred;
    test_time           :
        SET [0:1] OF active_test 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));
    resource_or_test    :
        SIZEOF(resource_time) + SIZEOF(test_time) = 1;
END_ENTITY;
  
```

5.2.7 inferred_diagnosis

EXPRESS-G view



Entity ‘inferred_diagnosis’ corresponds to a diagnosis that may be inferred by the reasoner. It is assumed the diagnosis can only take on a single value representing the “current” value. Since the **session** trace instantiates the diagnoses at each **step**, it is assumed the set of diagnoses is maintained at each **step**. Therefore, value changes for a diagnosis can be determined by traversing the trace.

Attribute ‘corresponds_to’ identifies the diagnosis in the diagnostic model that is available to be inferred in the **session**.

Attribute ‘state’ indicates the current “value” for the diagnosis. The type of this attribute is referenced from the EDIM, and this attribute can take on the value of either “good” or “candidate.” This attribute is optional since a reasoner may not have this information readily available.

Attribute ‘confidence’ indicates the current confidence in the value associated with this diagnosis. This attribute is optional since it is tied to **state**, which is also optional.

Inverse attribute ‘associated_step’ provides a pointer back to the **step** referring to this **inferred_diagnosis** entity.

Constraint ‘confidence_with_state’ ensures that neither a confidence attribute nor a state attribute exists or that they both exist.

```

ENTITY inferred_diagnosis;
    corresponds_to      :      diagnosis;
    state               :      OPTIONAL assertion;
    confidence          :      OPTIONAL confidence_value;
INVERSE
    associated_step     :      step FOR diagnoses;
  
```

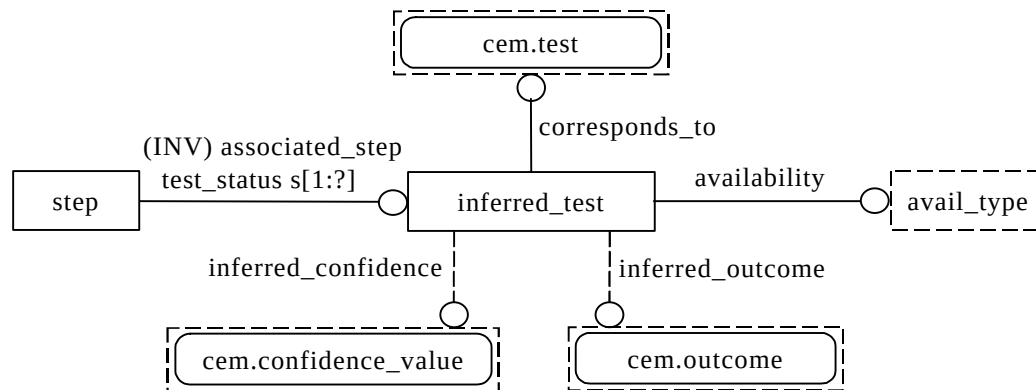
```

WHERE
    confidence_with_state :
        (NOT (EXISTS
            (confidence)) AND NOT
            (EXISTS (state))) OR
            (EXISTS (confidence) AND
            EXISTS (state));
END_ENTITY;

```

5.2.8 inferred_test

EXPRESS-G view



Entity ‘inferred_test’ corresponds to a test whose outcome is inferred by the reasoner. It is assumed 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.

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

Attribute ‘availability’ is a LOGICAL attribute indicating whether this test is available for the current **step**.

Attribute ‘inferred_outcome’ provides the current outcome associated with this test. This attribute is optional since the outcome may be tied to a test group or may be unknown.

Attribute ‘inferred_confidence’ provides the current confidence value associated with the **inferred_outcome** of the test. This attribute is optional since it is tied to the outcome, which is also optional.

Inverse attribute ‘associated_step’ provides a pointer back to the **step** referring to this **inferred_test** entity.

Constraint ‘confidence_with_outcome’ ensures that either an **inferred_outcome** is paired with an **inferred_confidence** or the **inferred_test** has neither an **inferred_outcome** nor **inferred_confidence**. In other words, both **inferred_outcome** and **inferred_confidence** must be present simultaneously, if they exist at all.

Constraint ‘outcome_is_valid’ crosschecks the outcome of the inferred test to ensure either that

- The outcome is identified in the CEM and is an outcome of the CEM test, or
- If the CEM test does not have associated outcomes, the `inferred_test` does not identify an outcome.

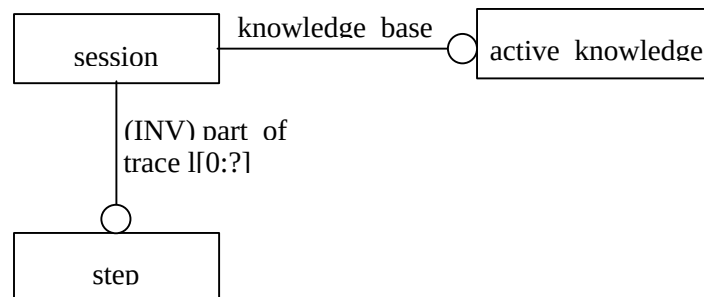
```

ENTITY inferred_test;
  corresponds_to      :      test;
  availability        :      avail_type;
  inferred_outcome    :      OPTIONAL outcome;
  inferred_confidence :      OPTIONAL confidence_value;
INVERSE
  associated_step      :      step FOR test_status;
WHERE
  confidence_with_outcome :
    (NOT(EXISTS(inferred_confidence)) AND
     NOT(EXISTS(inferred_outcome))) OR
    (EXISTS(inferred_confidence) AND
     EXISTS(inferred_outcome));
  outcome_is_valid      :
    (EXISTS(outcome) AND
     EXISTS(corresponds_to.test_outcome) AND
     (outcome IN corresponds_to.test_outcome)) XOR
    (NOT(EXISTS(outcome)) AND
     NOT(EXISTS(corresponds_to.test_outcome)));
END_ENTITY;

```

5.2.9 session

EXPRESS-G view



Entity ‘session’ provides a `session` trace for the current diagnostic `session`. It is assumed only one knowledge base (potentially consisting of multiple models) is used in a single `session`.

Attribute ‘knowledge_base’ indicates the knowledge base being used by diagnostic `session` corresponding to this `session` 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.

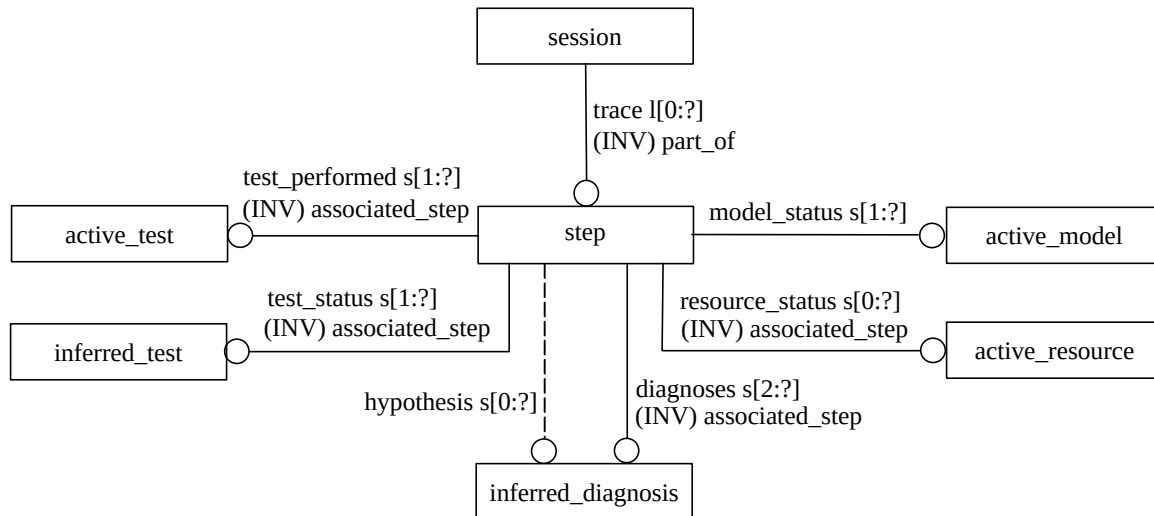
```

ENTITY session;
    knowledge_base      :    active_knowledge;
    trace               :    LIST [0:?] OF step;
END_ENTITY;

```

5.2.10 step

EXPRESS-G view



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.

Attribute ‘model_status’ lists all of the models associated with the current **session** and indicates their status at the current **step**.

Attribute ‘test_performed’ indicates the set of active tests performed at this **step** in the process.

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 may 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.

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.

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.

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.

Inverse attribute ‘part_of’ indicates the specific **session** trace to which this **step** belongs.

Constraint ‘model_active’ evaluates three predicates to ensure that the set of models listed in this **step** are active in the current **session**. The first predicate verifies that the model is identified by the

knowledge base. The second predicate verifies the models are expected in the current state. The third predicate verifies that all models are available.

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

Constraint ‘resource_active’ evaluates two predicates to ensure 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.

Constraint ‘diagnosis_in_state’ ensures the diagnoses in the hypothesis have been specified by the set of diagnoses given by the current state.

Constraint ‘diagnosis_candidate’ ensures that none of the diagnoses in the current hypothesis have been inferred to be “good” and thus eliminated from consideration.

Constraint ‘all_tests’ uses function `test_set_union` defined in 5.2.11.2. This constraint verifies that the union of the test sets from the available models equals the set listed in `test_status`.

Constraint ‘all_diagnoses’ uses function `diagnosis_set_union` defined in 5.2.11.3. This constraint verifies that the union of the diagnosis sets from the available models equals the set listed in `diagnoses`.

Constraint ‘all_active_models’ verifies that all of the models are specified by ‘active_knowledge’ and that all of the models listed are available.

```

ENTITY step;
    model_status      : SET [1:?] OF active_model;
    test_performed    : SET [1:?] OF active_test;
    hypothesis        : OPTIONAL SET [0:?] OF
                        inferred_diagnosis;
    resource_status    : SET [0:?] OF active_resource;
    test_status        : SET [1:?] OF inferred_test;
    diagnoses         : SET [2:?] OF inferred_diagnosis;
INVERSE
    part_of           : session FOR trace;
WHERE
    model_active      :
        (model_status <= part_of.knowledge_base.models) AND
        (SIZEOF(QUERY(temp <* model_status |
                        temp.availability = FALSE)) = 0);
    test_active       :
        active_tests_available(test_performed, 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 :
        (hypothesis <= diagnoses);
    diagnosis_candidate :
        (SIZEOF(QUERY(temp <* hypothesis |
                        temp.state = good)) = 0);
    all_tests         :
        (set_of_test(test_status) =
         test_set_union(model_status));
    all_diagnoses     :

```

```

        (set_of_diagnosis(diagnoses) =
          diagnosis_set_union(model_status));
    all_active_models :
        (model_status = model_status[1].part_of.models) AND
        (SIZEOF(QUERY(temp <* model_status |
                      temp.availability = FALSE)) = 0);
END_ENTITY;

```

5.2.11 Model constraint functions

The specification of the AI-ESTATE DCM constraint functions are defined in the following subsections.

5.2.11.1 active_tests_available

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 =
            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;

```

5.2.11.2 active_model_rollup

Function ‘active_model_rollup’ compares the models defined by the specified knowledge base and a separately specified set of models to determine whether the sets are the same. The function works by considering each model in the model_set and determining whether that model appears in the list of models from knowledge. Actually, this function does not determine equality. It only determines whether model_set is a subset of the models defined in knowledge. A separate function determines the converse.

```

FUNCTION active_model_rollup(knowledge : active_knowledge;
    model_set : SET [1:?] OF active_model) : BOOLEAN;
LOCAL
    result : BOOLEAN := TRUE;
END_LOCAL;

```

```

    REPEAT i := LOINDEX(model_set) TO HIINDEX(model_set);
      IF (model_set[i].part_of <> knowledge) THEN
        result := FALSE;
      END_IF;
    END_REPEAT;
    RETURN(result);
  END_FUNCTION;

```

5.2.11.3 diagnosis_set_union

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 +
      mdl[i].corresponds_to.model_diagnosis;
  END_REPEAT;
  RETURN(diagnosis_set);
END_FUNCTION;

```

5.2.11.4 model_in_knowledge

Function ‘model_in_knowledge’ tests to determine whether a model is in the set of models defined by `knowledge`.

```

FUNCTION model_in_knowledge mdl : active_model; knowledge :
  active_knowledge) : BOOLEAN;
  LOCAL
    result : LOGICAL := FALSE;
  END_LOCAL;
  REPEAT i := LOINDEX(knowledge.models) TO
    HIINDEX(knowledge.models);
    IF mdl = knowledge.models[i] THEN
      result := TRUE;
    END_IF;
  END_REPEAT;
  RETURN(result);
END_FUNCTION;

```

5.2.11.5 resources_available

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 : LOGICAL := 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;

```

5.2.11.6 resource_set_union

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 +
                                mdl[i].corresponds_to.model_resource;
    END_REPEAT;
    RETURN(resource_set);
END_FUNCTION;

```

5.2.11.7 set_of_cost

Function 'set_of_cost' constructs a set of non_time_costs as defined in the CEM when given a set of active_costs from the DCM.

```

FUNCTION set_of_cost ntc : SET [0:?] OF active_cost) :
    SET [0:?] OF non_time_cost;
    LOCAL
        cost_set : SET [0:?] OF non_time_cost := [];
    END_LOCAL;
    REPEAT i := LOINDEX(ntc) TO HIINDEX(ntc);
        cost_set := cost_set +
                                ntc[i].corresponds_to;
    END_REPEAT;
    RETURN(cost_set);
END_FUNCTION;

```

5.2.11.8 set_of_diagnosis

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;

```

5.2.11.9 set_of_resource

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;

```

5.2.11.10 set_of_test

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;

```

5.2.11.11 set_of_time

Function 'set_of_time' constructs a set of time_costs as defined in the CEM when given a set of active_time_costs from the DCM.

```

FUNCTION set_of_time(tc : SET [0:?] OF active_time_cost) :
    SET [0:?] OF time_cost;
    LOCAL
        time_set : SET [0:?] OF time_cost := [];
    END_LOCAL;
    REPEAT i := LOINDEX(tc) TO HIINDEX(tc);
        time_set := time_set +
            tc[i].corresponds_to;
    END_REPEAT;
    RETURN(time_set);
END_FUNCTION;

```

5.2.11.12 test_set_union

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 +
            mdl[i].corresponds_to.model_test;
    END_REPEAT;
    RETURN(test_set);
END_FUNCTION;
```

5.2.12 DCM EXPRESS specification

SCHEMA `dynamic_context_model`;

REFERENCE FROM `ai_estate_common_element_model`;
REFERENCE FROM
 `enhanced_diagnostic_inference_model`;

TYPE `avail_type` = LOGICAL;
END_TYPE;

(* ENTITY DEFINITIONS *)

```
ENTITY active_cost;
    corresponds_to      :      non_time_cost;
    actual_value        :      cost_value;
INVERSE
    resource_cost      :
        SET [0:1] OF active_resource FOR cost_incurred;
    test_cost          :
        SET [0:1] OF active_test 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));
    resource_or_test    :
        SIZEOF(resource_cost) + SIZEOF(test_cost) = 1;
END_ENTITY;
```

```
ENTITY active_knowledge;
    corresponds_to      :      diagnostic_knowledge;
    pathname            :      name_type;
    models              :      SET [1:?] of active_model;
WHERE
    models_is_rollup    :
```

```

        active_model_rollup(SELF,models);
END_ENTITY;

```

```

ENTITY active_model;
    corresponds_to      :      diagnostic_model;
    pathname            :      name_type;
    availability         :      avail_type;
INVERSE
    part_of             :      active_knowledge FOR models;
END_ENTITY;

```

```

ENTITY active_resource;
    corresponds_to      :      resource;
    availability         :      avail_type;
    cost_incurred       :      SET [0:?] OF active_cost;
    time_incurred       :      SET [0:?] OF active_time_cost;
INVERSE
    associated_step      :      step FOR resource_status;
WHERE
    active_cost_is_cost :
        set_of_cost(cost_incurred) <= corresponds_to.has_cost;
    active_time_is_time :
        set_of_time(time_incurred) <= corresponds_to.has_cost;
END_ENTITY;

```

```

ENTITY active_test;
    corresponds_to      :      test;
    actual_outcome       :      OPTIONAL outcome;
    actual_confidence    :      OPTIONAL confidence_value;
    resources_used       :      OPTIONAL SET [0:?] OF
                                active_resource;
    cost_incurred       :      SET [0:?] OF active_cost;
    time_incurred       :      SET [0:?] OF active_time_cost;
INVERSE associated_step:
    step FOR test_performed;
WHERE
    resources_valid      :
        resources_available(resources_used);
    confidence_with_outcome :
        (NOT(EXISTS(actual_confidence)) AND
         NOT(EXISTS(actual_outcome))) OR
        (EXISTS(actual_confidence) AND
         EXISTS(actual_outcome));
    active_cost_is_cost :
        set_of_cost(cost_incurred) <= corresponds_to.has_cost;
    active_time_is_time :
        set_of_time(time_incurred) <= corresponds_to.has_cost;
    active_resource_is_resource :
        set_of_resource(resources_used) <=
            corresponds_to.test_resource;
END_ENTITY;

```

```

ENTITY active_time_cost;
    corresponds_to      :      time_cost;
    actual_value         :      cost_value;
INVERSE
    resource_time       :
        SET [0:1] OF active_resource FOR time_incurred;

```

```

        test_time          :
            SET [0:1] OF active_test 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));
    resource_or_test        :
        SIZEOF(resource_time) + SIZEOF(test_time) = 1;
END_ENTITY;

ENTITY inferred_diagnosis;
    corresponds_to          :      diagnosis;
    state                   :      OPTIONAL assertion;
    confidence              :      OPTIONAL confidence_value;
INVERSE
    associated_step         :      step FOR diagnoses;
WHERE
    confidence_with_state   :
        (NOT (EXISTS
            (confidence)) AND NOT
            (EXISTS (state))) OR
            (EXISTS (confidence) AND
            EXISTS (state));
END_ENTITY;

ENTITY inferred_test;
    corresponds_to          :      test;
    availability            :      avail_type;
    inferred_outcome        :      OPTIONAL outcome;
    inferred_confidence     :      OPTIONAL confidence_value;
INVERSE
    associated_step         :      step FOR test_status;
WHERE
    confidence_with_outcome :
        (NOT(EXISTS(inferred_confidence)) AND
         NOT(EXISTS(inferred_outcome))) OR
        (EXISTS(inferred_confidence) AND
         EXISTS(inferred_outcome));
    outcome_is_valid        :
        (EXISTS(outcome) AND
         EXISTS(corresponds_to.test_outcome) AND
         (outcome IN corresponds_to.test_outcome)) XOR
        (NOT(EXISTS(outcome)) AND
         NOT(EXISTS(corresponds_to.test_outcome)));
END_ENTITY;

ENTITY session;
    knowledge_base          :      active_knowledge;
    trace                   :      LIST [0:?] OF step;
END_ENTITY;

ENTITY step;
    model_status            :      SET [1:?] OF active_model;
    test_performed          :      SET [1:?] OF active_test;
    hypothesis              :      OPTIONAL SET [0:?] OF
                                inferred_diagnosis;

```

```

        resource_status      :      SET [0:?] of active_resource;
        test_status          :      SET [1:?] OF inferred_test;
        diagnoses            :      SET [2:?] OF inferred_diagnosis;
INVERSE
    part_of                  :      session FOR trace;
WHERE
    model_active             :
        (model_status <= part_of.knowledge_base.models) AND
        (SIZEOF(QUERY(temp <* model_status |
                        temp.availability = FALSE)) = 0);
    test_active              :
        active_tests_available(test_performed, 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       :
        (hypothesis <= diagnoses);
    diagnosis_candidate      :
        (SIZEOF(QUERY(temp <* hypothesis |
                        temp.state = good)) = 0);
    all_tests                :
        (set_of_test(test_status) =
         test_set_union(model_status));
    all_diagnoses            :
        (set_of_diagnosis(diagnoses) =
         diagnosis_set_union(model_status));
    all_active_models        :
        (model_status = model_status[1].part_of.models) AND
        (SIZEOF(QUERY(temp <* model_status |
                        temp.availability = FALSE)) = 0);
END_ENTITY;

```

(* FUNCTION DEFINITIONS *)

(*

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 =
            inferred[j].corresponds_to) THEN
            member := TRUE;
            IF (inferred[j].availability = FALSE) THEN
                result := FALSE;
            END_IF;
        END_IF;
    END_IF;

```

```

    END_REPEAT;
    IF (member = FALSE) THEN
        result := FALSE;
    END_IF;
    END_REPEAT;
    RETURN (result);
END_FUNCTION;

```

(*
Function 'active_model_rollup' compares the models defined by the specified knowledge base and a separately specified set of models to determine whether the sets are the same. The function works by considering each model in the model_set and determining whether that model appears in the list of models from knowledge. Actually, this function does not determine equality. It only determines whether model_set is a subset of the models defined in knowledge. A separate function determines the converse.
*)

```

FUNCTION active_model_rollup(knowledge : active_knowledge;
    model_set : SET [1:?] OF active_model) : BOOLEAN;
    LOCAL
        result : BOOLEAN := TRUE;
    END_LOCAL;
    REPEAT i := LOINDEX(model_set) TO HIINDEX(model_set);
        IF (model_set[i].part_of <> knowledge) THEN
            result := FALSE;
        END_IF;
    END_REPEAT;
    RETURN(result);
END_FUNCTION;

```

(*
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 +
            mdl[i].corresponds_to.model_diagnosis;
    END_REPEAT;
    RETURN(diagnosis_set);
END_FUNCTION;

```

(*
Function 'model_in_knowledge' tests to determine whether a model is in the set of models defined by knowledge.
*)

```

FUNCTION model_in_knowledge(mdl : active_model; knowledge :
    active_knowledge) : BOOLEAN;
    LOCAL
        result : LOGICAL := FALSE;

```

```

    END_LOCAL;
    REPEAT i := LOINDEX(knowledge.models) TO
        HIINDEX(knowledge.models);
        IF mdl = knowledge.models[i] THEN
            result := TRUE;
        END_IF;
    END_REPEAT;
    RETURN(result);
END_FUNCTION;

```

(*
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 : LOGICAL := 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;

```

(*
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 +
            mdl[i].corresponds_to.model_resource;
    END_REPEAT;
    RETURN(resource_set);
END_FUNCTION;

```

(*
Function 'set_of_cost' constructs a set of non_time_costs as defined in the CEM when given a set of active_costs from the DCM.
*)

```

FUNCTION set_of_cost(ntc : SET [0:?] OF active_cost) :
    SET [0:?] OF non_time_cost;
    LOCAL
        cost_set : SET [0:?] OF non_time_cost := [];
    END_LOCAL;
    REPEAT i := LOINDEX(ntc) TO HIINDEX(ntc);
        cost_set := cost_set +

```

```

        ntc[i].corresponds_to;
    END_REPEAT;
    RETURN(cost_set);
END_FUNCTION;

(*
Function 'set_of_diagnosis' constructs a set of diagnoses as defined in
the CEM when given a set of inferred_diagnoses 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;

(*
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;

(*
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;

(*
Function 'set_of_time' constructs a set of time_costs as defined in the
CEM when given a set of active_time_costs from the DCM.
*)

```

```

FUNCTION set_of_time (tc : SET [0:?] OF active_time_cost) :
    SET [0:?] OF time_cost;
    LOCAL
        time_set : SET [0:?] OF time_cost := [];
    END_LOCAL;
    REPEAT i := LOINDEX(tc) TO HIINDEX(tc);
        time_set := time_set +
            tc[i].corresponds_to;
    END_REPEAT;
    RETURN(time_set);
END_FUNCTION;

```

(*
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 +
            mdl[i].corresponds_to.model_test;
    END_REPEAT;
    RETURN(test_set);
END_FUNCTION;

```

END_SCHEMA;

6. Services specification

6.1 Overview

This clause contains the specification for the diagnostic reasoner services for the AI-ESTATE DCM and CEM. The services are specified in the form of EXPRESS functions and procedures and are grouped according to the model entity they support.

Services are provided that allow users to create and delete each model entity. These services will request that the reasoner instantiate and destroy attributes of a model, respectively. Upon creation, each entity will possess the appropriate parent-child attribute links in an implementation-independent manner, therefore, reflecting the strong typing dictated by the model. Upon deletion, each entity shall be discarded in the view of the client application, again in an implementation-independent manner. In the case of either *create* or *delete* services, implementation-independent simply means that any memory management operations (i.e., pointer housekeeping) are the responsibility of the reasoner.

Services are also provided that allow users to get and put the attributes of each model entity, thus providing the means to examine, initialize, and revise the attributes of each entity. Again, the typing constraints of the model will be observed, but the implementation may use any underlying means necessary to accomplish this purpose. The parameter that references the value(s) the reasoner is to *put* or *get* shall be implemented as arrays in order to support multiple value returns. It is important to note that the value returned by any service shall be, at a minimum, a status indicator as defined in 6.2.

Reasoner service parameters shall be treated as “pass by value,” not “reference.” In other words, the invocation of a service shall have no effect locally on the value of the service’s input parameter. Unless otherwise specified, a service will have no effect on the state of the reasoning system. It will simply return its expected value.

The service definitions further support the following base and derived data types:

- Base
 - Float
 - Integer
 - Character
 - Enumerated
 - Boolean
- Derived
 - String
 - Array
 - Set
 - List

6.2 Status codes

An AI-ESTATE diagnostic reasoner shall provide a means for its clients to determine the success or failure of service requests. This means of determining success or failure shall be implemented by means of a global variable named *status_code*. This variable will assume one of the following mnemonic values:

- *operation_completed_successfully*
- *nonexistent_data_element_requested*
- *missing_or_invalid_argument*
- *invalid_model_schema*
- *service_not_available*
- *unknown_exception_raised*

The actual mapping will be provided by the language bindings.

In a multi-threaded environment, the variable shall be visible only to the local process in support of asynchronous applications.

6.2.1 Operation_completed_successfully

The *status_code* global variable will indicate *operation_completed_successfully* when the requested service was completed successfully and no other status code condition applies.

6.2.2 Nonexistent_data_element_requested

The *status_code* global variable will indicate *nonexistent_data_element_requested* when one of the static model accessor *get* services is requested for an optional data element of an entity and the requested data element does not exist.

6.2.3 Missing_or_invalid_argument

The *status_code* global variable will indicate *missing_or_invalid_argument* in the case when an ill-formed parameter list is provided as part of the service request.

6.2.4 Invalid_model_schema

The *status_code* global variable will indicate *invalid_model_schema* when a service is requested on a model that is inconsistent with a model's schema type. For example, applying an EDIM service request on a FTM is invalid.

6.2.5 Service_not_available

If a service is not fully elaborated, a conformant reasoner must, at a minimum, provide a stub service that returns the *service_not_available* status value. The *status_code* global variable will indicate this response when the requested service has not been further elaborated by the underlying application.

6.2.6 Unknown_exception_raised

The *status_code* global variable will indicate *unknown_exception_raised* when an error or other exceptional situation occurs during service execution causing normal execution to be abandoned. The requested service has detected the error or exceptional situation and has determined that no other status code is appropriate. As an example, this status code may be returned when the service requires system resources (e.g., memory, disk space) and the system fails to provide the resources.

6.3 DCM EXPRESS service specification

The specifications of the AI-ESTATE DCM services are defined in the following subsections. The service definitions are organized by entity as defined in Clause 5.

6.3.1 active_cost

The following functions are provided to create and delete instances of the DCM entity *active_cost* and to access (get and put) its attributes. The *active_cost* entity is defined in 5.2.1.

The *create* function will create an instance of the entity and return an instance identifier. The implementation of the instance identifier is dependent upon the target language (e.g., a pointer, a handle).

```
FUNCTION create_active_cost :
    active_cost;
END_FUNCTION;
```

The *get* functions will be given an entity instance identifier and will return the current value of the entity attribute identified by the function name. The *get* functions for entity *active_cost* are the following:

```
FUNCTION get_active_cost_corresponds_to
    ( ac_id : active_cost ) :
    non_time_cost;
END_FUNCTION;

FUNCTION get_active_cost_actual_value
    ( ac_id : active_cost ) :
    cost_value;
END_FUNCTION;
```

```
FUNCTION get_active_cost_resource_cost
  ( ac_id : active_cost ) :
  SET [0:1] OF active_resource;
END_FUNCTION;
```

```
FUNCTION get_active_cost_test_cost
  ( ac_id : active_cost ) :
  SET [0:1] OF active_test;
END_FUNCTION;
```

The *put* procedures will be given an entity instance identifier and a parameter value to be placed in the entity attribute identified by the procedure name. There is no return from a procedure. The *put* procedures for entity *active_cost* are the following:

```
PROCEDURE put_active_cost_corresponds_to
  ( ac_id          : active_cost;
    corresponds_to : non_time_cost );
END_PROCEDURE;
```

```
PROCEDURE put_active_cost_actual_value
  ( ac_id          : active_cost;
    actual_value   : cost_value );
END_PROCEDURE;
```

```
PROCEDURE put_active_cost_resource_cost
  ( ac_id          : active_cost;
    resource_cost  : SET [0:1] OF active_resource );
END_PROCEDURE;
```

```
PROCEDURE put_active_cost_test_cost
  ( ac_id          : active_cost;
    test_cost      : SET [0:1] OF active_test );
END_PROCEDURE;
```

The *delete* procedure will be given an entity instance identifier and will delete the instance. There is no return from a procedure.

```
PROCEDURE delete_active_cost
  ( ac_id : active_cost );
END_PROCEDURE;
```

6.3.2 active_knowledge

The following functions are given to create and delete instances of the DCM entity *active_knowledge* and to access (get and put) its attributes. The *active_knowledge* entity is defined in 5.2.2.

The *create* function will create an instance of the entity and return an instance identifier. The implementation of the instance identifier is dependent upon the target language (e.g., a pointer, a handle).

```
FUNCTION create_active_knowledge :
  active_knowledge;
END_FUNCTION;
```

The *get* functions will be given an entity instance identifier and will return the current value of the entity attribute identified by the function name. The *get* functions for entity *active_knowledge* are the following:

```

FUNCTION get_active_knowledge_corresponds_to
    ( ak_id : active_knowledge ) :
        diagnostic_knowledge;
END_FUNCTION;

FUNCTION get_active_knowledge_pathname
    ( ak_id : active_knowledge ) :
        name_type;
END_FUNCTION;

FUNCTION get_active_knowledge_models
    ( ak_id : active_knowledge ) :
        SET [1:?] OF active_model;
END_FUNCTION;

```

The *put* procedures will be given an entity instance identifier and a parameter value to be placed in the entity attribute identified by the procedure name. There is no return from a procedure. The *put* procedures for entity `active_knowledge` are the following:

```

PROCEDURE put_active_knowledge_corresponds_to
    ( ak_id      : active_knowledge;
      corresponds_to : diagnostic_knowledge );
END_PROCEDURE;

PROCEDURE put_active_knowledge_pathname
    ( ak_id      : active_knowledge;
      pathname   : name_type );
END_PROCEDURE;

PROCEDURE put_active_knowledge_models
    ( ak_id      : active_knowledge;
      models     : SET [1:?] OF active_model );
END_PROCEDURE;

```

The *delete* procedure will be given an entity instance identifier and will delete the instance. There is no return from a procedure.

```

PROCEDURE delete_active_knowledge
    ( ak_id : active_knowledge );
END_PROCEDURE;

```

6.3.3 active_model

The following functions are given to create and delete instances of the DCM entity `active_model` and to access (get and put) its attributes. The `active_model` entity is defined in 5.2.3.

The *create* function will create an instance of the entity and return an instance identifier. The implementation of the instance identifier is dependent upon the target language (e.g., a pointer, a handle).

```

FUNCTION create_active_model :
    active_model;
END_FUNCTION;

```

The *get* functions will be given an entity instance identifier and will return the current value of the entity attribute identified by the function name. The *get* functions for entity `active_model` are the following:

```

FUNCTION get_active_model_corresponds_to
    ( am_id : active_model ) :
        diagnostic_model;
END_FUNCTION;

FUNCTION get_active_model_pathname
    ( am_id : active_model ) :
        name_type;
END_FUNCTION;

FUNCTION get_active_model_availability
    ( am_id : active_model ) :
        avail_type;
END_FUNCTION;

FUNCTION get_active_model_part_of
    ( am_id : active_model ) :
        active_knowledge;
END_FUNCTION;

```

The *put* procedures will be given an entity instance identifier and a parameter value to be placed in the entity attribute identified by the procedure name. There is no return from a procedure. The *put* procedures for entity *active_model* are the following:

```

PROCEDURE put_active_model_corresponds_to
    ( am_id      : active_model;
      corresponds_to : diagnostic_model );
END_PROCEDURE;

PROCEDURE put_active_model_pathname
    ( am_id : active_model; pathname : name_type );
END_PROCEDURE;

PROCEDURE put_active_model_availability
    ( am_id      : active_model;
      availability : avail_type );
END_PROCEDURE;

PROCEDURE put_active_model_part_of
    ( am_id : active_model;
      part_of : active_knowledge );
END_PROCEDURE;

```

The *delete* procedure will be given an entity instance identifier and will delete the instance. There is no return from a procedure.

```

PROCEDURE delete_active_model
    ( am_id : active_model );
END_PROCEDURE;

```

6.3.4 active_resource

The following functions are provided to create and delete instances of the DCM entity *active_resource* and to access (get and put) its attributes. The *active_resource* entity is defined in 5.2.4.

The *create* function will create an instance of the entity and return an instance identifier. The implementation of the instance identifier is dependent upon the target language (e.g., a pointer, a handle).

```
FUNCTION create_active_resource :
    active_resource;
END_FUNCTION;
```

The *get* functions will be given an entity instance identifier and will return the current value of the entity attribute identified by the function name. The *get* functions for entity *active_resource* are the following:

```
FUNCTION get_active_resource_corresponds_to
    ( ar_id : active_resource ) :
    resource;
END_FUNCTION;
```

```
FUNCTION get_active_resource_availability
    ( ar_id : active_resource ) :
    avail_type;
END_FUNCTION;
```

```
FUNCTION get_active_resource_cost_incurred
    ( ar_id : active_resource ) :
    SET [0:?] OF active_cost;
END_FUNCTION;
```

```
FUNCTION get_active_resource_time_incurred
    ( ar_id : active_resource ) :
    SET [0:?] OF active_time_cost;
END_FUNCTION;
```

```
FUNCTION get_active_resource_associated_step
    ( ar_id : active_resource ) :
    step;
END_FUNCTION;
```

The *put* procedures will be given an entity instance identifier and a parameter value to be placed in the entity attribute identified by the procedure name. There is no return from a procedure. The *put* procedures for entity *active_resource* are the following:

```
PROCEDURE put_active_resource_corresponds_to
    ( ar_id : active_resource;
      corresponds_to : resource );
END_PROCEDURE;
```

```
PROCEDURE put_active_resource_availability
    ( ar_id : active_resource;
      availability : avail_type );
END_PROCEDURE;
```

```
PROCEDURE put_active_resource_cost_incurred
    ( ar_id : active_resource;
      cost_incurred : SET [0:?] OF active_cost );
END_PROCEDURE;
```

```

PROCEDURE put_active_resource_time_incurred
( ar_id      : active_resource;
  time_incurred : SET [0:?] OF active_time_cost );
END_PROCEDURE;

PROCEDURE put_active_resource_associated_step
( ar_id      : active_resource;
  associated_step : step
  );
END_PROCEDURE;

```

The *delete* procedure will be given an entity instance identifier and will delete the instance. There is no return from a procedure.

```

PROCEDURE delete_active_resource
( ar_id : active_resource );
END_PROCEDURE;

```

6.3.5 active_test

The following functions are provided to create and delete instances of the DCM entity *active_test* and to access (get and put) its attributes. The *active_test* entity is defined in 5.2.5.

The *create* function will create an instance of the entity and return an instance identifier. The implementation of the instance identifier is dependent upon the target language (e.g., a pointer, a handle).

```

FUNCTION create_active_test :
    active_test;
END_FUNCTION;

```

The *get* functions will be given an entity instance identifier and will return the current value of the entity attribute identified by the function name. The *get* functions for entity *active_test* are the following:

```

FUNCTION get_active_test_corresponds_to
( at_id : active_test ) :
    test;
END_FUNCTION;

FUNCTION get_active_test_actual_outcome
( at_id : active_test ) :
    outcome;
END_FUNCTION;

FUNCTION get_active_test_actual_confidence
( at_id : active_test ) :
    confidence_value;
END_FUNCTION;

FUNCTION get_active_test_resources_used
( at_id : active_test ) :
    SET [0:?] OF active_resource;
END_FUNCTION;

FUNCTION get_active_test_cost_incurred
( at_id : active_test ) :
    SET [0:?] OF active_cost;
END_FUNCTION;

```

```

FUNCTION get_active_test_time_incurred
  ( at_id : active_test ) :
    SET [0:?] OF active_time_cost;
END_FUNCTION;

```

```

FUNCTION get_active_test_associated_step
  ( at_id : active_test ) :
    step;
END_FUNCTION;

```

The *put* procedures will be given an entity instance identifier and a parameter value to be placed in the entity attribute identified by the procedure name. There is no return from a procedure. The *put* procedures for entity *active_test* are the following:

```

PROCEDURE put_active_test_corresponds_to
  ( at_id : active_test; corresponds_to : test );
END_PROCEDURE;

```

```

PROCEDURE put_active_test_actual_outcome
  ( at_id : active_test;
    actual_outcome : outcome );
END_PROCEDURE;

```

```

PROCEDURE put_active_test_actual_confidence
  ( at_id : active_test;
    actual_confidence : confidence_value );
END_PROCEDURE;

```

```

PROCEDURE put_active_test_resources_used
  ( at_id : active_test;
    resources_used : SET [0:?] OF active_resource );
END_PROCEDURE;

```

```

PROCEDURE put_active_test_cost_incurred
  ( at_id : active_test;
    cost_incurred : SET [0:?] OF active_cost );
END_PROCEDURE;

```

```

PROCEDURE put_active_test_time_incurred
  ( at_id : active_test;
    time_incurred : SET [0:?] OF active_time_cost );
END_PROCEDURE;

```

```

PROCEDURE put_active_test_associated_step
  ( at_id : active_test;
    associated_step : step );
END_PROCEDURE;

```

The *delete* procedure will be given an entity instance identifier and will delete the instance. There is no return from a procedure.

```

PROCEDURE delete_active_test
  ( at_id : active_test );
END_PROCEDURE;

```

6.3.6 active_time_cost

The following functions are provided to create and delete instances of the DCM entity `active_time_cost` and to access (get and put) its attributes. The `active_time_cost` entity is defined in 5.2.6.

The *create* function will create an instance of the entity and return an instance identifier. The implementation of the instance identifier is dependent upon the target language (e.g., a pointer, a handle).

```
FUNCTION create_active_time_cost :  
    active_time_cost;  
END_FUNCTION;
```

The *get* functions will be given an entity instance identifier and will return the current value of the entity attribute identified by the function name. The *get* functions for entity `active_time_cost` are the following:

```
FUNCTION get_active_time_cost_corresponds_to  
    ( atc_id : active_time_cost ) :  
    time_cost;  
END_FUNCTION;
```

```
FUNCTION get_active_time_cost_actual_value  
    ( atc_id : active_time_cost ) :  
    cost_value;  
END_FUNCTION;
```

```
FUNCTION get_active_time_cost_resource_time  
    ( atc_id : active_time_cost ) :  
    SET [0:1] OF active_resource;  
END_FUNCTION;
```

```
FUNCTION get_active_time_cost_test_time  
    ( atc_id : active_time_cost ) :  
    SET [0:1] OF active_test;  
END_FUNCTION;
```

The *put* procedures will be given an entity instance identifier and a parameter value to be placed in the entity attribute identified by the procedure name. There is no return from a procedure. The *put* procedures for entity `active_time_cost` are the following:

```
PROCEDURE put_active_time_cost_corresponds_to  
    ( atc_id      : active_time_cost;  
      corresponds_to : time_cost      );  
END_PROCEDURE;
```

```
PROCEDURE put_active_time_cost_actual_value  
    ( atc_id      : active_time_cost;  
      actual_value : cost_value      );  
END_PROCEDURE;
```

```
PROCEDURE put_active_time_cost_resource_time  
    ( atc_id      : active_time_cost;  
      resource_time : SET [0:1] OF active_resource );  
END_PROCEDURE;
```

```

PROCEDURE put_active_time_cost_test_time
    ( atc_id      : active_time_cost;
      test_time   : SET [0:1] OF active_test );
END_PROCEDURE;

```

The *delete* procedure will be given an entity instance identifier and will delete the instance. There is no return from a procedure.

```

PROCEDURE delete_active_time_cost
    ( atc_id : active_time_cost );
END_PROCEDURE;

```

6.3.7 inferred_diagnosis

The following functions are provided to create and delete instances of the DCM entity *inferred_diagnosis* and to access (get and put) its attributes. The *inferred_diagnosis* entity is defined in 5.2.7.

The *create* function will create an instance of the entity and return an instance identifier. The implementation of the instance identifier is dependent upon the target language (e.g., a pointer, a handle).

```

FUNCTION create_inferred_diagnosis :
    inferred_diagnosis;
END_FUNCTION;

```

The *get* functions will be given an entity instance identifier and will return the current value of the entity attribute identified by the function name. The *get* functions for entity *inferred_diagnosis* are the following:

```

FUNCTION get_inferred_diagnosis_corresponds_to
    ( id_id : inferred_diagnosis ) :
    diagnosis;
END_FUNCTION;

```

```

FUNCTION get_inferred_diagnosis_state
    ( id_id : inferred_diagnosis ) :
    assertion;
END_FUNCTION;

```

```

FUNCTION get_inferred_diagnosis_confidence
    ( id_id : inferred_diagnosis ) :
    confidence_value;
END_FUNCTION;

```

```

FUNCTION get_inferred_associated_step
    ( id_id : inferred_diagnosis ) :
    step;
END_FUNCTION;

```

The *put* procedures will be given an entity instance identifier and a parameter value to be placed in the entity attribute identified by the procedure name. There is no return from a procedure. The *put* procedures for entity *inferred_diagnosis* are the following: