

ISO TC184/SC4/WG11 N 137

Date: December 9, 2000

Supersedes SC4/WG11 N 076

PRODUCT DATA REPRESENTATION AND EXCHANGE

Part: 35

Title: Conformance Testing Methodology and Framework:
Abstract Test Methods for SDAI Implementations

Purpose of this document as it relates to the target document is:

- Primary Content
- Issue Discussion
- Alternate Proposal
- Partial Content

Current Status: working

ABSTRACT:

This part of ISO 10303 describes test methods for use in the conformance testing of software systems implementing, in one or more language binding, a level of functionality (implementation class) as specified in ISO 10303-22.

COMMENTS TO THE READER:

Since WG11N076 the majority of needed test cases (130) has been added. The SDAI_test_operation_schema has been improved and completed. Further work and checking of the SDAI_test_case_schema is needed. Annex C is added to show how the test-cases may be realized by an SDAI implementation. Nothing is done so far on formatting this document.

KEYWORDS:

Document Status/Dates

SDAI testing, test purpose,
test suite, language binding,
late binding, early binding,,
Data Access Interface

Part Documents

- Working Draft
- Project Draft
- Released Draft
- Technically Complete
- Editorially Complete
- ISO Committee Draft

Other SC4 Documents

- Working
- Released
- Confirmed
- Approved

Owner/Editor: Lothar Klein
Address: LKSoftWare GmbH
Steinweg 1
36093 Kuenzell
Germany

Telephone: +49 661 9339330
FAX: +49 661 9339332
E-Mail: lothar.klein@lksoft.com

Alternate:
Address:

Telephone:
FAX:
E-Mail:

this page intentionally blank

Foreword

The ISO (International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has a right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

Draft International Standards adopted by technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% of the member bodies casting a vote.

International Standard ISO 10303-22 is being prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration*, Subcommittee SC4, *Industrial data and global manufacturing languages*.

ISO 10303 consists of the following parts under the general title *Industrial automation systems and integration - Product data representation and exchange*:

- Part 1, Overview and fundamental principles;
- (to be added)

Should further parts be published, they will follow the same numbering pattern.

The reader may obtain information on the other Parts of ISO 10303 from the ISO Central Secretariat.

Annex A forms an integral part of this part of ISO 10303. Annex B is for information only.

Introduction

ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. The objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product, independent from any particular system. The nature of this description makes it suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases and archiving.

ISO 10303 is organized as a series of parts, each published separately. The parts of this International Standard fall into one of the following series: descriptive methods, integrated resources, application protocols, abstract test suites, implementation methods, and conformance testing. The series are described in ISO 10303-1. This part of ISO 10303 is a member of the conformance testing series.

This part of ISO 10303 specifies test methods used in the conformance testing of software systems implementing the Standard data access interface (SDAI) for one or more language bindings in one of the classes of implementation specified in ISO 10303-22.

Industrial automation systems and integration - Product data representation and exchange - Part 35: Conformance Testing Methodology and Framework: Abstract Test Methods for SDAI implementations

1 Scope

This part of ISO 10303 presents the abstract test methods and requirements for conformance testing of an implementation of a language binding of the SDAI. Since the SDAI is specified independently of any programming language, the abstract test methods presented in this part will be applicable to all language bindings specified for the SDAI. The abstract test methods specified herein will also address early and late bindings to application protocol schemata as well as the various classes of implementation detailed in ISO 10303-22.

The following are within the scope of this document:

- abstract test methods for software systems that implement the SDAI;
- conformance tests for all SDAI operations specified in ISO 10303-22;
- the specification, in a manner that is independent of any binding language¹, of algorithms and approaches for the testing of various SDAI operations;
- the specification and documentation of abstract test cases

The following are outside the scope of this document:

- the development of test data and/or test programs for specific language bindings;
- the specification of test methods, algorithms, or programs for the conformance testing of applications that interact with SDAI implementations;
- implementation policy for a conformance test system that realizes the test methods specified in this part of ISO 10303.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO 10303. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO 10303 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 10303-1 *Industrial automation systems and integration - Product data representation*

¹Binding-specific functions need to be tested as well. Although this part of ISO 10303 is language-independent, the issue of testing such functions needs to be addressed.

ISO TC184/SC4/WG11 N137

and exchange - Part 1: Overview and fundamental principles.

ISO 10303-11	<i>Industrial automation systems and integration - Product data representation and exchange - Part 11: Descriptive methods: The EXPRESS language reference manual.</i>
ISO 10303-21	<i>Industrial automation systems and integration - Product data representation and exchange - Part 21: Implementation methods: Clear text encoding of the exchange structure.</i>
ISO 10303-22	<i>Industrial automation systems and integration - Product data representation and exchange - Part 22: Implementation methods: Standard data access interface.</i>
ISO 10303-31	<i>Industrial automation systems and integration - Product data representation and exchange - Part 31: Conformance testing methodology and framework: General concepts.</i>
ISO 10303-32	<i>Industrial automation systems and integration - Product data representation and exchange - Part 32: Conformance testing methodology and framework: Requirements on testing laboratories and clients.</i>
ISO 10303-33	<i>Industrial automation systems and integration - Product data representation and exchange - Part 33: Conformance testing methodology and framework: Abstract test cases.</i>
ISO 10303-34	<i>Industrial automation systems and integration - Product data representation and exchange - Part 34: Conformance testing methodology and framework: Abstract test methods.</i>

3 Definitions

3.1 Terms defined in ISO 10303-1

This part of ISO 10303 makes use of the following terms defined in ISO 10303-1:

- abstract test suite;
- application protocol;
- conformance class;
- implementation method;
- Protocol Implementation Conformance Statement;
- PICS proforma;

3.2 Terms defined in ISO 10303-22

This part of ISO 10303 makes use of the following terms defined in ISO 10303-22:

- application schema;
- implementation class;
- domain equivalence;
- language binding;
- repository;
- schema instance;
- SDAI-model;
- session;
- validation;

3.3 Terms defined in ISO 10303-31

This part of ISO 10303 makes use of the following terms defined in ISO 10303-31:

- abstract test case;
- abstract test method;
- conformance;
- conformance log;
- conformance report;
- conformance testing;
- executable test case;
- FAIL (verdict);
- implementation under test;
- INCONCLUSIVE (verdict);
- PASS (verdict);
- Protocol Implementation eXtra Information for Testing;
- PIXIT proforma;

- test campaign;
- test case error;
- test laboratory;
- test report;
- (test) verdict;
- verdict criteria;

3.4 Abbreviations

The following abbreviations are used in this part of ISO 10303:

IUT	Implementation Under Test
PICS	Protocol Implementation Conformance Statement
PIXIT	Protocol Implementation eXtra Information for Testing
SDAI	Standard Data Access Interface

4 Requirements and overview

A conformance test system for SDAI implementations will possess the capability of testing a SDAI implementation (the IUT) for its claimed behavior against requirements specified in ISO 10303-22 and companion parts depending on the programming language the IUT supports.

This part of ISO 10303 describes abstract test methods that conformance test systems would implement in order to test SDAI implementations. General principles and an overall framework for conformance testing are provided in ISO 10303-31. Requirements on test laboratories are defined in ISO 10303-32. The methods for preparing, controlling, observing and analyzing implementations during testing are defined in this part of ISO 10303.

4.1 SDAI Testing requirements

These test methods described in this part of ISO 10303 will address the following testing requirements:

- An SDAI implementation may be early bound, late bound, or both. Test methods specified in this part of ISO 10303 address all such implementations.
- An SDAI implementation must obey the state model described in ISO 10303-22. The test methods specified in this part ensure this is tested.
- ISO 10303-22 is written independent of any programming language. The test methods specified for SDAI will therefore be generic.
- ISO 10303-22 specifies various levels of functionality that an SDAI implementation can support.

The test methods specified herein will facilitate the testing of implementations at these various levels or implementation classes.

- SDAI operations provide means for data manipulation and transactions. Testing these operations will provide assurance of their correctness and whether they had the desired effect on the persistent storage. These operations include create, delete, modify, validate and various manipulation operations that act on schema instances, SDAI-models, and instances of application schema entities.
- In situations of error, SDAI operations return error codes. Testing will encompass all reasonable error situations for an operation to ensure that appropriate error codes are returned.
- Dictionary information operations provide subtype/supertype and complex entity information, interface information, and domain equivalence information. In the case of late bound applications, the dictionary is *closed*², and will be tested implicitly.
- Environment and session operations provide the capability for changing the state of the SDAI session. Conformance testing will ascertain that only the permitted transitions take place and that only permitted operations for a state are allowed.
- The testing of error handling requires checking the returned error value against the actual error condition which triggered that return value.
- Aggregate operations play important roles in SDAI implementations, facilitating the successful completion of other complex operations or sequences of operations.
- Verdicts must be issued for all test cases executed.
- Test logs and reports that accurately capture the results of all test cases executed must be generated.
- Test campaigns with potentially large numbers of test cases will have to be managed, sequenced, and run.

4.2 SDAI implementation types

This clause describes the nature of SDAI implementations that are addressed by the test methods specified in this part. SDAI implementations can be of two types - implementations that accept requests encoded in the binding language of the implementation and respond to these requests in the manner required by the standard, and applications that make a series of these requests to achieve a meaningful objective.

- A SDAI implementation encapsulates repositories and presents an interface for the manipulation of these repositories and schema instances. This is the interface which applications would use to interact with SDAI implementations.
- Application programs use the SDAI interface to communicate with a SDAI implementation to perform their tasks. These tasks typically require access to repositories and schema instances. The SDAI implementation provides this access via the interface.

²This needs to be described in more detail.

This part of ISO 10303 presents test methods for the testing of implementations of the former kind. Test methods for applications that access SDAI implementations are not presented in this part.

4.3 Testing architecture

The architecture assumed by the test methods for SDAI implementations is a simple *client-server*³ relationship between the SDAI implementation (IUT) and the test engine. The test application, which is a part of the test engine, plays the role of an application by making SDAI operation requests to the IUT. The IUT returns output back to the test application. A simplified description is presented in Figure 1.

The test application processes the outputs and, in turn, communicates appropriate values to the test engine for recording.

The test application uses test data as appropriate to execute its test cases.

³The use of the term *client-server* has been viewed by some members of WG6 and 7 as carrying too much meaning, and thereby subject to misinterpretation by readers. It is still here since the editor of this part has not found a sufficiently high-quality substitute.

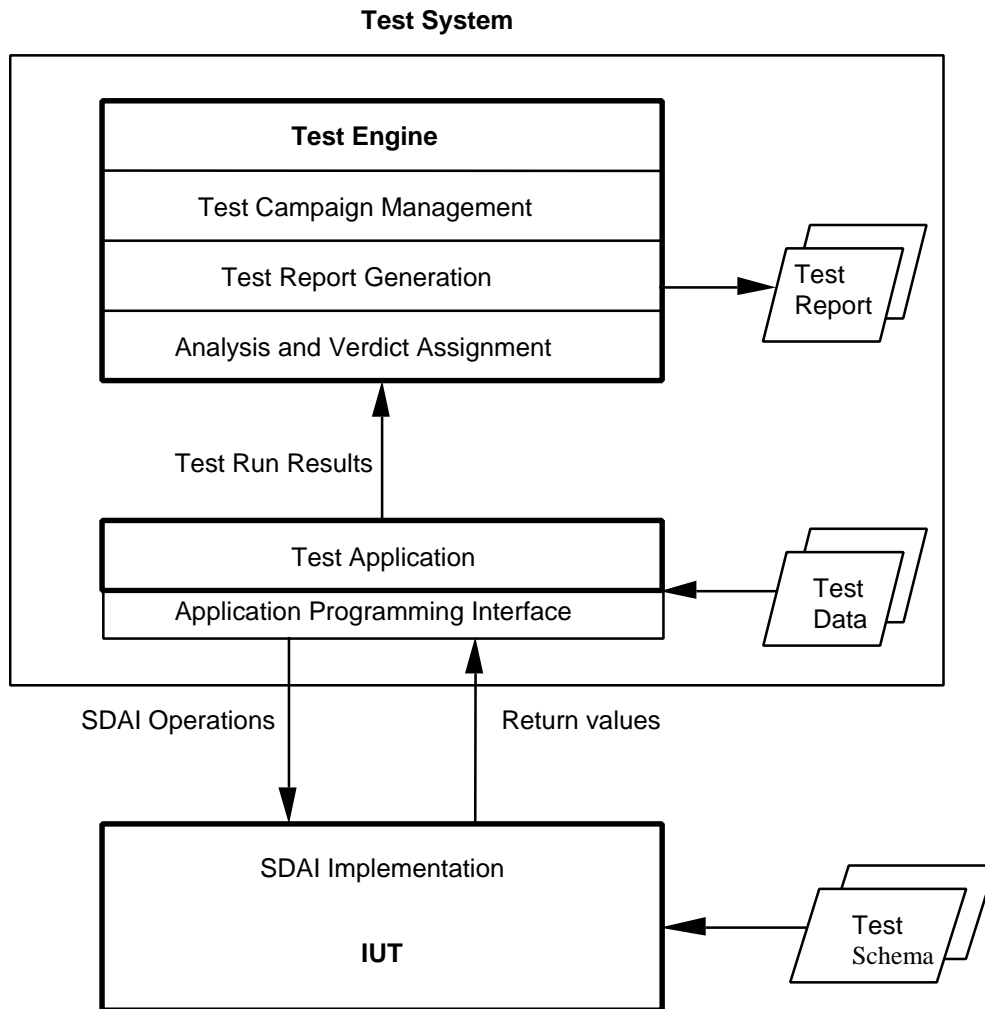


Figure 1 Testing architecture for SDAI implementations

5 Testing process

5.1 Preparation for testing

PICS and PIXIT proformas are completed by IUT vendors prior to testing. The information in these forms is used to guide test case selection, sequencing, and input data to be used in the execution of the test campaign.

Abstract test cases are built around SDAI operations specified in ISO 10303-22. Since there exists no guaranteed one-to-one relationship between operations in ISO 10303-22 and a particular language binding, executable test cases are developed to logically match these operations so that the executable test cases can be grouped according to implementation level.

ISO TC184/SC4/WG11 N137

Note: This part of ISO 10303 may also need to specify detailed guidelines for developing and documenting abstract test cases.

Executable test cases are built from the abstract test cases using PICS and PIXIT information for an IUT. The sequence of their execution in a test campaign is determined at this stage.

Note: In assembling executable tests for an IUT, of particular importance is information about the IUT's implementation class, whether it is early or late-bound, the application schema it is bound to (if early bound), and the programming language of choice.

For a further description of conformance testing preparations from the perspective of the testing laboratory, see ISO 10303-32.

5.2 Test campaign

A conformance test campaign is a sequenced execution of all required executable test cases (ETCs). The results of this sequenced execution determines conformance.

Modifications to the IUT are not permitted during a test campaign. Modifications to the ETCs or to the sequence of their execution is not permitted during a test campaign, except in the situation where the ETC is determined to be in error.

If an ETC is determined to be in error, a verdict of INCONCLUSIVE shall be assigned to its execution until the error is resolved and the test repeated.

5.3 Test conclusion

A test campaign may terminate for any reason. A normal termination of a test campaign occurs when all its executable tests have been run. A PASS verdict is assigned to a campaign if all the tests of the campaign have returned PASS verdicts and no violation of any ISO 10303 part is detected.

5.4 Test report production

A conformance test report is created after a test campaign terminates. As a minimum, this report shall contain information identifying the IUT and the capability that is being tested (late/early binding, SDAI implementation class(es) supported, binding language, AP or application schema information, etc.) and a detailed reports and verdicts on all the executable tests in the campaign including a list of those not executed, if any. Any relevant information on the testing environment will be included as well.

6 Test method - early-bound implementation

An early bound SDAI implementation provides an interface specific to the application schema to which it is bound at compile-time. The test application that will have to be written to test an early bound IUT will therefore depend on this application schema.

The set of early bound functions that need to be tested can be classified in two broad groups: those that are schema-specific and those that are schema-independent.

The schema-independent functions can be tested in a uniform manner regardless of the application

schema that is bound to the IUT. Test cases for the schema-specific functions will have to be developed for each application schema that may be early-bound to an IUT. The testing of schema-specific functions will typically require the use of schema-independent functions. Therefore, an efficient and realistic approach to testing would combine the two rather than test them in separate groups.

Note: Typically for an AP, a large number of schema-specific functions will be generated. WG11 will have to take a decision on whether it is necessary to exercise all these functions, or whether, an appropriately selected subset would suffice.

7 Test method - late-bound implementation

A late bound SDAI implementation provides a single generic interface, which may be used with any application schema. Nothing more can be said about the implementation since the implementor has complete freedom in adopting an implementation approach. In an extreme case, a late-bound implementation for a particular application schema could be early-bound, with an interface that mimics the interface of a late-bound implementation.

Therefore, the testing of late-bound implementations would also have to be in the context of a specific application schema, usually an AP. Testing would progress in a manner similar to the testing of early-bound implementations⁴.

8 Test Application

Test purposes are statements of requirements that are used in the development of abstract test cases. They are derived from two sources: ISO 10303-22, and the application schema(s) for which the implementation is being tested. For ISO 10303-22, they are derived from ISO 10303-22, clause 10 (operation test purposes) and clause 12 (state model test purposes). Each test case satisfies one or more test purposes.

Operation test purposes, derived from clause 10 of ISO 10303-22, specify the execution of operations in normal and under error conditions. For this purpose the SDAI operations are mapped into functions and procedures of the SDAI_test_operation_schema, see Annex A. In addition to the parameters specified in ISO 10303-22:10 parameters for the error code and error base are added. The Test Application shall implement this test-functions and procedures for the particular implementation and binding in such a way that the corresponding SDAI operation is invoked and the specified error is checked.

Operation test purposes do not contain the error indicators SS_NAVL, SY_ERR, and TR_EAB (see ISO 10303-22, clause 11).

Examples

1 - The test purpose

Open session

is an operation test purpose which requires that the operation open session be executed normally.

⁴How would an implementation advertising a general late-binding capability be tested? Would it have to be tested for each application schema they becomes available? Hopefully not!

ISO TC184/SC4/WG11 N137

2 - The test purpose

Open session terminating with SS_OPN.

is an operation test purpose which requires that the operation be executed such that the error indicator SS_OPN is returned.

3 - Although the error indicator SS_NAVL is a possible error indicator for this operation,

Open session terminating with SS_NAVL.

is not a valid test purpose.

9 Abstract Test Cases

This clause specifies abstract test cases for SDAI implementations. Each abstract test case addresses one or more test purposes from clause 8. All the test purposes and verdict criteria associated with the test case are explicitly referenced in the test case.

Each operation of a test case is specified on a numbered line. Verdicted operations have parenthesized verdict criteria next to them. If an operation needs an input, it is specified on the next line.

```
SCHEMA SDAI_test_cases_schema;  
USE SDAI_test_operation_schema;  
USE FROM SDAI_session_schema;  
USE FROM greek;
```

In the following test cases attributes of the session entities `sdai_session`, `sdai_transaction`, `implementation`, `sdai_repository`, `sdai_repository_contents`, `sdai_model`, `sdai_model_contents`, `entity_extent` and `schema_instance` will be accessed by usual EXPRESS expressions in read-only mode.

```
(*  
Testing the SDAI operation open session.  
*)  
PROCEDURE test_open_session();  
  LOCAL  
    session1 : sdai_session;  
    session2 : sdai_session;  
  END_LOCAL;  
  
  (* Ensures that open_session works correctly when SDAI session is closed.  
  *)  
  session1 := open_session(NO_ERROR, ?);  
  
  (* Ensures that open_session reports an SS_OPN error when SDAI session is  
  open. *)  
  session2 := open_session(SS_OPN, session1);  
  
END_PROCEDURE;
```

```
(*  
This test case prints the implementation information.  
An example of the printout:  
name = SDAI MULTIPLE  
level = Version 2.0.1 (Build 197, 2000-05-24)  
*)
```

```

sdai_version = { iso standard 10303 part(22) version(0) }
binding_version = { iso standard 10303 part(27) version(0) }
implementation_class = 5
transaction_level = 3
expression_level = 1
domain_equivalence_level = 2

```

Parameter:

session - an open SDAI session.

*)

```
PROCEDURE test_implementation(session:sdai_session);
```

```
  LOCAL
```

```
    imp : implementation := session.implementation;
```

```
  END_LOCAL;
```

```
  print('name = ' + imp.name);
```

```
  print('level = ' + imp.level);
```

```
  print('sdai_version = ' + imp.sdai_version);
```

```
  print('binding_version = ' + imp.binding_version);
```

```
  print('implementation_class = ' + format(imp.implementation_class, ''));
```

```
  print('transaction_level = ' + format(imp.transaction_level, ''));
```

```
  print('expression_level = ' + format(imp.expression_level, ''));
```

```
  print('recording_level = ' + format(imp.recording_level, ''));
```

```
  print('scope_level = ' + format(imp.scope_level, ''));
```

```
  print('domain_equivalence_level = ' +
```

```
format(imp.domain_equivalence_level, ''));
```

```
END_PROCEDURE;
```

(*

Testing the SDAI operation close session.

Parameter:

repo - an open repository (that is, from the active_servers set of the session).

*)

```
PROCEDURE test_close_session(repo:sdai_repository);
```

```
  LOCAL
```

```
    session : sdai_session := repo.session;
```

```
  END_LOCAL;
```

```
  (* Ensures that close_session works correctly when session is open. *)
```

```
  close_session(session, NO_ERROR, ?);
```

```
  (* Ensures that repositories are not available after session was closed. *)
```

```
  open_repository(session, repo, RP_NEXS, ?);
```

```
  (* Ensures that close_session reports an SS_NOPN error when SDAI session is closed. *)
```

```
  close_session(session, SS_NOPN, ?);
```

```
END_PROCEDURE;
```

(*

Testing the SDAI operation start transaction read-only access.

Parameter:

session - an open SDAI session.

*)

```
PROCEDURE test_start_transaction_read_only_access(session:sdai_session);
```

```
  LOCAL
```

ISO TC184/SC4/WG11 N137

```
        transaction : sdai_transaction;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that start_transaction_read_only_access works correctly when
    session
        is open and transaction does not exist.
    *)
    transaction := start_transaction_read_only_access(session, NO_ERROR, ?);

    (* Ensures that started transaction belongs to the active_transaction set
    of the session. *)
    bool := is_member(session.active_transaction, transaction, NO_ERROR, ?);
    assert(bool);

    (* Ensures that start_transaction_read_only_access reports a TR_EXS error
    when transaction is
        already started.
    *)
    transaction := start_transaction_read_only_access(session, TR_EXS,
    transaction);

END_PROCEDURE;

(*
Testing the SDAI operation open repository.
Parameter:
repo - a closed repository (that is, from the known_servers but not
active_servers
set of the session).
*)
PROCEDURE test_open_repository(repo:sdai_repository);
    LOCAL
        session : sdai_session := repo.session;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that open_repository works correctly when parameters are
    correct. *)
    open_repository(session, repo, NO_ERROR, ?);

    (* Ensures that opened repository belongs to the active_servers set of the
    session. *)
    bool := is_member(session.active_servers, repo, NO_ERROR, ?);
    assert(bool);

    (* Ensures that open_repository reports an RP_OPN error when repository is
    already open. *)
    open_repository(session, repo, RP_OPN, repo);

END_PROCEDURE;

(*
Testing the SDAI operation create SDAI-model.
Parameter:
repo - a closed repository (that is, from the known_servers but not
active_servers
set of the session).
*)
```



```

PROCEDURE test_create_sdai_model(repo:sdai_repository);
  LOCAL
    session : sdai_session := repo.session;
    transaction : sdai_transaction :=
start_transaction_read_only_access(session, NO_ERROR, ?);
    model : sdai_model;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that create_sdai_model reports a TR_NRW error when transaction
is not read-write. *)
  model := create_sdai_model(repo, 'exceptional_name_within_repo', 'GREEK',
TR_NRW, transaction);

  (* Starting transaction read-write access. *)
  end_transaction_access_and_abort(transaction, NO_ERROR, ?);
  transaction := start_transaction_read_write_access(session, NO_ERROR, ?);

  (* Ensures that create_sdai_model reports an RP_NOPN error when repository
within which
  an SDAI-model is to be created is closed. *)
  model := create_sdai_model(repo, 'exceptional_name_within_repo', 'GREEK',
RP_NOPN, repo);

  (* Opening the repository. *)
  open_repository(session, repo, NO_ERROR, ?);

  (* Ensures that create_sdai_model reports a VA_NSET error when the name of
an SDAI-model is
  not submitted.
  *)
  model := create_sdai_model(repo, ?, 'GREEK', VA_NSET, ?);

  (* Ensures that create_sdai_model reports an SD_NDEF error when schema
definition is
  not submitted.
  *)
  model := create_sdai_model(repo, 'exceptional_name_within_repo', ?,
SD_NDEF, ?);

  (* Ensures that create_sdai_model works correctly when all parameters are
correct. *)
  model := create_sdai_model(repo, 'exceptional_name_within_repo', 'GREEK',
NO_ERROR, ?);

  (* Ensures that created SDAI-model has no access mode. *)
  end_read_only_access(model, MX_NDEF, model);

  (* Ensures that a created SDAI-model belongs to the models set of the
sdai_repository_contents. *)
  bool := is_member(repo.contents.models, model, NO_ERROR, ?);
  assert(bool);

  (* Ensures that create_sdai_model reports a MO_DUP error when an SDAI-model
with the
  provided name already exists.
  *)
  model := create_sdai_model(repo, 'exceptional_name_within_repo', 'GREEK',
MO_DUP, model);

END_PROCEDURE;

```

ISO TC184/SC4/WG11 N137

```
(*
Testing the SDAI operation create schema instance.
Parameter:
repo - an open repository (that is, from the active_servers set of the
session).
*)
PROCEDURE test_create_schema_instance(repo:sdai_repository);
  LOCAL
    schema_inst : schema_instance;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that create_schema_instance reports a VA_NSET error when the
name of the
  schema instance to be created is not submitted.
  *)
  schema_inst := create_schema_instance(repo, ?, 'GREEK', VA_NSET, ?);

  (* Ensures that create_schema_instance reports an SD_NDEF error when schema
definition
  is not submitted.
  *)
  schema_inst := create_schema_instance(repo, 'exceptional_name_within_repo',
?, SD_NDEF, ?);

  (* Ensures that create_schema_instance works correctly when parameters are
correct. *)
  schema_inst := create_schema_instance(repo, 'exceptional_name_within_repo',
'GREEK', NO_ERROR, ?);

  (* Ensures that a created schema instance belongs to the schemas set of the
sdai_repository_contents. *)
  bool := is_member(repo.contents.schemas, schema_inst, NO_ERROR, ?);
  assert(bool);

  (* Ensures that create_schema_instance reports an SI_DUP error when a
schema instance with the
  provided name already exists.
  *)
  schema_inst := create_schema_instance(repo, 'exceptional_name_within_repo',
'GREEK', SI_DUP, schema_inst);

END_PROCEDURE;

(*
Testing the SDAI operation close repository.
Parameter:
repo - an open repository (that is, from the active_servers set of the
session).
*)
PROCEDURE test_close_repository(repo:sdai_repository);
  LOCAL
    transaction : sdai_transaction := repo.session.active_transaction[1];
    model : sdai_model := create_sdai_model(repo,
'exceptional_name_within_repo', 'GREEK', NO_ERROR, ?);
    inst : GENERIC_ENTITY;
    bool : BOOLEAN;
  END_LOCAL;
```

```

(* Ensures that close_repository reports a TR_RW error when some SDAI-model
  within the repository has been created.
*)
close_repository(repo, TR_RW, transaction);

(* Making persistent all changes in open repositories. *)
start_read_write_access(model, NO_ERROR, ?);
commit(transaction, NO_ERROR, ?);

(* Ensures that close_repository works correctly when all changes are
  resolved. *)
close_repository(repo, NO_ERROR, ?);

(* Ensures that closed repository does not belong to the active_servers
  set. *)
bool := is_member(repo.session.active_servers, repo, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that an SDAI-model in a closed repository does not belong to the
  active_models set. *)
bool := is_member(repo.session.active_models, model, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that SDAI-models in a closed repository are no longer
  accessible. *)
inst := create_entity_instance('GREEK.ALPHA', model, RP_NOPN, repo);

(* Ensures that close_repository reports an RP_NOPN error when repository
  is closed. *)
close_repository(repo, RP_NOPN, repo);

END_PROCEDURE;

(*
Testing the SDAI operation add SDAI-model.
Parameter:
schema_inst - a schema instance whose native schema is the greek schema.
*)
PROCEDURE test_add_sdai_model(schema_inst: schema_instance);
  LOCAL
    model : sdai_model := create_sdai_model(schema_inst.repository,
      'exceptional_name_within_repo',
      'GREEK', NO_ERROR, ?);
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that add_sdai_model reports a VA_NSET error when SDAI-model to
  be added is not submitted. *)
  add_sdai_model(schema_inst, ?, VA_NSET, ?);

  (* Ensures that add_sdai_model works correctly when parameters are correct.
  *)
  add_sdai_model(schema_inst, model, NO_ERROR, ?);

  (* Ensures that an added SDAI-model belongs to associated_models set of the
  schema instance. *)
  bool := is_member(schema_inst.associated_models, model, NO_ERROR, ?);
  assert(bool);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that schema instance belongs to associated_with set of the SDAI
model added. *)
bool := is_member(model.associated_with, schema_inst, NO_ERROR, ?);
assert(bool);
```

```
END_PROCEDURE;
```

```
(*
Testing the SDAI operation remove SDAI-model.
Parameter:
schema_inst - a schema instance whose native schema is the greek schema.
*)
PROCEDURE test_remove_sdai_model(schema_inst:schema_instance);
LOCAL
    model : sdai_model := create_sdai_model(schema_inst.repository,
'exceptional_name_within_repo',
    'GREEK', NO_ERROR, ?);
    bool : BOOLEAN;
END_LOCAL;

(* Initially, some SDAI-model shall be associated with a given schema
instance. *)
add_sdai_model(schema_inst, model, NO_ERROR, ?);

(* Ensures that remove_sdai_model reports a VA_NSET error when SDAI-model
to be removed is not submitted. *)
remove_sdai_model(schema_inst, ?, VA_NSET, ?);

(* Ensures that remove_sdai_model works correctly when parameters are
correct. *)
remove_sdai_model(schema_inst, model, NO_ERROR, ?);

(* Ensures that the removed SDAI-model does not belong to associated_models
set of the schema instance. *)
bool := is_member(schema_inst.associated_models, model, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that schema instance does not belong to associated_with set of
the SDAI-model removed. *)
bool := is_member(model.associated_with, schema_inst, NO_ERROR, ?);
assert(bool);

END_PROCEDURE;
```

```
(*
Testing the SDAI operation rename schema instance.
Parameter:
schema_inst - a schema instance whose native schema is the greek schema.
*)
PROCEDURE test_rename_schema_instance(schema_inst:schema_instance);
LOCAL
    schema_inst_created : schema_instance;
    bool : BOOLEAN;
END_LOCAL;

(* Ensures that rename_schema_instance reports a VA_NSET error when a new
name is not submitted. *)
rename_schema_instance(schema_inst, ?, VA_NSET, ?);
```

```

    (* Ensures that rename_schema_instance reports a SI_DUP error when schema
instance with a
    submitted name already exists in the repository.
    *)
    schema_inst_created := create_schema_instance(schema_inst.repository,
'exceptional_name_within_repo',
    'GREEK', NO_ERROR, ?);
    rename_schema_instance(schema_inst, 'exceptional_name_within_repo', SI_DUP,
schema_inst_created);

    (* Ensures that rename_schema_instance works correctly when the name
provided is
    different from the names of other schema instances in the same
repository.
    *)
    rename_schema_instance(schema_inst, 'another_exceptional_name_within_repo',
NO_ERROR, ?);

    (* Ensures that schema instance renamed stays in schemas set of the
sdai_repository_contents. *)
    bool := is_member(schema_inst.repository.contents.schemas, schema_inst,
NO_ERROR, ?);
    assert(bool);

END_PROCEDURE;

(*
Testing the SDAI operation delete schema instance.
Parameter:
schema_inst - a schema instance whose native schema is the greek schema.
*)
PROCEDURE test_delete_schema_instance(schema_inst:schema_instance);
    LOCAL
        modell : sdai_model := create_sdai_model(schema_inst.repository,
'exceptional_name_within_repo',
            'GREEK', NO_ERROR, ?);
        model2 : sdai_model := create_sdai_model(schema_inst.repository,
'another_exceptional_name_within_repo',
            'GREEK', NO_ERROR, ?);
        inst1 : GENERIC_ENTITY;
        inst2 : GENERIC_ENTITY;
        bool : BOOLEAN;
    END_LOCAL;

    (* Initially, two created SDAI-models are added to a given schema instance.
A reference between these models is also established.
    *)
    add_sdai_model(schema_inst, modell, NO_ERROR, ?);
    add_sdai_model(schema_inst, model2, NO_ERROR, ?);
    inst1 := create_entity_instance('GREEK.OMEGA', modell, NO_ERROR, ?);
    inst2 := create_entity_instance('GREEK.IOTA', model2, NO_ERROR, ?);
    put_attribute(inst1, 'GREEK.OMEGA.O0', inst2, NO_ERROR, ?);

    (* Ensures that delete_schema_instance works correctly. *)
    delete_schema_instance(schema_inst, NO_ERROR, ?);

    (* Ensures that schema instance deleted does not belong to the schemas set
of the
    sdai_repository_contents.
    *)

```

ISO TC184/SC4/WG11 N137

```
bool := is_member(schema_inst.repository.contents.schemas, schema_inst,
NO_ERROR, ?);
assert(NOT bool);

(* Ensures that schema instance, that was deleted, does not belong to
associated_with
set of any SDAI-model.
*)
bool := is_member(modell.associated_with, schema_inst, NO_ERROR, ?);
assert(bool);

(* Ensures that references between two SDAI-models, which were associated
with a
schema instance now deleted, are invalid.
*)
inst2 := get_attribute(inst1, 'GREEK.OMEGA.O0', VA_NVLD, ?);

END_PROCEDURE;

(*
Testing the SDAI operation start read-only access.
Parameter:
model - an SDAI-model with unset mode, based on the greek schema.
*)
PROCEDURE test_start_read_only_access(model:sdai_model);
LOCAL
bool : BOOLEAN;
END_LOCAL;

(* Ensures that start_read_only_access works correctly when SDAI-model
access is not started. *)
start_read_only_access(model, NO_ERROR, ?);

(* Ensures that an SDAI-model with access started belongs to active_models
set of the SDAI session. *)
bool := is_member(model.repository.session.active_models, model, NO_ERROR,
?);
assert(bool);

(* Ensures that start_read_only_access reports an MX_RO error when SDAI-
model is
already in read-only mode. Also this means that previous invocation of
start_read_only_access
correctly started the access of the SDAI-model.
*)
start_read_only_access(model, MX_RO, model);

(* Ensures that start_read_only_access reports an MX_RW error when an SDAI-
model is in read-write mode. *)
promote_sdai_model_to_read_write(model, NO_ERROR, ?);
start_read_only_access(model, MX_RW, model);

END_PROCEDURE;

(*
Testing the SDAI operation promote SDAI-model to read-write.
Parameter:
model - an SDAI-model with unset mode, based on the greek schema.
*)
```

```

PROCEDURE test_promote_sdai_model_to_read_write(model:sdai_model);
  LOCAL
    model_dict : sdai_model;
  END_LOCAL;

  (* Ensures that promote_sdai_model_to_read_write reports an MX_NDEF error
when SDAI-model
  access is not started.
  *)
  promote_sdai_model_to_read_write(model, MX_NDEF, model);

  (* Preparing SDAI-model for promote_sdai_model_to_read_write operation. *)
  start_read_only_access(model, NO_ERROR, ?);

  (* Ensures that promote_sdai_model_to_read_write works correctly when SDAI
model is in read-only mode. *)
  promote_sdai_model_to_read_write(model, NO_ERROR, ?);

  (* Ensures that promote_sdai_model_to_read_write reports an MX_RW error
when an SDAI-model is
  in read-write mode. Also this means that previous invocation of
  promote_sdai_model_to_read_write
  correctly established the access of the SDAI-model.
  *)
  promote_sdai_model_to_read_write(model, MX_RW, model);

  (* Ensures that promote_sdai_model_to_read_write reports an FN_NAVL error
when a data dictionary
  SDAI-model for promoting is submitted.
  *)
  model_dict := macro_get_data_dictionary_model();
  promote_sdai_model_to_read_write(model_dict, FN_NAVL, ?);

END_PROCEDURE;

```

```

(*
This macro shall return a data dictionary model.
*)
FUNCTION macro_get_data_dictionary_model() : sdai_model;
...
END_FUNCTION;

```

```

(*
Testing the SDAI operation end read-only access.
Parameter:
model - an SDAI-model in read-only mode, based on the greek schema.
*)
PROCEDURE test_end_read_only_access(model:sdai_model);
  LOCAL
    model_dict : sdai_model;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that end_read_only_access works correctly when SDAI-model is in
read-only mode. *)
  end_read_only_access(model, NO_ERROR, ?);

  (* Ensures that an SDAI-model with access ended does not belong to
active_models set of the SDAI session. *)

```

ISO TC184/SC4/WG11 N137

```
bool := is_member(model.repository.session.active_models, model, NO_ERROR,
?);
assert(NOT bool);

(* Ensures that end_read_only_access reports an MX_NDEF error when SDAI-
model
access is not started. Also this means that previous invocation of
end_read_only_access
correctly ended access of the SDAI-model.
*)
end_read_only_access(model, MX_NDEF, model);

(* Ensures that end_read_only_access reports an MX_RW error when an SDAI-
model is
in read-write mode.
*)
start_read_write_access(model, NO_ERROR, ?);
end_read_only_access(model, MX_RW, model);

(* Ensures that end_read_only_access reports an FN_NAVL error when a data
dictionary
SDAI-model for ending its access is submitted.
*)
model_dict := macro_get_data_dictionary_model();
end_read_only_access(model_dict, FN_NAVL, ?);

END_PROCEDURE;

(*
Testing the SDAI operation start read-write access.
Parameter:
model - an SDAI-model with unset mode, based on the greek schema.
*)
PROCEDURE test_start_read_write_access(model:sdai_model);
LOCAL
model_dict : sdai_model;
bool : BOOLEAN;
END_LOCAL;

(* Ensures that start_read_write_access works correctly when SDAI-model
access is not started. *)
start_read_write_access(model, NO_ERROR, ?);

(* Ensures that an SDAI-model with access started belongs to active_models
set of the SDAI session. *)
bool := is_member(model.repository.session.active_models, model, NO_ERROR,
?);
assert(bool);

(* Ensures that start_read_write_access reports an MX_RW error when SDAI-
model is
already in read-write mode. Also this means that previous invocation of
start_read_write_access
correctly started the access of the SDAI-model.
*)
start_read_write_access(model, MX_RW, model);

(* Ensures that start_read_write_access reports an MX_RO error when an
SDAI-model is
in read-only mode.
```



```

*)
end_read_write_access(model, NO_ERROR, ?);
start_read_only_access(model, NO_ERROR, ?);
start_read_write_access(model, MX_RO, model);

(* Ensures that start_read_write_access reports an FN_NAVL error when a
data dictionary
SDAI-model for its starting in read-write mode is submitted.
*)
model_dict := macro_get_data_dictionary_model();
start_read_write_access(model_dict, FN_NAVL, ?);

END_PROCEDURE;

(*
Testing the SDAI operation end read-write access.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_end_read_write_access(model:sdai_model);
LOCAL
transaction : sdai_transaction :=
model.repository.session.active_transaction[1];
inst1 : GENERIC_ENTITY;
inst2 : GENERIC_ENTITY;
bool : BOOLEAN;
END_LOCAL;

(* Ensures that end_read_write_access reports an TR_RW error when some
application
instance within the SDAI-model has been created.
*)
inst1 := create_entity_instance('GREEK.OMEGA', model, NO_ERROR, ?);
end_read_write_access(model, TR_RW, transaction);

(* Ensures that end_read_write_access reports an TR_RW error when some
application
instance within the SDAI-model has been modified.
*)
inst2 := create_entity_instance('GREEK.IOTA', model, NO_ERROR, ?);
commit(transaction, NO_ERROR, ?);
put_attribute(inst1, 'GREEK.OMEGA.OO', inst2, NO_ERROR, ?);
end_read_write_access(model, TR_RW, transaction);

(* Ensures that end_read_write_access reports an TR_RW error when some
application
instance within the SDAI-model has been deleted.
*)
commit(transaction, NO_ERROR, ?);
delete_application_instance(inst1, NO_ERROR, ?);
end_read_write_access(model, TR_RW, transaction);

(* Ensures that end_read_write_access works correctly when all changes
within the
SDAI-model are resolved.
*)
commit(transaction, NO_ERROR, ?);
end_read_write_access(model, NO_ERROR, ?);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that an SDAI-model with access ended does not belong to
active_models set of the SDAI session. *)
bool := is_member(model.repository.session.active_models, model, NO_ERROR,
?);
assert(NOT bool);

(* Ensures that end_read_write_access reports an MX_NDEF error when SDAI-
model
access is not started. Also this means that previous invocation of
end_read_write_access
correctly ended access of the SDAI-model.
*)
end_read_write_access(model, MX_NDEF, model);

(* Ensures that end_read_write_access reports an MX_RO error when the SDAI-
model is
in read-only mode.
*)
start_read_only_access(model, NO_ERROR, ?);
end_read_write_access(model, MX_RO, model);

END_PROCEDURE;

(*
Testing the SDAI operation delete SDAI-model.
Parameters:
modell - an SDAI-model in read-write mode, based on the greek schema;
model2 - an SDAI-model in read-write mode, based on the greek schema and
associated with a schema_instance whose native schema is greek schema; this
model
shall be different from modell and may even belong to a different repository.
*)
PROCEDURE test_delete_SDAI_model(modell:sdai_model; model2:sdai_model);
LOCAL
repo : sdai_repository := model2.repository;
schema_inst : schema_instance := model2.associated_with[1];
model_dict : sdai_model;
inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', modell,
NO_ERROR, ?);
inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.IOTA', model2,
NO_ERROR, ?);
bool : BOOLEAN;
END_LOCAL;

(* An instance in one SDAI-model references an instance in another SDAI-
model which will be deleted*)
put_attribute(inst1, 'GREEK.OMEGA.O0', inst2, NO_ERROR, ?);

(* Ensures that delete_sdai_model works correctly. *)
delete_sdai_model(model2, NO_ERROR, ?);

(* Ensures that deleted SDAI-model does not belong to models set of the
sdai_repository_contents. *)
bool := is_member(repo.contents.models, model2, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that deleted SDAI-model does not belong to active_models set of
the SDAI session. *)
bool := is_member(repo.session.active_models, model2, NO_ERROR, ?);
assert(NOT bool);
```

```

    (* Ensures that deleted SDAI-model does not belong to associated_models set
of the schema instance
    to which this SDAI-model earlier was added.
    *)
    bool := is_member(schema_inst.associated_models, model2, NO_ERROR, ?);
    assert(NOT bool);

    (* Ensures that references to instances of the deleted model from other
models become unset. *)
    inst2 := get_attribute(inst1, 'GREEK.OMEGA.O0', VA_NSET, ?);

    (* Ensures that delete_sdai_model reports a VT_NVLD error when a data
dictionary SDAI-model for
    deletion is submitted.
    *)
    model_dict := macro_get_data_dictionary_model();
    delete_sdai_model(model_dict, VT_NVLD, ?);

END_PROCEDURE;

(*
Testing the SDAI operation rename SDAI-model.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_rename_SDAI_model(model:sdai_model);
    LOCAL
        repo : sdai_repository := model.repository;
        model_created : sdai_model :=
            create_sdai_model(repo, 'exceptional_name_within_repo', 'GREEK',
NO_ERROR, ?);
        model_dict : sdai_model;
        inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
        inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.IOTA',
model_created, NO_ERROR, ?);
    END_LOCAL;

    (* An instance in one SDAI-model, which will be renamed, references an
instance in another SDAI-model*)
    put_attribute(inst1, 'GREEK.OMEGA.O0', inst2, NO_ERROR, ?);

    (* Ensures that rename_sdai_model reports a VA_NSET error when a new name
is not submitted. *)
    rename_sdai_model(model, ?, VA_NSET, ?);

    (* Ensures that rename_sdai_model reports an MO_DUP error when the new name
submitted coincides with
    that for some other model in the same repository.
    *)
    rename_sdai_model(model, 'exceptional_name_within_repo', MO_DUP, model);

    (* Ensures that rename_sdai_model works correctly when parameters are
correct. *)
    rename_sdai_model(model, 'another_exceptional_name_within_repo', NO_ERROR,
?);

    (* Ensures that references from the renamed SDAI-model to other SDAI-models
are retained. *)

```

ISO TC184/SC4/WG11 N137

```
inst2 := get_attribute(inst1, 'GREEK.OMEGA.O0', NO_ERROR, ?);

(* Ensures that rename_sdai_model reports a VT_NVLD error when a data
dictionary SDAI-model for
renaming is submitted.
*)
model_dict := macro_get_data_dictionary_model();
rename_sdai_model(model_dict, 'exceptional_name', VT_NVLD, ?);

END_PROCEDURE;

(*
Testing the SDAI operation get entity definition.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_get_entity_definition(model:sdai_model);
LOCAL
def : entity_definition;
END_LOCAL;

(* Ensures that get_entity_definition reports a VA_NSET error when entity
name is not submitted. *)
def := get_entity_definition(model, ?, VA_NSET, ?);

(* Ensures that get_entity_definition reports an ED_NDEF error when entity
name is not defined
or declared in the schema which is underlying for the specified model.
*)
def := get_entity_definition(model, 'EKS', ED_NDEF, ?);

(* Ensures that get_entity_definition works correctly when all parameters
are correct. *)
def := get_entity_definition(model, 'OMEGA', NO_ERROR, ?);
END_PROCEDURE;

(*
Testing the SDAI operation create entity instance.
Parameter:
repo - an open SDAI repository.
*)
PROCEDURE test_create_entity_instance(repo:sdai_repository);
LOCAL
model : sdai_model :=
create_sdai_model(repo, 'exceptional_name_within_repo', 'GREEK',
NO_ERROR, ?);
inst : GENERIC_ENTITY;
extent : entity_extent;
label : STRING;
bool : BOOLEAN;
END_LOCAL;

(* Ensures that create_entity_instance reports an ED_NDEF error when entity
definition is not submitted. *)
inst := create_entity_instance(?, model, ED_NDEF, ?);

(* Ensures that create_entity_instance reports an ED_NVLD error when
entity,
instance of which is to be created, is not defined
```

```

    or declared in the schema which is underlying for the specified model.
*)
inst := create_entity_instance('EKS', model, ED_NVLD, ?);

(* Ensures that create_entity_instance works correctly when parameters are
correct. *)
inst := create_entity_instance('OMEGA', model, NO_ERROR, ?);

(* Ensures that instance created belongs to instances set of the
sdai_model_contents. *)
bool := is_member(model.contents.instances, inst, NO_ERROR, ?);
assert(bool);

(* Ensures that entity extent containing created instance belongs to
populated_folders set
of the sdai_model_contents.
*)
extent := macro_get_entity_extent('OMEGA');
bool := macro_check_extent_if_populated(extent, model);
assert(bool);

(* Ensures that values of attributes of the created instance are unset. *)
bool := macro_check_instance_if_values_unset(inst);
assert(bool);

(* Ensures that a persistent label for the created instance is unique
within the
repository to which this instance belongs.
*)
label := get_persistent_label(inst, NO_ERROR, ?);
delete_application_instance(inst, NO_ERROR, ?);
inst := get_session_identifier(label, repo, EI_NEXS, ?);
END_PROCEDURE;

(* This macro shall return entity extent for the entity definition submitted.
*)
FUNCTION macro_get_entity_extent(def:entity_definition) : entity_extent;
...
END_FUNCTION;

(* This macro shall return 'true' if entity extent submitted belongs
to the 'populated_folders' set of the contents of the SDAI-model specified
by the second parameter; otherwise it shall return 'false'.
*)
FUNCTION macro_check_extent_if_populated(extent:entity_extent;
given_model:sdai_model) : BOOLEAN;
...
END_FUNCTION;

(* This macro shall return 'true' if all values of the entity instance
submitted
are unset; otherwise it shall return 'false'.
*)
FUNCTION macro_check_instance_if_values_unset(inst:GENERIC_ENTITY) : BOOLEAN;
...
END_FUNCTION;

(*)

```

ISO TC184/SC4/WG11 N137

Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute, unset attribute value, and
create aggregate instance for different types of the attribute.

Parameter:

model - an SDAI-model in read-write mode, based on the greek schema.

*)

```
PROCEDURE test_attribute_basic(model:sdai_model);

    (* The type of the attribute is NUMBER.*)
    test_attribute_number(model);

    (* The type of the attribute is REAL.*)
    test_attribute_real(model);

    (* The type of the attribute is INTEGER.*)
    test_attribute_integer(model);

    (* The type of the attribute is LOGICAL.*)
    test_attribute_logical(model);

    (* The type of the attribute is BOOLEAN.*)
    test_attribute_boolean(model);

    (* The type of the attribute is STRING.*)
    test_attribute_string(model);

    (* The type of the attribute is BINARY.*)
    test_attribute_binary(model);

    (* The type of the attribute is ENUMERATION.*)
    test_attribute_enumeration(model);

    (* The type of the attribute is select data type.*)
    test_attribute_select_defined_type(model);

    (* The type of the attribute is entity.*)
    test_attribute_entity(model);

    (* The type of the attribute is aggregation data type.*)
    test_attribute_aggregate(model);

    (* Testing if attribute is submitted to the SDAI operations and is a
    correct one.*)
    test_attribute_correctness(model);

END_PROCEDURE;
```

```
(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type NUMBER.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_number(model:sdai_model);
    LOCAL
        inst : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
        numb : NUMBER;
        bool : BOOLEAN;
```

```

END_LOCAL;

(* Ensures that the attribute is initially unset after creation of the
instance. *)
bool := test_attribute(inst, 'GREEK.OMEGA.O1', NO_ERROR, ?);
assert(NOT bool);

(* Ensures that get_attribute reports a VA_NSET error when the attribute
has no value. *)
numb := get_attribute(inst, 'GREEK.OMEGA.O1', VA_NSET, ?);

(* Ensures that put_attribute reports a VA_NSET error when an unset value
for the attribute is submitted. *)
put_attribute(inst, 'GREEK.OMEGA.O1', ?, VA_NSET, ?);

(* Ensures that put_attribute reports a VA_NVLD error when used for a value
of a wrong type. *)
put_attribute(inst, 'GREEK.OMEGA.O1', 'something', VT_NVLD, ?);

(* Ensures that put_attribute works correctly when all parameters are
correct. *)
put_attribute(inst, 'GREEK.OMEGA.O1', 3.4, NO_ERROR, ?);

(* Ensures that test_attribute returns TRUE after successfully invoking
put_attribute. *)
bool := test_attribute(inst, 'GREEK.OMEGA.O1', NO_ERROR, ?);
assert(bool);

(* Ensures that get_attribute retrieves the right value. *)
numb := get_attribute(inst, 'GREEK.OMEGA.O1', NO_ERROR, ?);
assert(numb = 3.4);

(* Ensures that the attribute value can be unset. *)
unset_attribute_value(inst, 'GREEK.OMEGA.O1', NO_ERROR, ?);

(* Ensures that the attribute is really unset after unset_attribute_value
operation. *)
bool := test_attribute(inst, 'GREEK.OMEGA.O1', NO_ERROR, ?);
assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type REAL.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_real(model:sdai_model);
  LOCAL
    inst : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    real_numb : REAL;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that the attribute is initially unset after creation of the
instance. *)
  bool := test_attribute(inst, 'GREEK.OMEGA.O2', NO_ERROR, ?);

```

ISO TC184/SC4/WG11 N137

```
    assert(NOT bool);

    (* Ensures that get_attribute reports a VA_NSET error when the attribute
    has no value. *)
    real_num := get_attribute(inst, 'GREEK.OMEGA.O2', VA_NSET, ?);

    (* Ensures that put_attribute reports a VA_NSET error when an unset value
    for the attribute is submitted. *)
    put_attribute(inst, 'GREEK.OMEGA.O2', ?, VA_NSET, ?);

    (* Ensures that put_attribute reports a VA_NVLD error when used for a value
    of a wrong type. *)
    put_attribute(inst, 'GREEK.OMEGA.O2', 'something', VT_NVLD, ?);

    (* Ensures that put_attribute works correctly when all parameters are
    correct. *)
    put_attribute(inst, 'GREEK.OMEGA.O2', 5.6, NO_ERROR, ?);

    (* Ensures that test_attribute returns TRUE after successfully invoking
    put_attribute. *)
    bool := test_attribute(inst, 'GREEK.OMEGA.O2', NO_ERROR, ?);
    assert(bool);

    (* Ensures that get_attribute retrieves the right value. *)
    real_num := get_attribute(inst, 'GREEK.OMEGA.O2', NO_ERROR, ?);
    assert(real_num = 5.6);

    (* Ensures that the attribute value can be unset. *)
    unset_attribute_value(inst, 'GREEK.OMEGA.O2', NO_ERROR, ?);

    (* Ensures that the attribute is really unset after unset_attribute_value
    operation. *)
    bool := test_attribute(inst, 'GREEK.OMEGA.O2', NO_ERROR, ?);
    assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type INTEGER.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_integer(model:sdai_model);
    LOCAL
        inst : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
        int : INTEGER;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that the attribute is initially unset after creation of the
    instance. *)
    bool := test_attribute(inst, 'GREEK.OMEGA.O3', NO_ERROR, ?);
    assert(NOT bool);

    (* Ensures that get_attribute reports a VA_NSET error when the attribute
    has no value. *)
    int := get_attribute(inst, 'GREEK.OMEGA.O3', VA_NSET, ?);
```



```

    (* Ensures that put_attribute reports a VA_NSET error when an unset value
    for the attribute is submitted. *)
    put_attribute(inst, 'GREEK.OMEGA.O3', ?, VA_NSET, ?);

    (* Ensures that put_attribute reports a VA_NVLD error when used for a value
    of a wrong type. *)
    put_attribute(inst, 'GREEK.OMEGA.O3', 'something', VT_NVLD, ?);

    (* Ensures that put_attribute works correctly when all parameters are
    correct. *)
    put_attribute(inst, 'GREEK.OMEGA.O3', 7, NO_ERROR, ?);

    (* Ensures that test_attribute returns TRUE after successfully invoking
    put_attribute. *)
    bool := test_attribute(inst, 'GREEK.OMEGA.O3', NO_ERROR, ?);
    assert(bool);

    (* Ensures that get_attribute retrieves the right value. *)
    int := get_attribute(inst, 'GREEK.OMEGA.O3', NO_ERROR, ?);
    assert(int = 7);

    (* Ensures that the attribute value can be unset. *)
    unset_attribute_value(inst, 'GREEK.OMEGA.O3', NO_ERROR, ?);

    (* Ensures that the attribute is really unset after unset_attribute_value
    operation. *)
    bool := test_attribute(inst, 'GREEK.OMEGA.O3', NO_ERROR, ?);
    assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type LOGICAL.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_logical(model:sdai_model);
    LOCAL
        inst : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
        logic : LOGICAL;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that the attribute is initially unset after creation of the
    instance. *)
    bool := test_attribute(inst, 'GREEK.OMEGA.O4', NO_ERROR, ?);
    assert(NOT bool);

    (* Ensures that get_attribute reports a VA_NSET error when the attribute
    has no value. *)
    logic := get_attribute(inst, 'GREEK.OMEGA.O4', VA_NSET, ?);

    (* Ensures that put_attribute reports a VA_NSET error when an unset value
    for the attribute is submitted. *)
    put_attribute(inst, 'GREEK.OMEGA.O4', ?, VA_NSET, ?);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that put_attribute reports a VA_NVLD error when used for a value
of a wrong type. *)
put_attribute(inst, 'GREEK.OMEGA.O4', 'something', VT_NVLD, ?);

(* Ensures that put_attribute works correctly when all parameters are
correct. *)
put_attribute(inst, 'GREEK.OMEGA.O4', UNKNOWN, NO_ERROR, ?);

(* Ensures that test_attribute returns TRUE after successfully invoking
put_attribute. *)
bool := test_attribute(inst, 'GREEK.OMEGA.O4', NO_ERROR, ?);
assert(bool);

(* Ensures that get_attribute retrieves the right value. *)
logic := get_attribute(inst, 'GREEK.OMEGA.O4', NO_ERROR, ?);
assert(logic = UNKNOWN);

(* Ensures that the attribute value can be unset. *)
unset_attribute_value(inst, 'GREEK.OMEGA.O4', NO_ERROR, ?);

(* Ensures that the attribute is really unset after unset_attribute_value
operation. *)
bool := test_attribute(inst, 'GREEK.OMEGA.O4', NO_ERROR, ?);
assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type BOOLEAN.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_boolean(model:sdai_model);
  LOCAL
    inst : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    bool_value : BOOLEAN;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that the attribute is initially unset after creation of the
instance. *)
  bool := test_attribute(inst, 'GREEK.OMEGA.O5', NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that get_attribute reports a VA_NSET error when the attribute
has no value. *)
  bool_value := get_attribute(inst, 'GREEK.OMEGA.O5', VA_NSET, ?);

  (* Ensures that put_attribute reports a VA_NSET error when an unset value
for the attribute is submitted. *)
  put_attribute(inst, 'GREEK.OMEGA.O5', ?, VA_NSET, ?);

  (* Ensures that put_attribute reports a VA_NVLD error when used for a value
of a wrong type. *)
  put_attribute(inst, 'GREEK.OMEGA.O5', 'something', VT_NVLD, ?);
```

```

    (* Ensures that put_attribute works correctly when all parameters are
    correct. *)
    put_attribute(inst, 'GREEK.OMEGA.O5', FALSE, NO_ERROR, ?);

    (* Ensures that test_attribute returns TRUE after successfully invoking
    put_attribute. *)
    bool := test_attribute(inst, 'GREEK.OMEGA.O5', NO_ERROR, ?);
    assert(bool);

    (* Ensures that get_attribute retrieves the right value. *)
    bool_value := get_attribute(inst, 'GREEK.OMEGA.O5', NO_ERROR, ?);
    assert(bool_value = FALSE);

    (* Ensures that the attribute value can be unset. *)
    unset_attribute_value(inst, 'GREEK.OMEGA.O5', NO_ERROR, ?);

    (* Ensures that the attribute is really unset after unset_attribute_value
    operation. *)
    bool := test_attribute(inst, 'GREEK.OMEGA.O5', NO_ERROR, ?);
    assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type STRING.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_string(model:sdai_model);
    LOCAL
        inst : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
        str : STRING;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that the attribute is initially unset after creation of the
    instance. *)
    bool := test_attribute(inst, 'GREEK.OMEGA.O6', NO_ERROR, ?);
    assert(NOT bool);

    (* Ensures that get_attribute reports a VA_NSET error when the attribute
    has no value. *)
    str := get_attribute(inst, 'GREEK.OMEGA.O6', VA_NSET, ?);

    (* Ensures that put_attribute reports a VA_NSET error when an unset value
    for the attribute is submitted. *)
    put_attribute(inst, 'GREEK.OMEGA.O6', ?, VA_NSET, ?);

    (* Ensures that put_attribute reports a VA_NVLD error when used for a value
    of a wrong type. *)
    put_attribute(inst, 'GREEK.OMEGA.O6', 7.7, VT_NVLD, ?);

    (* Ensures that put_attribute works correctly when all parameters are
    correct. *)
    put_attribute(inst, 'GREEK.OMEGA.O6', 'test-string', NO_ERROR, ?);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that test_attribute returns TRUE after successfully invoking
put_attribute. *)
bool := test_attribute(inst, 'GREEK.OMEGA.O6', NO_ERROR, ?);
assert(bool);

(* Ensures that get_attribute retrieves the right value. *)
str := get_attribute(inst, 'GREEK.OMEGA.O6', NO_ERROR, ?);
assert(str = 'test-string');

(* Ensures that the attribute value can be unset. *)
unset_attribute_value(inst, 'GREEK.OMEGA.O6', NO_ERROR, ?);

(* Ensures that the attribute is really unset after unset_attribute_value
operation. *)
bool := test_attribute(inst, 'GREEK.OMEGA.O6', NO_ERROR, ?);
assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type BINARY.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_binary(model:sdai_model);
  LOCAL
    inst : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    bin : BINARY;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that the attribute is initially unset after creation of the
instance. *)
  bool := test_attribute(inst, 'GREEK.OMEGA.O7', NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that get_attribute reports a VA_NSET error when the attribute
has no value. *)
  bin := get_attribute(inst, 'GREEK.OMEGA.O7', VA_NSET, ?);

  (* Ensures that put_attribute reports a VA_NSET error when an unset value
for the attribute is submitted. *)
  put_attribute(inst, 'GREEK.OMEGA.O7', ?, VA_NSET, ?);

  (* Ensures that put_attribute reports a VA_NVLD error when used for a value
of a wrong type. *)
  put_attribute(inst, 'GREEK.OMEGA.O7', 7.7, VT_NVLD, ?);

  (* Ensures that put_attribute works correctly when all parameters are
correct. *)
  put_attribute(inst, 'GREEK.OMEGA.O7', %111011, NO_ERROR, ?);

  (* Ensures that test_attribute returns TRUE after successfully invoking
put_attribute. *)
  bool := test_attribute(inst, 'GREEK.OMEGA.O7', NO_ERROR, ?);
  assert(bool);
```

```

(* Ensures that get_attribute retrieves the right value. *)
bin := get_attribute(inst, 'GREEK.OMEGA.O7', NO_ERROR, ?);
assert(bin = %111011);

(* Ensures that the attribute value can be unset. *)
unset_attribute_value(inst, 'GREEK.OMEGA.O7', NO_ERROR, ?);

(* Ensures that the attribute is really unset after unset_attribute_value
operation. *)
bool := test_attribute(inst, 'GREEK.OMEGA.O7', NO_ERROR, ?);
assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type ENUMERATION.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_enumeration(model:sdai_model);
  LOCAL
    inst : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    enum : ENUMERATION;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that the attribute is initially unset after creation of the
instance. *)
  bool := test_attribute(inst, 'GREEK.OMEGA.O8', NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that get_attribute reports a VA_NSET error when the attribute
has no value. *)
  enum := get_attribute(inst, 'GREEK.OMEGA.O8', VA_NSET, ?);

  (* Ensures that put_attribute reports a VA_NSET error when an unset value
for the attribute is submitted. *)
  put_attribute(inst, 'GREEK.OMEGA.O8', ?, VA_NSET, ?);

  (* Ensures that put_attribute reports a VA_NVLD error when used for a value
of a wrong type. *)
  put_attribute(inst, 'GREEK.OMEGA.O8', 7.7, VT_NVLD, ?);

  (* Ensures that put_attribute works correctly when all parameters are
correct. *)
  put_attribute(inst, 'GREEK.OMEGA.O8', tau.stigma, NO_ERROR, ?);

  (* Ensures that test_attribute returns TRUE after successfully invoking
put_attribute. *)
  bool := test_attribute(inst, 'GREEK.OMEGA.O8', NO_ERROR, ?);
  assert(bool);

  (* Ensures that get_attribute retrieves the right value. *)
  enum := get_attribute(inst, 'GREEK.OMEGA.O8', NO_ERROR, ?);
  assert(enum = tau.stigma);

  (* Ensures that the attribute value can be unset. *)

```

ISO TC184/SC4/WG11 N137

```
unset_attribute_value(inst, 'GREEK.OMEGA.O8', NO_ERROR, ?);

(* Ensures that the attribute is really unset after unset_attribute_value
operation. *)
bool := test_attribute(inst, 'GREEK.OMEGA.O8', NO_ERROR, ?);
assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of EXPRESS TYPE.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_select_defined_type(model:sdai_model);
  LOCAL
    inst : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    int : INTEGER;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that the attribute is initially unset after creation of the
instance. *)
  bool := test_attribute(inst, 'GREEK.OMEGA.O9', NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that get_attribute reports a VA_NSET error when the attribute
has no value. *)
  int := get_attribute(inst, 'GREEK.OMEGA.O9', VA_NSET, ?);

  (* Ensures that put_attribute reports a VA_NSET error when an unset value
for the attribute is submitted. *)
  put_attribute(inst, 'GREEK.OMEGA.O9', ?, VA_NSET, ?);

  (* Ensures that put_attribute reports a VA_NVLD error when used for a value
of a wrong type. *)
  put_attribute(inst, 'GREEK.OMEGA.O9', 'something', VT_NVLD, ?);

  (* Ensures that put_attribute works correctly when all parameters are
correct. *)
  put_attribute(inst, 'GREEK.OMEGA.O9', xi(89), NO_ERROR, ?);

  (* Ensures that test_attribute returns TRUE after successfully invoking
put_attribute. *)
  bool := test_attribute(inst, 'GREEK.OMEGA.O9', NO_ERROR, ?);
  assert(bool);

  (* Ensures that get_attribute retrieves the right value. *)
  int := get_attribute(inst, 'GREEK.OMEGA.O9', NO_ERROR, ?);
  assert(int = xi(89));

  (* Ensures that the attribute value can be unset. *)
  unset_attribute_value(inst, 'GREEK.OMEGA.O9', NO_ERROR, ?);

  (* Ensures that the attribute is really unset after unset_attribute_value
operation. *)
  bool := test_attribute(inst, 'GREEK.OMEGA.O9', NO_ERROR, ?);
```

```

    assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type ENTITY.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_entity(model:sdai_model);
    LOCAL
        inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
        inst2 : GENERIC_ENTITY;
        inst3 : GENERIC_ENTITY;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that the attribute is initially unset after creation of the
instance. *)
    bool := test_attribute(inst1, 'GREEK.OMEGA.O0', NO_ERROR, ?);
    assert(NOT bool);

    (* Ensures that get_attribute reports a VA_NSET error when the attribute
has no value. *)
    inst3 := get_attribute(inst1, 'GREEK.OMEGA.O0', VA_NSET, ?);

    (* Ensures that put_attribute reports a VA_NSET error when an unset value
for the attribute is submitted. *)
    put_attribute(inst1, 'GREEK.OMEGA.O0', ?, VA_NSET, ?);

    (* Ensures that put_attribute reports a VA_NVLD error when used for a value
of a wrong type. *)
    inst2 := create_entity_instance('GREEK.EPSILON', model, NO_ERROR, ?);
    put_attribute(inst1, 'GREEK.OMEGA.O0', inst2, VT_NVLD, ?);

    (* Ensures that put_attribute works correctly when all parameters are
correct. *)
    inst2 := create_entity_instance('GREEK.IOTA', model, NO_ERROR, ?);
    put_attribute(inst1, 'GREEK.OMEGA.O0', inst2, NO_ERROR, ?);

    (* Ensures that test_attribute returns TRUE after successfully invoking
put_attribute. *)
    bool := test_attribute(inst1, 'GREEK.OMEGA.O0', NO_ERROR, ?);
    assert(bool);

    (* Ensures that get_attribute retrieves the right value. *)
    inst3 := get_attribute(inst1, 'GREEK.OMEGA.O0', NO_ERROR, ?);
    assert(inst3 == inst2);

    (* Ensures that the attribute value can be unset. *)
    unset_attribute_value(inst1, 'GREEK.OMEGA.O0', NO_ERROR, ?);

    (* Ensures that the attribute is really unset after unset_attribute_value
operation. *)
    bool := test_attribute(inst1, 'GREEK.OMEGA.O0', NO_ERROR, ?);
    assert(NOT bool);

```

ISO TC184/SC4/WG11 N137

END_PROCEDURE;

```
(*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, create aggregate instance and unset attribute
value
for an attribute of type aggregate.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_aggregate(model:sdai_model);
  LOCAL
    inst : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON', model,
NO_ERROR, ?);
    aggr1 : LIST [1:?] OF nu;
    aggr2 : LIST [1:?] OF nu;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that the attribute is initially unset after creation of the
instance. *)
  bool := test_attribute(inst, 'GREEK.EPSILON.E2', NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that get_attribute reports a VA_NSET error when the attribute
has no value. *)
  aggr2 := get_attribute(inst, 'GREEK.EPSILON.E2', VA_NSET, ?);

  (* Ensures that create_aggregate_instance works correctly when parameters
are correct. *)
  aggr1 := create_aggregate_instance(inst, 'GREEK.EPSILON.E2', NO_ERROR, ?);

  (* Ensures that test_attribute returns TRUE after successfully invoking
create_aggregate_instance. *)
  bool := test_attribute(inst, 'GREEK.EPSILON.E2', NO_ERROR, ?);
  assert(bool);

  (* Ensures that get_attribute retrieves the right value. *)
  aggr2 := get_attribute(inst, 'GREEK.EPSILON.E2', NO_ERROR, ?);
  add_by_index(aggr1, 0, xi(100), NO_ERROR, ?);
  assert(xi(100) IN aggr2);

  (* Ensures that put_attribute cannot be applied to assign a value of type
aggregate. *)
  put_attribute(inst, 'GREEK.EPSILON.E2', aggr2, VT_NVLD, ?);

  (* Ensures that the attribute value can be unset. *)
  unset_attribute_value(inst, 'GREEK.EPSILON.E2', NO_ERROR, ?);

  (* Ensures that the attribute is really unset after unset_attribute_value
operation. *)
  bool := test_attribute(inst, 'GREEK.EPSILON.E2', NO_ERROR, ?);
  assert(NOT bool);

END_PROCEDURE;
```

```
(*
Testing if attribute is submitted to the SDAI operations
test attribute, get attribute, put attribute, create aggregate instance
```



```

and unset attribute value and is a correct one.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_attribute_correctness(model:sdai_model);
  LOCAL
    inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON', model,
NO_ERROR, ?);
    str : STRING;
    aggr : LIST [1:?] OF nu;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that put_attribute reports an AT_NDEF error when attribute is
not submitted. *)
  put_attribute(inst1, ?, 'test-string', AT_NDEF, ?);

  (* Ensures that put_attribute reports an AT_NVLD error when an attribute
submitted is not of this entity instance. *)
  put_attribute(inst1, 'GREEK.EPSILON.E1', 'test-string', AT_NVLD,
'GREEK.EPSILON.E1');

  (* Ensures that test_attribute reports an AT_NDEF error when attribute is
not submitted. *)
  bool := test_attribute(inst1, ?, AT_NDEF, ?);

  (* Ensures that test_attribute reports an AT_NVLD error when an attribute
submitted is not of this entity instance. *)
  bool := test_attribute(inst1, 'GREEK.EPSILON.E1', AT_NVLD,
'GREEK.EPSILON.E1');

  (* Ensures that get_attribute reports an AT_NDEF error when attribute is
not submitted. *)
  put_attribute(inst1, 'GREEK.OMEGA.O6', 'test-string', NO_ERROR, ?);
  str := get_attribute(inst1, ?, AT_NDEF, ?);

  (* Ensures that get_attribute reports an AT_NVLD error when an attribute
submitted is not of this entity instance. *)
  str := get_attribute(inst1, 'GREEK.EPSILON.E1', AT_NVLD,
'GREEK.EPSILON.E1');

  (* Ensures that unset_attribute_value reports an AT_NDEF error when
attribute is not submitted. *)
  unset_attribute_value(inst1, ?, AT_NDEF, ?);

  (* Ensures that unset_attribute_value reports an AT_NVLD error when an
attribute submitted is not of this entity instance. *)
  unset_attribute_value(inst1, 'GREEK.EPSILON.E1', AT_NVLD,
'GREEK.EPSILON.E1');

  (* Ensures that create_aggregate_instance reports an AT_NDEF error when
attribute is not submitted. *)
  aggr := create_aggregate_instance(inst2, ?, AT_NDEF, ?);

  (* Ensures that create_aggregate_instance reports an AT_NVLD error when an
attribute submitted is not of this entity instance. *)
  aggr := create_aggregate_instance(inst2, 'GREEK.OMEGA.O2', AT_NVLD,
'GREEK.OMEGA.O2');

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that create_aggregate_instance reports an AT_NVLD error when an
attribute type is not an aggregate. *)
aggr := create_aggregate_instance(inst2, 'GREEK.EPSILON.E1', AT_NVLD,
'GREEK.EPSILON.E1');

END_PROCEDURE;

(*
Testing get attribute operation for entity data type when referenced instance
belongs to a model which is in an open repository but which access is ended.
Parameters:
modell - an SDAI-model in read-write mode, based on the greek schema;
model2 - an SDAI-model in read-write mode, based on the greek schema; this
model
shall be different from modell and may even belong to a different repository.
*)
PROCEDURE test_attribute_entity_in_another_model(modell:sdai_model;
model2:sdai_model);
  LOCAL
    transaction : sdai_transaction :=
modell.repository.session.active_transaction[1];
    (* Creation of the referencing instance. *)
    inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', modell,
NO_ERROR, ?);
    (* Creation of the referenced instance. *)
    inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.IOTA', model2,
NO_ERROR, ?);
    inst3 : GENERIC_ENTITY;
  END_LOCAL;

  (* Setting a reference. *)
  put_attribute(inst1, 'GREEK.OMEGA.O0', inst2, NO_ERROR, ?);

  (* Commit operation settles all changes done; it is needed before ending
access of
the SDAI-model containing the referenced instance. *)
  commit(transaction, NO_ERROR, ?);

  (* A read-write access of the SDAI-model containing the referenced instance
is ended. *)
  end_read_write_access(model2, NO_ERROR, ?);

  (* Ensures that get_attribute retrieves the right value. *)
  inst3 := get_attribute(inst1, 'GREEK.OMEGA.O0', NO_ERROR, ?);
  assert(inst3 ::= inst2);

  (* Ensures that the read-only access of the closed model that is owning for
the instance
being referenced from another model is automatically started.
*)
  assert(model2.mode ::= access_type.read_only);

END_PROCEDURE;

(*
Testing get attribute operation for entity data type when referenced instance
belongs to a model in a closed repository.
Parameters:
modell - an SDAI-model in read-write mode, based on the greek schema;
```

```

model2 - an SDAI-model in read-write mode, based on the greek schema; this
model
shall belong to a different repository than model1 does.
*)
PROCEDURE test_attribute_entity_in_closed_repository(model1:sdai_model;
model2:sdai_model);
  LOCAL
    repo1 : sdai_repository := model1.repository;
    repo2 : sdai_repository := model1.repository;
    transaction : sdai_transaction := repo1.session.active_transaction[1];
    (* Creation of the referencing instance. *)
    inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model1,
NO_ERROR, ?);
    (* Creation of the referenced instance. *)
    inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.IOTA', model2,
NO_ERROR, ?);
    inst3 : GENERIC_ENTITY;
  END_LOCAL;

  (* Setting a reference. *)
  put_attribute(inst1, 'GREEK.OMEGA.O0', inst2, NO_ERROR, ?);

  (* Commit operation settles all changes done; it is needed before closing
repository
containing the referenced instance.
*)
  commit(transaction, NO_ERROR, ?);

  (* Repository containing the referenced instance is closed. *)
  close_repository(repo2, NO_ERROR, ?);

  (* Ensures that get_attribute reports an RP_NOPN error when the repository
containing an instance referenced by the specified attribute is closed.
*)
  inst3 := get_attribute(inst1, 'GREEK.OMEGA.O0', RP_NOPN, repo2);

END_PROCEDURE;

(*
Testing the SDAI operation find entity instance SDAI-model.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_find_entity_instance_SDAI_model(model:sdai_model);
  LOCAL
    model_found : sdai_model;
    inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    inst2 : GENERIC_ENTITY;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that the model found is the same as that in which the instance
submitted
for the method was created.
*)
  model_found := find_entity_instance_sdai_model(inst1, NO_ERROR, ?);
  inst2 := create_entity_instance('GREEK.EPSILON', model_found, NO_ERROR, ?);
  bool := is_member(model.contents.instances, inst2, NO_ERROR, ?);
  assert(bool);

```

ISO TC184/SC4/WG11 N137

```
END_PROCEDURE;
```

```
(*  
Testing the SDAI operation get instance type.  
Parameter:  
model - an SDAI-model in read-write mode, based on the greek schema.  
)  
PROCEDURE test_get_instance_type(model:sdai_model);  
  LOCAL  
    def1 : entity_definition := get_entity_definition(model, 'OMEGA',  
NO_ERROR, ?);  
    def2 : entity_definition;  
    inst : GENERIC_ENTITY := create_entity_instance(def, model, NO_ERROR, ?);  
    bool : BOOLEAN;  
  END_LOCAL;  
  
  (* Ensures that the correct entity definition of the instance is returned.  
)  
  def2 := get_instance_type(inst, NO_ERROR, ?);  
  assert(def1 :=: def2);  
  
END_PROCEDURE;
```

```
(*  
Testing the SDAI operation is instance of.  
Parameter:  
model - an SDAI-model in read-write mode, based on the greek schema.  
)  
PROCEDURE test_is_instance_of(model:sdai_model);  
  LOCAL  
    def : entity_definition := get_entity_definition(model, 'OMEGA',  
NO_ERROR, ?);  
    inst : GENERIC_ENTITY := create_entity_instance(def, model, NO_ERROR, ?);  
    bool : BOOLEAN;  
  END_LOCAL;  
  
  (* Ensures that is_instance_of reports an ED_NDEF error when entity  
definition is not submitted. *)  
  bool := is_instance_of(inst, ?, ED_NDEF, ?);  
  
  (* Ensures that is_instance_of returns TRUE when the instance checked is of  
the specified entity type. *)  
  bool := is_instance_of(inst, def, NO_ERROR, ?);  
  assert(bool);  
  
  (* Ensures that is_instance_of returns FALSE when the instance checked is  
of entity type  
that is different than the specified one.  
)  
  def := get_entity_definition(model, 'EPSILON', NO_ERROR, ?);  
  bool := is_instance_of(inst, def, NO_ERROR, ?);  
  assert(NOT bool);  
  
END_PROCEDURE;
```

```
(*  
Testing the SDAI operation is kind of.
```

```

Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_is_kind_of(model:sdai_model);
  LOCAL
    def : entity_definition;
    inst : GENERIC_ENTITY := create_entity_instance('GREEK.DELTA', model,
NO_ERROR, ?);
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that is_kind_of reports an ED_NDEF error when entity definition
is not submitted. *)
  bool := is_kind_of(inst, ?, ED_NDEF, ?);

  (* Ensures that is_kind_of returns TRUE when the instance checked is of
entity type
that is a subtype of the specified entity data type.
*)
  def := get_entity_definition(model, 'ALPHA', NO_ERROR, ?);
  bool := is_kind_of(inst, def, NO_ERROR, ?);
  assert(bool);

  (* Ensures that is_instance_of returns FALSE when the instance checked is
of entity type
that is not a subtype of the specified entity data type.
*)
  def := get_entity_definition(model, 'LAMDA', NO_ERROR, ?);
  bool := is_kind_of(inst, def, NO_ERROR, ?);
  assert(NOT bool);

END_PROCEDURE;

(*
Testing the SDAI operations get_persistent_label and get_session_identifier.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_persistent_label_and_session_identifier(model:sdai_model);
  LOCAL
    repo : sdai_repository := model.repository;
    inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    inst2 : GENERIC_ENTITY;
    label : STRING;
  END_LOCAL;

  (* Ensures that get_persistent_label performs correctly when an instance is
specified. *)
  label := get_persistent_label(inst1, NO_ERROR, ?);

  (* Ensures that the label returned by get_persistent_label is unique within
a repository. *)
  inst2 := get_session_identifier(label, repo, NO_ERROR, ?);
  assert(inst2 :=: inst1);

  (* Ensures that get_session_identifier reports an VA_NSET error when label
is not submitted. *)
  inst2 := get_session_identifier(?, repo, VA_NSET, ?);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that get_session_identifier reports an EI_NEXS error when an
instance requested
is not found in the repository.
```

```
*)
delete_application_instance(inst1, NO_ERROR, ?);
inst2 := get_session_identifier(label, repo, EI_NEXS, ?);
```

```
(* Ensures that get_session_identifier reports an RP_NOPN error when an
instance requested
belongs to the closed repository.
```

```
*)
inst1 := create_entity_instance('GREEK.EPSILON', model, NO_ERROR, ?);
label := get_persistent_label(inst1, NO_ERROR, ?);
close_repository(repo, NO_ERROR, ?);
inst2 := get_session_identifier(label, repo, RP_NOPN, repo);
```

```
END_PROCEDURE;
```

```
(*
Testing the SDAI operation copy application instance.
An instance is copied into a model created into the same repository.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema
and associated with a schema_instance whose native schema is greek schema.
```

```
*)
PROCEDURE test_copy_application_instance(model:sdai_model);
LOCAL
  schema_inst : schema_instance := model.associated_with[1];
  model_target : sdai_model :=
    create_sdai_model(model.repository, 'exceptional_name_within_repo',
'GREEK', NO_ERROR, ?);
  inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON',
some_mod, NO_ERROR, ?);
  inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA',
another_mod, NO_ERROR, ?);
  inst3 : GENERIC_ENTITY;
  inst4 : GENERIC_ENTITY;
  aggr1 : LIST [1:?] OF nu;
  aggr2 : LIST [1:?] OF nu;
  aggr1_element1 : LIST [1:3] OF REAL;
  aggr2_element1 : LIST [1:3] OF REAL;
  aggr1_element2 : tau;
  aggr2_element2 : tau;
  int : INTEGER;
  bool : BOOLEAN;
END_LOCAL;
```

```
-- preparation of the test data
```

```
(* An instance to be copied is prepared. An attribute of entity type is set
with value. *)
```

```
put_attribute(inst1, 'GREEK.EPSILON.E1', inst2, NO_ERROR, ?);
```

```
(* An attribute of aggregation type is set with value that is an aggregate
containing
```

```
two members one of which is again aggregate.
```

```
*)
aggr1 := create_aggregate_instance(inst1, 'GREEK.EPSILON.E2', NO_ERROR, ?);
aggr1_element1 := create_aggregate_instance_by_index(aggr1, 1,
['GREEK.CHI'], NO_ERROR, ?);
add_by_index(aggr1, 2, tau.stigma, NO_ERROR, ?);
```

```

(* An attribute of simple type is set with value. *)
put_attribute(inst1, 'GREEK.EPSILON.E3', 7, NO_ERROR, ?);

-- starting of tests
(* Ensures that copy_application_instance reports an MX_NRW error when a
target model
to which the specified instance needs to be copied is not in read&write
mode.
*)
inst3 := copy_application_instance(inst1, model_target, MX_NRW, ?);

(* Read-write mode for the target model is started. *)
start_read_write_access(model_target, NO_ERROR, ?);

(* Ensures that copy_application_instance reports an SI_NEXS error if there
are no
schema instance with which both the model being an owner of the specified
instance
and the target model are associated.
*)
inst3 := copy_application_instance(inst1, model_target, SI_NEXS, ?);

(* The target model is added to the given schema instance. *)
add_sdai_model(schema_inst, model_target, NO_ERROR, ?);

(* Ensures that copy_application_instance works correctly when parameters
are correct. *)
inst3 := copy_application_instance(inst1, model_target, NO_ERROR, ?);

(* Ensures that both entity instances, original and its copy, reference the
same instance. *)
inst4 := get_attribute(inst3, 'GREEK.EPSILON.E1', NO_ERROR, ?);
assert(inst2 == inst4);

(* Ensures that original entity instance and its copy have different
aggregates as values
of the same attribute.
*)
add_by_index(aggr1, 3, 100, NO_ERROR, ?);
aggr2 := get_attribute(inst3, 'GREEK.EPSILON.E2', NO_ERROR, ?);
bool := is_member(aggr2, 100, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that original entity instance and its copy have different
aggregates at any level
of nesting.
*)
add_by_index(aggr1_element1, 1, 7.7, NO_ERROR, ?);
aggr2_element1 := get_by_index(aggr2, 1, NO_ERROR, ?);
bool := is_member(aggr2_element1, 7.7, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that original entity instance and its copy have the same values
of simple types
at any level of nesting.
*)
aggr1_element2 := get_by_index(aggr1, 2, NO_ERROR, ?);
aggr2_element2 := get_by_index(aggr2, 2, NO_ERROR, ?);
assert(aggr1_element2 == aggr2_element2);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that original entity instance and its copy have the same values
of simple types. *)
int := get_attribute(inst3, 'GREEK.EPSILON.E3', NO_ERROR, ?);
assert(int = 7);
```

```
END_PROCEDURE;
```

```
(*
Testing the SDAI operation delete application instance.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema
and associated with a schema_instance whose native schema is greek schema.
*)
PROCEDURE test_delete_application_instance(model:sdai_model);
  LOCAL
    model_created : sdai_model :=
      create_sdai_model(model.repository, 'exceptional_name_within_repo',
'GREEK', NO_ERROR, ?);
    extent : entity_extent;
    def : entity_definition;
    inst1 : create_entity_instance('GREEK.EPSILON', model, NO_ERROR, ?);
    inst2 : create_entity_instance('GREEK.OMEGA', model_created, NO_ERROR,
?);
    bool : BOOLEAN;
  END_LOCAL;

  (* An instance references another instance which is planned to be deleted.
*)
  put_attribute(inst1, 'GREEK.EPSILON.E1', inst2, NO_ERROR, ?);

  (* Saving the entity definition of an instance to be deleted. *)
  def := get_instance_type(inst2, NO_ERROR, ?);

  (* Deleting of an application instance. *)
  delete_application_instance(inst2, NO_ERROR, ?);

  (* Ensures that a reference to an application instance that was deleted
becomes unset. *)
  bool := test_attribute(inst1, 'GREEK.EPSILON.E1', NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that deleted application instance does not belong to instances
set of the sdai_model_contents. *)
  bool := is_member(model_created.contents.instances, inst2, NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that emptied entity extent does not belong to populated_folders
set of the sdai_model_contents. *)
  extent := macro_get_entity_extent(def);
  bool := macro_check_extent_if_populated(extent, model_created);
  assert(NOT bool);

END_PROCEDURE;
```

```
(*
Testing the SDAI operation find entity instance users.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema
and associated with a schema_instance whose native schema is greek schema.
```



```

*)
PROCEDURE test_find_entity_instance_users(model:sdai_model);
  LOCAL
    schema_inst : schema_instance := model.associated_with[1];
    inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON', model,
NO_ERROR, ?);
    inst3 : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON', model,
NO_ERROR, ?);
    inst4 : GENERIC_ENTITY := create_entity_instance('GREEK.IOTA', model,
NO_ERROR, ?);
    aggr : LIST [1:?] OF nu;
    non_persist_list_schemas : LIST [0:?] OF GENERIC_ENTITY :=
create_non_persistent_list(NO_ERROR, ?);
    non_persist_list_instances : LIST [0:?] OF GENERIC_ENTITY :=
create_non_persistent_list(NO_ERROR, ?);
    count : INTEGER;
    bool : BOOLEAN;
  END_LOCAL;

  (* Instance inst2 references instance inst1. *)
  put_attribute(inst2, 'GREEK.EPSILON.E1', inst1, NO_ERROR, ?);

  (* Instance inst3 references instance inst1 (through the aggregate and
through the attribute of
entity type).
*)
  aggr := create_aggregate_instance(inst3, 'GREEK.EPSILON.E2', NO_ERROR, ?);
  add_by_index(aggr, 1, inst1, NO_ERROR, ?);
  add_by_index(aggr, 2, inst1, NO_ERROR, ?);
  put_attribute(inst3, 'GREEK.EPSILON.E5', inst1, NO_ERROR, ?);

  (* Ensures that find_entity_instance_users reports an AI_NEXS error when a
non-persistent
list for writing a result is not provided.
*)
  find_entity_instance_users(inst1, non_persist_list_schemas, ?, AI_NEXS, ?);

  (* Ensures that find_entity_instance_users reports an AI_NVLD error when an
aggregate different
than non-persistent list for writing a result is submitted.
*)
  find_entity_instance_users(inst1, non_persist_list_schemas, aggr, AI_NVLD,
aggr);

  (* Ensures that find_entity_instance_users reports an AI_NEXS error when a
non-persistent
list specifying the domain of entity instances is not submitted.
*)
  find_entity_instance_users(inst1, ?, non_persist_list_instances, AI_NEXS,
?);

  (* Ensures that find_entity_instance_users performs correctly when all
parameters are correct. *)
  find_entity_instance_users(inst1, non_persist_list_schemas,
non_persist_list_instances, NO_ERROR, ?);

  (* Ensures that the resulting aggregate is empty (because a non-persistent
list specifying
the domain of entity instances is empty).

```

ISO TC184/SC4/WG11 N137

```
*)
count := get_member_count(non_persist_list_instances, NO_ERROR, ?);
assert(count = 0);

(* Making the non-persistent list specifying the domain of entity instances
to contain
   a given schema instance.
*)
add_by_index(non_persist_list_schemas, 1, schema_inst, NO_ERROR, ?);

(* Ensures that find_entity_instance_users performs correctly and writes
the instances
   referencing the specified instance into a submitted aggregate.
*)
find_entity_instance_users(inst1, non_persist_list_schemas,
non_persist_list_instances, NO_ERROR, ?);
bool := macro_compare_aggregates(non_persist_list_instances, [inst2, inst3,
inst3, inst3]);
assert(bool);

(* Instance inst1 references instance inst4. *)
put_attribute(inst1, 'GREEK.OMEGA.O0', inst4, NO_ERROR, ?);

(* Ensures that find_entity_instance_users performs correctly when a non-
persistent list
   submitted for apending instances is nonempty.
*)
find_entity_instance_users(inst4, non_persist_list_schemas,
non_persist_list_instances, NO_ERROR, ?);

(* Ensures that the instance referencing the specified instance is added to
a
   non-persistent list submitted for writing the result and all instances
earlier included into it are retained.
*)
bool := is_member(non_persist_list_instances, inst1, NO_ERROR, ?);
assert(bool);
count := get_member_count(non_persist_list_instances, NO_ERROR, ?);
assert(count = 5);

END_PROCEDURE;
```

```
(*
This macro compares the contents of two aggregates. The value TRUE is returned
if
and only if both aggregates contain exactly the same members with the same
repetition.
The order in which elements are stored in each of the aggregates is ignored.
*)
FUNCTION macro_compare_aggregates(aggr1:LIST [0:?] OF GENERIC_ENTITY;
aggr2:BAG [0:?] OF GENERIC_ENTITY) : BOOLEAN;
...
END_FUNCTION;
```

```
(*
Testing the SDAI operation find entity instance usedin.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema
and associated with a schema_instance whose native schema is greek schema.
```

```

*)
PROCEDURE test_find_entity_instance_usedin(model:sdai_model);
  LOCAL
    schema_inst : schema_instance := model.associated_with[1];
    inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON', model,
NO_ERROR, ?);
    inst3 : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON', model,
NO_ERROR, ?);
    inst4 : GENERIC_ENTITY := create_entity_instance('GREEK.IOTA', model,
NO_ERROR, ?);
    aggr1 : LIST [1:?] OF nu;
    aggr2 : LIST [1:?] OF nu;
    non_persist_list_schemas : LIST [0:?] OF GENERIC_ENTITY :=
create_non_persistent_list(NO_ERROR, ?);
    non_persist_list_instances : LIST [0:?] OF GENERIC_ENTITY :=
create_non_persistent_list(NO_ERROR, ?);
    count : INTEGER;
    bool : BOOLEAN;
  END_LOCAL;

  (* Instance inst2 references (through the aggregate) instance inst1. *)
  aggr1 := create_aggregate_instance(inst2, 'GREEK.EPSILON.E2', NO_ERROR, ?);
  add_by_index(aggr1, 1, inst1, NO_ERROR, ?);

  (* Instance inst3 references instance inst1 (through the aggregate and
through the attribute of
entity type).
*)
  aggr2 := create_aggregate_instance(inst3, 'GREEK.EPSILON.E2', NO_ERROR, ?);
  add_by_index(aggr2, 1, inst1, NO_ERROR, ?);
  add_by_index(aggr2, 2, inst1, NO_ERROR, ?);
  put_attribute(inst3, 'GREEK.EPSILON.E5', inst1, NO_ERROR, ?);

  (* Ensures that find_entity_instance_usedin reports an AI_NEXS error when a
non-persistent
list for writing a result is not provided.
*)
  find_entity_instance_usedin(inst1, 'GREEK.EPSILON.E2',
non_persist_list_schemas, ?, AI_NEXS, ?);

  (* Ensures that find_entity_instance_usedin reports an AI_NVLD error when
an aggregate different
than non-persistent list for writing a result is submitted.
*)
  find_entity_instance_usedin(inst1, 'GREEK.EPSILON.E2',
non_persist_list_schemas, aggr1, AI_NVLD, aggr1);

  (* Ensures that find_entity_instance_usedin reports an AI_NEXS error when a
non-persistent
list specifying the domain of entity instances is not submitted.
*)
  find_entity_instance_usedin(inst1, 'GREEK.EPSILON.E2', ?,
non_persist_list_instances, AI_NEXS, ?);

  (* Ensures that find_entity_instance_usedin performs correctly when all
parameters are correct. *)
  find_entity_instance_usedin(inst1, 'GREEK.EPSILON.E2',
non_persist_list_schemas,
non_persist_list_instances, NO_ERROR, ?);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that the resulting aggregate is empty (because a non-persistent
list specifying
the domain of entity instances is empty).
*)
count := get_member_count(non_persist_list_instances, NO_ERROR, ?);
assert(count = 0);

(* Making the non-persistent list specifying the domain of entity instances
to contain
a given schema instance.
*)
add_by_index(non_persist_list_schemas, 1, schema_inst, NO_ERROR, ?);

(* Ensures that find_entity_instance_usedin reports an AT_NDEF error when
an attribute
specifying the role is not submitted.
*)
find_entity_instance_usedin(inst1, ?, non_persist_list_schemas,
non_persist_list_instances, AT_NDEF, ?);

(* Ensures that find_entity_instance_usedin performs correctly and writes
the instances
referencing the specified instance into a submitted aggregate.
*)
find_entity_instance_usedin(inst1, 'GREEK.EPSILON.E2',
non_persist_list_schemas,
non_persist_list_instances, NO_ERROR, ?);
bool := macro_compare_aggregates(non_persist_list_instances, [inst2, inst3,
inst3]);
assert(bool);

(* Instance inst1 references instance inst4. *)
put_attribute(inst1, 'GREEK.OMEGA.O0', inst4, NO_ERROR, ?);

(* Ensures that find_entity_instance_usedin performs correctly when a non-
persistent list
submitted for apending instances is nonempty.
*)
find_entity_instance_usedin(inst4, 'GREEK.OMEGA.O0',
non_persist_list_schemas,
non_persist_list_instances, NO_ERROR, ?);

(* Ensures that the instance referencing the specified instance is added to
a
non-persistent list submitted for writing the result and all instances
earlier included into it are retained.
*)
bool := is_member(non_persist_list_instances, inst1, NO_ERROR, ?);
assert(bool);
count := get_member_count(non_persist_list_instances, NO_ERROR, ?);
assert(count = 4);

END_PROCEDURE;

(*
Testing the SDAI operation find instance roles.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema
and associated with a schema_instance whose native schema is greek schema.
```

```

*)
PROCEDURE test_find_instance_roles(model:sdai_model);
  LOCAL
    schema_inst : schema_instance := model.associated_with[1];
    inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON', model,
NO_ERROR, ?);
    inst3 : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON', model,
NO_ERROR, ?);
    inst4 : GENERIC_ENTITY := create_entity_instance('GREEK.IOTA', model,
NO_ERROR, ?);
    aggr : LIST [1:?] OF nu;
    non_persist_list_schemas : LIST [0:?] OF GENERIC_ENTITY :=
create_non_persistent_list(NO_ERROR, ?);
    non_persist_list_instances : LIST [0:?] OF GENERIC_ENTITY :=
create_non_persistent_list(NO_ERROR, ?);
    count : INTEGER;
    bool : BOOLEAN;
  END_LOCAL;

  (* Instance inst2 references instance inst1. *)
  put_attribute(inst2, 'GREEK.EPSILON.E1', inst1, NO_ERROR, ?);

  (* Instance inst3 references instance inst1 (through the aggregate and
through the attribute of
entity type).
*)
  aggr := create_aggregate_instance(inst3, 'GREEK.EPSILON.E2', NO_ERROR, ?);
  add_by_index(aggr, 1, inst1, NO_ERROR, ?);
  add_by_index(aggr, 2, inst1, NO_ERROR, ?);
  put_attribute(inst3, 'GREEK.EPSILON.E5', inst1, NO_ERROR, ?);

  (* Ensures that find_instance_roles reports an AI_NEXS error when a non-
persistent
list for writing a result is not provided.
*)
  find_instance_roles(inst1, non_persist_list_schemas, ?, AI_NEXS, ?);

  (* Ensures that find_instance_roles reports an AI_NVLD error when an
aggregate different
than non-persistent list for writing a result is submitted.
*)
  find_instance_roles(inst1, non_persist_list_schemas, aggr, AI_NVLD, aggr);

  (* Ensures that find_instance_roles reports an AI_NEXS error when a non-
persistent
list specifying the domain of entity instances is not submitted.
*)
  find_instance_roles(inst1, ?, non_persist_list_instances, AI_NEXS, ?);

  (* Ensures that find_instance_roles performs correctly when all parameters
are correct. *)
  find_instance_roles(inst1, non_persist_list_schemas,
non_persist_list_instances, NO_ERROR, ?);

  (* Ensures that the resulting aggregate is empty (because a non-persistent
list specifying
the domain of entity instances is empty).
*)
  count := get_member_count(non_persist_list_instances, NO_ERROR, ?);

```

ISO TC184/SC4/WG11 N137

```
    assert(count = 0);

    (* Making the non-persistent list specifying the domain of entity instances
    to contain
       a given schema instance.
    *)
    add_by_index(non_persist_list_schemas, 1, schema_inst, NO_ERROR, ?);

    (* Ensures that find_instance_roles performs correctly and writes the
    attributes
       referencing the specified instance into a submitted aggregate.
    *)
    find_instance_roles(inst1, non_persist_list_schemas,
non_persist_list_instances, NO_ERROR, ?);
    bool := macro_compare_aggregates(non_persist_list_instances,
    ['GREEK.EPSILON.E1', 'GREEK.EPSILON.E2', 'GREEK.EPSILON.E5']);
    assert(bool);

    (* Instance inst1 references instance inst4. *)
    put_attribute(inst1, 'GREEK.OMEGA.O0', inst4, NO_ERROR, ?);

    (* Ensures that find_instance_roles performs correctly when a non-
    persistent list
       submitted for apending attributes is nonempty.
    *)
    find_instance_roles(inst4, non_persist_list_schemas,
non_persist_list_instances, NO_ERROR, ?);

    (* Ensures that the new attribute is added to a non-persistent list
    submitted for writing
       the result and all attributes earlier included into it are retained.
    *)
    bool := is_member(non_persist_list_instances, 'GREEK.OMEGA.O0', NO_ERROR,
?);
    assert(bool);
    count := get_member_count(non_persist_list_instances, NO_ERROR, ?);
    assert(count = 4);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation add by index
for an aggregate of type LIST of NUMBER.
Parameter:
aggr - an aggregate of type LIST of NUMBER; the aggregate shall be empty.
*)
PROCEDURE test_aggregate_number_list_add_by_index(aggr:LIST [0:?] OF NUMBER);

    (* Ensures that indexing in the LIST starts from 1. *)
    add_by_index(aggr, 0, 1.5, IX_NVLD, aggr);

    (* Ensures that add_by_index reports a VA_NSET error when an unset value is
    submitted. *)
    add_by_index(aggr, 1, ?, VA_NSET, ?);

    (* Ensures that add_by_index reports a VT_NVLD error when value of a wrong
    type is submitted. *)
    add_by_index(aggr, 1, 'something', VT_NVLD, ?);
```

```
(* Ensures that add_by_index works correctly when all parameters are
correct. *)
```

```
add_by_index(aggr, 1, 1.5, NO_ERROR, ?);
add_by_index(aggr, 2, 77, NO_ERROR, ?);
add_by_index(aggr, 1, 9.99, NO_ERROR, ?);
add_by_index(aggr, 3, 3.14, NO_ERROR, ?);
```

```
(* Ensures that add_by_index reports an IX_NVLD error when the index
submitted exceeds
```

```
the count of aggregate members plus one. *)
add_by_index(aggr, 6, 1.5, IX_NVLD, aggr);
```

```
END_PROCEDURE;
```

```
(*
Testing the basic functionality of the SDAI operation get by index
for an aggregate of type LIST of NUMBER.
```

```
Parameter:
```

```
aggr - an aggregate of type LIST of NUMBER;
the aggregate shall be that processed by
test_aggregate_number_list_add_by_index.
```

```
*)
```

```
PROCEDURE test_aggregate_number_list_get_by_index(aggr:LIST [0:?] OF NUMBER);
```

```
LOCAL
```

```
numb : NUMBER;
```

```
END_LOCAL;
```

```
(* Ensures that get_by_index reports an IX_NVLD error when the index
submitted is outside of the legal range. *)
```

```
numb := get_by_index(aggr, 0, IX_NVLD, aggr);
numb := get_by_index(aggr, 5, IX_NVLD, aggr);
```

```
(* Ensures that get_by_index works correctly when the index submitted is
from the legal range. *)
```

```
numb := get_by_index(aggr, 1, NO_ERROR, ?);
assert(numb = 9.99);
numb := get_by_index(aggr, 2, NO_ERROR, ?);
assert(numb = 1.5);
numb := get_by_index(aggr, 3, NO_ERROR, ?);
assert(numb = 3.14);
numb := get_by_index(aggr, 4, NO_ERROR, ?);
assert(numb = 77);
```

```
END_PROCEDURE;
```

```
(*
Testing the basic functionality of the SDAI operation put by index
for an aggregate of type LIST of NUMBER.
```

```
Parameter:
```

```
aggr - an aggregate of type LIST of NUMBER;
the aggregate shall be that processed by
test_aggregate_number_list_add_by_index.
```

```
*)
```

```
PROCEDURE test_aggregate_number_list_put_by_index(aggr:LIST [0:?] OF NUMBER);
```

```
LOCAL
```

```
numb : NUMBER;
```

```
END_LOCAL;
```

ISO TC184/SC4/WG11 N137

```
(* Ensures that put_by_index reports an IX_NVLD error when the index
submitted is outside of the legal range. *)
put_by_index(aggr, 0, 3.147, IX_NVLD, aggr);
put_by_index(aggr, 5, 3.147, IX_NVLD, aggr);

(* Ensures that put_by_index reports a VA_NSET error when an unset value is
submitted. *)
put_by_index(aggr, 3, ?, VA_NSET, ?);

(* Ensures that put_by_index reports a VT_NVLD error when value of a wrong
type is submitted. *)
put_by_index(aggr, 3, 'something', VT_NVLD, ?);

(* Ensures that put_by_index works correctly when all parameters are
correct. *)
put_by_index(aggr, 3, 3.147, NO_ERROR, ?);
numb := get_by_index(aggr, 3, NO_ERROR, ?);
assert(numb = 3.147);
```

END_PROCEDURE;

```
(*
Testing the basic functionality of the SDAI operations is_member and get_member_count
for an aggregate of type LIST of NUMBER.
Parameter:
aggr - an aggregate of type LIST of NUMBER;
the aggregate shall be that processed by
test_aggregate_number_list_put_by_index.
*)
PROCEDURE test_aggregate_number_list_is_member(aggr:LIST [0:?] OF NUMBER);
  LOCAL
    bool : BOOLEAN;
    count : INTEGER;
  END_LOCAL;

  (* Ensures that is_member returns TRUE when value submitted belongs to the
aggregate. *)
  bool := is_member(aggr, 77.0, NO_ERROR, ?);
  assert(bool);

  (* Ensures that is_member returns FALSE when value submitted does not
belong to the aggregate. *)
  bool := is_member(aggr, 9.9, NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that get_member_count works correctly. *)
  count := get_member_count(aggr, NO_ERROR, ?);
  assert(count = 4);
```

END_PROCEDURE;

```
(*
Testing the basic functionality of the SDAI operation remove by index
for an aggregate of type LIST of NUMBER.
Parameter:
aggr - an aggregate of type LIST of NUMBER;
the aggregate shall be that processed by
test_aggregate_number_list_put_by_index.
```



```

*)
PROCEDURE test_aggregate_number_list_remove_by_index(aggr:LIST [0:?] OF
NUMBER);
  LOCAL
    numb : NUMBER;
  END_LOCAL;

  (* Ensures that remove_by_index reports an IX_NVLD error when the index
submitted is outside of the legal range. *)
  remove_by_index(aggr, 0, IX_NVLD, aggr);
  remove_by_index(aggr, 5, IX_NVLD, aggr);

  (* Ensures that remove_by_index works correctly when the index submitted is
from a legal range. *)
  remove_by_index(aggr, 2, NO_ERROR, ?);
  numb := get_by_index(aggr, 2, NO_ERROR, ?);
  assert(numb = 3.147);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation add before current
member
for an aggregate of type LIST of NUMBER. Also testing the SDAI operation add
after current member in the case when such an aggregate is empty.
Parameter:
iter - an iterator over an aggregate of type LIST of NUMBER;
the aggregate shall be empty.
*)
PROCEDURE test_aggregate_number_list_add_before_current_member(iter:iterator);

  (* Ensures that after add_after_current_member operation for an empty list
iterator is positioned at
the beginning of the list (there is no current member). *)
  atEnd(iter, NO_ERROR, ?);
  add_after_current_member(iter, 59.5, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  assert(bool);
  bool := remove_current_member(iter, NO_ERROR, ?);

  (* Ensures that add_before_current_member reports an IR_NEXS error when
iterator is not provided. *)
  add_before_current_member(?, 3.3, IR_NEXS, ?);

  (* Ensures that add_before_current_member reports a VA_NSET error when an
unset value is submitted. *)
  add_before_current_member(iter, ?, VA_NSET, ?);

  (* Ensures that add_before_current_member reports a VT_NVLD error when
value of a wrong type is submitted. *)
  add_before_current_member(iter, 'something', VT_NVLD, ?);

  (* Ensures that after add_before_current_member operation for an empty list
iterator is positioned at
the end of the list (there is no current member). *)
  beginning(iter, NO_ERROR, ?);
  add_before_current_member(iter, 3.3, NO_ERROR, ?);
  bool := previous(iter, NO_ERROR, ?);
  assert(bool);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that add_before_current_member works correctly when all
parameters are correct. *)
add_before_current_member(iter, 2.2, NO_ERROR, ?);
atEnd(iter, NO_ERROR, ?);
add_before_current_member(iter, 4.4, NO_ERROR, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation add after current member
for an aggregate of type LIST of NUMBER.
Parameter:
iter - an iterator over an aggregate of type LIST of NUMBER;
the aggregate shall be that processed by
test_aggregate_number_list_add_before_current_member.
*)
PROCEDURE test_aggregate_number_list_add_after_current_member(iter:iterator);

    (* Positioning of the iterator at the end of the aggregate. *)
    atEnd(iter, NO_ERROR, ?);

    (* Ensures that add_after_current_member reports an IR_NEXS error when
iterator is not provided. *)
    add_after_current_member(?, 5.5, IR_NEXS, ?);

    (* Ensures that add_after_current_member reports a VA_NSET error when an
unset value is submitted. *)
    add_after_current_member(iter, ?, VA_NSET, ?);

    (* Ensures that add_after_current_member reports a VT_NVLD error when value
of a wrong type is submitted. *)
    add_after_current_member(iter, 'something', VT_NVLD, ?);

    (* Ensures that add_after_current_member works correctly when all
parameters are correct. *)
    add_after_current_member(iter, 5.5, NO_ERROR, ?);
    add_after_current_member(iter, 6.6, NO_ERROR, ?);
    beginning(iter, NO_ERROR, ?);
    add_after_current_member(iter, 1.1, NO_ERROR, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation get current member
for an aggregate of type LIST of NUMBER.
Parameter:
iter - an iterator over an aggregate of type LIST of NUMBER;
the aggregate shall be that processed by
test_aggregate_number_list_add_after_current_member.
*)
PROCEDURE test_aggregate_number_list_get_current_member(iter:iterator);
    LOCAL
        numb : NUMBER;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that get_current_member reports an IR_NEXS error when iterator
is not provided. *)
    numb := get_current_member(?, IR_NEXS, ?);
```

```
(* Ensures that get_current_member reports an IR_NSET error when iterator
has no current member set. *)
```

```
beginning(iter, NO_ERROR, ?);
numb := get_current_member(iter, IR_NSET, ?);
atEnd(iter, NO_ERROR, ?);
numb := get_current_member(iter, IR_NSET, ?);
```

```
(* Ensures that get_current_member works correctly when iterator has
current member set. *)
```

```
bool := previous(iter, NO_ERROR, ?);
numb := get_current_member(iter, NO_ERROR, ?);
assert(numb = 6.6);
bool := previous(iter, NO_ERROR, ?);
numb := get_current_member(iter, NO_ERROR, ?);
assert(numb = 5.5);
bool := previous(iter, NO_ERROR, ?);
numb := get_current_member(iter, NO_ERROR, ?);
assert(numb = 4.4);
bool := previous(iter, NO_ERROR, ?);
numb := get_current_member(iter, NO_ERROR, ?);
assert(numb = 3.3);
bool := previous(iter, NO_ERROR, ?);
numb := get_current_member(iter, NO_ERROR, ?);
assert(numb = 2.2);
bool := previous(iter, NO_ERROR, ?);
numb := get_current_member(iter, NO_ERROR, ?);
assert(numb = 1.1);
```

```
END_PROCEDURE;
```

```
(*
Testing the basic functionality of the SDAI operation put current member
for an aggregate of type LIST of NUMBER.
```

```
Parameter:
```

```
iter - an iterator over an aggregate of type LIST of NUMBER;
```

```
the aggregate shall be that processed by
```

```
test_aggregate_number_list_add_after_current_member.
```

```
*)
```

```
PROCEDURE test_aggregate_number_list_put_current_member(iter:iterator);
```

```
LOCAL
```

```
    numb : NUMBER;
```

```
    bool : BOOLEAN;
```

```
END_LOCAL;
```

```
(* Ensures that put_current_member reports an IR_NEXS error when iterabr
is not provided. *)
```

```
put_current_member(?, 9.99, IR_NEXS, ?);
```

```
(* Ensures that put_current_member reports an IR_NSET error when iterator
has no current member set. *)
```

```
beginning(iter, NO_ERROR, ?);
put_current_member(iter, 9.99, IR_NSET, ?);
atEnd(iter, NO_ERROR, ?);
put_current_member(iter, 9.99, IR_NSET, ?);
```

```
(* Ensures that put_current_member reports a VA_NSET error when an unset
value is submitted. *)
```

```
put_current_member(iter, ?, VA_NSET, ?);
```

ISO TC184/SC4/WG11 N137

```
(* Ensures that put_current_member reports a VT_NVLD error when value of a
wrong type is submitted. *)
put_current_member(iter, 'something', VT_NVLD, ?);
```

```
(* Ensures that put_current_member works correctly when all parameters are
correct. *)
bool := previous(iter, NO_ERROR, ?);
put_current_member(iter, 9.99, NO_ERROR, ?);
numb := get_current_member(iter, NO_ERROR, ?);
assert(numb = 9.99);
```

```
END_PROCEDURE;
```

```
(*
Testing the basic functionality of the SDAI operation remove current member
for an aggregate of type LIST of NUMBER.
Parameter:
iter - an iterator over an aggregate of type LIST of NUMBER;
the aggregate shall be that processed by
test_aggregate_number_list_put_current_member.
*)
```

```
PROCEDURE test_aggregate_number_list_remove_current_member(iter:iterator);
LOCAL
  numb : NUMBER;
  bool : BOOLEAN;
END_LOCAL;
```

```
(* Ensures that remove_current_member reports an IR_NEXS error when
iterator is not provided. *)
bool := remove_current_member(?, IR_NEXS, ?);
```

```
(* Ensures that remove_current_member reports an IR_NSET error when
iterator has no current member set. *)
atEnd(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);
beginning(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);
```

```
(* Ensures that remove_current_member works correctly when iterator has
current member set. *)
bool := next(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);
assert(bool);
numb := get_current_member(iter, NO_ERROR, ?);
assert(numb = 2.2);
```

```
(* Ensures that remove_current_member returns FALSE when iterator refersto
the last member of the aggregate. *)
atEnd(iter, NO_ERROR, ?);
bool := previous(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);
assert(NOT bool);
```

```
END_PROCEDURE;
```

```
(*
Testing the basic functionality of the SDAI operations
get by index, put by index, add by index, remove by index, is member,
add before current member, add after current member, get current member,
```

```

put current member, and remove current member
for an aggregate of type LIST of NUMBER.
Parameters:
aggr - an aggregate of type LIST of NUMBER;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_number_list(aggr:LIST [0:?] OF NUMBER;
iter:iterator);

    (* Making the given aggregate empty. *)
    macro_clear_aggregate(aggr);

    (* Testing of the list operation add by index. *)
    test_aggregate_number_list_add_by_index(aggr);

    (* Testing of the aggregate operation get by index. *)
    test_aggregate_number_list_get_by_index(aggr);

    (* Testing of the aggregate operation put by index. *)
    test_aggregate_number_list_put_by_index(aggr);

    (* Testing of the aggregate operation is member. *)
    test_aggregate_number_list_is_member(aggr);

    (* Testing of the list operation remove by index. *)
    test_aggregate_number_list_remove_by_index(aggr);

    (* Making the given aggregate empty. *)
    macro_clear_aggregate(aggr);

    (* Testing of the list operation add before current member. *)
    test_aggregate_number_list_add_before_current_member(iter);

    (* Testing of the list operation add after current member. *)
    test_aggregate_number_list_add_after_current_member(iter);

    (* Testing of the aggregate operation get current member. *)
    test_aggregate_number_list_get_current_member(iter);

    (* Testing of the aggregate operation put current member. *)
    test_aggregate_number_list_put_current_member(iter);

    (* Testing of the aggregate operation remove current member. *)
    test_aggregate_number_list_remove_current_member(iter);

END_PROCEDURE;

(*
This macro makes the submitted aggregate empty.
*)
PROCEDURE macro_clear_aggregate(aggr:AGGREGATE OF GENERIC);
...
END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation add by index
for an aggregate of type LIST of entity instances.
Parameters:

```

ISO TC184/SC4/WG11 N137

aggr - an aggregate of type LIST of entity instances; the aggregate shall be empty;

aggr_with_data - an aggregate containing entity instances that will be added to the list specified by the first parameter.

*)

```
PROCEDURE test_aggregate_entity_list_add_by_index(aggr:LIST [1:?] OF
GENERIC_ENTITY;
```

```
    aggr_with_data:BAG [0:?] OF GENERIC_ENTITY);
```

```
    (* Ensures that indexing in the LIST starts from 1. *)
```

```
    add_by_index(aggr, 0, aggr_with_data[1], IX_NVLD, aggr);
```

```
    (* Ensures that add_by_index reports a VA_NSET error when an unset value is
submitted. *)
```

```
    add_by_index(aggr, 1, ?, VA_NSET, ?);
```

```
    (* Ensures that add_by_index reports a VT_NVLD error when value of a wrong
type is submitted. *)
```

```
    add_by_index(aggr, 1, 'something', VT_NVLD, ?);
```

```
    (* Ensures that add_by_index works correctly when all parameters are
correct. *)
```

```
    add_by_index(aggr, 1, aggr_with_data[2], NO_ERROR, ?);
```

```
    add_by_index(aggr, 2, aggr_with_data[4], NO_ERROR, ?);
```

```
    add_by_index(aggr, 1, aggr_with_data[1], NO_ERROR, ?);
```

```
    add_by_index(aggr, 3, aggr_with_data[3], NO_ERROR, ?);
```

```
    (* Ensures that add_by_index reports an IX_NVLD error when the index
submitted exceeds
```

```
    the count of aggregate members plus one. *)
```

```
    add_by_index(aggr, 6, aggr_with_data[1], IX_NVLD, aggr);
```

```
END_PROCEDURE;
```

```
(*
```

```
Testing the basic functionality of the SDAI operation get by index
for an aggregate of type LIST of entity instances.
```

```
Parameters:
```

```
aggr - an aggregate of type LIST of entity instances;
```

```
the aggregate shall be that processed by
```

```
test_aggregate_entity_list_add_by_index.
```

```
aggr_with_data - an aggregate containing entity instances that will be
compared with elements in the list specified by the first parameter.
```

```
*)
```

```
PROCEDURE test_aggregate_entity_list_get_by_index(aggr:LIST [1:?] OF
GENERIC_ENTITY;
```

```
    aggr_with_data:BAG [0:?] OF GENERIC_ENTITY);
```

```
    LOCAL
```

```
        inst : GENERIC_ENTITY;
```

```
    END_LOCAL;
```

```
    (* Ensures that get_by_index reports an IX_NVLD error when the index
submitted is outside of the legal range. *)
```

```
    inst := get_by_index(aggr, 0, IX_NVLD, aggr);
```

```
    inst := get_by_index(aggr, 5, IX_NVLD, aggr);
```

```
    (* Ensures that get_by_index works correctly when the index submitted is
from a legal range. *)
```

```
    inst := get_by_index(aggr, 1, NO_ERROR, ?);
```

```
    assert(inst == aggr_with_data[1]);
```

```

    inst := get_by_index(aggr, 2, NO_ERROR, ?);
    assert(inst :=: aggr_with_data[2]);
    inst := get_by_index(aggr, 3, NO_ERROR, ?);
    assert(inst :=: aggr_with_data[3]);
    inst := get_by_index(aggr, 4, NO_ERROR, ?);
    assert(inst :=: aggr_with_data[4]);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation put by index
for an aggregate of type LIST of entity instances.
Parameters:
aggr - an aggregate of type LIST of entity instances;
the aggregate shall be that processed by
test_aggregate_entity_list_add_by_index;
inst - an entity instance.
*)
PROCEDURE test_aggregate_entity_list_put_by_index(aggr:LIST [1:?] OF
GENERIC_ENTITY; inst:GENERIC_ENTITY);
    LOCAL
        inst2 : GENERIC_ENTITY;
    END_LOCAL;

    (* Ensures that put_by_index reports an IX_NVLD error when the index
submitted is outside of the legal range. *)
    put_by_index(aggr, 0, inst, IX_NVLD, aggr);
    put_by_index(aggr, 5, inst, IX_NVLD, aggr);

    (* Ensures that put_by_index reports a VA_NSET error when an unset value is
submitted. *)
    put_by_index(aggr, 3, ?, VA_NSET, ?);

    (* Ensures that put_by_index reports a VT_NVLD error when value of a wrong
type is submitted. *)
    put_by_index(aggr, 3, 'something', VT_NVLD, ?);

    (* Ensures that put_by_index works correctly when all parameters are
correct. *)
    put_by_index(aggr, 3, inst, NO_ERROR, ?);
    inst2 := get_by_index(aggr, 3, NO_ERROR, ?);
    assert(inst2 :=: inst);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation is member
for an aggregate of type LIST of entity instances.
Parameters:
aggr - an aggregate of type LIST of entity instances;
the aggregate shall be that processed by
test_aggregate_entity_list_put_by_index;
aggr_with_data - an aggregate containing entity instances that will be
checked for inclusion in the list specified by the first parameter.
*)
PROCEDURE test_aggregate_entity_list_is_member(aggr:LIST [1:?] OF
GENERIC_ENTITY;
        aggr_with_data:BAG [0:?] OF GENERIC_ENTITY);
    LOCAL

```

ISO TC184/SC4/WG11 N137

```
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that is_member returns TRUE when value submitted belongs to the
  aggregate. *)
  bool := is_member(aggr, aggr_with_data[4], NO_ERROR, ?);
  assert(bool);

  (* Ensures that is_member returns FALSE when value submitted does not
  belong to the aggregate. *)
  bool := is_member(aggr, aggr_with_data[3], NO_ERROR, ?);
  assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation remove by index
for an aggregate of type LIST of entity instances.
Parameters:
aggr - an aggregate of type LIST of entity instances;
the aggregate shall be that processed by
test_aggregate_entity_list_put_by_index;
inst - an entity instance.
*)
PROCEDURE test_aggregate_entity_list_remove_by_index(aggr:LIST [1:?] OF
GENERIC_ENTITY; inst:GENERIC_ENTITY);
  LOCAL
    inst2 : GENERIC_ENTITY;
  END_LOCAL;

  (* Ensures that remove_by_index reports an IX_NVLD error when the index
  submitted is outside of the legal range. *)
  remove_by_index(aggr, 0, IX_NVLD, aggr);
  remove_by_index(aggr, 5, IX_NVLD, aggr);

  (* Ensures that remove_by_index works correctly when the index submitted is
  from a legal range. *)
  remove_by_index(aggr, 2, NO_ERROR, ?);
  inst2 := get_by_index(aggr, 2, NO_ERROR, ?);
  assert(inst2 :=: inst);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation add before current
member
for an aggregate of type LIST of entity instances. Also testing the SDAI
operation add
after current member in the case when such an aggregate is empty.
Parameters:
iter - an iterator over an aggregate of type LIST of entity instances;
the aggregate shall be empty;
aggr_with_data - an aggregate containing entity instances that will be
added to the list specified by the iterator.
*)
PROCEDURE test_aggregate_entity_list_add_before_current_member(iter:iterator;
  aggr_with_data:BAG [0:?] OF GENERIC_ENTITY);
```



```

    (* Ensures that after add_after_current_member operation for an empty list
    iterator is positioned at
    the beginning of the list (there is no current member). *)
    atEnd(iter, NO_ERROR, ?);
    add_after_current_member(iter, aggr_with_data[1], NO_ERROR, ?);
    bool := next(iter, NO_ERROR, ?);
    assert(bool);
    bool := remove_current_member(iter, NO_ERROR, ?);

    (* Ensures that add_before_current_member reports an IR_NEXS error when
    iterator is not provided. *)
    add_before_current_member(?, aggr_with_data[3], IR_NEXS, ?);

    (* Ensures that add_before_current_member reports a VA_NSET error when an
    unset value is submitted. *)
    add_before_current_member(iter, ?, VA_NSET, ?);

    (* Ensures that add_before_current_member reports a VT_NVLD error when
    value of a wrong type is submitted. *)
    add_before_current_member(iter, 'something', VT_NVLD, ?);

    (* Ensures that after add_before_current_member operation for an empty list
    iterator is positioned at
    the end of the list (there is no current member). *)
    beginning(iter, NO_ERROR, ?);
    add_before_current_member(iter, aggr_with_data[3], NO_ERROR, ?);
    bool := previous(iter, NO_ERROR, ?);
    assert(bool);

    (* Ensures that add_before_current_member works correctly when all
    parameters are correct. *)
    add_before_current_member(iter, aggr_with_data[2], NO_ERROR, ?);
    atEnd(iter, NO_ERROR, ?);
    add_before_current_member(iter, aggr_with_data[4], NO_ERROR, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation add after current member
for an aggregate of type LIST of entity instances.
Parameters:
iter - an iterator over an aggregate of type LIST of entity instances;
the aggregate shall be that processed by
test_aggregate_entity_list_add_before_current_member;
aggr_with_data - an aggregate containing entity instances that will be
added to the list specified by the iterator.
*)
PROCEDURE test_aggregate_entity_list_add_after_current_member(iter:iterator;
    aggr_with_data:BAG [0:?] OF GENERIC_ENTITY);

    (* Positioning of the iterator at the end of the aggregate. *)
    atEnd(iter, NO_ERROR, ?);

    (* Ensures that add_after_current_member reports an IR_NEXS error when
    iterator is not provided. *)
    add_after_current_member(?, aggr_with_data[5], IR_NEXS, ?);

    (* Ensures that add_after_current_member reports a VA_NSET error when an
    unset value is submitted. *)
    add_after_current_member(iter, ?, VA_NSET, ?);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that add_after_current_member reports a VT_NVLD error when value
of a wrong type is submitted. *)
```

```
add_after_current_member(iter, 'something', VT_NVLD, ?);
```

```
(* Ensures that add_after_current_member works correctly when all
parameters are correct. *)
```

```
add_after_current_member(iter, aggr_with_data[5], NO_ERROR, ?);
```

```
add_after_current_member(iter, aggr_with_data[6], NO_ERROR, ?);
```

```
beginning(iter, NO_ERROR, ?);
```

```
add_after_current_member(iter, aggr_with_data[1], NO_ERROR, ?);
```

```
END_PROCEDURE;
```

```
(*
Testing the basic functionality of the SDAI operation get current member
for an aggregate of type LIST of entity instances.
```

```
Parameters:
```

```
iter - an iterator over an aggregate of type LIST of entity instances;
the aggregate shall be that processed by
```

```
test_aggregate_entity_list_add_after_current_member;
```

```
aggr_with_data - an aggregate containing entity instances that will be
compared with elements in the list specified by the iterator.
```

```
*)
```

```
PROCEDURE test_aggregate_entity_list_get_current_member(iter:iterator;
aggr_with_data:BAG [0:?] OF GENERIC_ENTITY);
```

```
LOCAL
```

```
inst : GENERIC_ENTITY;
```

```
bool : BOOLEAN;
```

```
END_LOCAL;
```

```
(* Ensures that get_current_member reports an IR_NEXS error when iterator
is not provided. *)
```

```
inst := get_current_member(?, IR_NEXS, ?);
```

```
(* Ensures that get_current_member reports an IR_NSET error when iterator
has no current member set. *)
```

```
beginning(iter, NO_ERROR, ?);
```

```
inst := get_current_member(iter, IR_NSET, ?);
```

```
atEnd(iter, NO_ERROR, ?);
```

```
inst := get_current_member(iter, IR_NSET, ?);
```

```
(* Ensures that get_current_member works correctly when iterator has
current member set. *)
```

```
bool := previous(iter, NO_ERROR, ?);
```

```
inst := get_current_member(iter, NO_ERROR, ?);
```

```
assert(inst = aggr_with_data[6]);
```

```
bool := previous(iter, NO_ERROR, ?);
```

```
inst := get_current_member(iter, NO_ERROR, ?);
```

```
assert(inst = aggr_with_data[5]);
```

```
bool := previous(iter, NO_ERROR, ?);
```

```
inst := get_current_member(iter, NO_ERROR, ?);
```

```
assert(inst = aggr_with_data[4]);
```

```
bool := previous(iter, NO_ERROR, ?);
```

```
inst := get_current_member(iter, NO_ERROR, ?);
```

```
assert(inst = aggr_with_data[3]);
```

```
bool := previous(iter, NO_ERROR, ?);
```

```
inst := get_current_member(iter, NO_ERROR, ?);
```

```
assert(inst = aggr_with_data[2]);
```

```
bool := previous(iter, NO_ERROR, ?);
```

```

    inst := get_current_member(iter, NO_ERROR, ?);
    assert(inst = aggr_with_data[1]);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation put current member
for an aggregate of type LIST of entity instances.
Parameters:
iter - an iterator over an aggregate of type LIST of entity instances;
the aggregate shall be that processed by
test_aggregate_entity_list_add_after_current_member;
inst - an entity instance.
*)
PROCEDURE test_aggregate_entity_list_put_current_member(iter:iterator;
inst:GENERIC_ENTITY);
    LOCAL
        inst2 : GENERIC_ENTITY;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that put_current_member reports an IR_NEXS error when iterator
is not provided. *)
    put_current_member(?, inst, IR_NEXS, ?);

    (* Ensures that put_current_member reports an IR_NSET error when iterator
has no current member set. *)
    beginning(iter, NO_ERROR, ?);
    put_current_member(iter, inst, IR_NSET, ?);
    atEnd(iter, NO_ERROR, ?);
    put_current_member(iter, inst, IR_NSET, ?);

    (* Ensures that put_current_member reports a VA_NSET error when an unset
value is submitted. *)
    put_current_member(iter, ?, VA_NSET, ?);

    (* Ensures that put_current_member reports a VT_NVLD error when value of a
wrong type is submitted. *)
    put_current_member(iter, 'something', VT_NVLD, ?);

    (* Ensures that put_current_member works correctly when all parameters are
correct. *)
    bool := previous(iter, NO_ERROR, ?);
    put_current_member(iter, inst, NO_ERROR, ?);
    inst2 := get_current_member(iter, NO_ERROR, ?);
    assert(inst2 = inst);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation remove current member
for an aggregate of type LIST of entity instances.
Parameters:
iter - an iterator over an aggregate of type LIST of entity instances;
the aggregate shall be that processed by
test_aggregate_entity_list_put_current_member;
inst - an entity instance.
*)

```

ISO TC184/SC4/WG11 N137

```
PROCEDURE test_aggregate_entity_list_remove_current_member(iter:iterator;
inst:GENERIC_ENTITY);
  LOCAL
    inst2 : GENERIC_ENTITY;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that remove_current_member reports an IR_NEXS error when
iterator is not provided. *)
  bool := remove_current_member(?, IR_NEXS, ?);

  (* Ensures that remove_current_member reports an IR_NSET error when
iterator has no current member set. *)
  atEnd(iter, NO_ERROR, ?);
  bool := remove_current_member(iter, NO_ERROR, ?);
  beginning(iter, NO_ERROR, ?);
  bool := remove_current_member(iter, NO_ERROR, ?);

  (* Ensures that remove_current_member works correctly when iterator has
current member set. *)
  bool := next(iter, NO_ERROR, ?);
  bool := remove_current_member(iter, NO_ERROR, ?);
  assert(bool);
  inst2 := get_current_member(iter, NO_ERROR, ?);
  assert(inst2 = inst);

  (* Ensures that remove_current_member returns FALSE when iterator refers to
the last member of the aggregate. *)
  atEnd(iter, NO_ERROR, ?);
  bool := previous(iter, NO_ERROR, ?);
  bool := remove_current_member(iter, NO_ERROR, ?);
  assert(NOT bool);

END_PROCEDURE;
```

```
(*
Testing the basic functionality of the SDAI operations
get by index, put by index, add by index, remove by index, is member,
add before current member, add after current member, get current member,
put current member, and remove current member
for an aggregate of type LIST of entity instances.
Parameters:
aggr - an aggregate of type LIST of instances of entity nu in greek schema;
the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter;
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_aggregate_entity_list(aggr:LIST [1:?] OF nu; iter:iterator;
model:sdai_model);
  LOCAL
    inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    inst3 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    inst4 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    inst5 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
```

```

    inst6 : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
    END_LOCAL;

    (* Testing of the list operation add by index. *)
    test_aggregate_entity_list_add_by_index(aggr, [inst1, inst2, inst3,
inst4]);

    (* Testing of the aggregate operation get by index. *)
    test_aggregate_entity_list_get_by_index(aggr, [inst1, inst2, inst3,
inst4]);

    (* Testing of the aggregate operation put by index. *)
    test_aggregate_entity_list_put_by_index(aggr, inst4);

    (* Testing of the aggregate operation is member. *)
    test_aggregate_entity_list_is_member(aggr, [inst1, inst2, inst3, inst4]);

    (* Testing of the list operation remove by index. *)
    test_aggregate_entity_list_remove_by_index(aggr, inst4);

    (* Making the given aggregate empty. *)
    macro_clear_aggregate(aggr);

    (* Testing of the list operation add before current member. *)
    test_aggregate_entity_list_add_before_current_member(iter, [inst1, inst2,
inst3, inst4]);

    (* Testing of the list operation add after current member. *)
    test_aggregate_entity_list_add_after_current_member(iter, [inst1, inst2,
inst3, inst4, inst5, inst6]);

    (* Testing of the aggregate operation get current member. *)
    test_aggregate_entity_list_get_current_member(iter, [inst1, inst2, inst3,
inst4, inst5, inst6]);

    (* Testing of the aggregate operation put current member. *)
    test_aggregate_entity_list_put_current_member(iter, inst1);

    (* Testing of the aggregate operation remove current member. *)
    test_aggregate_entity_list_remove_current_member(iter, inst2);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation add unordered
for an aggregate of type SET of LOGICAL.
Parameters:
aggr - an aggregate of type SET of LOGICAL; the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_logical_set_add_unordered(aggr:SET [2:4] OF LOGICAL;
iter:iterator);
    LOCAL
        aggr_auxiliary : SET [0:?] OF LOGICAL;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that add_unordered reports a VA_NSET error when an unset value
is submitted. *)

```

ISO TC184/SC4/WG11 N137

```
add_unordered(aggr, ?, VA_NSET, ?);

(* Ensures that add_unordered reports a VT_NVLD error when value of a wrong
type is submitted. *)
add_unordered(aggr, 'something', VT_NVLD, ?);

(* Ensures that add_unordered works correctly when all parameters are
correct. *)
add_unordered(aggr, UNKNOWN, NO_ERROR, ?);
add_unordered(aggr, FALSE, NO_ERROR, ?);
add_unordered(aggr, TRUE, NO_ERROR, ?);
beginning(iter, NO_ERROR, ?);
bool := next(iter, NO_ERROR, ?);
aggr_auxiliary[1] := get_current_member(iter, NO_ERROR, ?);
bool := next(iter, NO_ERROR, ?);
aggr_auxiliary[2] := get_current_member(iter, NO_ERROR, ?);
bool := next(iter, NO_ERROR, ?);
aggr_auxiliary[3] := get_current_member(iter, NO_ERROR, ?);
bool := macro_compare_aggregates(aggr, aggr_auxiliary);
assert(bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation remove unordered
for an aggregate of type SET of LOGICAL.
Parameter:
aggr - an aggregate of type SET of LOGICAL;
the aggregate shall be that processed by
test_aggregate_logical_set_add_unordered.
*)
PROCEDURE test_aggregate_logical_set_remove_unordered(aggr:SET [2:4] OF
LOGICAL);

(* Ensures that remove_unordered reports a VA_NSET error when an unset
value is submitted. *)
remove_unordered(aggr, ?, VA_NSET, ?);

(* Ensures that remove_unordered reports a VT_NVLD error when value of a
wrong type is submitted. *)
remove_unordered(aggr, 'something', VT_NVLD, ?);

(* Ensures that remove_unordered works correctly when parameters are
correct. *)
remove_unordered(aggr, FALSE, NO_ERROR, ?);

(* Ensures that remove_unordered reports a VA_NEXS error when value does
not exist in the aggregate. *)
remove_unordered(aggr, FALSE, VA_NEXS, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations is member and get
member count
for an aggregate of type SET of LOGICAL.
Parameter:
aggr - an aggregate of type SET of LOGICAL;
```

```

the aggregate shall be that processed by
test_aggregate_logical_set_remove_unordered.
*)
PROCEDURE test_aggregate_logical_set_is_member(aggr:SET [2:4] OF LOGICAL);
  LOCAL
    bool : BOOLEAN;
    count : INTEGER;
  END_LOCAL;

  (* Ensures that is_member returns TRUE when value submitted belongs to the
  aggregate. *)
  bool := is_member(aggr, UNKNOWN, NO_ERROR, ?);
  assert(bool);

  (* Ensures that is_member returns FALSE when value submitted does not
  belong to the aggregate. *)
  bool := is_member(aggr, FALSE, NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that get_member_count works correctly. *)
  count := get_member_count(aggr, NO_ERROR, ?);
  assert(count = 2);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation get current member
for an aggregate of type SET of LOGICAL.
Parameter:
iter - an iterator over an aggregate of type SET of LOGICAL;
the aggregate shall contain values UNKNOWN, FALSE and TRUE.
*)
PROCEDURE test_aggregate_logical_set_get_current_member(iter:iterator);
  LOCAL
    aggr : SET [0:?] OF LOGICAL;
    logic : LOGICAL;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that get_current_member reports an IR_NEXS error when iterator
  is not provided. *)
  logic := get_current_member(?, IR_NEXS, ?);

  (* Ensures that get_current_member reports an IR_NSET error when iterator
  is at the beginning. *)
  beginning(iter, NO_ERROR, ?);
  logic := get_current_member(iter, IR_NSET, ?);

  (* Ensures that get_current_member works correctly when iterator has
  current member set. *)
  bool := next(iter, NO_ERROR, ?);
  aggr[1] := get_current_member(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr[2] := get_current_member(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr[3] := get_current_member(iter, NO_ERROR, ?);
  bool := macro_compare_aggregates(aggr, [UNKNOWN, FALSE, TRUE]);
  assert(bool);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that get_current_member reports an IR_NSET error when iterator
is at the end. *)
bool := next(iter, NO_ERROR, ?);
logic := get_current_member(iter, IR_NSET, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation put current member
for an aggregate of type SET of LOGICAL.
Parameters:
aggr - an aggregate of type SET of LOGICAL; its elements shall be UNKNOWN and
TRUE;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_logical_set_put_current_member(aggr:SET [2:4] OF
LOGICAL; iter:iterator);
  LOCAL
    logic : LOGICAL;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that put_current_member reports an IR_NEXS error when iterator
is not provided. *)
  put_current_member(?, FALSE, IR_NEXS, ?);

  (* Ensures that put_current_member reports an IR_NSET error when iterator
is at the beginning. *)
  beginning(iter, NO_ERROR, ?);
  put_current_member(iter, FALSE, IR_NSET, ?);

  (* Ensures that put_current_member reports a VA_NSET error when an unset
value is submitted. *)
  bool := next(iter, NO_ERROR, ?);
  put_current_member(iter, ?, VA_NSET, ?);

  (* Ensures that put_current_member reports a VT_NVLD error when value of a
wrong type is submitted. *)
  put_current_member(iter, 'something', VT_NVLD, ?);

  (* Ensures that put_current_member works correctly when all parameters are
correct. *)
  logic := get_current_member(iter, NO_ERROR, ?);
  put_current_member(iter, FALSE, NO_ERROR, ?);
  bool := is_member(aggr, FALSE, NO_ERROR, ?);
  assert(bool);
  bool := is_member(aggr, logic, NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that put_current_member reports an IR_NSET error when iterator
is at the end. *)
  bool := next(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  put_current_member(iter, FALSE, IR_NSET, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation remove current member
```



```

for an aggregate of type SET of LOGICAL.
Parameters:
aggr - an aggregate of type SET of LOGICAL; its elements shall be UNKNOWN,
FALSE and TRUE;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_logical_set_remove_current_member(aggr:SET [2:4] OF
LOGICAL; iter:iterator);
  LOCAL
    logic : LOGICAL;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that remove_current_member reports an IR_NEXS error when
iterator is not provided. *)
  bool := remove_current_member(?, IR_NEXS, ?);

  (* Ensures that remove_current_member reports an IR_NSET error when
iterator is at the beginning. *)
  beginning(iter, NO_ERROR, ?);
  bool := remove_current_member(iter, NO_ERROR, ?);

  (* Ensures that remove_current_member works correctly when iterator has
current member set. *)
  bool := next(iter, NO_ERROR, ?);
  logic := get_current_member(iter, NO_ERROR, ?);
  bool := remove_current_member(iter, NO_ERROR, ?);
  assert(bool);
  bool := is_member(aggr, logic, NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that remove_current_member returns FALSE when iterator refers to
the last member of the aggregate. *)
  bool := next(iter, NO_ERROR, ?);
  bool := remove_current_member(iter, NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that remove_current_member reports an IR_NSET error when
iterator is at the end. *)
  bool := remove_current_member(iter, NO_ERROR, ?);

END_PROCEDURE;

(*)
Testing the basic functionality of the SDAI operations
add unordered, remove unordered, is member,
get current member, put current member, and remove current member
for an aggregate of type SET of LOGICAL.
Parameters:
aggr - an aggregate of type SET of LOGICAL; the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_logical_set(aggr:SET [2:4] OF LOGICAL;
iter:iterator);

  (* Testing of the aggregate operation add unordered. *)
  test_aggregate_logical_set_add_unordered(aggr, iter);

  (* Testing of the aggregate operation remove unordered. *)
  test_aggregate_logical_set_remove_unordered(aggr);

```

ISO TC184/SC4/WG11 N137

```
(* Testing of the aggregate operation is member. *)
test_aggregate_logical_set_is_member(aggr);

(* Making the given set empty. *)
macro_clear_aggregate(aggr);

(* Initializing the set with all three logical values. *)
add_unordered(aggr, UNKNOWN, NO_ERROR, ?);
add_unordered(aggr, FALSE, NO_ERROR, ?);
add_unordered(aggr, TRUE, NO_ERROR, ?);

(* Testing of the aggregate operation get current member. *)
test_aggregate_logical_set_get_current_member(iter);

(* Testing of the aggregate operation remove current member. *)
test_aggregate_logical_set_remove_current_member(aggr, iter);

(* Making the given set empty. *)
macro_clear_aggregate(aggr);

(* Initializing the set with logical values UNKNOWN and TRUE. *)
add_unordered(aggr, UNKNOWN, NO_ERROR, ?);
add_unordered(aggr, TRUE, NO_ERROR, ?);

(* Testing of the aggregate operation put current member. *)
test_aggregate_logical_set_put_current_member(aggr, iter);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation add unordered
for an aggregate of type SET of EXPRESS TYPE.
Parameters:
aggr - an aggregate of type SET of EXPRESS TYPE; the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_select_defined_type_set_add_unordered(aggr:SET [1:?]
OF xi; iter:iterator);
  LOCAL
    aggr_auxiliary : SET [0:?] OF xi;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that add_unordered reports a VA_NSET error when an unset value
is submitted. *)
  add_unordered(aggr, ?, VA_NSET, ?);

  (* Ensures that add_unordered reports a VT_NVLD error when value of a wrong
type is submitted. *)
  add_unordered(aggr, 'something', VT_NVLD, ?);

  (* Ensures that add_unordered works correctly when all parameters are
correct. *)
  add_unordered(aggr, xi(10), NO_ERROR, ?);
  add_unordered(aggr, xi(20), NO_ERROR, ?);
  add_unordered(aggr, xi(30), NO_ERROR, ?);
  beginning(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr_auxiliary[1] := get_current_member(iter, NO_ERROR, ?);
```

```

bool := next(iter, NO_ERROR, ?);
aggr_auxiliary[2] := get_current_member(iter, NO_ERROR, ?);
bool := next(iter, NO_ERROR, ?);
aggr_auxiliary[3] := get_current_member(iter, NO_ERROR, ?);
bool := macro_compare_aggregates(aggr, aggr_auxiliary);
assert(bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation remove unordered
for an aggregate of type SET of EXPRESS TYPE.
Parameter:
aggr - an aggregate of type SET of EXPRESS TYPE;
the aggregate shall be that processed by
test_aggregate_select_defined_type_set_add_unordered.
*)
PROCEDURE test_aggregate_select_defined_type_set_remove_unordered(aggr:SET
[1:?] OF xi);

    (* Ensures that remove_unordered reports a VA_NSET error when an unset
value is submitted. *)
    remove_unordered(aggr, ?, VA_NSET, ?);

    (* Ensures that remove_unordered reports a VT_NVLD error when value of a
wrong type is submitted. *)
    remove_unordered(aggr, 'something', VT_NVLD, ?);

    (* Ensures that remove_unordered works correctly when parameters are
correct. *)
    remove_unordered(aggr, xi(20), NO_ERROR, ?);

    (* Ensures that remove_unordered reports a VA_NEXS error when value does
not exist in the aggregate. *)
    remove_unordered(aggr, xi(20), VA_NEXS, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation is member
for an aggregate of type SET of EXPRESS TYPE.
Parameter:
aggr - an aggregate of type SET of EXPRESS TYPE;
the aggregate shall be that processed by
test_aggregate_select_defined_type_set_remove_unordered.
*)
PROCEDURE test_aggregate_select_defined_type_set_is_member(aggr:SET [1:?] OF
xi);
    LOCAL
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that is_member returns TRUE when value submitted belongs to the
aggregate. *)
    bool := is_member(aggr, xi(10), NO_ERROR, ?);
    assert(bool);

    (* Ensures that is_member returns FALSE when value submitted does not
belong to the aggregate. *)

```

ISO TC184/SC4/WG11 N137

```
    bool := is_member(aggr, xi(20), NO_ERROR, ?);
    assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation get current member
for an aggregate of type SET of EXPRESS TYPE.
Parameter:
iter - an iterator over an aggregate of type SET of EXPRESS TYPE;
the aggregate shall contain values 10, 20 and 30.
*)
PROCEDURE
test_aggregate_select_defined_type_set_get_current_member(iter:iterator);
    LOCAL
        aggr : SET [0:?] OF xi;
        int : INTEGER;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that get_current_member reports an IR_NEXS error when iterator
is not provided. *)
    logic := get_current_member(?, IR_NEXS, ?);

    (* Ensures that get_current_member reports an IR_NSET error when iterator
is at the beginning. *)
    beginning(iter, NO_ERROR, ?);
    int := get_current_member(iter, IR_NSET, ?);

    (* Ensures that get_current_member works correctly when iterator has
current member set. *)
    bool := next(iter, NO_ERROR, ?);
    aggr[1] := get_current_member(iter, NO_ERROR, ?);
    bool := next(iter, NO_ERROR, ?);
    aggr[2] := get_current_member(iter, NO_ERROR, ?);
    bool := next(iter, NO_ERROR, ?);
    aggr[3] := get_current_member(iter, NO_ERROR, ?);
    bool := macro_compare_aggregates(aggr, [10, 20, 30]);
    assert(bool);

    (* Ensures that get_current_member reports an IR_NSET error when iterator
is at the end. *)
    bool := next(iter, NO_ERROR, ?);
    int := get_current_member(iter, IR_NSET, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation put current member
for an aggregate of type SET of EXPRESS TYPE.
Parameters:
aggr - an aggregate of type SET of EXPRESS TYPE; its elements shall be 10 and
30;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_select_defined_type_set_put_current_member(aggr:SET
[1:?] OF xi; iter:iterator);
    LOCAL
        int : INTEGER;
```

```

    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that put_current_member reports an IR_NEXS error when iterator
  is not provided. *)
  put_current_member(?, xi(20), IR_NEXS, ?);

  (* Ensures that put_current_member reports an IR_NSET error when iterator
  is at the beginning. *)
  beginning(iter, NO_ERROR, ?);
  put_current_member(iter, xi(20), IR_NSET, ?);

  (* Ensures that put_current_member reports a VA_NSET error when an unset
  value is submitted. *)
  bool := next(iter, NO_ERROR, ?);
  put_current_member(iter, ?, VA_NSET, ?);

  (* Ensures that put_current_member reports a VT_NVLD error when value of a
  wrong type is submitted. *)
  put_current_member(iter, 'something', VT_NVLD, ?);

  (* Ensures that put_current_member works correctly when all parameters are
  correct. *)
  int := get_current_member(iter, NO_ERROR, ?);
  put_current_member(iter, xi(20), NO_ERROR, ?);
  bool := is_member(aggr, xi(20), NO_ERROR, ?);
  assert(bool);
  bool := is_member(aggr, int, NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that put_current_member reports an IR_NSET error when iterator
  is at the end. *)
  bool := next(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  put_current_member(iter, xi(20), IR_NSET, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation remove current member
for an aggregate of type SET of EXPRESS TYPE.
Parameters:
aggr - an aggregate of type SET of EXPRESS TYPE; its elements shall be 10, 20
and 30;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE
test_aggregate_select_defined_type_set_remove_current_member(aggr:SET [1?] OF
xi; iter:iterator);
  LOCAL
    int : INTEGER;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that remove_current_member reports an IR_NEXS error when
  iterator is not provided. *)
  bool := remove_current_member(?, IR_NEXS, ?);

  (* Ensures that remove_current_member reports an IR_NSET error when
  iterator is at the beginning. *)

```

ISO TC184/SC4/WG11 N137

```
beginning(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);

(* Ensures that remove_current_member works correctly when iterator has
current member set. *)
bool := next(iter, NO_ERROR, ?);
int := get_current_member(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);
assert(bool);
bool := is_member(aggr, int, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that remove_current_member returns FALSE when iterator refers to
the last member of the aggregate. *)
bool := next(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that remove_current_member reports an IR_NSET error when
iterator is at the end. *)
bool := remove_current_member(iter, NO_ERROR, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
add unordered, remove unordered, is member,
get current member, put current member, and remove current member
for an aggregate of type SET of EXPRESS TYPE.
Parameters:
aggr - an aggregate of type SET of EXPRESS TYPE; the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_select_defined_type_set(aggr:SET [1:?] OF xi;
iter:iterator);

(* Testing of the aggregate operation add unordered. *)
test_aggregate_select_defined_type_set_add_unordered(aggr, iter);

(* Testing of the aggregate operation remove unordered. *)
test_aggregate_select_defined_type_set_remove_unordered(aggr);

(* Testing of the aggregate operation is member. *)
test_aggregate_select_defined_type_set_is_member(aggr);

(* Making the given set empty. *)
macro_clear_aggregate(aggr);

(* Initializing the set with three integer values. *)
add_unordered(aggr, xi(10), NO_ERROR, ?);
add_unordered(aggr, xi(20), NO_ERROR, ?);
add_unordered(aggr, xi(30), NO_ERROR, ?);

(* Testing of the aggregate operation get current member. *)
test_aggregate_select_defined_type_set_get_current_member(iter);

(* Testing of the aggregate operation remove current member. *)
test_aggregate_select_defined_type_set_remove_current_member(aggr, iter);

(* Making the given set empty. *)
```

```

macro_clear_aggregate(aggr);

(* Initializing the set with integer values 10 and 30. *)
add_unordered(aggr, xi(10), NO_ERROR, ?);
add_unordered(aggr, xi(30), NO_ERROR, ?);

(* Testing of the aggregate operation put current member. *)
test_aggregate_select_defined_type_set_put_current_member(aggr, iter);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation add unordered
for an aggregate of type BAG of STRING.
Parameters:
aggr - an aggregate of type BAG of STRING; the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_string_bag_add_unordered(aggr:BAG [2:4] OF STRING;
iter:iterator);
  LOCAL
    aggr_auxiliary : BAG [0:?] OF STRING;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that add_unordered reports a VA_NSET error when an unset value
is submitted. *)
  add_unordered(aggr, ?, VA_NSET, ?);

  (* Ensures that add_unordered reports a VT_NVLD error when value of a wrong
type is submitted. *)
  add_unordered(aggr, 5.5, VT_NVLD, ?);

  (* Ensures that add_unordered works correctly when all parameters are
correct. *)
  add_unordered(aggr, 'first', NO_ERROR, ?);
  add_unordered(aggr, 'second', NO_ERROR, ?);
  add_unordered(aggr, 'second', NO_ERROR, ?);
  beginning(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr_auxiliary[1] := get_current_member(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr_auxiliary[2] := get_current_member(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr_auxiliary[3] := get_current_member(iter, NO_ERROR, ?);
  bool := macro_compare_aggregates(aggr, aggr_auxiliary);
  assert(bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation remove unordered
for an aggregate of type BAG of STRING.
Parameters:
aggr - an aggregate of type BAG of STRING;
the aggregate shall be that processed by
test_aggregate_string_bag_add_unordered;
iter - an iterator over an aggregate specified by the first parameter.
*)

```

ISO TC184/SC4/WG11 N137

```
PROCEDURE test_aggregate_string_bag_remove_unordered(aggr:BAG [2:4] OF STRING;
iter:iterator);
  LOCAL
    str1 : STRING;
    str2 : STRING;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that remove_unordered reports a VA_NSET error when an unset
value is submitted. *)
  remove_unordered(aggr, ?, VA_NSET, ?);

  (* Ensures that remove_unordered reports a VT_NVLD error when value of a
wrong type is submitted. *)
  remove_unordered(aggr, 5.5, VT_NVLD, ?);

  (* Ensures that remove_unordered works correctly when parameters are
correct and removes only
one occurrence of the specified value. *)
  remove_unordered(aggr, 'second', NO_ERROR, ?);
  beginning(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  str1 := get_current_member(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  str2 := get_current_member(iter, NO_ERROR, ?);
  bool := macro_compare_aggregates([str1, str2], ['first', 'second']);
  assert(bool);

  (* Ensures that remove_unordered reports a VA_NEXS error when value does
not exist in the aggregate. *)
  remove_unordered(aggr, 'third', VA_NEXS, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations is member get member
count
for an aggregate of type BAG of STRING.
Parameter:
aggr - an aggregate of type BAG of STRING;
the aggregate shall be that processed by
test_aggregate_string_bag_remove_unordered.
*)
PROCEDURE test_aggregate_string_bag_is_member(aggr:BAG [2:4] OF STRING);
  LOCAL
    bool : BOOLEAN;
    count : INTEGER;
  END_LOCAL;

  (* Ensures that is_member returns TRUE when value submitted belongs to the
aggregate. *)
  bool := is_member(aggr, 'first', NO_ERROR, ?);
  assert(bool);

  (* Ensures that is_member returns FALSE when value submitted does not
belong to the aggregate. *)
  bool := is_member(aggr, 'third', NO_ERROR, ?);
  assert(NOT bool);

  (* Ensures that get_member_count works correctly. *)
```



```

    count := get_member_count(aggr, NO_ERROR, ?);
    assert(count = 2);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation get current member
for an aggregate of type BAG of STRING.
Parameters:
aggr - an aggregate of type BAG of STRING;
the aggregate shall contain three values of type STRING;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_string_bag_get_current_member(aggr:BAG [2:4] OF
STRING; iter:iterator);
    LOCAL
        aggr_auxiliary : BAG [0:?] OF STRING;
        str : STRING;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that get_current_member reports an IR_NEXS error when iterator
is not provided. *)
    str := get_current_member(?, IR_NEXS, ?);

    (* Ensures that get_current_member reports an IR_NSET error when iterator
is at the beginning. *)
    beginning(iter, NO_ERROR, ?);
    str := get_current_member(iter, IR_NSET, ?);

    (* Ensures that get_current_member works correctly when iterator has
current member set. *)
    bool := next(iter, NO_ERROR, ?);
    aggr_auxiliary[1] := get_current_member(iter, NO_ERROR, ?);
    bool := next(iter, NO_ERROR, ?);
    aggr_auxiliary[2] := get_current_member(iter, NO_ERROR, ?);
    bool := next(iter, NO_ERROR, ?);
    aggr_auxiliary[3] := get_current_member(iter, NO_ERROR, ?);
    bool := macro_compare_aggregates(aggr, aggr_auxiliary);
    assert(bool);

    (* Ensures that get_current_member reports an IR_NSET error when iterator
is at the end. *)
    bool := next(iter, NO_ERROR, ?);
    str := get_current_member(iter, IR_NSET, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation put current member
for an aggregate of type BAG of STRING.
Parameters:
aggr - an aggregate of type BAG of STRING;
its elements shall be strings different than string 'third';
iter - an iterator over an aggregate specified by the first parameter
*)
PROCEDURE test_aggregate_string_bag_put_current_member(aggr:BAG [2:4] OF
STRING; iter:iterator);
    LOCAL

```

ISO TC184/SC4/WG11 N137

```
    str : STRING;
    bool : BOOLEAN;
END_LOCAL;

(* Ensures that put_current_member reports an IR_NEXS error when iterator
is not provided. *)
put_current_member(?, 'third', IR_NEXS, ?);

(* Ensures that put_current_member reports an IR_NSET error when iterator
is at the beginning. *)
beginning(iter, NO_ERROR, ?);
put_current_member(iter, 'third', IR_NSET, ?);

(* Ensures that put_current_member reports a VA_NSET error when an unset
value is submitted. *)
bool := next(iter, NO_ERROR, ?);
put_current_member(iter, ?, VA_NSET, ?);

(* Ensures that put_current_member reports a VT_NVLD error when value of a
wrong type is submitted. *)
put_current_member(iter, 5.5, VT_NVLD, ?);

(* Ensures that put_current_member works correctly when all parameters are
correct. *)
str := get_current_member(iter, NO_ERROR, ?);
put_current_member(iter, 'third', NO_ERROR, ?);
bool := is_member(aggr, 'third', NO_ERROR, ?);
assert(bool);
bool := is_member(aggr, str, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that put_current_member reports an IR_NSET error when iterator
is at the end. *)
bool := next(iter, NO_ERROR, ?);
bool := next(iter, NO_ERROR, ?);
put_current_member(iter, 'third', IR_NSET, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation remove current member
for an aggregate of type BAG of STRING.
Parameters:
aggr - an aggregate of type BAG of STRING;
it shall contain three values of type STRING, first of which is unique;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_string_bag_remove_current_member(aggr:BAG [2:4] OF
STRING; iter:iterator);
    LOCAL
        str : STRING;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that remove_current_member reports an IR_NEXS error when
iterator is not provided. *)
    bool := remove_current_member(?, IR_NEXS, ?);

    (* Ensures that remove_current_member reports an IR_NSET error when
iterator is at the beginning. *)
```

```

beginning(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);

(* Ensures that remove_current_member works correctly when iterator has
current member set. *)
bool := next(iter, NO_ERROR, ?);
str := get_current_member(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);
assert(bool);
bool := is_member(aggr, str, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that remove_current_member returns FALSE when iterator refers to
the last member of the aggregate. *)
bool := next(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);
assert(NOT bool);

(* Ensures that remove_current_member reports an IR_NSET error when
iterator is at the end. *)
bool := remove_current_member(iter, NO_ERROR, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations
add unordered, remove unordered, is member,
get current member, put current member, and remove current member
for an aggregate of type BAG of STRING.
Parameters:
aggr - an aggregate of type BAG of STRING;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_string_bag(aggr:BAG [2:4] OF STRING; iter:iterator);

(* Making the given bag empty. *)
macro_clear_aggregate(aggr);

(* Testing of the aggregate operation add unordered. *)
test_aggregate_string_bag_add_unordered(aggr, iter);

(* Testing of the aggregate operation remove unordered. *)
test_aggregate_string_bag_remove_unordered(aggr, iter);

(* Testing of the aggregate operation is member. *)
test_aggregate_string_bag_is_member(aggr);

(* Making the given bag empty. *)
macro_clear_aggregate(aggr);

(* Initializing the bag with three STRING values. *)
add_unordered(aggr, 'first', NO_ERROR, ?);
add_unordered(aggr, 'second', NO_ERROR, ?);
add_unordered(aggr, 'second', NO_ERROR, ?);

(* Testing of the aggregate operation get current member. *)
test_aggregate_string_bag_get_current_member(aggr, iter);

(* Testing of the aggregate operation remove current member. *)
test_aggregate_string_bag_remove_current_member(aggr, iter);

```

ISO TC184/SC4/WG11 N137

```
(* Making the given bag empty. *)
macro_clear_aggregate(aggr);

(* Initializing the bag with two STRING values. *)
add_unordered(aggr, 'first', NO_ERROR, ?);
add_unordered(aggr, 'second', NO_ERROR, ?);

(* Testing of the aggregate operation put current member. *)
test_aggregate_string_bag_put_current_member(aggr, iter);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation add unordered
for an aggregate of type BAG of REAL.
Parameters:
aggr - an aggregate of type BAG of REAL; the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_real_bag_add_unordered(aggr:BAG [0:?] OF REAL;
iter:iterator);
    LOCAL
        aggr_auxiliary : BAG [0:?] OF REAL;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that add_unordered reports a VA_NSET error when an unset value
is submitted. *)
    add_unordered(aggr, ?, VA_NSET, ?);

    (* Ensures that add_unordered reports a VT_NVLD error when value of a wrong
type is submitted. *)
    add_unordered(aggr, 'something', VT_NVLD, ?);

    (* Ensures that add_unordered works correctly when all parameters are
correct. *)
    add_unordered(aggr, 1.1, NO_ERROR, ?);
    add_unordered(aggr, 2.2, NO_ERROR, ?);
    add_unordered(aggr, 2.2, NO_ERROR, ?);
    beginning(iter, NO_ERROR, ?);
    bool := next(iter, NO_ERROR, ?);
    aggr_auxiliary[1] := get_current_member(iter, NO_ERROR, ?);
    bool := next(iter, NO_ERROR, ?);
    aggr_auxiliary[2] := get_current_member(iter, NO_ERROR, ?);
    bool := next(iter, NO_ERROR, ?);
    aggr_auxiliary[3] := get_current_member(iter, NO_ERROR, ?);
    bool := macro_compare_aggregates(aggr, aggr_auxiliary);
    assert(bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation remove unordered
for an aggregate of type BAG of REAL.
Parameters:
aggr - an aggregate of type BAG of REAL;
the aggregate shall be that processed by
test_aggregate_real_bag_add_unordered;

```

```

iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_real_bag_remove_unordered(aggr:BAG [0:?] OF REAL;
iter:iterator);
  LOCAL
    real1 : REAL;
    real2 : REAL;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that remove_unordered reports a VA_NSET error when an unset
value is submitted. *)
  remove_unordered(aggr, ?, VA_NSET, ?);

  (* Ensures that remove_unordered reports a VT_NVLD error when value of a
wrong type is submitted. *)
  remove_unordered(aggr, 'something', VT_NVLD, ?);

  (* Ensures that remove_unordered works correctly when parameters are
correct and removes only
one occurrence of the specified value. *)
  remove_unordered(aggr, 2.2, NO_ERROR, ?);
  beginning(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  real1 := get_current_member(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  real2 := get_current_member(iter, NO_ERROR, ?);
  bool := macro_compare_aggregates([real1, real2], [1.1, 2.2]);
  assert(bool);

  (* Ensures that remove_unordered reports a VA_NEXS error when value does
not exist in the aggregate. *)
  remove_unordered(aggr, 3.3, VA_NEXS, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation is_member
for an aggregate of type BAG of REAL.
Parameter:
aggr - an aggregate of type BAG of REAL;
the aggregate shall be that processed by
test_aggregate_real_bag_remove_unordered.
*)
PROCEDURE test_aggregate_real_bag_is_member(aggr:BAG [0:?] OF REAL);
  LOCAL
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that is_member returns TRUE when value submitted belongs to the
aggregate. *)
  bool := is_member(aggr, 1.1, NO_ERROR, ?);
  assert(bool);

  (* Ensures that is_member returns FALSE when value submitted does not
belong to the aggregate. *)
  bool := is_member(aggr, 3.3, NO_ERROR, ?);
  assert(NOT bool);

END_PROCEDURE;

```

ISO TC184/SC4/WG11 N137

```
(*
Testing the basic functionality of the SDAI operation get current member
for an aggregate of type BAG of STRING.
Parameters:
aggr - an aggregate of type BAG of REAL;
the aggregate shall contain three values of type REAL;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_real_bag_get_current_member(aggr:BAG [0:?] OF REAL;
iter:iterator);
  LOCAL
    aggr_auxiliary : BAG [0:?] OF REAL;
    re : REAL;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that get_current_member reports an IR_NEXS error when iterator
is not provided. *)
  re := get_current_member(?, IR_NEXS, ?);

  (* Ensures that get_current_member reports an IR_NSET error when iterator
is at the beginning. *)
  beginning(iter, NO_ERROR, ?);
  re := get_current_member(iter, IR_NSET, ?);

  (* Ensures that get_current_member works correctly when iterator has
current member set. *)
  bool := next(iter, NO_ERROR, ?);
  aggr_auxiliary[1] := get_current_member(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr_auxiliary[2] := get_current_member(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr_auxiliary[3] := get_current_member(iter, NO_ERROR, ?);
  bool := macro_compare_aggregates(aggr, aggr_auxiliary);
  assert(bool);

  (* Ensures that get_current_member reports an IR_NSET error when iterator
is at the end. *)
  bool := next(iter, NO_ERROR, ?);
  re := get_current_member(iter, IR_NSET, ?);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation put current member
for an aggregate of type BAG of REAL.
Parameters:
aggr - an aggregate of type BAG of REAL;
its elements shall be real numbers different than 3.3;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_real_bag_put_current_member(aggr:BAG [0:?] OF REAL;
iter:iterator);
  LOCAL
    re : REAL;
    bool : BOOLEAN;
  END_LOCAL;
```

```

    (* Ensures that put_current_member reports an IR_NEXS error when iterator
    is not provided. *)
    put_current_member(?, 3.3, IR_NEXS, ?);

    (* Ensures that put_current_member reports an IR_NSET error when iterator
    is at the beginning. *)
    beginning(iter, NO_ERROR, ?);
    put_current_member(iter, 3.3, IR_NSET, ?);

    (* Ensures that put_current_member reports a VA_NSET error when an unset
    value is submitted. *)
    bool := next(iter, NO_ERROR, ?);
    put_current_member(iter, ?, VA_NSET, ?);

    (* Ensures that put_current_member reports a VT_NVLD error when value of a
    wrong type is submitted. *)
    put_current_member(iter, 'something', VT_NVLD, ?);

    (* Ensures that put_current_member works correctly when all parameters are
    correct. *)
    re := get_current_member(iter, NO_ERROR, ?);
    put_current_member(iter, 3.3, NO_ERROR, ?);
    bool := is_member(aggr, 3.3, NO_ERROR, ?);
    assert(bool);
    bool := is_member(aggr, re, NO_ERROR, ?);
    assert(NOT bool);

    (* Ensures that put_current_member reports an IR_NSET error when iterator
    is at the end. *)
    bool := next(iter, NO_ERROR, ?);
    bool := next(iter, NO_ERROR, ?);
    put_current_member(iter, 3.3, IR_NSET, ?);

```

```
END_PROCEDURE;
```

```

(*)
Testing the basic functionality of the SDAI operation remove current member
for an aggregate of type BAG of REAL.
Parameters:
aggr - an aggregate of type BAG of REAL;
it shall contain three values of type REAL, first of which is unique;
iter - an iterator over an aggregate specified by the first parameter.
*)

```

```
PROCEDURE test_aggregate_real_bag_remove_current_member(aggr:BAG [0:?] OF
REAL; iter:iterator);
```

```

    LOCAL
        re : STRING;
        bool : BOOLEAN;
    END_LOCAL;
```

```

    (* Ensures that remove_current_member reports an IR_NEXS error when
    iterator is not provided. *)
    bool := remove_current_member(?, IR_NEXS, ?);

```

```

    (* Ensures that remove_current_member reports an IR_NSET error when
    iterator is at the beginning. *)
    beginning(iter, NO_ERROR, ?);
    bool := remove_current_member(iter, NO_ERROR, ?);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that remove_current_member works correctly when iterator has
current member set. *)
```

```
bool := next(iter, NO_ERROR, ?);
re := get_current_member(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);
assert(bool);
bool := is_member(aggr, re, NO_ERROR, ?);
assert(NOT bool);
```

```
(* Ensures that remove_current_member returns FALSE when iterator refers to
the last member of the aggregate. *)
```

```
bool := next(iter, NO_ERROR, ?);
bool := remove_current_member(iter, NO_ERROR, ?);
assert(NOT bool);
```

```
(* Ensures that remove_current_member reports an IR_NSET error when
iterator is at the end. *)
```

```
bool := remove_current_member(iter, NO_ERROR, ?);
```

```
END_PROCEDURE;
```

```
(*
```

```
Testing the basic functionality of the SDAI operations
add unordered, remove unordered, is member,
get current member, put current member, and remove current member
for an aggregate of type BAG of REAL.
```

```
Parameters:
```

```
aggr - an aggregate of type BAG of REAL;
```

```
iter - an iterator over an aggregate specified by the first parameter.
```

```
*)
```

```
PROCEDURE test_aggregate_real_bag(aggr:BAG [0:?] OF REAL; iter:iterator);
```

```
(* Making the given bag empty. *)
```

```
macro_clear_aggregate(aggr);
```

```
(* Testing of the aggregate operation add unordered. *)
```

```
test_aggregate_real_bag_add_unordered(aggr, iter);
```

```
(* Testing of the aggregate operation remove unordered. *)
```

```
test_aggregate_real_bag_remove_unordered(aggr, iter);
```

```
(* Testing of the aggregate operation is member. *)
```

```
test_aggregate_real_bag_is_member(aggr);
```

```
(* Making the given bag empty. *)
```

```
macro_clear_aggregate(aggr);
```

```
(* Initializing the bag with three REAL values. *)
```

```
add_unordered(aggr, 1.1, NO_ERROR, ?);
```

```
add_unordered(aggr, 2.2, NO_ERROR, ?);
```

```
add_unordered(aggr, 2.2, NO_ERROR, ?);
```

```
(* Testing of the aggregate operation get current member. *)
```

```
test_aggregate_real_bag_get_current_member(aggr, iter);
```

```
(* Testing of the aggregate operation remove current member. *)
```

```
test_aggregate_real_bag_remove_current_member(aggr, iter);
```

```
(* Making the given bag empty. *)
```

```
macro_clear_aggregate(aggr);
```



```

(* Initializing the bag with two REAL values. *)
add_unordered(aggr, 1.1, NO_ERROR, ?);
add_unordered(aggr, 2.2, NO_ERROR, ?);

(* Testing of the aggregate operation put current member. *)
test_aggregate_real_bag_put_current_member(aggr, iter);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation get by index
for an aggregate of type ARRAY of INTEGER.
Parameter:
aggr - an aggregate of type ARRAY of INTEGER;
it shall have unset value in the first position.
*)
PROCEDURE test_aggregate_integer_array_get_by_index(aggr:ARRAY [0:4] OF
INTEGER);
  LOCAL
    int : INTEGER;
  END_LOCAL;

  (* Ensures that get_by_index reports an IX_NVLD error when the index
submitted is outside of the legal range. *)
  int := get_by_index(aggr, -1, IX_NVLD, aggr);
  int := get_by_index(aggr, 5, IX_NVLD, aggr);

  (* Ensures that get_by_index reports a VA_NSET error when the array has no
value at the
  specified position.
  *)
  int := get_by_index(aggr, 0, VA_NSET, ?);

  (* Ensures that get_by_index works correctly when the index submitted is
from a legal range. *)
  put_by_index(aggr, 1, 100, NO_ERROR, ?);
  int := get_by_index(aggr, 1, NO_ERROR, ?);
  assert(int = 100);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation test by index
for an aggregate of type ARRAY of INTEGER.
Parameter:
aggr - an aggregate of type ARRAY of INTEGER;
it shall have unset value in the first position.
*)
PROCEDURE test_aggregate_integer_array_test_by_index(aggr:ARRAY [0:4] OF
INTEGER);
  LOCAL
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that test_by_index reports an IX_NVLD error when the index
submitted is outside of the legal range. *)
  bool := test_by_index(aggr, -1, IX_NVLD, aggr);
  bool := test_by_index(aggr, 5, IX_NVLD, aggr);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that test_by_index returns FALSE when the array has no value at the specified position. *)
```

```
bool := test_by_index(aggr, 0, NO_ERROR, ?);  
assert(NOT bool);
```

```
(* Ensures that test_by_index returns TRUE when the array has some value at the specified position. *)
```

```
put_by_index(aggr, 2, 200, NO_ERROR, ?);  
bool := test_by_index(aggr, 2, NO_ERROR, ?);  
assert(bool);
```

```
END_PROCEDURE;
```

```
(*  
Testing the basic functionality of the SDAI operation put by index  
for an aggregate of type ARRAY of INTEGER.
```

```
Parameter:
```

```
aggr - an aggregate of type ARRAY of INTEGER.
```

```
*)
```

```
PROCEDURE test_aggregate_integer_array_put_by_index(aggr:ARRAY [0:4] OF  
INTEGER);
```

```
LOCAL
```

```
int : INTEGER;
```

```
END_LOCAL;
```

```
(* Ensures that put_by_index reports an IX_NVLD error when the index submitted is outside of the legal range. *)
```

```
put_by_index(aggr, -1, 100, IX_NVLD, aggr);  
put_by_index(aggr, 5, 100, IX_NVLD, aggr);
```

```
(* Ensures that put_by_index reports a VA_NSET error when an unset value is submitted. *)
```

```
put_by_index(aggr, 1, ?, VA_NSET, ?);
```

```
(* Ensures that put_by_index reports a VT_NVLD error when value of a wrong type is submitted. *)
```

```
put_by_index(aggr, 1, 'something', VT_NVLD, ?);
```

```
(* Ensures that put_by_index works correctly when all parameters are correct. *)
```

```
put_by_index(aggr, 3, 300, NO_ERROR, ?);  
int := get_by_index(aggr, 3, NO_ERROR, ?);  
assert(int = 300);
```

```
END_PROCEDURE;
```

```
(*  
Testing the basic functionality of the SDAI operation unset value by index  
for an aggregate of type ARRAY of INTEGER.
```

```
Parameter:
```

```
aggr - an aggregate of type ARRAY of INTEGER;
```

```
it shall have some value in the position with index 1.
```

```
*)
```

```
PROCEDURE test_aggregate_integer_array_unset_value_by_index(aggr:ARRAY [0:4]  
OF INTEGER);
```

```
LOCAL
```

```
bool : BOOLEAN;
```

```
END_LOCAL;
```

```

    (* Ensures that unset_value_by_index reports an IX_NVLD error when the
    index submitted is outside of the legal range. *)
    unset_value_by_index(aggr, -1, IX_NVLD, aggr);
    unset_value_by_index(aggr, 5, IX_NVLD, aggr);

    (* Ensures that unset_value_by_index works correctly when the index
    submitted is from a legal range. *)
    unset_value_by_index(aggr, 1, NO_ERROR, ?);

    (* Ensures that array value is really unset after unset_value_by_index
    operation. *)
    bool := test_by_index(aggr, 1, NO_ERROR, ?);
    assert(NOT bool);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operations is member and get
member count
for an aggregate of type ARRAY of INTEGER.
Parameter:
aggr - an aggregate of type ARRAY of INTEGER;
it shall contain value 300 but not 400.
*)
PROCEDURE test_aggregate_integer_array_is_member(aggr:ARRAY [0:4] OF INTEGER);
    LOCAL
        bool : BOOLEAN;
        count : INTEGER;
    END_LOCAL;

    (* Ensures that is_member returns TRUE when value submitted belongs to the
    aggregate. *)
    bool := is_member(aggr, 300, NO_ERROR, ?);
    assert(bool);

    (* Ensures that is_member returns FALSE when value submitted does not
    belong to the aggregate. *)
    bool := is_member(aggr, 400, NO_ERROR, ?);
    assert(NOT bool);

    (* Ensures that get_member_count works correctly. *)
    count := get_member_count(aggr, NO_ERROR, ?);
    assert(count = 5);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation get current member
for an aggregate of type ARRAY of INTEGER.
Parameter:
iter - an iterator over an aggregate of type ARRAY of INTEGER;
the aggregate shall be that processed by
test_aggregate_integer_array_put_current_member.
*)
PROCEDURE test_aggregate_integer_array_get_current_member(iter:iterator);
    LOCAL
        int : INTEGER;
        bool : BOOLEAN;

```

ISO TC184/SC4/WG11 N137

```
END_LOCAL;

(* Ensures that get_current_member reports an IR_NEXS error when iterator
is not provided. *)
int := get_current_member(?, IR_NEXS, ?);

(* Ensures that get_current_member reports an IR_NSET error when iterator
has no current member set. *)
atEnd(iter, NO_ERROR, ?);
int := get_current_member(iter, IR_NSET, ?);
beginning(iter, NO_ERROR, ?);
int := get_current_member(iter, IR_NSET, ?);

(* Ensures that get_current_member reports a VA_NSET error when iterator
has current member unset. *)
bool := next(iter, NO_ERROR, ?);
int := get_current_member(iter, VA_NSET, ?);

(* Ensures that get_current_member works correctly when iterator has
current member set. *)
bool := next(iter, NO_ERROR, ?);
int := get_current_member(iter, NO_ERROR, ?);
assert(int = 100);

END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation put current member
for an aggregate of type ARRAY of INTEGER.
Parameter:
iter - an iterator over an aggregate of type ARRAY of INTEGER.
*)
PROCEDURE test_aggregate_integer_array_put_current_member(iter:iterator);
LOCAL
    bool : BOOLEAN;
END_LOCAL;

(* Ensures that put_current_member reports an IR_NEXS error when iterator
is not provided. *)
put_current_member(?, 100, IR_NEXS, ?);

(* Ensures that put_current_member reports an IR_NSET error when iterator
has no current member set. *)
atEnd(iter, NO_ERROR, ?);
put_current_member(iter, 100, IR_NSET, ?);
beginning(iter, NO_ERROR, ?);
put_current_member(iter, 100, IR_NSET, ?);

(* Ensures that put_current_member reports a VA_NSET error when an unset
value is submitted. *)
bool := next(iter, NO_ERROR, ?);
put_current_member(iter, ?, VA_NSET, ?);

(* Ensures that put_current_member reports a VT_NVLD error when value of a
wrong type is submitted. *)
put_current_member(iter, 'something', VT_NVLD, ?);

(* Ensures that put_current_member works correctly when parameters are
correct. *)
bool := next(iter, NO_ERROR, ?);
```

```

    put_current_member(iter, 100, NO_ERROR, ?);
END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation test current member
for an aggregate of type ARRAY of INTEGER.
Parameter:
iter - an iterator over an aggregate of type ARRAY of INTEGER;
the aggregate shall be that processed by
test_aggregate_integer_array_put_current_member.
*)
PROCEDURE test_aggregate_integer_array_test_current_member(iter:iterator);
    LOCAL
        int : INTEGER;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that test_current_member reports an IR_NEXS error when iterator
is not provided. *)
    bool := test_current_member(?, IR_NEXS, ?);

    (* Ensures that test_current_member reports an IR_NSET error when iterator
has no current member set. *)
    atEnd(iter, NO_ERROR, ?);
    bool := test_current_member(iter, IR_NSET, ?);
    beginning(iter, NO_ERROR, ?);
    bool := test_current_member(iter, IR_NSET, ?);

    (* Ensures that test_current_member returns FALSE when value is unset at a
position referred by the iterator. *)
    bool := next(iter, NO_ERROR, ?);
    bool := test_current_member(iter, NO_ERROR, ?);
    assert(NOT bool);

    (* Ensures that test_current_member returns TRUE when value is set at a
position referred by the iterator. *)
    bool := next(iter, NO_ERROR, ?);
    bool := test_current_member(iter, NO_ERROR, ?);
    assert(bool);
END_PROCEDURE;

(*
Testing the basic functionality of the SDAI operation unset value current
member
for an aggregate of type ARRAY of INTEGER.
Parameter:
iter - an iterator over an aggregate of type ARRAY of INTEGER;
the aggregate shall be that processed by
test_aggregate_integer_array_put_current_member.
*)
PROCEDURE
test_aggregate_integer_array_unset_value_current_member(iter:iterator);
    LOCAL
        bool : BOOLEAN;
    END_LOCAL;

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that unset_value_current_member reports an IR_NEXS error when
iterator is not provided. *)
unset_value_current_member(?, IR_NEXS, ?);
```

```
(* Ensures that unset_value_current_member reports an IR_NSET error when
iterator has no current member set. *)
atEnd(iter, NO_ERROR, ?);
unset_value_current_member(iter, IR_NSET, ?);
beginning(iter, NO_ERROR, ?);
unset_value_current_member(iter, IR_NSET, ?);
```

```
(* Ensures that unset_value_current_member works correctly when iterator
has current member. *)
bool := next(iter, NO_ERROR, ?);
bool := next(iter, NO_ERROR, ?);
unset_value_current_member(iter, NO_ERROR, ?);
```

```
(* Ensures that array value is really unset after
unset_value_current_member operation. *)
bool := test_current_member(iter, NO_ERROR, ?);
assert(NOT bool);
```

```
END_PROCEDURE;
```

```
(*
Testing the basic functionality of the SDAI operations
get by index, test by index, put by index, unset value by index, is member,
get current member, put current member, test current member,
and unset value current member for an aggregate of type ARRAY of INTEGER.
Parameters:
aggr - an aggregate of type ARRAY of INTEGER;
the aggregate shall have all values unset;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_integer_array(aggr:ARRAY [0:4] OF INTEGER;
iter:iterator);
```

```
(* Testing of the aggregate operation get by index. *)
test_aggregate_integer_array_get_by_index(aggr);
```

```
(* Testing of the array operation test by index. *)
test_aggregate_integer_array_test_by_index(aggr);
```

```
(* Testing of the aggregate operation put by index. *)
test_aggregate_integer_array_put_by_index(aggr);
```

```
(* Testing of the array operation unset value by index. *)
test_aggregate_integer_array_unset_value_by_index(aggr);
```

```
(* Testing of the aggregate operation is member. *)
test_aggregate_integer_array_is_member(aggr);
```

```
(* Testing of the aggregate operation put current member. *)
test_aggregate_integer_array_put_current_member(iter);
```

```
(* Testing of the aggregate operation get current member. *)
test_aggregate_integer_array_get_current_member(iter);
```

```
(* Testing of the array operation test current member. *)
test_aggregate_integer_array_test_current_member(iter);
```

```

    (* Testing of the array operation unset value current member. *)
    test_aggregate_integer_array_unset_value_current_member(iter);

END_PROCEDURE;

(*
Testing the SDAI operations for non-nested aggregates.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_aggregate_simple(model:sdai_model);
    LOCAL
        inst1 : GENERIC_ENTITY := create_entity_instance('GREEK.SIGMA', model,
NO_ERROR, ?);
        inst2 : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON', model,
NO_ERROR, ?);
        aggr : AGGREGATE OF GENERIC;
        iter : iterator;
    END_LOCAL;

    (* An aggregate of type LIST of NUMBER and its iterator are created. *)
    aggr := create_aggregate_instance(inst1, 'GREEK.SIGMA.S1', NO_ERROR, ?);
    iter := create_iterator(aggr, NO_ERROR, ?);

    (* Testing of the SDAI operations on iterator creation, deletion and
positioning. *)
    test_iterator(aggr, iter);

    (* Checks cases when aggregate operations are illegal for the LIST. *)
    test_aggregate_operations_disallowed_for_list(aggr, iter);

    (* Testing the basic functionality of the SDAI operations for an aggregate
of type
    LIST of NUMBER.
    *)
    test_aggregate_number_list(aggr, iter);

    (* An aggregate of type LIST of entity instances and its iterator are
created. *)
    aggr := create_aggregate_instance(inst2, 'GREEK.EPSILON.E2', NO_ERROR, ?);
    iter := create_iterator(aggr, NO_ERROR, ?);

    (* Testing the basic functionality of the SDAI operations for an aggregate
of type
    LIST of entity instances.
    *)
    test_aggregate_entity_list(aggr, iter, model);

    (* An aggregate of type SET of LOGICAL and its iterator are created. *)
    aggr := create_aggregate_instance(inst1, 'GREEK.SIGMA.S4', NO_ERROR, ?);
    iter := create_iterator(aggr, NO_ERROR, ?);

    (* Checks cases when aggregate operations are illegal for the SET. *)
    test_aggregate_operations_disallowed_for_set(aggr, iter);

    (* Testing the basic functionality of the SDAI operations for an aggregate
of type
    SET of LOGICAL.
    *)

```

ISO TC184/SC4/WG11 N137

```
test_aggregate_logical_set(aggr, iter);

(* An aggregate of type SET of EXPRESS TYPE and its iterator are created.
*)
aggr := create_aggregate_instance(inst1, 'GREEK.SIGMA.S9', NO_ERROR, ?);
iter := create_iterator(aggr, NO_ERROR, ?);

(* Testing the basic functionality of the SDAI operations for an aggregate
of type
    SET of EXPRESS TYPE.
*)
test_aggregate_select_defined_type_set(aggr, iter);

(* An aggregate of type BAG of STRING and its iterator are created. *)
aggr := create_aggregate_instance(inst1, 'GREEK.SIGMA.S6', NO_ERROR, ?);
iter := create_iterator(aggr, NO_ERROR, ?);

(* Testing iterator operation next for the unordered collection. *)
test_iterator_next_for_unordered_collection(aggr, iter);

(* Testing the basic functionality of the SDAI operations for an aggregate
of type
    BAG of STRING.
*)
test_aggregate_string_bag(aggr, iter);

(* An aggregate of type BAG of REAL and its iterator are created. *)
aggr := create_aggregate_instance(inst1, 'GREEK.SIGMA.S2', NO_ERROR, ?);
iter := create_iterator(aggr, NO_ERROR, ?);

(* Testing the basic functionality of the SDAI operations for an aggregate
of type
    BAG of REAL.
*)
test_aggregate_real_bag(aggr, iter);

(* An aggregate of type ARRAY of INTEGER and its iterator are created. *)
aggr := create_aggregate_instance(inst1, 'GREEK.SIGMA.S3', NO_ERROR, ?);
iter := create_iterator(aggr, NO_ERROR, ?);

(* Checks cases when aggregate operations are illegal for the ARRAY. *)
test_aggregate_operations_disallowed_for_array(aggr, iter);

(* Testing the basic functionality of the SDAI operations for an aggregate
of type
    ARRAY of INTEGER.
*)
test_aggregate_integer_array(aggr, iter);

END_PROCEDURE;

(*
Testing of the SDAI operations on iterator creation, deletion and positioning:
create iterator, delete iterator, beginning, end, next, and previous.
Parameters:
aggr - an aggregate of type LIST of NUMBER; the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_iterator(aggr:LIST [0:?] OF NUMBER; iter:iterator);
```



```

(* Testing iterator operations create iterator and delete iterator. *)
test_aggregate_iterator_create_delete(aggr);

(* Testing iterator operations beginning and end. *)
test_iterator_beginning_end(iter);

(* Testing iterator operation next for the ordered collection. *)
test_iterator_next_for_ordered_collection(iter);

(* Testing iterator operation previous. *)
test_iterator_previous(iter);

END_PROCEDURE;

(*
Testing of the SDAI operations create iterator and delete iterator.
Parameter:
agg - an aggregate of type LIST of NUMBER; the aggregate shall be empty.
*)
PROCEDURE test_aggregate_iterator_create_delete(aggr:LIST [0:?] OF NUMBER);
  LOCAL
    iter : iterator;
    bool : BOOLEAN;
  END_LOCAL;

  (* Making the aggregate to contain one element. *)
  add_by_index(aggr, 1, 5.0, NO_ERROR, ?);

  (* Ensures that create_iterator works correctly. *)
  iter := create_iterator(aggr, NO_ERROR, ?);

  (* Ensures that newly created iterator is positioned at the beginning and
has no current member. *)
  bool := next(iter, NO_ERROR, ?);
  assert(bool);

  (* Ensures that delete_iterator reports an IR_NEXS error when iterator is
not provided. *)
  delete_iterator(?, IR_NEXS, ?);

  (* Ensures that delete_iterator works correctly when iterator is submitted.
*)
  delete_iterator(iter, NO_ERROR, ?);

  (* Ensures that iterator was deleted by the delete_iterator operation. *)
  beginning(iter, IR_NEXS, ?);

END_PROCEDURE;

(*
Testing of the SDAI operations for iterator: beginning and end.
Parameter:
iter - an iterator over an aggregate of type LIST of NUMBER;
the aggregate shall be that processed by
test_aggregate_iterator_create_delete.
*)
PROCEDURE test_iterator_beginning_end(iter:iterator);
  LOCAL
    numb : NUMBER;

```

ISO TC184/SC4/WG11 N137

```
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that beginning reports an IR_NEXS error when iterator is not
  provided. *)
  beginning(?, IR_NEXS, ?);

  (* Ensures that after beginning operation iterator is positioned at the
  beginning of the aggregate
  (there is no current member).
  *)
  beginning(iter, NO_ERROR, ?);
  numb := get_current_member(iter, IR_NSET, ?);
  bool := next(iter, NO_ERROR, ?);
  assert(bool);

  (* Ensures that atEnd reports an IR_NEXS error when iterator is not
  provided. *)
  atEnd(?, IR_NEXS, ?);

  (* Ensures that after end operation iterator is positioned at the end of
  the aggregate
  (there is no current member).
  *)
  atEnd(iter, NO_ERROR, ?);
  numb := get_current_member(iter, IR_NSET, ?);
  bool := previous(iter, NO_ERROR, ?);
  assert(bool);

END_PROCEDURE;

(*
Testing of the SDAI operation next for ordered collections.
Parameter:
iter - an iterator over an aggregate of type LIST of NUMBER;
the aggregate shall be that processed by
test_aggregate_iterator_create_delete.
*)
PROCEDURE test_iterator_next_for_ordered_collection(iter:iterator);
  LOCAL
    numb : NUMBER;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that next reports an IR_NEXS error when iterator is not
  provided. *)
  bool := next(?, IR_NEXS, ?);

  (* Ensures that next works correctly when before this operation iterator is
  at the beginning. *)
  beginning(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  numb := get_current_member(iter, NO_ERROR, ?);
  assert(numb = 5.0);

  (* Ensures that next works correctly when before this operation iterator is
  at the end. *)
  atEnd(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  assert(NOT bool);
```

```

bool := previous(iter, NO_ERROR, ?);
assert(bool);

(* Ensures that next works correctly when before this operation iterator is
at the last
  member of the aggregate.
*)
bool := next(iter, NO_ERROR, ?);
assert(NOT bool);
bool := previous(iter, NO_ERROR, ?);
assert(bool);

END_PROCEDURE;

(*
Testing of the SDAI operation previous.
Parameter:
iter - an iterator over an aggregate of type LIST of NUMBER;
the aggregate shall be that processed by
test_aggregate_iterator_create_delete.
*)
PROCEDURE test_iterator_previous(iter:iterator);
  LOCAL
    numb : NUMBER;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that previous reports an IR_NEXS error when iterator is not
provided. *)
  bool := previous(?, IR_NEXS, ?);

  (* Ensures that previous works correctly when before this operation
iterator is at the end. *)
  atEnd(iter, NO_ERROR, ?);
  bool := previous(iter, NO_ERROR, ?);
  numb := get_current_member(iter, NO_ERROR, ?);
  assert(numb = 5.0);

  (* Ensures that previous works correctly when before this operation
iterator is at the beginning. *)
  beginning(iter, NO_ERROR, ?);
  bool := previous(iter, NO_ERROR, ?);
  assert(NOT bool);
  bool := next(iter, NO_ERROR, ?);
  assert(bool);

  (* Ensures that previous works correctly when before this operation
iterator is at the
  first member of the aggregate.
*)
  bool := previous(iter, NO_ERROR, ?);
  assert(NOT bool);
  bool := next(iter, NO_ERROR, ?);
  assert(bool);

END_PROCEDURE;

(*
Testing of the SDAI operation next for unordered collections.

```

ISO TC184/SC4/WG11 N137

Parameters:

agg - an aggregate of type BAG of STRING; the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter.
*)

```
PROCEDURE test_iterator_next_for_unordered_collection(agg:BAG [2:4] OF  
STRING; iter:iterator);
```

```
    LOCAL  
        str1, str2 : STRING;  
        bool : BOOLEAN;  
    END_LOCAL;
```

```
    (* Adding two values to the aggregate. *)  
    add_unordered(agg, 'first', NO_ERROR, ?);  
    add_unordered(agg, 'second', NO_ERROR, ?);
```

```
    (* Ensures that for an unordered collection the repeated application of the  
next operation
```

```
    gives all members of the collection.
```

```
    *)  
    beginning(iter, NO_ERROR, ?);  
    bool := next(iter, NO_ERROR, ?);  
    str1 := get_current_member(iter, NO_ERROR, ?);  
    bool := next(iter, NO_ERROR, ?);  
    str2 := get_current_member(iter, NO_ERROR, ?);  
    bool := macro_compare_aggregates(agg, [str1, str2]);  
    assert(bool);
```

```
END_PROCEDURE;
```

```
(*  
Testing the cases when an SDAI operation on aggregates is applied to an  
aggregate  
whose type is LIST and this is disallowed for that operation. In all such  
cases an  
error indicator AI_NVLD shall be reported.
```

Parameters:

agg - an aggregate of type LIST;
iter - an iterator over an aggregate specified by the first parameter.
*)

```
PROCEDURE test_aggregate_operations_disallowed_for_list(agg:AGGREGATE OF  
GENERIC; iter:iterator);
```

```
    LOCAL  
        bool : BOOLEAN;  
    END_LOCAL;
```

```
    (* Ensures that an AI_NVLD error is reported when SET and BAG operations  
are applied to a LIST. *)
```

```
    add_unordered(agg, 7.7, AI_NVLD, agg);  
    remove_unordered(agg, 7.7, AI_NVLD, agg);
```

```
    (* Ensures that an AI_NVLD error is reported when ARRAY operations are  
applied to a LIST. *)
```

```
    test_by_index(agg, 1, AI_NVLD, agg);  
    unset_value_by_index(agg, 1, AI_NVLD, agg);  
    bool := test_current_member(iter, AI_NVLD, agg);  
    unset_value_current_member(iter, AI_NVLD, agg);
```

```
END_PROCEDURE;
```

```

(*)
Testing the cases when an SDAI operation on aggregates is applied to an
aggregate
whose type is SET and this is disallowed for that operation. In all such cases
an
error indicator AI_NVLD shall be reported.
Parameters:
aggr - an aggregate of type SET;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_operations_disallowed_for_set(aggr:AGGREGATE OF
GENERIC; iter:iterator);
  LOCAL
    logic : LOGICAL;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that an AI_NVLD error is reported when LIST and ARRAY operations
are applied to a SET. *)
  logic := get_by_index(aggr, 1, AI_NVLD, aggr);
  atEnd(iter, AI_NVLD, aggr);
  bool := previous(iter, AI_NVLD, aggr);
  put_by_index(aggr, 1, UNKNOWN, AI_NVLD, aggr);

  (* Ensures that an AI_NVLD error is reported when LIST operations are
applied to a SET. *)
  add_by_index(aggr, 1, UNKNOWN, AI_NVLD, aggr);
  remove_by_index(aggr, 1, AI_NVLD, aggr);
  add_before_current_member(iter, UNKNOWN, AI_NVLD, aggr);
  add_after_current_member(iter, UNKNOWN, AI_NVLD, aggr);

  (* Ensures that an AI_NVLD error is reported when ARRAY operations are
applied to a SET. *)
  test_by_index(aggr, 1, AI_NVLD, aggr);
  unset_value_by_index(aggr, 1, AI_NVLD, aggr);
  bool := test_current_member(iter, AI_NVLD, aggr);
  unset_value_current_member(iter, AI_NVLD, aggr);

END_PROCEDURE;

(*)
Testing the cases when an SDAI operation on aggregates is applied to an
aggregate
whose type is ARRAY and this is disallowed for that operation. In all such
cases an
error indicator AI_NVLD shall be reported.
Parameters:
aggr - an aggregate of type ARRAY;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_operations_disallowed_for_array(aggr:AGGREGATE OF
GENERIC; iter:iterator);
  LOCAL
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that an AI_NVLD error is reported when LIST operations are
applied to an ARRAY. *)
  add_by_index(aggr, 1, 10, AI_NVLD, aggr);
  remove_by_index(aggr, 1, AI_NVLD, aggr);

```

ISO TC184/SC4/WG11 N137

```
add_before_current_member(iter, 10, AI_NVLD, aggr);
add_after_current_member(iter, 10, AI_NVLD, aggr);

(* Ensures that an AI_NVLD error is reported when SET and BAG operations
are applied to an ARRAY. *)
add_unordered(aggr, 10, AI_NVLD, aggr);
remove_unordered(aggr, 10, AI_NVLD, aggr);

(* Ensures that an AI_NVLD error is reported when LIST, SET and BAG
operation remove_current_member
is applied to an ARRAY.
*)
bool := remove_current_member(iter, AI_NVLD, aggr);

END_PROCEDURE;

(*
Testing the SDAI operation add aggregate instance by index
for an aggregate of type LIST of select data type, where the set of selections
in the select data type includes LIST of REAL.
Parameters:
aggr - an aggregate of type LIST of select data type; the aggregate shall be
empty.
*)
PROCEDURE
test_aggregate_list_of_list_add_aggregate_instance_by_index(aggr:LIST [1:?] OF
nu);
LOCAL
aggr_added : LIST [1:3] OF REAL;
aggr_returned : LIST [1:3] OF REAL;
count : INTEGER;
bool : BOOLEAN;
END_LOCAL;

(* Ensures that indexing in the LIST starts from 1. *)
aggr_added := add_aggregate_instance_by_index(aggr, 0, ['GREEK.CHI'],
IX_NVLD, aggr);

(* Making an aggregate initially nonempty. *)
add_by_index(aggr, 1, xi(10), NO_ERROR, ?);
add_by_index(aggr, 2, xi(20), NO_ERROR, ?);
add_by_index(aggr, 3, xi(30), NO_ERROR, ?);

(* Ensures that add_aggregate_instance_by_index reports a VT_NVLD error
when
list of defined types is either empty (but should not be such) or
contains wrong elements
or is not submitted at all though is needed.
*)
aggr_added := add_aggregate_instance_by_index(aggr, 3, [], VT_NVLD, ?);
aggr_returned := add_aggregate_instance_by_index(aggr, 3, ['GREEK.XI'],
VT_NVLD, ?);
aggr_returned := add_aggregate_instance_by_index(aggr, 3, ?, VT_NVLD, ?);

(* Ensures that add_aggregate_instance_by_index works correctly when all
parameters are correct. *)
aggr_returned := add_aggregate_instance_by_index(aggr, 3, ['GREEK.CHI'],
NO_ERROR, ?);

(* Ensures that an aggregate created is empty. *)
```

```

count := get_member_count(aggr_added, NO_ERROR, ?);
assert(count = 0);

(* Ensures that an aggregate of a required type is created and is added in
the correct place. *)
add_by_index(aggr_added, 1, 5.5, NO_ERROR, ?);
aggr_returned := get_by_index(aggr, 3, NO_ERROR, ?);
bool := is_member(aggr_returned, 5.5, NO_ERROR, ?);
assert(bool);

(* Ensures that add_aggregate_instance_by_index reports an IX_NVLD error
when the index submitted exceeds
the count of aggregate members plus one. *)
aggr_added := add_aggregate_instance_by_index(aggr, 6, ['GREEK.CHI'],
IX_NVLD, aggr);

END_PROCEDURE;

(*
Testing the SDAI operation create aggregate instance by index
for an aggregate of type LIST of select data type, where the set of selections
in the select data type includes LIST of REAL.
Parameter:
aggr - an aggregate of type LIST of select data type;
the aggregate shall be that processed by
test_aggregate_list_of_list_add_aggregate_instance_by_index.
*)
PROCEDURE
test_aggregate_list_of_list_create_aggregate_instance_by_index(aggr:LIST [1:?]
OF nu);
LOCAL
aggr_created : LIST [1:3] OF REAL;
aggr_returned : LIST [1:3] OF REAL;
count : INTEGER;
bool : BOOLEAN;
END_LOCAL;

(* Ensures that create_aggregate_instance_by_index reports an IX_NVLD error
when the index submitted
is outside of the legal range.
*)
aggr_created := create_aggregate_instance_by_index(aggr, 0, ['GREEK.CHI'],
IX_NVLD, aggr);
aggr_created := create_aggregate_instance_by_index(aggr, 5, ['GREEK.CHI'],
IX_NVLD, aggr);

(* Ensures that create_aggregate_instance_by_index reports a VT_NVLD error
when
list of defined types is either empty (but should not be such) or
contains wrong elements
or is not submitted at all though is needed.
*)
aggr_created := create_aggregate_instance_by_index(aggr, 1, [], VT_NVLD,
?);
aggr_created := create_aggregate_instance_by_index(aggr, 1, ['GREEK.XI'],
VT_NVLD, ?);
aggr_created := create_aggregate_instance_by_index(aggr, 1, ?, VT_NVLD, ?);

(* Ensures that create_aggregate_instance_by_index works correctly when all
parameters are correct. *)

```

ISO TC184/SC4/WG11 N137

```
    aggr_created := create_aggregate_instance_by_index(aggr, 1, ['GREEK.CHI'],
NO_ERROR, ?);
```

```
    (* Ensures that an aggregate created is empty. *)
    count := get_member_count(aggr_created, NO_ERROR, ?);
    assert(count = 0);
```

```
    (* Ensures that an aggregate of a required type is created and is assigned
to the correct place. *)
```

```
    add_by_index(aggr_created, 1, 99.99, NO_ERROR, ?);
    aggr_returned := get_by_index(aggr, 1, NO_ERROR, ?);
    bool := is_member(aggr_returned, 99.99, NO_ERROR, ?);
    assert(bool);
```

```
END_PROCEDURE;
```

```
(*
Testing the SDAI operation create aggregate instance before current member
for an aggregate of type LIST of select data type, where the set of selections
in the select data type includes LIST of REAL.
```

```
Parameter:
```

```
iter - an iterator over an aggregate of type LIST of select data type;
the aggregate shall be empty;
```

```
*)
```

```
PROCEDURE
```

```
test_aggregate_list_of_list_create_aggregate_instance_before_current_member(it
er:iterator);
```

```
    LOCAL
```

```
        aggr_added : LIST [1:3] OF REAL;
```

```
        re : REAL;
```

```
        bool : BOOLEAN;
```

```
    END_LOCAL;
```

```
    (* Ensures that create_aggregate_instance_before_current_member reports an
IR_NEXS error
```

```
        when iterator is not provided.
```

```
    *)
```

```
    aggr_added := create_aggregate_instance_before_current_member(?,
['GREEK.CHI'], IR_NEXS, ?);
```

```
    (* Ensures that create_aggregate_instance_before_current_member reports a
VT_NVLD error when
```

```
        list of defined types is either empty (but should not be such) or
contains wrong elements
```

```
        or is not submitted at all though is needed.
```

```
    *)
```

```
    beginning(iter, NO_ERROR, ?);
```

```
    aggr_added := create_aggregate_instance_before_current_member(iter, [],
VT_NVLD, ?);
```

```
    aggr_added := create_aggregate_instance_before_current_member(iter,
['GREEK.XI'], VT_NVLD, ?);
```

```
    aggr_added := create_aggregate_instance_before_current_member(iter, ?,
VT_NVLD, ?);
```

```
    (* Ensures that after create_aggregate_instance_before_current_member
operation for an empty list
```

```
        iterator is positioned at the end of the list (there is no current
member).
```

```
    *)
```



```

    aggr_added := create_aggregate_instance_before_current_member(iter,
['GREEK.CHI'], NO_ERROR, ?);
    add_by_index(aggr_added, 1, 2.2, NO_ERROR, ?);
    bool := previous(iter, NO_ERROR, ?);
    assert(bool);

    (* Ensures that create_aggregate_instance_before_current_member works
correctly
    when all parameters are correct.
    *)
    aggr_added := create_aggregate_instance_before_current_member(iter,
['GREEK.CHI'], NO_ERROR, ?);
    add_by_index(aggr_added, 1, 1.1, NO_ERROR, ?);
    atEnd(iter, NO_ERROR, ?);
    aggr_added := create_aggregate_instance_before_current_member(iter,
['GREEK.CHI'], NO_ERROR, ?);
    add_by_index(aggr_added, 1, 3.3, NO_ERROR, ?);

    (* Checking if created aggregates were placed in correct positions. *)
    beginning(iter, NO_ERROR, ?);
    bool := next(iter, NO_ERROR, ?);
    aggr_added := get_current_member(iter, NO_ERROR, ?);
    re := get_by_index(aggr_added, 1, NO_ERROR, ?);
    assert(re = 1.1);
    bool := next(iter, NO_ERROR, ?);
    aggr_added := get_current_member(iter, NO_ERROR, ?);
    re := get_by_index(aggr_added, 1, NO_ERROR, ?);
    assert(re = 2.2);
    bool := next(iter, NO_ERROR, ?);
    aggr_added := get_current_member(iter, NO_ERROR, ?);
    re := get_by_index(aggr_added, 1, NO_ERROR, ?);
    assert(re = 3.3);

```

END_PROCEDURE;

```

(*)
Testing the SDAI operation create aggregate instance after current member
for an aggregate of type LIST of select data type, where the set of selections
in the select data type includes LIST of REAL.
Parameter:
iter - an iterator over an aggregate of type LIST of select data type;
the aggregate shall be empty;
*)
PROCEDURE
test_aggregate_list_of_list_create_aggregate_instance_after_current_member(ite
r:iterator);
    LOCAL
        aggr_added : LIST [1:3] OF REAL;
        re : REAL;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that create_aggregate_instance_after_current_member reports an
IR_NEXS error
    when iterator is not provided.
    *)
    aggr_added := create_aggregate_instance_after_current_member(?,
['GREEK.CHI'], IR_NEXS, ?);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that create_aggregate_instance_after_current_member reports a
VT_NVLD error when
    list of defined types is either empty (but should not be such) or
contains wrong elements
    or is not submitted at all though is needed.
*)
beginning(iter, NO_ERROR, ?);
aggr_added := create_aggregate_instance_after_current_member(iter, [],
VT_NVLD, ?);
aggr_added := create_aggregate_instance_after_current_member(iter,
['GREEK.XI'], VT_NVLD, ?);
aggr_added := create_aggregate_instance_after_current_member(iter, ?,
VT_NVLD, ?);

(* Ensures that after create_aggregate_instance_after_current_member
operation for an empty list
    iterator is positioned at the beginning of the list (there is no current
member).
*)
aggr_added := create_aggregate_instance_after_current_member(iter,
['GREEK.CHI'], NO_ERROR, ?);
add_by_index(aggr_added, 1, 2.2, NO_ERROR, ?);
bool := next(iter, NO_ERROR, ?);
assert(bool);

(* Ensures that create_aggregate_instance_after_current_member works
correctly
    when all parameters are correct.
*)
aggr_added := create_aggregate_instance_after_current_member(iter,
['GREEK.CHI'], NO_ERROR, ?);
add_by_index(aggr_added, 1, 3.3, NO_ERROR, ?);
beginning(iter, NO_ERROR, ?);
aggr_added := create_aggregate_instance_after_current_member(iter,
['GREEK.CHI'], NO_ERROR, ?);
add_by_index(aggr_added, 1, 1.1, NO_ERROR, ?);

(* Checking if created aggregates were placed in correct positions. *)
beginning(iter, NO_ERROR, ?);
bool := next(iter, NO_ERROR, ?);
aggr_added := get_current_member(iter, NO_ERROR, ?);
re := get_by_index(aggr_added, 1, NO_ERROR, ?);
assert(re = 1.1);
bool := next(iter, NO_ERROR, ?);
aggr_added := get_current_member(iter, NO_ERROR, ?);
re := get_by_index(aggr_added, 1, NO_ERROR, ?);
assert(re = 2.2);
bool := next(iter, NO_ERROR, ?);
aggr_added := get_current_member(iter, NO_ERROR, ?);
re := get_by_index(aggr_added, 1, NO_ERROR, ?);
assert(re = 3.3);

END_PROCEDURE;

(*
Testing the SDAI operation create_aggregate_instance_as_current_member
for an aggregate of type LIST of select data type, where the set of selections
in the select data type includes LIST of REAL.
Parameter:
iter - an iterator over an aggregate of type LIST of select data type;
```

```

the aggregate shall be that processed by
test_aggregate_list_of_list_create_aggregate_instance_after_current_member.
*)
PROCEDURE
test_aggregate_list_of_list_create_aggregate_instance_as_current_member(iter:i
erator);
  LOCAL
    aggr_created : LIST [1:3] OF REAL;
    aggr_returned : LIST [1:3] OF REAL;
    re : REAL;
    count : INTEGER;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that create_aggregate_instance_as_current_member reports an
  IR_NEXS error
  when iterator is not provided.
  *)
  aggr_created := create_aggregate_instance_as_current_member(?,
['GREEK.CHI'], IR_NEXS, ?);

  (* Ensures that create_aggregate_instance_as_current_member reports an
  IR_NSET error
  when iterator has no current member set.
  *)
  atEnd(iter, NO_ERROR, ?);
  aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.CHI'], IR_NSET, ?);
  beginning(iter, NO_ERROR, ?);
  aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.CHI'], IR_NSET, ?);

  (* Ensures that create_aggregate_instance_as_current_member reports a
  VT_NVLD error when
  list of defined types is either empty (but should not be such) or
  contains wrong elements
  or is not submitted at all though is needed.
  *)
  bool := next(iter, NO_ERROR, ?);
  aggr_created := create_aggregate_instance_as_current_member(iter, [],
VT_NVLD, ?);
  aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.XI'], VT_NVLD, ?);
  aggr_created := create_aggregate_instance_as_current_member(iter, ?,
VT_NVLD, ?);

  (* Ensures that create_aggregate_instance_as_current_member works correctly
  when all parameters are correct.
  *)
  aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.CHI'], NO_ERROR, ?);

  (* Ensures that an aggregate created is empty. *)
  count := get_member_count(aggr_created, NO_ERROR, ?);
  assert(count = 0);

  (* Ensures that an aggregate of a required type is created and is assigned
  to the correct place. *)
  add_by_index(aggr_created, 1, 7.7, NO_ERROR, ?);
  aggr_returned := get_current_member(iter, NO_ERROR, ?);

```

ISO TC184/SC4/WG11 N137

```
    re := get_by_index(aggr_returned, 1, NO_ERROR, ?);
    assert(re = 7.7);

END_PROCEDURE;

(*
Testing the SDAI operations add aggregate instance by index,
create aggregate instance by index, create aggregate instance before current
member,
create aggregate instance after current member, and
create aggregate instance as current member
for an aggregate of type LIST of select data type, where the set of selections
in the select data type includes LIST of REAL.
Parameters:
aggr - an aggregate of type LIST of select data type nu in greek schema;
the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_list_of_list(aggr:LIST [1:?] OF nu; iter:iterator);

    (* Testing of the list operation add aggregate instance by index. *)
    test_aggregate_list_of_list_add_aggregate_instance_by_index(aggr);

    (* Testing of the aggregate operation create aggregate instance by index.
*)
    test_aggregate_list_of_list_create_aggregate_instance_by_index(aggr);

    (* Making the given aggregate empty. *)
    macro_clear_aggregate(aggr);

    (* Testing of the list operation create aggregate instance before current
member. *)
    test_aggregate_list_of_list_create_aggregate_instance_before_current_member
(iter);

    (* Making the given aggregate empty. *)
    macro_clear_aggregate(aggr);

    (* Testing of the list operation create aggregate instance after current
member. *)
    test_aggregate_list_of_list_create_aggregate_instance_after_current_member(
iter);

    (* Testing of the aggregate operation create aggregate instance as current
member. *)
    test_aggregate_list_of_list_create_aggregate_instance_as_current_member(ite
r);

END_PROCEDURE;

(*
Testing the SDAI operation create aggregate instance unordered
for an aggregate of type SET of select data type, where the set of selections
in the select data type includes SET of data type nu in greek schema.
Parameters:
aggr - an aggregate of type SET of select data type; the aggregate shall be
empty.
*)
```

```

PROCEDURE
test_aggregate_set_of_set_create_aggregate_instance_unordered(aggr:SET [1:?]
OF rho);
  LOCAL
    aggr_created : SET [0:3] OF nu;
    aggr_auxiliary : SET [1:?] OF rho;
    count : INTEGER;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that create_aggregate_instance_unordered reports a VT_NVLD error
when
  list of defined types is either empty (but should not be such) or
contains wrong elements
  or is not submitted at all though is needed.
*)
  aggr_created := create_aggregate_instance_unordered(aggr, [], VT_NVLD, ?);
  aggr_created := create_aggregate_instance_unordered(aggr, ['GREEK.XI'],
VT_NVLD, ?);
  aggr_created := create_aggregate_instance_unordered(aggr, ?, VT_NVLD, ?);

  (* Ensures that create_aggregate_instance_unordered works correctly when
all parameters are correct. *)
  aggr_created := create_aggregate_instance_unordered(aggr,
['GREEK.UPSILON'], NO_ERROR, ?);
  aggr_created := create_aggregate_instance_unordered(aggr,
['GREEK.UPSILON'], NO_ERROR, ?);
  aggr_created := create_aggregate_instance_unordered(aggr,
['GREEK.UPSILON'], NO_ERROR, ?);

  (* Ensures that an aggregate created is empty. *)
  count := get_member_count(aggr_created, NO_ERROR, ?);
  assert(count = 0);

  (* Ensures that an aggregate of a required type is created. *)
  add_unordered(aggr_created, xi(10), NO_ERROR, ?);

  (* Ensures that all created aggregates are added to the top level
aggregate. *)
  beginning(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr_auxiliary[1] := get_current_member(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr_auxiliary[2] := get_current_member(iter, NO_ERROR, ?);
  bool := next(iter, NO_ERROR, ?);
  aggr_auxiliary[3] := get_current_member(iter, NO_ERROR, ?);
  bool := macro_compare_aggregates(aggr, aggr_auxiliary);
  assert(bool);

END_PROCEDURE;

(*
Testing the SDAI operation create aggregate instance as current member
for an aggregate of type SET of select data type, where the set of selections
in the select data type includes SET of data type nu in greek schema.
Parameters:
aggr - an aggregate of type SET of select data type;
the aggregate shall be that processed by
test_aggregate_set_of_set_create_aggregate_instance_unordered;
iter - an iterator over an aggregate specified by the first parameter.

```

ISO TC184/SC4/WG11 N137

```
*)
PROCEDURE
test_aggregate_set_of_set_create_aggregate_instance_as_current_member(aggr:SET
[1:?] OF rho;
    iter:iterator);
    LOCAL
        aggr_created : SET [0:3] OF nu;
        count : INTEGER;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that create_aggregate_instance_as_current_member reports an
IR_NEXS error
    when iterator is not provided.
    *)
    aggr_created := create_aggregate_instance_as_current_member(? ,
['GREEK.UPSILON'], IR_NEXS, ?);

    (* Ensures that create_aggregate_instance_as_current_member reports an
IR_NSET error
    when iterator has no current member set.
    *)
    atEnd(iter, NO_ERROR, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.UPSILON'], IR_NSET, ?);
    beginning(iter, NO_ERROR, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.UPSILON'], IR_NSET, ?);

    (* Ensures that create_aggregate_instance_as_current_member reports a
VT_NVLD error when
    list of defined types is either empty (but should not be such) or
contains wrong elements
    or is not submitted at all though is needed.
    *)
    bool := next(iter, NO_ERROR, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter, [],
VT_NVLD, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.XI'], VT_NVLD, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter, ?,
VT_NVLD, ?);

    (* Ensures that create_aggregate_instance_as_current_member works correctly
    when all parameters are correct.
    *)
    aggr_old := get_current_member(iter, NO_ERROR, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.UPSILON'], NO_ERROR, ?);
    bool := is_member(aggr, aggr_created, NO_ERROR, ?);
    assert(bool);
    bool := is_member(aggr, aggr_old, NO_ERROR, ?);
    assert(NOT bool);

    (* Ensures that an aggregate created is empty. *)
    count := get_member_count(aggr_created, NO_ERROR, ?);
    assert(count = 0);

    (* Ensures that an aggregate of a required type is created. *)
    add_unordered(aggr_created, xi(10), NO_ERROR, ?);
```

```
END_PROCEDURE;
```

```
( *
Testing the SDAI operations create aggregate instance unordered and
create aggregate instance as current member
for an aggregate of type SET of select data type, where the set of selections
in the select data type includes SET of data type nu in greek schema.
Parameters:
aggr - an aggregate of type SET of select data type rho in greek schema;
the aggregate shall be empty;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_set_of_set(aggr:SET [1:?] OF rho; iter:iterator);

    (* Testing of the aggregate operation create aggregate instance unordered.
*)
    test_aggregate_set_of_set_create_aggregate_instance_unordered(aggr);

    (* Testing of the aggregate operation create aggregate instance as current
member. *)
    test_aggregate_set_of_set_create_aggregate_instance_as_current_member(aggr,
iter);
END_PROCEDURE;
```

```
( *
Testing the cases when an SDAI operation on aggregate creation is applied to
an aggregate
whose type is SET and this is disallowed for that operation. In all such cases
an
error indicator AI_NVLD shall be reported.
Parameters:
aggr - an aggregate of type SET;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE test_aggregate_creation_operations_disallowed_for_set(aggr:AGGREGATE
OF GENERIC; iter:iterator);
    LOCAL
        aggr_created : SET [0:3] OF nu;
    END_LOCAL;

    (* Ensures that an AI_NVLD error is reported when LIST and ARRAY operation
create aggregate instance by index is applied to a SET.
*)
    aggr_created := create_aggregate_instance_by_index(aggr, 1,
['GREEK.UPSILON'], AI_NVLD, aggr);

    (* Ensures that an AI_NVLD error is reported when LIST operations are
applied to a SET. *)
    aggr_created := add_aggregate_instance_by_index(aggr, 1, ['GREEK.UPSILON'],
AI_NVLD, aggr);
    aggr_created := create_aggregate_instance_before_current_member(iter,
['GREEK.UPSILON'], AI_NVLD, aggr);
    aggr_created := create_aggregate_instance_after_current_member(iter,
['GREEK.UPSILON'], AI_NVLD, aggr);
END_PROCEDURE;
```

ISO TC184/SC4/WG11 N137

```
(*
Testing the SDAI operation create aggregate instance by index
for an aggregate of type ARRAY of select data type, where the set of
selections
in the select data type includes LIST of entity data type.
Parameters:
aggr - an aggregate of type ARRAY of select data type;
inst - an entity instance.
*)
PROCEDURE
test_aggregate_array_of_list_create_aggregate_instance_by_index(aggr:ARRAY
[1:3] OF omicron;
  inst:GENERIC_ENTITY);
  LOCAL
    aggr_created : LIST [1:?] OF omega;
    aggr_returned : LIST [1:?] OF omega;
    count : INTEGER;
    bool : BOOLEAN;
  END_LOCAL;

  (* Ensures that create_aggregate_instance_by_index reports an IX_NVLD error
when the index submitted
  is outside of the legal range.
  *)
  aggr_created := create_aggregate_instance_by_index(aggr, 0, ['GREEK.PHI'],
IX_NVLD, aggr);
  aggr_created := create_aggregate_instance_by_index(aggr, 4, ['GREEK.PHI'],
IX_NVLD, aggr);

  (* Ensures that create_aggregate_instance_by_index reports a VT_NVLD error
when
  list of defined types is either empty (but should not be such) or
contains wrong elements
  or is not submitted at all though is needed.
  *)
  aggr_created := create_aggregate_instance_by_index(aggr, 1, [], VT_NVLD,
?);
  aggr_created := create_aggregate_instance_by_index(aggr, 1, ['GREEK.XI'],
VT_NVLD, ?);
  aggr_created := create_aggregate_instance_by_index(aggr, 1, ?, VT_NVLD, ?);

  (* Ensures that create_aggregate_instance_by_index works correctly when all
parameters are correct. *)
  aggr_created := create_aggregate_instance_by_index(aggr, 1, ['GREEK.PHI'],
NO_ERROR, ?);

  (* Ensures that an aggregate created is empty. *)
  count := get_member_count(aggr_created, NO_ERROR, ?);
  assert(count = 0);

  (* Ensures that an aggregate of a required type is created and is assigned
to the correct place. *)
  add_by_index(aggr_created, 1, inst, NO_ERROR, ?);
  aggr_returned := get_by_index(aggr, 1, NO_ERROR, ?);
  bool := is_member(aggr_returned, inst, NO_ERROR, ?);
  assert(bool);

END_PROCEDURE;

(*
```



```

Testing the SDAI operation create aggregate instance as current member
for an aggregate of type ARRAY of select data type, where the set of
selections
in the select data type includes LIST of entity data type.
Parameters:
iter - an iterator over an aggregate of type ARRAY of select data type;
inst - an entity instance.
*)
PROCEDURE
test_aggregate_array_of_list_create_aggregate_instance_as_current_member(iter:
iterator;
    inst:GENERIC_ENTITY);
    LOCAL
        aggr_created : LIST [1:?] OF omega;
        aggr_returned : LIST [1:?] OF omega;
        count : INTEGER;
        bool : BOOLEAN;
    END_LOCAL;

    (* Ensures that create_aggregate_instance_as_current_member reports an
IR_NEXS error
    when iterator is not provided.
    *)
    aggr_created := create_aggregate_instance_as_current_member(?,
['GREEK.PHI'], IR_NEXS, ?);

    (* Ensures that create_aggregate_instance_as_current_member reports an
IR_NSET error
    when iterator has no current member set.
    *)
    atEnd(iter, NO_ERROR, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.PHI'], IR_NSET, ?);
    beginning(iter, NO_ERROR, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.PHI'], IR_NSET, ?);

    (* Ensures that create_aggregate_instance_as_current_member reports a
VT_NVLD error when
    list of defined types is either empty (but should not be such) or
contains wrong elements
    or is not submitted at all though is needed.
    *)
    bool := next(iter, NO_ERROR, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter, [],
VT_NVLD, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.XI'], VT_NVLD, ?);
    aggr_created := create_aggregate_instance_as_current_member(iter, ?,
VT_NVLD, ?);

    (* Ensures that create_aggregate_instance_as_current_member works correctly
    when all parameters are correct.
    *)
    aggr_created := create_aggregate_instance_as_current_member(iter,
['GREEK.PHI'], NO_ERROR, ?);

    (* Ensures that an aggregate created is empty. *)
    count := get_member_count(aggr_created, NO_ERROR, ?);
    assert(count = 0);

```

ISO TC184/SC4/WG11 N137

```
(* Ensures that an aggregate of a required type is created and is assigned
to the correct place. *)
add_by_index(aggr_created, 1, inst, NO_ERROR, ?);
aggr_returned := get_current_member(iter, NO_ERROR, ?);
bool := is_member(aggr_returned, inst, NO_ERROR, ?);
assert(bool);

END_PROCEDURE;
```

```
(*
Testing the SDAI operations create aggregate instance by index and
create aggregate instance as current member
for an aggregate of type ARRAY of select data type, where the set of
selections
in the select data type includes LIST of entity data type.
Parameters:
aggr - an aggregate of type ARRAY of select data type omicron (what is
tantamount to nu)
in greek schema;
iter - an iterator over an aggregate specified by the first parameter;
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_aggregate_array_of_list(aggr:ARRAY [1:3] OF omicron;
iter:iterator; model:sdai_model);
LOCAL
inst : GENERIC_ENTITY := create_entity_instance('GREEK.OMEGA', model,
NO_ERROR, ?);
END_LOCAL;

(* Testing of the aggregate operation create aggregate instance by index.
*)
test_aggregate_array_of_list_create_aggregate_instance_by_index(aggr,
inst);

(* Testing of the aggregate operation create aggregate instance as current
member. *)
test_aggregate_array_of_list_create_aggregate_instance_as_current_member(ag
gr, iter, inst);

END_PROCEDURE;
```

```
(*
Testing the cases when an SDAI operation on aggregate creation is applied to
an aggregate
whose type is ARRAY and this is disallowed for that operation. In all such
cases an
error indicator AI_NVLD shall be reported.
Parameters:
aggr - an aggregate of type ARRAY;
iter - an iterator over an aggregate specified by the first parameter.
*)
PROCEDURE
test_aggregate_creation_operations_disallowed_for_array(aggr:AGGREGATE OF
GENERIC; iter:iterator);
LOCAL
bool : BOOLEAN;
END_LOCAL;
```

```

    (* Ensures that an AI_NVLD error is reported when LIST operations are
    applied to an ARRAY. *)
    aggr_created := add_aggregate_instance_by_index(aggr, 1, ['GREEK.PHI'],
    AI_NVLD, aggr);
    aggr_created := create_aggregate_instance_before_current_member(iter,
    ['GREEK.PHI'], AI_NVLD, aggr);
    aggr_created := create_aggregate_instance_after_current_member(iter,
    ['GREEK.PHI'], AI_NVLD, aggr);

    (* Ensures that an AI_NVLD error is reported when SET and BAG operation
    create aggregate instance unordered is applied to an ARRAY.
    *)
    aggr_created := create_aggregate_instance_unordered(aggr, ['GREEK.PHI'],
    AI_NVLD, aggr);

END_PROCEDURE;

(*
Testing the SDAI operations for non-nested aggregates.
Parameter:
model - an SDAI-model in read-write mode, based on the greek schema.
*)
PROCEDURE test_aggregate_nested(model:sdai_model);
    LOCAL
        inst : GENERIC_ENTITY := create_entity_instance('GREEK.EPSILON', model,
    NO_ERROR, ?);
        aggr : AGGREGATE OF GENERIC;
        iter : iterator;
        aggr_created : LIST [1:3] OF REAL;
    END_LOCAL;

    (* An aggregate of type LIST of select type nu in greek schema and its
    iterator are created. *)
    aggr := create_aggregate_instance(inst, 'GREEK.EPSILON.E2', NO_ERROR, ?);
    iter := create_iterator(aggr, NO_ERROR, ?);

    (* Ensures that an AI_NVLD error is reported when SET and BAG operation
    create aggregate instance unordered is applied to a LIST.
    *)
    aggr_created := create_aggregate_instance_unordered(aggr, ['GREEK.CHI'],
    AI_NVLD, aggr);

    (* Testing the SDAI operations on aggregate creation for an aggregate of
    type LIST of LIST. *)
    test_aggregate_list_of_list(aggr, iter);

    (* An aggregate of type SET of select type rho in greek schema and its
    iterator are created. *)
    aggr := create_aggregate_instance(inst, 'GREEK.EPSILON.E6', NO_ERROR, ?);
    iter := create_iterator(aggr, NO_ERROR, ?);

    (* Checks cases when aggregate creation operations are illegal for the SET.
    *)
    test_aggregate_creation_operations_disallowed_for_set(aggr, iter);

    (* Testing the SDAI operations on aggregate creation for an aggregate of
    type SET of SET. *)
    test_aggregate_set_of_set(aggr, iter);

```

ISO TC184/SC4/WG11 N137

```
(* An aggregate of type ARRAY of select type nu in greek schema and its
iterator are created. *)
aggr := create_aggregate_instance(inst, 'GREEK.EPSILON.E4', NO_ERROR, ?);
iter := create_iterator(aggr, NO_ERROR, ?);

(* Checks cases when aggregate creation operations are illegal for the
ARRAY. *)
test_aggregate_creation_operations_disallowed_for_array(aggr, iter);

(* Testing the SDAI operations on aggregate creation for an aggregate of
type ARRAY of LIST. *)
test_aggregate_array_of_list(aggr, iter, model);

END_PROCEDURE;

END_SCHEMA;
```


Annex A
(normative)
SDAI test operation schema

(*

The test sequences are defined in a kind of pseudo EXPRESS code. For this the SDAI operations are translated into EXPRESS procedures, assuming that the session entities are together with the application schema available.

The express given here uses the GENERIC_ENTITY from the proposed EXPRESS ammendment. Furthermore we assume that for every defined type which is not a select type, there is a implicit definition of a function with the same name which returned this type and which accept the base type as parameter. In the case of a defined type which is a select type then all the possible underlying types of this select are accepted.

Not mapped operations:

- 10.4.1 Record error
- 10.4.2 Start event recording
- 10.4.3 Stop event recording
- 10.8.1 Add to scope
- 10.8.2 Is scope owner
- 10.8.3 Get scope
- 10.8.4 Remove from scope
- 10.8.5 Add to export list
- 10.8.6 Remove from export list
- 10.8.7 Scoped delete
- 10.8.8 Scoped copy
- 10.8.9 Validate scope referenece restrictions
- 10.9.3 Is SDAI subtype of
- 10.10.7 Is SDAI kind of
- 10.11.18 Validate real precision

*)

SCHEMA SDAI_test_operation_schema;

USE FROM SDAI_session_schema;

CONSTANT

- NO_ERROR : INTEGER := 0;
- SS_OPN : INTEGER := 10;
- SS_NAVL : INTEGER := 20;
- SS_NOPN : INTEGER := 30;
- RP_NEXS : INTEGER := 40;
- RP_NAVL : INTEGER := 50;
- RP_OPN : INTEGER := 60;
- RP_NOPN : INTEGER := 70;
- RP_DUP : INTEGER := 75;
- TR_EAB : INTEGER := 80;
- TR_EXS : INTEGER := 90;
- TR_NAVL : INTEGER := 100;
- TR_RW : INTEGER := 110;

```

TR_NRW : INTEGER := 120;
TR_NEXS : INTEGER := 130;
MO_NDEQ : INTEGER := 140;
MO_NEXS : INTEGER := 150;
MO_NVLD : INTEGER := 160;
MO_DUP : INTEGER := 170;
MX_NRW : INTEGER := 180;
MX_NDEF : INTEGER := 190;
MX_RW : INTEGER := 200;
MX_RO : INTEGER := 210;
SD_NDEF : INTEGER := 220;
ED_NDEF : INTEGER := 230;
ED_NDEQ : INTEGER := 240;
ED_NVLD : INTEGER := 250;
RU_NDEF : INTEGER := 260;
EX_NSUP : INTEGER := 270;
AT_NVLD : INTEGER := 280;
AT_NDEF : INTEGER := 290;
SI_DUP : INTEGER := 300;
SI_NEXS : INTEGER := 310;
EI_NEXS : INTEGER := 320;
EI_NAVL : INTEGER := 330;
EI_NVLD : INTEGER := 340;
EI_NEXP : INTEGER := 350;
SC_NEX : INTEGER := 360;
SC_EXS : INTEGER := 370;
AI_NEXS : INTEGER := 380;
AI_NVLD : INTEGER := 390;
AI_NSET : INTEGER := 400;
VA_NVLD : INTEGER := 410;
VA_NEXS : INTEGER := 420;
VA_NSET : INTEGER := 430;
VT_NVLD : INTEGER := 440;
IR_NEXS : INTEGER := 450;
IR_NSET : INTEGER := 460;
IX_NVLD : INTEGER := 470;
ER_NSET : INTEGER := 480;
OP_NVLD : INTEGER := 490;
FN_NAVL : INTEGER := 500;
SY_ERR : INTEGER := 1000;
END_CONSTANT;

-- 10.3.1 Open session
FUNCTION open_session(error:INTEGER; base:GENERIC) : sdai_session;
{} END_FUNCTION;

-- 10.4.4 Close session
PROCEDURE close_session(session:sdai_session; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.4.5 Open repository
PROCEDURE open_repository(session:sdai_session; repository:sdai_repository;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.4.6 Start transaction read-write access
FUNCTION start_transaction_read_write_access(session:sdai_session;
error:INTEGER; base:GENERIC) : sdai_transaction;
{} END_FUNCTION;

-- 10.4.7 Start transaction read-only access

```

ISO TC184/SC4/WG11 N137

```
FUNCTION start_transaction_read_only_access(session:sdai_session;
error:INTEGER; base:GENERIC) : sdai_transaction;
{} END_FUNCTION;

-- 10.4.8 Commit
PROCEDURE commit(transaction:sdai_transaction; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.4.9 Abort
PROCEDURE abort(transaction:sdai_transaction; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.4.10 End transaction access and commit
PROCEDURE end_transaction_access_and_commit(transaction:sdai_transaction;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.4.11 End transaction access and abort
PROCEDURE end_transaction_access_and_abort(transaction:sdai_transaction;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.4.12 Create non-persistent list
FUNCTION create_non_persistent_list(error:INTEGER; base:GENERIC) LIST OF
GENERIC_ENTITY;
{} END_PROCEDURE;

-- 10.4.13 delete_non_persistent_list
PROCEDURE delete_non_persistent_list(the_list:LIST OF GENERIC_ENTITY;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.4.14 SDAI query where source is a non-nested, persistent or non-
persistent aggregate
FUNCTION query_aggregate(source:LIST OF GENERIC_ENTITY;
query_expression:STRING; the_entity:GENERIC_ENTITY; result:LIST OF
GENERIC_ENTITY; error:INTEGER; base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.4.14 SDAI query where source is an sdai_model
FUNCTION query_model(source:sdai_model; query_expression:STRING;
the_entity:GENERIC_ENTITY; result:LIST OF GENERIC_ENTITY; error:INTEGER;
base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.4.14 SDAI query where source is a schema_instance
FUNCTION query_schema_instance(source:sdai_model; query_expression:STRING;
the_entity:GENERIC_ENTITY; result:LIST OF GENERIC_ENTITY; error:INTEGER;
base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.4.14 SDAI query where source is an sdai_repository
FUNCTION query_repository(source:sdai_repository; query_expression:STRING;
the_entity:GENERIC_ENTITY; result:LIST OF GENERIC_ENTITY; error:INTEGER;
base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.5.1 Create SDAI model
FUNCTION create_sdai_model(repository:sdai_repository; name:String;
underlying_schema:schema_definition; error:INTEGER; base:GENERIC):sdai_model;
{} END_FUNCTION;
```



```

-- 10.5.2 Create schema instance
FUNCTION create_schema_instance(repository:sdai_repository; name:String;
native_schema:schema_definition; error:INTEGER; base:GENERIC):schema_instance;
{} END_FUNCTION;

-- 10.5.3 Close repository
PROCEDURE close_repository(repository:sdai_repository; error:INTEGER;
base:GENERIC);
{} END_PROCEDURE;

-- 10.6.1 Delete schema instance
PROCEDURE delete_schema_instance(the_schema_instance:schema_instance;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.6.2 Rename schema instance
PROCEDURE rename_schema_instance(the_schema_instance:schema_instance;
name:String; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.6.3 Add SDAI-model
PROCEDURE add_sdai_model(the_schema_instance:schema_instance;
model:sdai_model; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.6.4 Remove SDAI-model
PROCEDURE remove_sdai_model(the_schema_instance:schema_instance;
model:sdai_model; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.6.5 Validate global rule
FUNCTION validate_global_rule(the_schema_instance:schema:instance;
rule:global_rule; VAR nonConf:LIST OF where_rule; error:INTEGER;
base:GENERIC):LOGICAL;
{} END_FUNCTION;

-- 10.6.6 Validate uniqueness rule
FUNCTION validate_uniqueness_rule(the_schema_instance:schema:instance;
rule:uniqueness_rule; VAR nonConf:LIST OF GENERIC_ENTITY; error:INTEGER;
base:GENERIC):LOGICAL;
{} END_FUNCTION;

-- 10.6.7 Validate instance reference domain
FUNCTION
validate_instance_reference_domain(the_schema_instance:schema:instance;
object:GENERIC_ENTITY; VAR nonConf:LIST OF where_rule; error:INTEGER;
base:GENERIC):LOGICAL;
{} END_FUNCTION;

-- 10.6.5 Validate schema instance
FUNCTION validate_schema_instance(the_schema_instance:schema:instance;
error:INTEGER; base:GENERIC):LOGICAL;
{} END_FUNCTION;

-- 10.6.9 Is validation current
FUNCTION is_validation_current(the_schema_instance:schema_instance;
error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.7.1 Delete SDAI-model

```

ISO TC184/SC4/WG11 N137

```
PROCEDURE delete_sdai_model(model:sdai_model; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.7.2 Rename SDAI-model
PROCEDURE rename_sdai_model(model:sdai_model; name:String; error:INTEGER;
base:GENERIC);
{} END_PROCEDURE;

-- 10.7.3 Start read-only access
PROCEDURE start_read_only_access(model:sdai_model; error:INTEGER;
base:GENERIC);
{} END_PROCEDURE;

-- 10.7.4 Promote SDAI-model to read-write
PROCEDURE promote_sdai_model_to_read_write(model:sdai_model; error:INTEGER;
base:GENERIC);
{} END_PROCEDURE;

-- 10.7.5 End read-only access
PROCEDURE end_read_only_access(model:sdai_model; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.7.6 Start read_write access
PROCEDURE start_read_write_access(model:sdai_model; error:INTEGER;
base:GENERIC);
{} END_PROCEDURE;

-- 10.7.7 End read-write access
PROCEDURE end_read_write_access(model:sdai_model; error:INTEGER;
base:GENERIC);
{} END_PROCEDURE;

-- 10.7.8 Get entity definition
FUNCTION get_entity_definition(model:sdai_model; entityname:STRING;
error:INTEGER; base:GENERIC):entity_definition;
{} END_FUNCTION;

-- 10.7.9 Create entity instance
FUNCTION create_entity_instance(the_type:entity_definition; model:sdai_model;
error:INTEGER; base:GENERIC):GENERIC_ENTITY;
{} END_FUNCTION;

-- 10.7.10 Undo changes
PROCEDURE undo_changes(model:sdai_model; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.7.11 Save changes
FUNCTION save_changes(model:sdai_model; error:INTEGER; base:GENERIC);
{} END_FUNCTION;

-- 10.9.1 Get complex entity definition
FUNCTION get_complex_entity_definition(types:LIST OF entity_definition;
error:INTEGER; base:GENERIC):entity_definition;
{} END_FUNCTION;

-- 10.9.2 Is subtype of
FUNCTION is_subtype_of(the_type:entity_definition;
comp_Type:entity_definition; error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.9.4 Is domain equivalent with
```

```

FUNCTION is_domain_equivalent_with(the_type:entity_definition;
comp_type:entity_definition; error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.10.1 Get attribute
FUNCTION get_attribute(object:GENERIC_ENTITY; the_attribute:attribute;
error:INTEGER; base:GENERIC):GENERIC;
{} END_FUNCTION;

-- 10.10.1.a Get inverse attribute (new)
FUNCTION get_attribute(object:GENERIC_ENTITY; the_attribute:attribute;
domain:AGGREGATE OF schema_instance; error:INTEGER; base:GENERIC):GENERIC;
{} END_FUNCTION;

-- 10.10.2 Test attribute
FUNCTION test_attribute(object:GENERIC_ENTITY; the_attribute:attribute;
error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.10.3 Find entity instance SDAI-model
FUNCTION find_entity_instance_sdai_model(object:GENERIC_ENTITY; error:INTEGER;
base:GENERIC):sdai_model;
{} END_FUNCTION;

-- 10:10:4 Get instance type
FUNCTION get_instance_type(object:GENERIC_ENTITY; error:INTEGER;
base:GENERIC):entity_definition;
{} END_FUNCTION;

-- 10:10:5 Is instance of
FUNCTION is_instance_of(object:GENERIC_ENTITY; check_type:entity_definition;
error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.10.6 Is kind of
FUNCTION is_kind_of(object:GENERIC_ENTITY; check_type:entity_definition;
error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.10.8 Find entity instance users
PROCEDURE find_entity_instance_users(object:GENERIC_ENTITY; domain:AGGREGATE OF
schema_instance; VAR result:LIST OF GENERIC_ENTITY; error:INTEGER;
base:GENERIC);
{} END_PROCEDURE;

-- 10.10.9 Find entity instance usedin
PROCEDURE find_entity_instance_usedin(object:GENERIC_ENTITY; role:attribute;
domain:AGGREGATE OF schema_instance; result:LIST OF GENERIC_ENTITY;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.10.10 Get attribute value bound
FUNCTION get_attribute_value_bound(object:GENERIC_ENTITY;
the_attribute:attribute; error:INTEGER; base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.10.11 Find instance_roles
PROCEDURE find_instance_roles(object:GENERIC_ENTITY; domain:LIST OF
schema_instance; VAR result:LIST of attributes; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

```

ISO TC184/SC4/WG11 N137

```
-- 10.10.12 Find instance data types
PROCEDURE find_instance_data_types(object:entity_instance; VAR result:LIST OF
named_type; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.11.1 Copy application instance
FUNCTION copy_application_instance(object:GENERIC_ENTITY;
targetModel:sdai_model; error:INTEGER; base:GENERIC):GENERIC_ENTITY;
{} END_FUNCTION;

-- 10.11.2 Delete application instance
PROCEDURE delete_application_instance(object:GENERIC_ENTITY; error:INTEGER;
base:GENERIC);
{} END_PROCEDURE;

-- 10.11.3 Put attribute
PROCEDURE put_attribute(object:GENERIC_ENTITY; attribute:explicit_attribute;
value:GENERIC; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.11.4 Unset attribute value
PROCEDURE unset_attribute_value(object:GENERIC_ENTITY;
attribute:explicit_attribute; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.11.5 Create aggregate instance
FUNCTION create_aggregate_instance(object:GENERIC_ENTITY;
attribute:explicit_attribute; error:INTEGER; base:GENERIC):AGGREGATE OF
GENERIC_ENTITY;
{} END_FUNCTION;

-- 10.11.6 Get persistent lable
FUNCTION get_persistent_lable(object:GENERIC_ENTITY; error:INTEGER;
base:GENERIC):STRING;
{} END_FUNCTION;

-- 10.11.7 Get session identifier
FUNCTION get_session_identifier(label:STRING; repository:sdai_repository;
error:INTEGER; base:GENERIC):GENERIC_ENTITY;
{} END_FUNCTION;

-- 10.11.8 Get description
FUNCTION get_description(object:GENERIC_ENTITY; error:INTEGER;
base:GENERIC):STRING;
{} END_FUNCTION;

-- 10.11.9 Validate where rule
FUNCTION validate_where_rule(object:GENERIC_ENTITY; the_rule:where_rule;
error:INTEGER; base:GENERIC):LOGICAL;
{} END_FUNCTION;

-- 10.11.10 Validate required explicit attribute assigned
FUNCTION validate_required_explicit_attribute_assigned(object:GENERIC_ENTITY;
VAR nonConf:LIST OF GENERIC; error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.11.11 Validate inverse attributes
FUNCTION validate_inverse_attributes(object:GENERIC_ENTITY; VAR nonConf:LIST
OF GENERIC; error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;
```

```

-- 10.11.12 Validate explicit attribute references
FUNCTION validate_explicit_attribute_references(object:GENERIC_ENTITY; VAR
nonConf:LIST OF GENERIC; error:INTEGER; base:GENERIC):LOGICAL;
{} END_FUNCTION;

-- 10.11.13 Validate aggregate size
FUNCTION validate_aggregate_size(object:GENERIC_ENTITY; VAR nonConf:LIST OF
GENERIC; error:INTEGER; base:GENERIC):LOGICAL;
{} END_FUNCTION;

-- 10.11.14 Validate aggregates uniqueness
FUNCTION validate_aggregate_uniqueness(object:GENERIC_ENTITY; VAR nonConf:LIST
OF GENERIC; error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.11.15 Validate array not optional
FUNCTION validate_array_not_optional(object:GENERIC_ENTITY; VAR nonConf:LIST
OF GENERIC; error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.11.16 Validate string width
FUNCTION validate_string_width(object:GENERIC_ENTITY; VAR nonConf:LIST OF
GENERIC; error:INTEGER; base:GENERIC):LOGICAL;
{} END_FUNCTION;

-- 10.11.17 Validate binary width
FUNCTION validate_binary_width(object:GENERIC_ENTITY; VAR nonConf:LIST OF
GENERIC; error:INTEGER; base:GENERIC):LOGICAL;
{} END_FUNCTION;

-- 10.12.1 Get member count
FUNCTION get_member_count(the_aggregate:AGGREGATE OF GENERIC; error:INTEGER;
base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.12.2 Is member
FUNCTION is_member(the_aggregate:AGGREGATE OF GENERIC; value: GENERIC;
error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.12.3 Create iterator
FUNCTION create_iterator(aggregate:AGGREGATE OF GENERIC; error:INTEGER;
base:GENERIC):iterator;
{} END_FUNCTION;

--10.12.4 Delete iterator
PROCEDURE delete_iterator(iter:iterator; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

--10.12.5 Beginning
PROCEDURE beginning(iter:iterator; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.12.6 Next
FUNCTION next(iter:iterator; error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.12.7 Get current member
FUNCTION get_current_member(iter:iterator; error:INTEGER;
base:GENERIC):GENERIC;
{} END_FUNCTION;

```

ISO TC184/SC4/WG11 N137

```
-- 10.12.8 Get value bound by iterator
FUNCTION get_value_bound_by_iterator(iter:iterator; error:INTEGER;
base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.12.9 Get lower bound
FUNCTION get_lower_bound(iter:iterator; error:INTEGER; base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.12.10 Get upper bound
FUNCTION get_upper_bound(iter:iterator; error:INTEGER; base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.13.1 Create aggregate instance as current member
FUNCTION create_aggregate_instance_as_current_member(iter:iterator;
error:INTEGER; base:GENERIC):AGGREGATE OF GENERIC;
{} END_FUNCTION;

-- 10.13.2 Put current member
PROCEDURE put_current_member(iter:iterator, value:GENERIC; error:INTEGER;
base:GENERIC);
{} END_PROCEDURE;

-- 10.13.3 Remove current member
FUNCTION remove_current_member(iter:iterator; error:INTEGER;
base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.14.1 Add unordered
PROCEDURE add_unordered(aggregate:AGGREGATE OF GENERIC; value:GENERIC;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.14.2 Create aggregate instance unordered
FUNCTION create_aggregate_instance_unordered(agg:AGGREGATE OF GENERIC;
error:INTEGER; base:GENERIC):AGGREGATE OF GENERIC; -- specifications for
select needed
{} END_FUNCTION;

-- 10.14.3 Remove unordered
PROCEDURE remove_unordered(agg:AGGREGATE OF GENERIC; value:GENERIC;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.15.1 Get by index
FUNCTION get_by_index(agg:AGGREGATE OF GENERIC; index:INTEGER; error:INTEGER;
base:GENERIC):GENERIC;
{} END_FUNCTION;

-- 10.15.2 End
PROCEDURE end(iter:iterator; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.15.3 Previous
FUNCTION previous(iter:iterator; error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.15.4 Get value bound by index
FUNCTION get_value_bound_by_index(agg:AGGREGATE OF GENERIC; index:INTEGER;
error:INTEGER; base:GENERIC):INTEGER;
```

```

{} END_FUNCTION;

-- 10.16.1 Put by index
PROCEDURE put_by_index(agg:AGGREGATE OF GENERIC; index:INTEGER; value:GENERIC;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.16.2 Create aggregate instance by index
FUNCTION create_aggregate_instance_by_index(agg:AGGREGATE OF GENERIC;
inde:INTEGER; error:INTEGER; base:GENERIC):AGGREGATE OF GENERIC; -- select
specification needed
{} END_FUNCTION;

-- 10.17.1 Test by index
FUNCTION test_by_index(agg:ARRAY OF GENERIC; index:INTEGER; error:INTEGER;
base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.17.2 Test current member
FUNCTION test_current_member(iter:iterator; error:INTEGER; base:GENERIC;
error:INTEGER; base:GENERIC):BOOLEAN;
{} END_FUNCTION;

-- 10.17.3 Get lower index
FUNCTION get_lower_index(agg:AGGREGATE OF GENERIC; error:INTEGER;
base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.17.4 Get upper index
FUNCTION get_upper_index(agg:ARRAY OF GENERIC; error:INTEGER;
base:GENERIC):INTEGER;
{} END_FUNCTION;

-- 10.18.1 Unset value by index
PROCEDURE unset_value_by_index(agg: ARRAY OF GENERIC; index:INTEGER;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.18.2 Unset value current member
PROCEDURE unset_value_current_member(iter:iterator; error:INTEGER;
base:GENERIC);
{} END_PROCEDURE;

-- 10.18.3 Reindex array
PROCEDURE reindex_array(agg: ARAY OF GENERIC; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.18.4 Reset array index
PROCEDURE reset_array_index(agg:ARRAY OF GENERIC; lower:INTEGER;
upper:INTEGER; error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.19.1 Add before current member
PROCEDURE add_before_current_member(iter:iterator; value:GENERIC;
error:INTEGER; base:GENERIC);
{} END_PROCEDURE;

-- 10.19.2 Add after current member
PROCEDURE add_after_current_member(iter:iterator; value:GENERIC;
error:INTEGER; base:GENERIC);

```

ISO TC184/SC4/WG11 N137

```
{ } END_PROCEDURE;

-- 10.19.3 Add by index
PROCEDURE add_by_index(agg: LIST OF GENERIC; index:INTEGER; value:GENERIC;
error:INTEGER; base:GENERIC);
{ } END_PROCEDURE;

-- 10.19.4 Create aggregate instance before current member
FUNCTION create_aggregate_instance_before_current_member(iter:iterator;
error:INTEGER; base:GENERIC):AGGREGATE OF GENERIC; -- select specification
needed
{ } END_FUNCTION;

-- 10.19.5 Create aggregate instance after current member
FUNCTION create_aggregate_instance_after_current_member(iter:iterator;
error:INTEGER; base:GENERIC):AGGREGATE OF GENERIC; -- select specification
needed
{ } END_FUNCTION;

-- 10.19.6 Add aggregate instance by index
FUNCTION add_aggregate_instance_by_index(agg:LIST OF GENERIC; index:INTEGER;
error:INTEGER; base:GENERIC):AGGREGATE OF GENERIC; -- select specification
needed
{ } END_FUNCTION;

-- 19.19.7 Remove by index
PROCEDURE remove_by_index(agg:LIST OF GENERIC; index:INTEGER; error:INTEGER;
base:GENERIC);
{ } END_PROCEDURE;

-- auxiliary procedure. Checks that both instances are value equal
PROCEDURE check_instance(instance1: GENERIC; instance2: GENERIC);
{ } END_PROCEDURE;

-- auxiliary procedure. Prints out the string
PROCEDURE print(str : STRING);
{ } END_PROCEDURE;

END_SCHEMA;
```


Annex B
(normative)
The "Greek" Test Application schema

```

SCHEMA greek;

TYPE chi = LIST [1:3] OF REAL;
END_TYPE;

TYPE omicron = nu;
END_TYPE;

TYPE phi = LIST [1:?] OF omega;
END_TYPE;

TYPE pie = xi;
END_TYPE;

TYPE psi = LIST [1:?] OF phi;
END_TYPE;

TYPE upsilon = SET [0:3] OF nu;
END_TYPE;

TYPE xi = INTEGER;
END_TYPE;

TYPE tau = ENUMERATION OF
    (stigma,
     digamma,
     koppa,
     sampi);
END_TYPE;

TYPE alpha_or_kappa = SELECT
    (alpha,
     kappa);
END_TYPE;

TYPE nu = SELECT
    (phi,
     psi,
     chi,
     sigma,
     omega,
     tau,
     xi,
     pie);
END_TYPE;

TYPE rho = SELECT
    (nu,
     omicron,
     upsilon);
END_TYPE;

```

ISO TC184/SC4/WG11 N137

```
ENTITY alpha;
  a1 : kappa;
  a2 : zeta;
END_ENTITY;

ENTITY beta
  SUBTYPE OF (alpha);
  xxx : INTEGER;
  yyy : REAL;
END_ENTITY;

ENTITY delta
  SUBTYPE OF (beta,gamma);
  DERIVE
    a1 : mu := xxx;
END_ENTITY;

ENTITY epsilon;
  e1 : nu;
  e2 : LIST [1:?] OF nu;
  e3 : omicron;
  e4 : ARRAY [1:3] OF omicron;
  e5 : rho;
  e6 : SET [1:?] OF rho;
END_ENTITY;

ENTITY eta
  SUBTYPE OF (zeta);
  INVERSE
    SELF\zeta.z2 : SET[1:?] OF alpha FOR a2;
END_ENTITY;

ENTITY gamma
  SUBTYPE OF (alpha,kappa);
  SELF\alpha.a1 : lamda;
  xxx : INTEGER;
  yyy : STRING;
END_ENTITY;

ENTITY iota;
END_ENTITY;

ENTITY kappa;
  k1 : INTEGER;
  DERIVE
    k2 : INTEGER := k1;
  INVERSE
    k3 : alpha FOR a1;
END_ENTITY;

ENTITY lamda
  SUBTYPE OF (kappa);
  SELF\kappa.k1 : xi;
  DERIVE
    SELF\kappa.k2 : xi := k1;
END_ENTITY;

ENTITY mu
  SUBTYPE OF (lamda);
  SELF\lamda.k1 : pie;
  DERIVE
```

```

    SELF\lamda.k2 : pie := k1;
END_ENTITY;

```

```

ENTITY omega;
    o0 : iota;
    o1 : NUMBER;
    o2 : REAL;
    o3 : INTEGER;
    o4 : LOGICAL;
    o5 : BOOLEAN;
    o6 : STRING;
    o7 : BINARY;
    o8 : tau;
    o9 : xi;
END_ENTITY;

```

```

ENTITY sigma;
    s1 : LIST [0:?] OF NUMBER;
    s2 : BAG [0:?] OF REAL;
    s3 : ARRAY [0:4] OF INTEGER;
    s4 : SET [2:4] OF LOGICAL;
    s5 : LIST [2:4] OF BOOLEAN;
    s6 : BAG [2:4] OF STRING;
    s7 : ARRAY [2:4] OF BINARY;
    s8 : LIST [1:?] OF tau;
    s9 : SET [1:?] OF xi;
END_ENTITY;

```

```

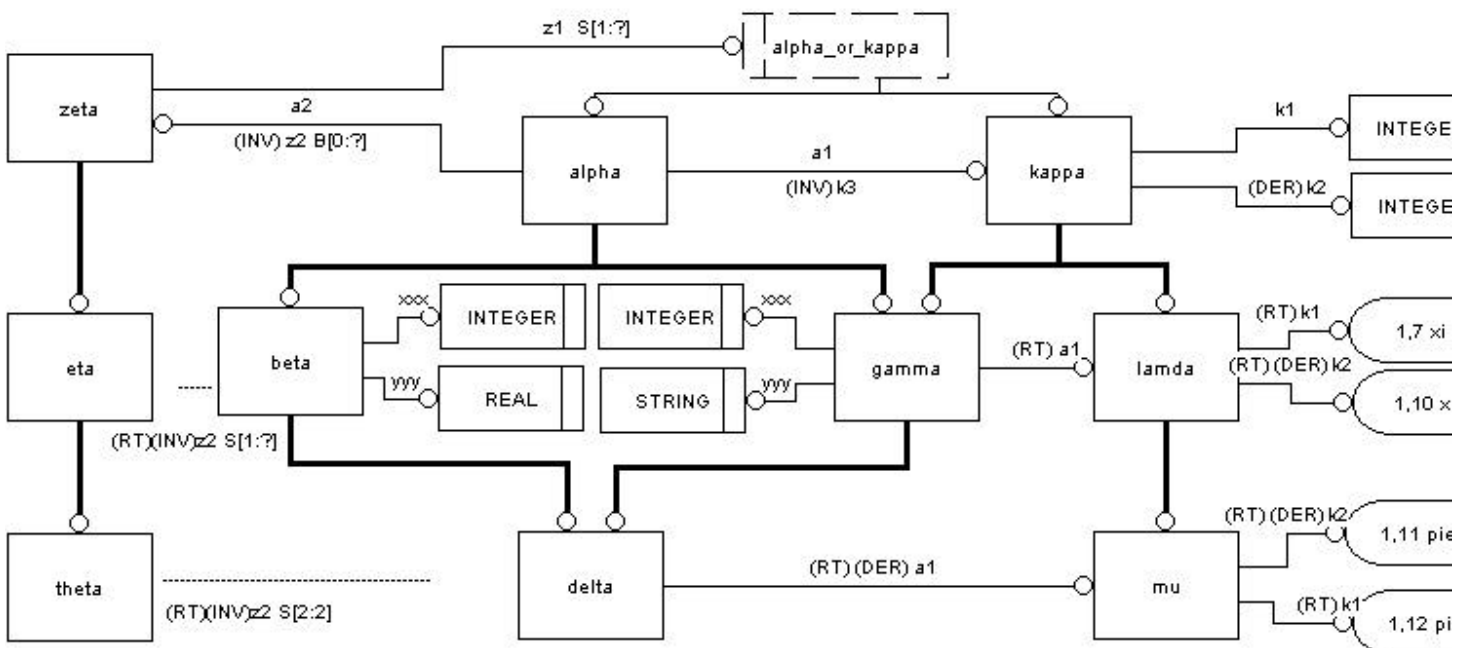
ENTITY theta
    SUBTYPE OF (eta);
    INVERSE
        SELF\eta.z2 : SET[2:2] OF alpha FOR a2;
END_ENTITY;

```

```

ENTITY zeta;
    z1 : alpha_or_kappa;
    INVERSE
        z2 : BAG[0:?] OF alpha FOR a2;
END_ENTITY;

```



Annex C (informative) Example

This example shows some test-cases mapped into into a particular SDAI implementation, based on the Java language binding to the SDAI, ISO 10303-27.

```
import java.io.*;
import SDAI.lang.*;
import SDAI.dictionary.*;
import SDAI.SGreek.*;

public class TestCases {

    /*
    Testing the SDAI operation delete SDAI-model.
    Parameters:
    modell - an SDAI-model in read-write mode, based on the greek schema;
    model2 - an SDAI-model in read-write mode, based on the greek schema and
    associated with a schema_instance whose native schema is greek schema; this
    model
    shall be different from modell and may even belong to a different repository.
    */
    public void test_delete_SDAI_model(SdaiModel modell, SdaiModel model2)
    throws SdaiException {

        /* Declarations and initializations. */
        SdaiRepository repo = model2.getRepository();
        ASchemaInstance aggr_schemas = model2.getAssociatedWith();
        SchemaInstance schema_instance = aggr_schemas.getByIndex(1);
        SdaiModel model_dict;
        EOmega inst1 = (EOmega)modell.createEntityInstance(EOmega.class);
        EIota inst2 = (EIota)model2.createEntityInstance(EIota.class);

        /* An instance in one SDAI-model references an instance in
        another SDAI-model which will be deleted.
        */
        inst1.setO0(null, inst2);

        /* Ensures that delete_sdai_model works correctly. */
        try {
            model2.deleteSdaiModel();
        } catch (SdaiException ex) {
            System.out.println("Test failed: no SdaiException can be thrown.");
        }

        /* Ensures that deleted SDAI-model does not belong to models set
        of the sdai_repository_contents.
        */
        if (repo.getModels().isMember(model2)) {
            System.out.println("Test failed: deleted SDAI-model still belongs to
            models set of the repository contents.");
        }

        /* Ensures that deleted SDAI-model does not belong to active_models set
        of the SDAI session. */
        if (repo.getSession().getActiveModels().isMember(model2)) {
```

ISO TC184/SC4/WG11 N137

```
        System.out.println("Test failed: deleted SDAI-model still belongs to
active_models set of the SDAI session.");
    }

    /* Ensures that deleted SDAI-model does not belong to associated_models
set of the schema instance
to which this SDAI-model earlier was added.
*/
    if (schema_instance.getAssociatedModels().isMember(model2)) {
        System.out.println("Test failed: deleted SDAI-model still belongs to
associated_models set of the schema instance.");
    }

    /* Ensures that references to instances of the deleted model from other
models become unset. */
    try {
        inst2 = inst1.getO0(null);
        System.out.println("Test failed: references to instances of the
deleted model from other models shall be unset.");
    } catch (SdaiException ex) {
        if (ex.getErrorId() != SdaiException.VA_NSET) {
            System.out.println("Test failed: VA_NSET error shall be reported.");
        }
    }

    /* Ensures that delete_sdai_model reports a VT_NVLD error when a data
dictionary SDAI-model for
deletion is submitted.
*/
    model_dict = macro_get_data_dictionary_model();
    try {
        model_dict.deleteSdaiModel();
        System.out.println("Test failed: data dictionary SDAI-model is
forbiden for deleting.");
    } catch (SdaiException ex) {
        if (ex.getErrorId() != SdaiException.VT_NVLD) {
            System.out.println("Test failed: VT_NVLD error shall be reported.");
        }
    }
}

/*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type REAL.
Early binding versions of the methods for these operations are checked.
*/
public void test_attribute_real_early_binding(SdaiModel model) throws
SdaiException {

    /* Declarations and initializations. */
    EOmega inst = (EOmega)model.createEntityInstance(EOmega.class);
    double real_num;

    /* Ensures that the attribute is initially unset after creation of the
instance. */
    if (inst.testO2(null)) {
        System.out.println("Test failed: attribute shall be initially
unset.");
    }
}
```

```

    /* Ensures that get_attribute reports a VA_NSET error when the attribute
has no value. */
    try {
        real_numb = inst.getO2(null);
        System.out.println("Test failed: VA_NSET error shall be reported when
the attribute has no value.");
    } catch (SdaiException ex) {
        if (ex.getErrorId() != SdaiException.VA_NSET) {
            System.out.println("Test failed: VA_NSET error shall be reported
when the attribute has no value.");
        }
    }

    /* Ensures that put_attribute reports a VA_NSET error when an unset value
for the attribute is submitted.
*/
    try {
        inst.setO2(null, Double.NaN);
        System.out.println("Test failed: VA_NSET error shall be reported when
value is not submitted.");
    } catch (SdaiException ex) {
        if (ex.getErrorId() != SdaiException.VA_NSET) {
            System.out.println("Test failed: VA_NSET error shall be reported
when value is not submitted.");
        }
    }

    /* Ensures that put_attribute works correctly when all parameters are
correct. */
    try {
        inst.setO2(null, 5.6);
    } catch (SdaiException ex) {
        System.out.println("Test failed: no SdaiException can be thrown.");
    }

    /* Ensures that test_attribute reports TRUE after successfully invoking
put_attribute. */
    if (!inst.testO2(null)) {
        System.out.println("Test failed: attribute cannot be unset after
assigning a value to it.");
    }

    /* Ensures that get_attribute retrieves the right value. */
    if (inst.getO2(null) != 5.6) {
        System.out.println("Test failed: attribute must have the value which
was assigned.");
    }

    /* Ensures that the attribute value can be unset. */
    try {
        inst.unsetO2(null);
    } catch (SdaiException ex) {
        System.out.println("Test failed: no SdaiException can be thrown.");
    }

    /* Ensures that the attribute is really unset after unset_attribute_value
operation. */
    if (inst.testO2(null)) {
        System.out.println("Test failed: attribute shall be unset after unset
attribute value operation.");
    }

```

ISO TC184/SC4/WG11 N137

```
    }
}

/*
Testing the basic functionality of the SDAI operations
test attribute, get attribute, put attribute and unset attribute value
for an attribute of type REAL.
Late binding versions of the methods for these operations are checked.
*/
public void test_attribute_real_late_binding(SdaiModel model) throws
SdaiException {

    /* Declarations and initializations. */
    EOmega inst = (EOmega)model.createEntityInstance(EOmega.class);
    double real_num;
    EAttribute attr = inst.getAttributeDefinition("o2");

    /* Ensures that the attribute is initially unset after creation of the
instance. */
    if (inst.testAttribute(attr, null) != 0) {
        System.out.println("Test failed: attribute shall be initially
unset.");
    }

    /* Ensures that get_attribute reports a VA_NSET error when the attribute
has no value. */
    try {
        real_num = inst.get_double(attr);
        System.out.println("Test failed: VA_NSET error shall be reported when
the attribute has no value.");
    } catch (SdaiException ex) {
        if (ex.getErrorId() != SdaiException.VA_NSET) {
            System.out.println("Test failed: VA_NSET error shall be reported
when the attribute has no value.");
        }
    }

    /* Ensures that put_attribute reports a VA_NSET error when an unset value
for the attribute is submitted.
*/
    try {
        inst.set(attr, Double.NaN, null);
        System.out.println("Test failed: VA_NSET error shall be reported when
value is not submitted.");
    } catch (SdaiException ex) {
        if (ex.getErrorId() != SdaiException.VA_NSET) {
            System.out.println("Test failed: VA_NSET error shall be reported
when value is not submitted.");
        }
    }

    /* Ensures that put_attribute reports a VA_NVLD error when used for a
value of a wrong type. */
    try {
        inst.set(attr, "something", null);
        System.out.println("Test failed: VT_NVLD error shall be reported when
value of a wrong type is submitted.");
    } catch (SdaiException ex) {
        if (ex.getErrorId() != SdaiException.VT_NVLD) {
```



```

        System.out.println("Test failed: VT_NVLD error shall be reported
when value of a wrong type is submitted.");
    }
}

/* Ensures that put_attribute works correctly when all parameters are
correct. */
try {
    inst.set(attr, 5.6, null);
} catch (SdaiException ex) {
    System.out.println("Test failed: no SdaiException can be thrown.");
}

/* Ensures that test_attribute reports TRUE after successfully invoking
put_attribute. */
if (inst.testAttribute(attr, null) == 3) {
    System.out.println("Test failed: attribute cannot be unset after
assigning a value to it.");
}

/* Ensures that get_attribute retrieves the right value. */
if (inst.get_double(attr) != 5.6) {
    System.out.println("Test failed: attribute must have the value which
was assigned.");
}

/* Ensures that the attribute value can be unset. */
try {
    inst.unsetAttributeValue(attr);
} catch (SdaiException ex) {
    System.out.println("Test failed: no SdaiException can be thrown.");
}

/* Ensures that the attribute is really unset after unset_attribute_value
operation. */
if (inst.testAttribute(attr, null) != 0) {
    System.out.println("Test failed: attribute shall be unset after unset
attribute value operation.");
}
}

public SdaiModel macro_get_data_dictionary_model() throws SdaiException {
    SdaiSession session = SdaiSession.getSession();
    SchemaInstance schema_instance = session.getDataDictionary();
    return schema_instance.getAssociatedModels().getByIndex(1);
}

public TestCases() {
}

public static final void main(String argv[]) throws SdaiException {
    SdaiModel model1, model2;
    SdaiRepository repo1, repo2;
    SchemaInstance schema_instance;

    if (argv.length != 0) {
        System.out.println("USAGE: java TestCases");
        return;
    }
    TestCases tc = new TestCases();
}

```

ISO TC184/SC4/WG11 N137

```
SdaiSession.setLogWriter(new PrintWriter(System.out, true));
SdaiSession session = SdaiSession.openSession();
SdaiTransaction trans = session.startTransactionReadWriteAccess();

System.out.println("TestCases started");
repo1 = session.createRepository("Test_repo1", null);
repo1.openRepository();
repo2 = session.createRepository("Test_repo2", null);
repo2.openRepository();
modell1 = repo1.createSdaiModel("Test_model1", SGreek.class);
modell1.startReadWriteAccess();
model2 = repo2.createSdaiModel("Test_model2", SGreek.class);
model2.startReadWriteAccess();
schema_instance = repo2.createSchemaInstance("Test_schema_instance",
SGreek.class);
schema_instance.addSdaiModel(modell1);
schema_instance.addSdaiModel(model2);
tc.test_delete_SDAI_model(modell1, model2);
System.out.println("TestCases finished");
trans.abort();
repo1.deleteRepository();
repo2.deleteRepository();
session.closeSession();
}
}
```

Annex D
(informative)
Issues Log

Each issue is set out as follows:

Issue number these are assigned sequentially

Issue date the date given on the written issue or, where this is not given, the date on which the issue was received.

Author the author of the issue may be an individual or a committee.

Status the current status of the issue. This is categorized as follows:

Open this issue has not been resolved by WG6

Resolved a resolution to the issue has been agreed by WG6; a brief discussion of the resolution is given together with a reference to the version of the Part document in which the resolution is fully documented.

Rejected the issue has been discussed by WG6 but has not been accepted. A brief statement of the reason for rejection of the issue is given.

Clarification sought the issue is not understood by WG6; the author has been asked to supply additional information.

Deferred the issue has been discussed by WG6 and, while it has been agreed that this issue is valid, it has been agreed that resolution will not be possible until a future version of the Part. A brief statement of the reasons for the deferral are given.

Other projects are named (WGN/Pm) if WG6 has found it necessary or expedient to consult with other projects or working groups in resolving, rejecting or deferring an issue.

Note: Where other projects are indicated for an **open** issue, this indicates that WG6 (or, in some cases, the document editor) has identified a requirement to consult with other projects or working groups.

Issue text the text of the issue as documented by its author.

Technical discussion see comments above regarding the statement of reasons for the acceptance, rejection or deferral of an issue.

Issues

Issue 1

Date: 12 June 1995

Author: Shantanu Dhar

Status: Open, to be resolved, Lothar Klein, 1999/11/07

The Guidelines for the development of abstract test suites address the testing of preprocessors and postprocessors, i.e. only the exchange structure is addressed. The specification, development, and documentation of abstract test cases and suites has to be addressed. Part 35 or a companion document would have to address this.

Technical Discussion: Part 35 will contain all abstract test cases as EXPRESS procedures in the SDAI_test_cases_schema. In addition to this the corresponding part 21 input and output will be given as comments. It is up to specific implementation and language binding to specify proper test suites. For the complete Java(TM) programming language binding to the SDAI test suites for late and early binding will be given in an informative annex.

Issue 2

Date: 12 June 1995

Author: Shantanu Dhar

Status: Open, to be resolved, Lothar Klein, 1999/11/07

WG 6 and 7 need to resolve the issue of exhaustive testing of early bound implementations. The question is: Is it necessary to exhaustively test every function for the creation, deletion, and modification of every entity type in the application schema? Or does a judiciously chosen representative set of such functions adequately address testing needs.

Technical Discussion: This selection is done by specifying the artificial "Greek" test schema. Instances of the specified entities are created, deleted and modified in the given test cases.

Issue 3

Date: 12 June 1995

Author: Shantanu Dhar

Status: Open, to be resolved, Lothar Klein 1999/11/07

ISO 10303-22 specifies error conditions for all operations. Testing will have to check whether an implementation does indeed return the appropriate error values when error conditions are encountered. This implies error test cases would have to be part of SDAI test suites.

Technical Discussion: The SDAI operations are mapped into EXPRESS functions and procedures of the SDAI_test_operation_schema with additional parameters for expected error codes and error base. The

given test cases specify which error conditions of the many possible to test.

Issue 4

Date: 7 November 1999

Author: Lothar Klein

Status: Open

In the original EXPRESS (ISO 10303-22 + TC1 + TC2) there is no predefined method for type casting simple data type values for named types. Especially for select data types of named data types this is a real problem. The only way to solve this in EXPRESS 1 is to defined functions with the proper return type. Part 35 requires a default mechanism in EXPRESS to do such casting for the EXPRESS planned Amendment 1.1. This should consider either the "type_mark" from EXPRESS 2 (WG11N81) or (better) for every named type declaration defines an implicit function with the same name which returns this type and accept as it's parameter the underlying type.

Technical Discussion:

Index