
**Technical Report**
RAL-TR-96-043

# The MIDAS SDAI Implementation

D Thomas and C Greenough

July 1996

COUNCIL FOR THE CENTRAL LABORATORY OF THE RESEARCH COUNCILS

# MIDAS SDAI implementation

Mrs D Thomas and Dr C Greenough

February 1996

**Abstract**

This report describes the implementation of the STEP Standard Data Access Interface (SDAI) used within the EC Project MIDAS.

MIDAS (ESPRIT project 7294) was a three year project which brought together the software from different vendors in the areas of solid modelling, advanced mesh generation, stress analysis and electromagnetic analysis to provide an integrated suite of software with a common user interface and a common database. The common data management and exchange mechanism was based on the STEP technology.

STEP is now starting to be used as an integration tool between CAD applications. Electromagnetic analysis is not covered presently by STEP but the existing models and methodology can be used and extended into this area.

Part of the STEP methodology includes the definition of a STEP Data Access Interface (SDAI) and how the STEP information models can be accessed in a common database using this interface.

The reports describes how a C late binding SDAI has been implemented on top of the DEVA database in order to access the STEP compatible data models defined within the MIDAS project for electromagnetic analysis.

---

Mathematical Software Group
Advanced Interactive Systems Division
Rutherford Appleton Laboratory
Chilton, Didcot
Oxfordshire OX11 0QX

# Contents

# 1 Introduction

The International Standard for data exchange ISO 10303, known as STEP, is now starting to be used as an integration tool between CAD applications. Electromagnetic analysis is not covered presently by STEP but the existing models and methodology can be used and extended into this area.

Part of the STEP methodology includes the definition of a STEP Data Access Interface (SDAI) [6] and how the STEP information models can be accessed in a common database using this interface. This reports describes how a C late binding SDAI [5] has been implemented on top of the DEVA database in order to access the STEP compatible data models previously defined within the MIDAS project [1] [2].

# 2 DEVA Database

The database which was used by the MIDAS environment was called DEVA. DEVA provides a database management system, I/O control and a data access interface. A prototype version of DEVA was written at Rutherford Appleton Laboratory under the ESPRIT II program, Project 5172(IDAM) and has been developed further under MIDAS Project to meet the needs of the MIDAS Environment.

One of the first tasks undertaken in MIDAS was to enhance DEVA to run as a client/server database. In this mode a server database is initiated on any accessible machine. This server is then available for any application to act as a client and to access the data. Each application may perform various actions i.e. place data, retrieve data, save datasets and rollback, via function calls. When application processes are closed the server database remains active until explicitly closed.

Like any other database, the DEVA database needs a schema which describes the data to be stored and accessed. DEVA is STEP compatible because it uses as a starting point the entity definitions of the Application Protocols and Integrated Resource Models of STEP. MIDAS has added to and modified the STEP *EXPRESS* descriptions to meet the requirements of electromagnetic applications.

DEVA is closely linked to the *EXPRESS* compiler 'EX' which was developed in the IDAM project. One of the outputs from EX is a Keyword Definition File (KDF) which forms the basis of the DEVA data dictionary. Additional information is required by the DEVA database about block sizes and array bounds.

DEVA originally had its own subroutine interface but this has now been replaced by a subset of a C late binding SDAI.

# 3 STEP Overview

ISO 10303, known informally as STEP (STandard for Exchange of Product model data), 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.

Information models are defined using the formally defined data specification language, *EXPRESS* [3], which was developed as part of STEP.

Implementation methods are also defined as part of STEP, these are:

**physical file:** sequential, free format file exchange [4]. Defines the mapping from *EXPRESS* to the physical file. The syntax is defined in Wirth Syntax Notation (WSN).

**SDAI:** Standard data access interface [6]. Formal specification of the interface between an application and instances of data in a form specified by an *EXPRESS* schema. The data may be stored in a database, a file or an in-memory working form. This specification is independent of any implementation language.

**late bindings:** specification of the SDAI for specific target languages which is independent of the *EXPRESS* schema being implemented. Currently being defined for C.

**early bindings:** specification of the SDAI for specific target languages which is dependent on the *EXPRESS* schema being implemented. Currently being defined for C++

The way that information models are implemented into a specific type of database will not be defined in STEP, only the interface to access the data conforming to the model.

# 4 Implementation Overview

ISO 10303 Part 22 [6], which defines the STEP Standard Data Access Interface, also specifies the possible implementation classes to which an SDAI implementation must conform. The MIDAS SDAI satisfies many of the requirements of a Class 1 implementation. This supports:

- Level 1 of transaction support. This level consists of an implementation providing no support of the session transaction operations nor the SDAI-model save and undo operations.
- Level 1 of expression evaluation support. This level consists of all validation operations always returning UNKNOWN as their result and support only for Get of derived attributes defined in the SDAI session schema and no support required for any access to the SDAI dictionary entity instances. Constraints defined in SDAI session schema are enforced at all times regardless of support for the validation operations.
- Level 1 of session record support. This level consists of no support for the session event recording function. At this level of support, no instances of the `error_event` entity type are created,the `sdai_session.errors` list is always empty and `session.recording_active` is always FALSE.
- Level 1 of scope support. This level consists of no support for the scope operations.
- Level 1 of domain equivalence support. This level consists of no support for the declaration of domain equivalent entity types and their use by application instances based upon different schema definitions.

Annex I gives the operations required by implementation class.

The files that make up the DEVA software are organised into several directories. The directories `Clients`, `Server` and `Alone` contain routines which are specific to the client, server and standalone versions of DEVA respectively. All of the header files are in a directory called `Include`. The directory `Library` contains all the routines which are common to both versions of DEVA. As many of the routines as possible are placed in the `Library` directory to ensure consistency and ease of maintenance.

The SDAI functions have been added to the `Library` directory as they have been made common between both versions of DEVA. This has been made possible by ensuring that no SDAI routines access the DEVA database directly. The only access is made through DEVA routines. Thus the SDAI forms a separate layer sitting above the DEVA interface level.

Only a few changes have been made to the existing DEVA routines. Where the SDAI routines could not be based upon existing DEVA routines then new DEVA routines have been designed and written for both versions and placed in the relevant directories. These new routines are described in detail in Section 7.

A testbed program has been designed and written. This serves to test out the new DEVA and SDAI routines as they are implemented and may also be used as an example program for new users of the SDAI.

# 5 SDAI Header File

There is a C late binding header file called <sdai.h> which is included as an Annex to Part 24. Parts of this header file are necessary to understanding the function prototypes given in a later section so the relevant parts are reproduced here. In addition, some of the types are deliberately left undefined as they are intended to be implementation dependent. The MIDAS definitions of these are given in a separate file called <sdai_deva.h> which is also given below.

## 5.1 Extracts from sdai.h

```
/*** Constant declarations ****************************************/

/* LOGICAL and BOOLEAN value elements: */

#define sdaiFALSE   0
#define sdaiTRUE    1
#define sdaiUNKNOWN 2

/*** Type declarations ********************************************/

typedef unsigned char   SdaiBit;
/* C late binding primitive data types: */
typedef long        SdaiInteger;
typedef double          SdaiReal;
typedef int         SdaiBoolean;
typedef int         SdaiLogical;
typedef char        *SdaiString;
typedef SdaiBit         *SdaiBinary;
/* enumeration data type:                                        */
typedef char        *SdaiEnum;
/* aggregate data types:                                         */
typedef SdaiAggrId      SdaiAggr;
typedef SdaiAggr        SdaiOrderedAggr;
typedef SdaiAggr        SdaiUnorderedAggr;
typedef SdaiOrderedAggr SdaiArray;
```

```
typedef SdaiOrderedAggr SdaiList;
typedef SdaiUnorderedAggr     SdaiSet;
typedef SdaiUnorderedAggr     SdaiBag;
/* entity instance identifier type:                                    */
typedef SdaiId                SdaiInstance
/* SDAI instance identifier types:                                     */
typedef SdaiInstance          SdaiAppInstance;
typedef SdaiInstance          SdaiModel;
typedef SdaiInstance          SdaiRep;
typedef SdaiInstance          SdaiSession;
typedef SdaiInstance          SdaiAttr;
typedef SdaiAttr              SdaiExplicitAttr;
typedef SdaiInstance          SdaiEntity;
typedef SdaiInstance          SdaiWhereRule;
typedef SdaiInstance          SdaiUniRule;
typedef SdaiInstance          SdaiGlobalRule;
typedef SdaiInstance          SdaiSchema;
typedef SdaiInstance          SdaiScope;
typedef SdaiInstance          SdaiSchemaInstance;
typedef SdaiInstance          SdaiTrx;
/* SDAI iterator identifier type:                                      */
typedef SdaiItrId             SdaiIterator;
/* access mode data type:                                              */
typedef enum {sdaiRO, sdaiRW) SdaiAccessMode;
/* C late binding ADB identifier type:                                 */
typedef SdaiADBId             SdaiADB;
/* attribute type data type:                                           */
typedef enum {
            sdaiADB, sdaiAGGR, sdaiBOOLEAN, sdaiBINARY, sdaiENUM,
            sdaiINSTANCE, sdaiINTEGER, sdaiLOGICAL, sdaiNOTYPE,
            sdaiREAL, sdaiSTRING
            } SdaiPrimitiveType;
/* error code data type:                                               */
typedef SdaiErrorId           SdaiErrorCode;
/* error handler data type:                                            */
typedef void      (*SdaiErrorHandler)(SdaiErrorCode);
/* transaction commit mode data type:                                  */
typedef enum {sdaiCOMMIT, sdaiABORT} SdaiCommitMode;
```

## 5.2 Extracts from sdai_deva.h

```
struct SdaiId {
       int   type;       /* number from 1 to 7*/
       int   ds;         /* dataset number for entity ids, else 0 */
       int   ent_type;   /* entity type number for entity ids, else 0 */
       int   id;         /* instance number dependent on type */
 };

struct SdaiAggrId {
       int   level;      /* aggregate level = 1 for 1D aggregates*/
```

```
        int   ds;         /* dataset number*/
        int   ent_type;   /* entity type number*/
        int   id;         /* entity instance number */
        int   attr;       /* attribute number */
};

struct ModelTable {
        int id;     /* SDAI-model instance number */
        char name[MAXNAMESZ];     /* name of SDAI-model */
        int status;  /* =0 when created =1 when exported ?? */
};

struct ArrayStore {
        struct SdaiAggrId id;
        int highest; /* highest index actually stored */
        int length; /* length of array values type */
        char * values; /* pointer to buffer of values */
};

/* Type definitions for global variables used in sdai.c and sdai_deva.c */
typedef int SdaiErrorId;
#define SDAI_REF_IS  struct SdaiId
#define MAX_DATASETS 10
extern SdaiErrorId errorcode;
extern int highest_model;
extern struct ModelTable models[MAX_DATASETS];
extern struct ArrayStore putarray;
extern struct ArrayStore getarray;

/* Function prototypes for the functions in sdai_deva.c */
struct SdaiId get_sdai_id(int type, int instance_no  );
struct SdaiId deva_to_sdai(struct Inst_Id deva_id );
struct Inst_Id sdai_to_deva(struct SdaiId ent_id);
struct Inst_Id aggr_to_deva(struct SdaiAggrId aggr_id);
struct SdaiAggrId get_aggr_id(struct SdaiId sdai_id, int level,
                              int attrnum);
int get_session_id(void);
int cmp_aggr_id(struct SdaiAggrId aggr_id_1, struct SdaiAggrId aggr_id_2);
int put_array();
```

# 6 General Concepts

This section details some implementation issues which are general to more than one of the C late binding SDAI functions.

## 6.1 Global identifiers

The SDAI assumes that the database can provide identifiers for several types of instances of information. These must be unique over all the instances in the database during a session. This identifier is

provided to the application via SDAI routines. The application then uses the identifier in further SDAI calls.

In the header file all of the relevant types are defined to be of type `SdaiInstance` which is itself defined to be of type `SdaiId`. `SdaiId` is an implementation specific handle. The handle serves as the identifier of the instance. Identifiers are not persistent. Identifiers shall be unique globally over all types of instances and unchanging within a simple session for any particular instance.

It is proposed to compose an `SdaiId` from four specific DEVA identifiers. This composition will then ensure global uniqueness of all SDAI identifiers. The four integers which are used to form the unique identifier for each SDAI type are as follows:

1. SDAI type - this is an integer giving the type of the SDAI entity. The value for each type is:
   - `SdaiAppInstance` - 1
   - `SdaiModel` - 2
   - `SdaiRep` - 3
   - `SdaiSession` - 4
   - `SdaiAttr` - 5
   - `SdaiEntity` - 6
   - `SdaiSchema` - 7
2. DEVA dataset number - only for application instances ie. Type 1, for other types this is set to zero.
3. DEVA entity type number - only for application instances ie. Type 1, for other types this is set to zero.
4. instance number - an identifying number which is unique within that type. The meaning of this number for each type is:
   - `SdaiAppInstance` - DEVA entity identifier
   - `SdaiModel` - dataset number
   - `SdaiRep` - repository number
   - `SdaiSession` - session number
   - `SdaiAttr` - attribute number (order in *EXPRESS* model)
   - `SdaiEntity` - DEVA entity type number
   - `SdaiSchema` - schema number

The problem is that if the identifier is generated from combining four integers of unknown size then the result could be too big for a single integer variable. Three possible solutions are:

1. Make the identifier a structure and store the four integers separately. This has the advantage of being easy to generate and separate but is difficult for the application to do comparisons with. If the application simply receives the id from one routine and passes it on to another routine then it could be sufficient.

2. Make the identifier of type `char *` and store the four integers separated by a non-numeric character, for example '.'. This is easy to generate and fairly easy to separate although not as easy as 1. It is easier for the application to deal with a single variable for comparisons but string comparisons will be necessary.

3. Make the identifier of type `int`. Combine the four integers together and apply a hashing algorithm to compress the number into a single integer. This results in a single integer identifier which is easy for the application to deal with but a routine will always be needed to generate and separate it.

The initial implementation will use solution 1. The structure chosen is included in `<sdai_deva.h>` as given above.

## 6.2 Concept mappings

There are more levels in the SDAI than in DEVA. The following is an attempt to match SDAI concepts to those present in DEVA.

- **Session:** The set of operations that occur between the initiation and termination of the use of an SDAI implementation by one application.

  There will be one single session in the initial SDAI implementation. There is no comparable concept in DEVA so the session will exist in the SDAI layer. When the session is opened a predefined session identifier will be allocated and several global variables will be initialised.

- **Repository:** Repositories are data storage facilities. A repository may be implemented in memory, as a single database, multiple databases, a single file, a collection of files, or any other method. Multiple repositories may be opened in a single session.

  For the MIDAS SDAI implementation it will be possible to open one repository in a session. When DEVA is initialised in reads a single schema into its data dictionary. It is not possible to change this data dictionary during that run of DEVA. When an SDAI repository is opened this will have the effect of initialising DEVA and reading in a data dictionary. The name of the repository will be used for the name of the data dictionary file to be read in.

- **Schema instance:** Schema instances are logical collections of SDAI-models from which a set of entity instances can be derived. This set of entity instances defines the domain over which references between entity instances and global rule validation are supported. Schema instances must be created within a repository.

  These are not implemented in DEVA and are not necessary in the SDAI. They will not be implemented.

- **SDAI-model:** Entity instances are created within SDAI-models which must be created within a repository. The entity instances making up each SDAI-model are based on a single *EXPRESS* schema. One SDAI-may be associated with more than one schema instance.

  A single SDAI-model will be mapped to a single DEVA dataset.

## 6.3 Basic types

The following table shows the mapping between SDAI types, DEVA types and basic C types.

| SDAI | C type | DEVA | C type |
|---|---|---|---|
| SdaiInteger | long | BIG_INT | long |
| SdaiReal | double | BIG_REAL | double |
| SdaiBoolean | int | BOOLEAN | unsigned char |
| SdaiLogical | int | LOGICAL | unsigned char |
| SdaiString | char* | VLEN_THING | char* |
| SdaiEnum | char* | ENUM_REF | unsigned int |

Table 1: SDAI and DEVA type mappings

### 6.4 Aggregates

DEVA currently only deals with arrays and lists of a fixed length (they get treated like arrays). It does not deal with sets, bags or lists of variable length. An array is stored and extracted by casting the array as a `char*` and treating it like any other attribute, ie. storing the array with its values along with the other attribute values for an entity instance. Arrays can only be stored and extracted as a complete block not as individual values.There are no specific routines for arrays.

The SDAI treats aggregates differently from other attributes. The aggregate has its own unique identifier and is stored separately from the entity instance. The relevant attribute of the entity instance contains a pointer to the aggregate instance. Values are placed into and extracted from the array individually.

To enable this SDAI functionality to be built on top of DEVA, the SDAI interface layer manages the array values in local dynamically allocated storage. This enables values to be put and got individually by the SDAI but as a whole array within DEVA.

Consider first an array that is present in the DEVA database. The first time that a SDAI routine attempts to access the array, sufficient local storage is obtained to store the whole array which is then copied from the DEVA database into the local memory. This is called `getarray`. It is most likely that the next routine will ask for another value from the same array and in this case the value is simply obtained from local storage without the need to access the database. The array continues to reside in local memory until a read request is received for a different array when the local memory is released and fresh memory obtained for the new array.

Considering an array that is having values set in the database by the SDAI. The first time that a value is set in an array, enough local memory is obtained for the whole array. The value is written into this local memory and not into the database. This array is called `putarray`. It is most likely that the next value to be set will also be for this same array and in which case the value may be written straight into the local storage. This continues until either a routine wants to write a value into a different array in which case the whole array is then down-loaded to the DEVA database, the memory released and fresh memory obtained for the new array, or a routine wants to read a value from the current array in which case the array is both down-loaded to the database and transferred to the 'getarray' storage.

### 6.5 Implementation type

Implementations either support SDAI-model Save and Undo operations or Commit and Abort transactions. The DEVA SDAI will support Save and Undo.

# 7 New DEVA Routines

The following new DEVA routines have been implemented. They were all necessary to provide functionality needed by the SDAI which was not available from existing DEVA routines.

It was necessary for the SDAI to have the name of the attributes as given in the *EXPRESS* data model available in the database data dictionary. At the start of MIDAS, DEVA stored entity names but not attribute names. Therefore, EX has been extended to add attribute names from the *EXPRESS* data model when outputting a DEVA data dictionary file. DEVA has also been modified so that it now reads in this attribute name and stores in an extended version of the attribute structure in the data dictionary. This then enabled the following new routines which use attribute name to be written.

## 7.1 deva_get_attr_name

int deva_get_attr_name(char ent_name[], int attr_number, char * attr_name)

This routine gets the name of an attribute given the entity name and the attribute number. The number of the attribute is taken from the order of attributes as given in the *EXPRESS* model with the first having number 0.

## 7.2 deva_get_attr_num

int deva_get_attr_num(int index, char * attr_name, int * attr_num)

This routine gets the number of an attribute given the entity index and the attribute name. The number of the attribute is taken from the order of attributes as given in the *EXPRESS* model with the first having number 0. The first argument should be the index of the *EXPRESS* entity. This can be obtained from the entity name by calling the DEVA routine `deva_q_atoi`.

## 7.3 deva_get_attr_type

int deva_get_attr_type(int index, int attr_number, int * attr_type)

This routine gets the type of an attribute given the entity index and the attribute number. The number of the attribute is taken from the order of attributes as given in the *EXPRESS* model with the first having number 0. The first argument should be the index of the *EXPRESS* entity. This can be obtained from the entity name by calling the DEVA routine `deva_q_atoi`.

## 7.4 deva_get_aggr_lim

int deva_get_aggr_lim(int index, int attr_number, int * limit)

This routine gets the limit of a single level array or list given the entity index and the attribute number. The number of the attribute is taken from the order of attributes as given in the *EXPRESS* model with the first having number 0. The size of the array or list as given in the data dictionary is returned in limit. Only one number is stored in the data dictionary so an array of dimension [5:9] will have limit 4. For a list the limit is the maximum size of the list.

### 7.5 deva_get_aggr_type

int deva_get_aggr_type(int index, int attr_number, int * type)

This routine gets the type of the elements of a single level array or list given the entity index and the attribute number. The number of the attribute is taken from the order of attributes as given in the *EXPRESS* model with the first having number 0. The type of the elements or the array or list as given in the data dictionary is returned in `type`.

### 7.6 deva_get_array

int deva_get_array(struct Inst_Id id, int attrnum, int * maxsize, void * attrp)

This routine gets a copy of all the values in an array for a specified attribute of a specified entity instance when that attribute is of type `ARRAY`. The maximum size of the array is returned in `Mmaxsize`. The array is returned in `attrp` which should be cast as `(char*)` by the caller.

### 7.7 deva_put_array

int deva_put_array(struct Inst_Id id, int attrnum, int put_size, void * attrp)

This routine sets the value for a number of members in an array for a specified attribute of a specified entity instance when that attribute is of type `ARRAY`. The third argument is the size of the array in bytes, this allows less than the maximum size of the array to be put. The array of values should be given in the final argument and cast as `(char*)` by the caller.

### 7.8 deva_q_u_name

int deva_q_u_name(int index, int attr_number, char * enum_name)

This routine gets the enumeration name of an attribute which is of type enumeration given the entity index and the attribute number. The number of the attribute is taken from the order of attributes as given in the *EXPRESS* model with the first having number 0. The first argument should be the index of the *EXPRESS* entity. This can be obtained from the entity name by calling the DEVA routine `deva_q_atoi`.

### 7.9 deva_put_entity_id

int deva_put_entity_id(struct Inst_Id id)

This routine creates an entity instance with no attribute values.

### 7.10 deva_put_attribute

int deva_put_attribute(struct Inst_Id id, int attrnum, int type, void* attrp)

This routine is adapted from the original DEVA routine `deva_put_entity` which put values for all attributes into an entity instance. `deva_put_attribute` sets a value for a single attribute of an entity instance.

The first two arguments are the instance identifier of the entity instance and the attribute number which is being set. The third argument is the type of the attribute which is one of the DEVA types given in the Table 1 in Section 6.4.

The final argument is the value of the attribute which is of the appropriate type but must be cast to a `void*`.

### 7.11 deva_get_attribute

int deva_get_attribute(struct Inst_Id id, int attrnum, int * type, void* attrp)

This routine is adapted from the original DEVA routine `deva_get_entity` which got values for all attributes from an entity instance. `deva_put_attribute` gets a value for a single attribute of an entity instance.

The first two arguments are the instance identifier of the entity instance and the attribute number which is being set. The third argument returns the type of the attribute which is one of the DEVA types given in the Table 1 in Section 6.4.

The final argument should be a variable of the appropriate type which should be cast to `void*` and will be filled with the value of the attribute.

## 8 C Late Binding Routines

This section lists each C function defined in STEP Part 24 [5] which is required for implementation Class 1. Some of the functions given in Part 24 are specific to the C late binding and are not based on Part 22 [6], these functions are only included below if it is considered that they would be useful for the MIDAS SDAI.

For each function the following information is given if relevant:

- C function prototype
- priority of implementation within the MIDAS project. The priority numbers have the following meanings:
    1. Of top priority (Priority 1), these will be the first routines to be implemented for the first release of the SDAI
    2. Of medium priority (Priority 2), will be implemented in the second release of the SDAI.
    3. Of low priority (Priority 3)
    4. Routine not needed for the MIDAS project, will not be implemented (Priority 4) (In this case the following information will not be given)
- input, output and return values for the function
- effect on SDAI environment, only a relevant subset is given
- mappings to relevant DEVA functions and variables
- problem areas

## 8.1 Open Session (6.1.1 )

This operation initiates the SDAI implementation and commences a new SDAI session.

```
SdaiSession sdaiOpenSession (void);
```

**Priority: 1**

**Return:** session instance identifier

**Effect:** An instance of `sdai_session` is created. An instance of implementation is created. The `known_servers` attribute shall be initialised with a set instance whose members are the instances of `sdai_repository` available to the application for this session.

**Status:** Implemented. For the first release of the MIDAS SDAI there is a single session with a predefined identifier. When this routine is called it initialises several global variables, for example, `errorcode` is set to `SdaiNO_ERR`

## 8.2 Error Query (6.1.3.1 )

```
SdaiErrorCode sdaiErrorQuery (void);
```

**Priority: 2**

**Return:** The error code stored in the conceptual single valued error buffer.

**Status:** This has been implemented by using a single external integer variable called `errorcode`. The definition of this is included in the additional header file `sdai_deva.h`. Implemented and tested.

## 8.3 Record event (6.2.1 )

**Priority: 4**

## 8.4 Close session (6.2.3 )

This function shall terminate the current SDAI session. It sets the error indicator to tt sdaiSS_NOPN. After invoking this function, subsequent C late binding function calls shall no longer be operating in an SDAI environment. All instance identifiers shall no longer be valid once the session is closed.

```
void sdaiCloseSession (SdaiSession session);
```

**Priority: 1**

**Status:** Not implemented. Active `sdai_models` should be returned to an appropriate state before terminating.

### 8.5 Open repository (6.2.4 )

The Open Repository function shall make the repository and its SDAI-models available to the session.

SdaiRep sdaiOpenRepository (SdaiSession session, SdaiRep repository);

SdaiRep sdaiOpenRepositoryBN (SdaiSession session, SdaiString repositoryName);

**Priority: 1**

**Input:**

`session` - identifier of the session in which the repository is to be opened

`repository` - depository identifier

`repositoryName` - repository name

**Return:**

`repository` instance identifier

**Status:** Implemented. For the first release of the MIDAS SDAI there will only be repository available. `deva_init` is called using the repository name as the name of the data dictionary file to be opened.

### 8.6 Create non-persistent list (6.2.8 )

This function creates a non-bounded, non-persistent list (NPL) as a container of entity instances. It is the intention that NPLs are possibly to access by any C late binding function which has a parameter of the type `SdaiList`.

**Priority: 4**

### 8.7 Delete non-persistent list (6.2.9 )

**Priority: 4**

### 8.8 SDAI query (6.2.10 )

This function determines those entity instances from a NPL, which meet a specified criteria and puts them to the result NPL. The criteria for this function are limited to a subset of EXPRESS expressions.

**Priority: 4**

### 8.9 Delete non-persistent list (6.11.2.1 )

**Priority: 4**

### 8.10 Create ADB (6.2.12.1 )

An ADB is an Attribute Data Block. This is needed for `sdaiPutAllAttrs`.

**Priority: 2**

### 8.11 Get ADB value (6.2.12.2 )

This function is needed to get the values out of the ADB set passed to the SDAI by the routine `sdaiGetAllAttrs`.

void *sdaiGetADBValue (SdaiADB block, SdaiPrimitiveType valueType, void *value);

**Priority: 2**

**Input:**

`block` - attribute data block containing the data to be retrieved

`valueType` - type of value

`value` - handle matching or convertible to the `valueType`

**Output:**

`value` - the handle filled with the primitive or identifier value retrieved from the ADB

### 8.12 Put ADB value (6.2.12.3 )

**Priority: 2**

### 8.13 Get ADB type (6.2.12.4 )

**Priority: 2**

### 8.14 Delete ADB (6.2.12.5 )

**Priority: 2**

### 8.15 Create schema instance (6.3.1 )

Schema instances do not map directly into anything available in DEVA and are not necessary for the SDAI implementation.

**Priority: 4**

### 8.16 Add SDAI-model (6.3.2 )

**Priority: 4**

### 8.17 Remove SDAI-model (6.3.3 )

**Priority: 4**

### 8.18 Delete schema instance (6.3.4 )

**Priority: 4**

### 8.19 Create SDAI-model (6.4.1 )

This function shall create a new SDAI-model in the specified repository.

SdaiModel sdaiCreateModel (SdaiRep repository, SdaiString modelName, SdaiSchema schema);

SdaiModel sdaiCreateModelBN (SdaiRep repository, SdaiString modelName, SdaiString schemaName);

**Priority: 1**

**Input:**

`repository` - the identifier of the repository in which the SDAI-model is to be created.

`modelName` - the name of the new SDAI-model, shall be unique within the repository.

`schema` - the identifier of the schema that is associated with the SDAI-model

`schemaName` - the schema is identified by its name instead of the identifier

**Return:** identifier of the SDAI-model instance

**Status:** Implemented. A model table is constructed from an array of structures and made globally available in the SDAI. Each structure contains the number of an SDAI-model and its name. The number is the number of the DEVA dataset which is allocated to that SDAI-model.

When this routine is called the model table is checked and a new entry created if the `modelName` is not present. Only one data model can be used in a single DEVA server so the input schema is ignored in the first version and will be checked in a later release.

### 8.20 Close repository (6.4.2 )

This function shall close the specified repository.

**Priority: 1**

**Status:** Not yet implemented

This will call `deva_finish` and export any unsaved datasets.

### 8.21 Get schema definition (6.4.3.1 )

This function shall get the schema definition entity to a given schema name.

**Priority: 4**

### 8.22 Get schema instance (6.4.3.2 )

This function shall get the schema instance identifier to a given schema instance name.

**Priority: 3**

### 8.23 Delete SDAI-model (6.5.1 )

This function shall delete an SDAI-model along with all of the instances which it contains.

void sdaiDeleteModel (SdaiModel model);

**Priority: 1**

**Input:**

`model` - the SDAI-model to delete

**Status:** Not yet implemented. This could be mapped to `deva_del_dataset`.

### 8.24 Rename SDAI-model (6.5.2 )

This function shall assign a new name to a SDAI-model.

**Priority: 2**

**Status:** This can be implemented at the SDAI interface level by simply changing the name in the model table.

### 8.25 Start SDAI-model access (6.5.3 )

SdaiModel sdaiAccessModel (SdaiModel model, SdaiAccessMode mode);

SdaiModel sdaiAccessModelBN (SdaiRep repository, SdaiString modelName, SdaiAccessMode mode);

**Priority: 1**

**Input:**

`model` - the identifier of the SDAI-model whose access mode is to be assigned.

`repository` - the identifier of the repository containing the SDAI-model.

`modelName` - the name of the SDAI-model, shall be unique within the repository.

`mode` - the access mode to be assigned to the SDAI-model, can be: `sdaiRO` for read-only, `sdaiRW` for read-write

**Return:** identifier of the SDAI-model instance

**Status:** Implemented. This routine checks the model table. If the `modelName` is not present then it adds a new entry to the table. It then imports the SDAI-model using `deva_fimport`. The input parameter access mode is ignored as this cannot be set in DEVA.

### 8.26 Promote SDAI-model to read-write access (6.5.4 )

**Priority: 4**

## 8.27 End SDAI-model access (6.5.5 )

This function shall end access to a given SDAI-model.

void sdaiEndModelAccess (SdaiModel model);

**Priority: 1**

**Input:**

`model` - the identifier of the SDAI-model whose access mode is to be terminated.

**Status:** Implemented and tested.

This routine checks the model table. If it finds a match then it exports that dataset to the filename given in the name attribute of the model structure. Before exporting the dataset this routine checks whether the array in `putarray` is in this dataset and, if so, commits the array to memory.

## 8.28 Get entity definition (6.5.6 )

This function shall retrieve the entity definition for the specified entity name within the schema associated to the specified SDAI-model.

SdaiEntity sdaiGetEntity (SdaiModel model, SdaiString name);

**Priority: 1**

**Input:**

`model` - the model of which the entity type is part of its schema

`name` - the entity type name

**Return:** the identifier of the entity definition

**Status:** Implemented using `deva_q_atoi`.

There is only one schema present in a DEVA database so the first parameter, model, will be ignored. The entity definition is obtained by a call to `deva_q_atoi`.

## 8.29 Create entity instance (6.5.7 )

This function shall create a new application instance of a specified type in an application SDAI-model.

SdaiAppInstance sdaiCreateInstance (SdaiModel model, SdaiEntity entity);

SdaiAppInstance sdaiCreateInstanceBN (SdaiModel modelName, SdaiString entityName);

**Priority: 1**

**Input:**

`model` - identifier of the SDAI-model in which an entity instance will be created

`entity` - identifier of an entity definition

`modelName` - the name of the SDAI-model in which in entity instance will be created

`entityName` - the name of the entity type

**Return:** identifier of an entity instance

**Status:** Implemented and tested.

Map model to dataset. Convert SDAI identifier to DEVA identifier.

Use `deva_get_etot (id, total, high)` to get the total number of entities of that type plus the highest id. Add one to highest id. Then call new DEVA routine, `deva_put_entity_id`, to create an entity instance with no attribute values.

### 8.30 Undo changes (6.5.8 )

**Priority: 2**

Will use DEVA routine `deva_do_rollback`.

### 8.31 Save changes (6.5.9 )

**Priority: 2**

Will use DEVA routine `deva_set_rollback`.

### 8.32 Create complex entity instance (6.5.10.1 )

This function shall create a new application instance of a specified type determined by a constructed entity type which is made up of the supplied simple entity types in an application SDAI-model.

**Priority: 4**

### 8.33 Get entity extent (6.5.10.2 )

This function shall retrieve an entity folder of a given entity type belonging to a SDAI-model.

SdaiSet sdaiGetEntityExtent (SdaiModel model, SdaiEntity entity);

SdaiSet sdaiGetEntityExtentBN (SdaiModel model, SdaiString name);

**Priority: 2**

**Input:**

`model` - the identifier of the model to which the folder belongs

`entity` - the identifier of the entity type definition for the folder

`name` - the name of the entity type

**Return:** identifier of a set containing entity instances

Map model to dataset. Call `deva_q_atoi` if BN. Call `deva_get_etot` to get highest id. Call `deva_get_entity` and place valid entity ids (composed into valid `SdaiIds`) into set. Return pointer to set.

### 8.34 Type operations (6.7 )

**Priority: 4**

### 8.35 Get attribute definition (6.7.5.1 )

SdaiAttr sdaiGetAttrDefinition (SdaiEntity entity, SdaiString attrName);

SdaiAttr sdaiGetAttrDefinitionBN (SdaiString schemaName, SdaiString entityName, SdaiString attrName);

**Priority: 2**

**Input:**

`entity` - identifier of an entity definition

`attrName` - name of an attribute

`schemaName` - name of the schema to which the entity type belongs

`entityName` - name of an entity type

**Return:** identifier of the specified attribute

Need identifiers for all attribute definitions globally. Concatenate `attr` number with entity type number and dataset number.

### 8.36 Get attribute (6.8.1 )

This function shall get, and may convert, a primitive or instance value of an attribute from an SDAI instance.

void* sdaiGetAttr (SdaiInstance instance, SdaiAttr attribute, SdaiPrimitiveType valueType, void *value);

void* sdaiGetAttrBN (SdaiInstance instance, SdaiString attributeName, SdaiPrimitiveType valueType, void *value);

**Priority: 1**

**Input:**

`instance` - the instance of the entity whose attribute is being retrieved `attribute` - an instance of an attribute definition from the meta-data schema `attributename` - the name of the attribute to be retrieved `valueType` - the type of the read attribute `value` - handle matching or convertible to the `valueType`

**Output:** `value` - handle filled with the primitive or identifier value retrieved from the attribute. If the attribute is an aggregate or instance, an identifier shall be returned that may be used in subsequent C late binding functions to examine their contents.

**Status:** Implemented using new DEVA routine, `deva_get_attribute`. Tested and working for integers, reals, strings, enumerations, entity references and select references.

## 8.37 Test attribute (6.8.2 )

SdaiBoolean sdaiTestAttr (SdaiInstance instance, SdaiAttr attribute);

SdaiBoolean sdaiTestAttrBN (SdaiInstance instance, SdaiString attributeName);

**Priority: 2**

**Input:**

`instance` - the instance of the entity whose attribute is being tested

`attribute` - an `SdaiAttr` instance from the meta-data schema

`attributename` - the name of the attribute being tested

**Return:** This function shall return `sdaiTRUE` if the attribute is assigned a value or `sdaiFALSE` if the attribute value is undefined.

## 8.38 Find entity instance SDAI-model (6.8.3 )

SdaiModel sdaiGetInstanceModel (SdaiInstance instance);

**Priority: 2**

**Input:**

`instance` - the instance identifier whose SDAI-model is to be found

**Return:** identifier of the found SDAI-model

This can be returned by decomposing the `id` to determine the dataset number and if necessary re-composing into a globally unique number. It is not necessary to access the database.

## 8.39 Get instance type (6.8.4 )

SdaiEntity sdaiGetInstanceType (SdaiInstance instance);

**Priority: 2**

**Input:**

`instance` - identifier of an instance

**Return:** identifier of an entity definition

It should be possible to just decompose the `SdaiId` into its constituents one of which is the entity definition number and then regenerate this into a valid `SdaiId`.

### 8.40 Is instance of (6.8.5 )

SdaiBoolean sdaiIsInstanceOf (SdaiInstance instance, SdaiEntity entity);

SdaiBoolean sdaiIsInstanceOfBN (SdaiInstance instance, SdaiString entityName);

**Priority: 2**

**Input:**

`instance` - identifier of an instance

`entity` - identifier of an entity definition

`entityName` - identifier of an entity name

**Return:**

`sdaiTRUE` - if the instance is of the given type

`sdaiFALSE` - if the instance is not of the given type

Again this can be generated from the `SdaiId` and no access is necessary to the DEVA database.

### 8.41 Is kind of (6.8.6 )

**Priority: 4**

This is equivalent to 6.8.5 since subtypes are not implemented in DEVA.

### 8.42 Is SDAI kind of (6.8.7 )

**Priority: 4**

This is equivalent to 6.8.5 since subtypes are not implemented in DEVA.

### 8.43 Find entity instance users (6.8.8 )

SdaiNPL sdaiFindInstanceUsers (SdaiInstance instance, SdaiNPL domain, SdaiNPL resultList);

**Priority: 3**

**Input:**

`instance` - identifier of the entity instance whose users are requested

`domain` - identifier of a NPL containing the `SdaiSchemaInstance` schema instances which define the domain of the function request

`resultList` - identifier of the pre-existing NPL to which the `SdaiInstance` instance identifiers for those entity instances are added meeting the specified criteria.

**Return:** identifier of the result NPL

## 8.44 Get attributes (6.8.9.1 )

void sdaiGetAttrs (SdaiInstance instance, SdaiInteger numberAttr, SdaiAttr attribute, SdaiPrimitiveType valueType, void *value,...);

void sdaiGetAttrsBN (SdaiInstance instance, SdaiInteger numberAttr, SdaiString attributeName, SdaiPrimitiveType valueType, void *value,...);

**Priority: 2**

**Input:**

`instance` - identifier of the instance whose attributes are being retrieved.

`numberAttr` - number of attributes to be retrieved and indirectly number of arguments specified in call

`attribute` - the attribute definition from the meta-data schema of the attribute to be retrieved

`attributeName` - the name of the attribute to be retrieved

`valueType` - the type of the read attribute

`value` - handle matching or convertible to the `valueType`

**Output:**

`value` - these functions shall return the value argument(s) filled with the primitive or identifier attribute value retrieved from the instance.

If the `valueType` is of `sdaiADB`, `sdaiAGGR` or `sdaiINSTANCE`, an identifier shall be returned that may be used in subsequent C late binding functions to examine their contents. The input parameters `attribute`, `valueType` and `value` shall be repeated in the given order as often as the value of `numberAttr` says.

## 8.45 Get all attributes (6.8.9.2 )

SdaiADB *sdaiGetAllAttrs (SdaiInstance instance, SdaiInteger *numberAttr);

**Priority: 2**

**Input:**

`instance` - identifier of the instance whose attributes are being retrieved

**Output:**

`numberAttr` - number of attribute values returned.

**Return:** This function shall return an C array containing identifiers of internally created `SdaiADBs` with the value arguments filled with the primitive or identifier attribute values retrieved from the instance. If the attribute is an aggregate or instance, an identifier shall be put into the ADB that may be used in subsequent C late binding functions to examine their contents.

The SdaiADB identifiers returned may be used in subsequent C late binding functions to examine their contents. The array returned contains all of the attribute values in the order defined in ISO 10303-21 [4].

This cannot be implemented by using deva_get_entity as the parameter list for deva_get_entity is dependent upon the entity type. This needs to be present at compile-time but the information about the entity type is not present until run-time as sdaiGetAllAttrs is a generic function for all entities. The alternative is to implement is by using a new DEVA function to get each attribute one at a time. This function will be written for the implementation of sdaiGetAttr and will then be used to implement sdaiGetAllAttrs as a next priority.

### 8.46 Copy application instance in same SDAI-model (6.9.1 )

**Priority: 4**

### 8.47 Copy application instance to other SDAI-model (6.9.2 )

**Priority: 4**

### 8.48 Delete application instance (6.9.3 )

void sdaiDeleteInstance (SdaiAppInstance instance);

**Priority: 1**

**Input:**

instance - identifier of the instance to be deleted

This will be implemented using deva_del_entity.

### 8.49 Put attribute (6.9.4 )

void sdaiPutAttr (SdaiAppInstance instance, SdaiExplicitAttr attribute, SdaiPrimitiveType valueType,...);

void sdaiPutAttrBN (SdaiAppInstance instance, SdaiString attributeName, SdaiPrimitiveType valueType,...);

**Priority: 2**

**Input:**

instance - identifier of the application instance

attribute - an attribute definition from the meta-data schema

attributeName - the name of the attribute to be set

valueType - the type of the attribute to be put

... - value of SdaiInteger, SdaiReal, SdaiBoolean, SdaiLogical type or handle matching of convertible to the valueType and given as a function parameter with the specified C late binding type.

**Status:** Implemented using new DEVA routine, deva_get_attribute. Tested and working for integers, reals, strings, enumerations, entity references and select references.

It is not necessary to use this routine for arrays as the routine `sdaiCreateAggr` associates the aggregate with an entity instance.

### 8.50 Unset attribute value (6.9.5 )

**Priority: 4**

### 8.51 Create aggregate instance (6.9.6 )

This function creates an empty aggregate-valued attribute for an entity instance. It creates an aggregate instance identifier associated with the specified attribute of the specified instance. If the aggregate-valued attribute had already been created (via a precious Create Aggregate Instance operation) the existing contents are lost.

SdaiAggr sdaiCreateAggr (SdaiAppInstance instance, SdaiExplicitAttr attribute);

SdaiAggr sdaiCreateAggrBN (SdaiAppInstance instance, SdaiString attributeName);

**Priority: 1**

**Input:**

`instance` - identifier of an application instance

`attribute` - identifier of the aggregate valued attribute definition of the instance

`attributeName` - name of the attribute in the entity definition

**Return:** identifier of the new aggregate

**Status:** Implemented and tested.

This routine creates an aggregate identifier which is associated with the specified entity instance.

### 8.52 Get persistent label (6.9.7 )

**Priority: 4**

### 8.53 Get session identifier (6.9.8 )

**Priority: 4**

### 8.54 Put attributes (6.9.19.1 )

void sdaiPutAttrs (SdaiAppInstance appInstance, SdaiInteger numberAttr, SdaiAttr attribute, SdaiPrimitiveType valueType,...);

void sdaiPutAttrsBN (SdaiAppInstance appInstance, SdaiInteger numberAttr, SdaiString attributeName, SdaiPrimitiveType valueType,...);

**Priority: 3**

**Input:**

`appInstance` - identifier of the application instance whose attributes are to be set

`numberAttr` - number of attributes to be set and indirectly number of arguments specified in the call

`attribute` - an attribute definition from the meta-data schema of the attribute to be set

`attributeName` - the name of the attribute to be set

`valueType` - type of the attribute to be set

`...` - value of `SdaiInteger`, `SdaiReal`, `SdaiBoolean`, `SdaiLogical` type or handle matching or convertible to the `valueType`, and given as a function parameter with the specified C late binding type.

The input parameters `attribute` (or `attributeName`),`valueType` and the value parameter shall be repeated in the given order as often as the value of the `numberAttr` says.

## 8.55 Put all attributes (6.9.19.2 )

void sdaiPutAllAttrs (SdaiAppInstance appInstance, SdaiInteger numberAttr, SdaiADB *values);

**Priority: 2**

**Input:**

`appInstance` - identifier of the application instance whose attributes are being set

`numberAttr` - number of attribute values are being provided, determines the length of the values array

`values` - an array of C structures `SdaiADB` containing the types and values of the attributes in the order defined in ISO 10303-21.

This cannot be implemented with `deva_put_entity` as the variable parameter list cannot be generated at run-time. It will be implemented on top of the new DEVA routine `deva_put_attribute`.

## 8.56 Get member count (6.10.1 )

SdaiInteger sdaiGetMemberCount (SdaiAggr aggregate);

**Priority: 2**

**Input:**

`aggregate` - identifier of an aggregate

**Return:** the number of elements or the size of an array

## 8.57 Is member (6.10.2 )

**Priority: 4**

### 8.58 Create iterator (6.10.3 )

**Priority: 4**

### 8.59 Delete iterator (6.10.4 )

**Priority: 4**

### 8.60 Beginning (6.10.5 )

**Priority: 4**

### 8.61 Next (6.10.6 )

**Priority: 4**

### 8.62 Get current member by iterator (6.10.7 )

**Priority: 4**

### 8.63 Application instance aggregate operations (6.11 )

**Priority: 4**

These all use iterators

### 8.64 Application instance unordered collection operations (6.12 )

**Priority: 4**

An unordered collection is a SET or a BAG which are not implemented in DEVA.

### 8.65 Indexed get aggregate member (6.13.1 )

void* sdaiGetAggrByIndex (SdaiOrderedAggr aggregate, SdaiAggrIndex index, SdaiPrimitiveType valueType, void *value);

**Priority: 1**

**Input:**

`aggregate` - identifier of an ordered aggregate

`index` - position in the aggregate of the value to be returned

`valueType` - type of the value to be read

`value` - handle matching or convertible to the `valueType`

**Output:**

`value` - handle filled with the primitive or identifier value retrieved from the aggregate. If the `valueType` is of `sdaiADB`, `sdaiAGGR`, or `sdaiINSTANCE`, an identifier shall be returned that may be used in subsequent C late binding functions to examine their contents.

**Status:** Implemented and tested for integers, reals and entity references. These are the only types of array values in the MIDAS data model.

This routine has been implemented by getting the whole array from DEVA the first time the array is accessed. The array is placed in memory. If the next access is to the same array then it is obtained from memory. If the next access is to a different array then the memory is freed and the required array is retrieved from the database.

### 8.66 End (6.13.2 )

**Priority: 4**

### 8.67 Previous (6.13.3 )

**Priority: 4**

### 8.68 Indexed put aggregate member (6.14.1 )

void sdaiPutAggrByIndex (SdaiOrderedAggr aggregate, SdaiAggrIndex index, SdaiPrimitiveType valueType, ...);

**Priority: 1**

**Input:**

`aggregate` - identifier of an aggregate

`index` - position in the aggregate of the value to be set

`valueType` - Type of the value to be set

`...` - value or handle matching or convertible to the `valueType`

**Status:** Implemented and tested. This routine works for integers, reals and entity references which are the only types of arrays present in the MIDAS data model.

This routine has been implemented by creating a space in memory for the whole array when it is called for the first time. If the next call to the routine is to put another value into the same array then this value is placed into memory. If the next call is related to a different array then the previous array is transferred to the database, the memory is freed and memory obtained for the new array. The array is also transferred to the database when the array is accessed by `sdaiGetAggrByIndex` or when the access to the SDAI-model containing the array is stopped by a call to `sdaiEndModelAccess`.

### 8.69 Indexed create nested aggregate instance (6.14.2 )

SdaiAggr sdaiCreateNestedAggrByIndex (SdaiOrderedAggr aggregate, SdaiAggrIndex index);

**Priority: 3**

**Input:**

`aggregate` - identifier of a multiple dimensioned aggregate

`index` - the index at where the new aggregate will be inserted

### 8.70 Entity instance array operations (6.15 )

**Priority: 4**

### 8.71 Application instance array operations (6.16 )

**Priority: 2**

### 8.72 Application instance list operations (6.17 )

**Priority: 3**

### 8.73 C late binding specific type path operations (6.18 )

**Priority: 4**

## References


[1] MIDAS.RAL.95.1 "Data Modelling for Electromagnetic and Stress Analysis Integration", Mrs D Thomas and Dr C Greenough, 30 March 1995.

[2] MIDAS.RAL.95.2 "The MIDAS Data Model for Electromagnetic and Stress Analysis Integration, Version 1.2", Mrs D Thomas and Dr C Greenough, 30 March 1995.

[3] ISO 10303-11, "Part 11: Description methods:*EXPRESS* Language reference manual".

[4] ISO 10303-21, "Part 21: Implementation Methods: Clear text encoding of the exchange structure",

[5] ISO TC184/SC4/WG7 N394, "Product data representation and exchange, Part24, Standard data access interface - C language late binding", 28 July 1995.

[6] ISO TC184/SC4/WG7 N382, "Product data representation and exchange, Part22, Standard Data Access Interface", 31 May 1995.

# A Implementation classes

This table describes the operations required for each implementation class defined in the STEP SDAI [6]. The operations are named and numbered according to the SDAI definitions in Part 22 not the C late binding functions defined in Part 24.

| Operation | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 11.3.1 Open Session | Y | Y | Y | Y | Y | Y |
| 11.4.1 Start event recording | N | N | N | Y | Y | Y |
| 11.4.2 Stop event recording | N | N | N | Y | Y | Y |
| 11.4.3 Close session | Y | Y | Y | Y | Y | Y |
| 11.4.4 Open repository | Y | Y | Y | Y | Y | Y |
| 11.4.5 Start transaction read-write access | N | N | N | N | Y | Y |
| 11.4.6 Start transaction read-only access | N | N | N | N | Y | Y |
| 11.4.7 Commit | N | N | N | N | Y | Y |
| 11.4.8 Abort | N | N | N | N | Y | Y |
| 11.4.9 End transaction access and commit | N | N | N | N | Y | Y |
| 11.4.10 End transaction access and abort | N | N | N | N | Y | Y |
| 11.4.11 Create non-persistent list | Y | Y | Y | Y | Y | Y |
| 11.4.12 Is non-persistent list of | Y | Y | Y | Y | Y | Y |
| 11.4.13 SDAI-query | ? | | | | | |
| 11.4.14 Record event | N | N | N | Y | Y | Y |
| 11.5.1 Create schema instance | Y | Y | Y | Y | Y | Y |
| 11.5.2 Add SDAI-model | P | P | Y | P | Y | Y |
| 11.5.3 Remove SDAI-model | Y | Y | Y | Y | Y | Y |
| 11.5.4 Delete schema instance | Y | Y | Y | Y | Y | Y |
| 11.5.5 Validate global rule | N | P | P | P | P | Y |
| 11.5.6 Validate uniqueness rule | N | Y | Y | Y | Y | Y |
| 11.5.7 Validate instance reference domain | N | Y | Y | Y | Y | Y |
| 11.5.1-11.5.2 Repository operations | Y | Y | Y | Y | Y | Y |
| 11.6.1-11.6.9 SDAI-model operations | Y | Y | Y | Y | Y | Y |
| 11.6.10-11.6.11 Save and Undo | N | N | N | Y | N | N |
| 11.7.1-11.7.10 Scope operations | N | Y | Y | N | N | Y |
| 11.8.1 Get Complex Entity | N | ? | ? | ? | ? | Y |
| 11.8.2 Is subtype of | Y | Y | Y | Y | Y | Y |
| 11.8.3 Is SDAI subtype of | Y | Y | Y | Y | Y | Y |
| 11.8.4 Is interoperable with | N | N | Y | N | Y | Y |
| 11.9.1 Get Attribute | P | P | P | P | P | Y |
| 11.9.2-11.9.7 Entity instance operations | Y | Y | Y | Y | Y | Y |
| 11.10.1-11.10.6 Application instance op.s | Y | Y | Y | Y | Y | Y |
| 11.10.7-11.10.16 " validation | N | P | P | P | P | Y |
| 11.11.1-11.18.4 Aggregate operations | Y | Y | Y | Y | Y | Y |