INTERNATIONAL STANDARD

ISO 10303-23

First edition
2000-12-01

Industrial automation systems and integration — Product data representation and exchange —

Part 23: Implementation methods: C++ language binding to the standard data access interface

Systèmes d'automatisation industrielle et intégration — Représentation et échange de données de produits —

Partie 23: Méthodes de mise en application: Liant de langage C++ à l'interface d'accès aux données normalisées

Reference number
ISO 10303-23:2000(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.ch
Web www.iso.ch

Printed in Switzerland

Contents

Page

Figures

Tables

Foreword

ISO (the 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 the 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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

Draft International Standards adopted by the 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.

Attention is drawn to the possibility that some of the elements of this part of ISO 10303 may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

International Standard ISO 10303-23 was prepared by Technical Committee ISO/TC 184, *Industrial automation and systems and integration*, Subcommittee SC 4, *Industrial data*.

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;

— Part 11, Description methods: The EXPRESS language reference manual;

— Part 12, Description method: The EXPRESS-I language reference manual;

— Part 21, Implementation methods: Clear text encoding of the exchange structure;

— Part 22, Implementation method: Standard data access interface specification;

— Part 23, Implementation method: C++ language binding to the standard data access interface;

— Part 24, Implementation method: C language binding to the standard data access interface;

— Part 26, Implementation method: Interface definition language binding to the standard data access;

— Part 31, Conformance testing methodology and framework: General concepts;

— Part 32, Conformance testing methodology and framework: Requirements on testing laboratories and clients;

— Part 34, Conformance testing methodology and framework: Abstract test methods;

— Part 35, Conformance testing methodology and framework: Abstract test methods for SDAI implementations;

— Part 41, Integrated generic resources: Fundamentals of product description and support;

— Part 42, Integrated generic resources: Geometric and topological representation;

— Part 43, Integrated generic resources: Representation structures;

— Part 44, Integrated generic resources: Product structure configuration;

— Part 45, Integrated generic resource: Materials;

— Part 46, Integrated generic resources: Visual presentation;

— Part 47, Integrated generic resource: Shape variation tolerances;

— Part 49, Integrated generic resource: Process structure and properties;

— Part 101, Integrated application resource: Draughting;

— Part 104, Integrated application resource: Finite element analysis;

— Part 105, Integrated application resource: Kinematics;

— Part 106, Integrated application resource: Building construction core model;

— Part 107, Engineering Analysis Core Application reference model (EA C-ARM);

— Part 201, Application protocol: Explicit draughting;

— Part 202, Application protocol: Associative draughting;

— Part 203, Application protocol: Configuration controlled design;

— Part 204, Application protocol: Mechanical design using boundary representation;

— Part 205, Application protocol: Mechanical design using surface representation;

— Part 207, Application protocol: Sheet metal die planning and design;

— Part 208, Application protocol: Life cycle management - Change process;

— Part 209, Application protocol: Composite and metallic structural analysis and related design;

— Part 210, Application protocol: Electronic assembly, interconnect, and packaging design;

— Part 212, Application protocol: Electrotechnical design and installation

— Part 213, Application protocol: Numerical control process plans for machined parts;

— Part 214, Application protocol: Core data for automotive design;

— Part 215, Application protocol: Ship arrangement;

— Part 216, Application protocol: Ship moulded forms;

— Part 217, Application protocol: Ship piping;

— Part 218, Application protocol: Ship structures;

— Part 221, Application protocol: Functional data and their schematic representation for process plant;

— Part 222, Application protocol: Exchange of product data for composite structures;

— Part 223, Application protocol: Exchange of design and manufacturing product information for casting parts;

— Part 224, Application protocol: Mechanical product definition for process plans using machining features;

— Part 225, Application protocol: Building elements using explicit shape representation;

— Part 226, Application protocol: Ship mechanical systems;

— Part 227, Application protocol: Plant spatial configuration;

— Part 229, Application protocol: Exchange of design and manufacturing product information for forged parts;

— Part 230, Application protocol: Building structural frame: Steelwork;

— Part 231, Application protocol: Process engineering data: Process design and process specification of major equipment;

— Part 232, Application protocol: Technical data packaging core information and exchange;

— Part 301, Abstract test suite: Explicit draughting;

— Part 302, Abstract test suite: Associative draughting;

— Part 303, Abstract test suite: Configuration controlled design;

— Part 304, Abstract test suite: Mechanical design using boundary representation;

— Part 305, Abstract test suite: Mechanical design using surface representation;

— Part 307, Abstract test suite: Sheet metal die planning and design;

— Part 308, Abstract test suite: Life cycle management - Change process;

— Part 309, Abstract test suite: Composite and metallic structural analysis and related design;

— Part 310, Abstract test suite: Electronic assembly, interconnect, and packaging design;

— Part 312, Abstract test suite: Electrotechnical design and installation;

— Part 313, Abstract test suite: Numerical control process plans for machined parts;

— Part 314, Abstract test suite: Core data for automotive mechanical design;

— Part 315, Abstract test suite: Ship arrangement;

— Part 316, Abstract test suite: Ship moulded forms;

— Part 317, Abstract test suite: Ship piping;

— Part 318, Abstract test suite: Ship structures;

— Part 321, Abstract test suite: Functional data and their schematic representation for process plant;

— Part 322, Abstract test suite: Exchange of product data for composite structures;

— Part 323, Abstract test suite: Exchange of design and manufacturing product information for casting parts;

— Part 324, Abstract test suite: Mechanical product definition for process plans using machining features;

— Part 325, Abstract test suite: Building elements using explicit shape representation;

— Part 326, Abstract test suite: Ship mechanical systems;

— Part 327, Abstract test suite: Plant spatial configuration;

— Part 329, Abstract test suite: Exchange of design and manufacturing product information for forged parts;

— Part 330, Abstract test suite: Building structural frame: Steelwork;

— Part 331, Abstract test suite: Process engineering data: Process design and process specification of major equipment;

— Part 332, Abstract test suite: Technical data packaging core information and exchange;

— Part 501, Application interpreted construct: Edge-based wireframe;

— Part 502, Application interpreted construct: Shell-based wireframe;

— Part 503, Application interpreted construct: Geometrically bounded 2D wireframe;

— Part 504, Application interpreted construct: Draughting annotation;

— Part 505, Application interpreted construct: Drawing structure and administration;

— Part 506, Application interpreted construct: Draughting elements;

— Part 507, Application interpreted construct: Geometrically bounded surface;

— Part 508, Application interpreted construct: Non-manifold surface;

— Part 509, Application interpreted construct: Manifold surface;

— Part 510, Application interpreted construct: Geometrically bounded wireframe;

— Part 511, Application interpreted construct: Topologically bounded surface;

— Part 512, Application interpreted construct: Faceted boundary representation;

— Part 513, Application interpreted construct: Elementary boundary representation;

— Part 514, Application interpreted construct: Advanced boundary representation;

— Part 515, Application interpreted construct: Constructive solid geometry;

— Part 517, Application interpreted construct: Mechanical design geometric presentation;

— Part 518, Application interpreted construct: Mechanical design shaded representation;

— Part 519, Application interpreted construct: Geometric tolerances;

— Part 520, Application interpreted construct: Associative draughting.

The structure of this International Standard is described in ISO 10303-1. The numbering of the parts of the International Standard reflects its structure:

— Parts 11 to 12 specify the description methods,

— Parts 21 to 26 specify the implementation methods,

— Parts 31 to 35 specify the conformance testing methodology and framework,

— Parts 41 to 49 specify the integrated generic resources,

— Parts 101 to 107 specify the integrated application resources,

— Parts 201 to 232 specify the application protocols,

— Parts 301 to 332 specify the abstract test suites, and

— Parts 501 to 520 specify the application interpreted constructs.

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

Annex A forms a normative 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.

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

This part of ISO 10303 specifies a C++ language binding to the standard data access interface (SDAI). SDAI is the access interface specification to data which has been defined using ISO 10303-11. SDAI is specified in ISO 10303-22.

Computer application systems are implemented using computing or definition languages. The specification of the functionality defined in ISO 10303-22 in a particular computing or definition language is referred to as an SDAI language binding. Since there are many computing and definition languages, many SDAI language bindings are possible.

Readers of this part of ISO 10303 are advised that this part of ISO 10303 presupposes a working knowledge of the functionality of the SDAI which is described in ISO 10303-22. Moreover, this part of ISO 10303 presupposes a working knowledge of the C++ programming language. The source for the C++ programming language is specified in the normative references of this part of ISO 10303.

Major subdivisions of this part of ISO 10303 are:

— language binding characteristics, constants and data types found in clauses 4 and 5;

— language bindings to SDAI operations and error handling found in clauses 6 and 7.

Industrial automation systems and integration — Product data representation and exchange —

Part 23:
Implementation methods: C++ language binding to the standard data access interface

1 Scope

This part of ISO 10303 specifies the implementation of the functional interface specified in the standard data access interface (SDAI), ISO 10303-22, in the C++ programming language.

The following are within the scope of this part of ISO 10303:

— access to and manipulation of data types and entities which are specified in ISO 10303-22;

— convenience functions suitable to this language binding;

— binding of functions to operations and attributes specified in ISO 10303-22 with the linking of application schema definition at either compile-time or run-time;

— implementation mechanisms for the handling of errors as specified in ISO 10303-22;

— implementation mechanisms for the validation of constraints as specified in ISO 10303-22.

The following is outside the scope of this part of ISO 10303:

— all items listed as out of scope in ISO 10303-22.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO 10303. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO 10303 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

Ellis, Margaret A. and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, New York, 1990.

IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985.

ISO/IEC 8824-1:1998, *Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation.*

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

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

ISO 10303-21:1994, *Industrial automation systems and integration - Product data representation and exchange - Part 21: Implementation methods: Clear text encoding of the exchange structure*.

ISO 10303-22:1998, *Industrial automation systems and integration - Product data representation and exchange - Part 22: Implementation methods: Standard data access interface*.

ISO 10303-31:1994, *Industrial automation systems and integration - Product data representation and exchange - Part 31: Conformance testing methodology and framework: General concepts*.

3 Definitions and abbreviations

For the purposes of this part of ISO 10303, the following definitions and naming conventions apply.

3.1 Definitions

3.1.1 Terms defined in ISO 10303-1

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

— application;

— application protocol;

— data;

— implementation method;

— information;

— information model;

— product information model;

— structure.

3.1.2 Terms defined in ISO 10303-11

This part of ISO 10303 makes use of the following terms defined in ISO 10303-11:

— complex entity data type;

— data type;

— entity;

— entity data type;

— entity instance;

— instance;

— population;

— value.

3.1.3 Terms defined in ISO 10303-22

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

— application schema;

— concurrent access;

— constraint;

— current schema;

— external schema;

— foreign schema;

— identifier;

— implementation class;

— iterator;

— native schema;

— repository;

— schema instance;

— SDAI-model;

— session;

— validation.

3.1.4 Terms defined in The Annotated C++ Reference Manual

This part of ISO 10303 makes use of the following terms defined in The Annotated C++ Reference Manual:

— base class;

— C++ programming language;

— class;

— derived class;

— lvalue;

— member function;

— placement;

— virtual base class;

— user-defined conversion operator.

3.1.5 Term defined in IEEE standard

This part of ISO 10303 makes use of the following term defined in the IEEE standard for binary floating-point arithmetic:

— NaN.

3.1.6 Other definitions

For the purpose of this part of ISO 10303, the following definitions apply.

3.1.6.1 accessor member function: a member function of a C++ class that is used to set, retrieve, unset or test an attribute of that class.

3.1.6.2 convenience function: a function provided by a language binding in order to provide a more convenient implementation.

3.1.6.3 copy constructor: a C++ constructor which makes a copy of an instance of the same class.

3.1.6.4 deep copy: a copy of an instance of a class in which member objects pointed to are recursively copied.

3.1.6.5 default constructor: any C++ constructor which takes no arguments.

3.1.6.6 handle: an abstract data type which consists of a reference to a C++ class. A handle to class type <T> may hold a null reference (i.e. a reference to no instance) or a reference to an instance of class <T> or a reference to an instance of a subclass of type <T>.

3.1.6.7 narrow: to assign an object reference of a given class to an object reference of a derived class of the given class.

3.1.6.8 primitive: a term used to describe the following EXPRESS data types, INTEGER, REAL, LOGICAL, BOOLEAN, STRING, BINARY, and their corresponding C++ data types.

3.1.6.9 receiver: an instance upon which a specified member function is invoked.

3.1.6.10 release: to indicate to the SDAI implementation that an object will not be used again.

3.1.6.11 SDAI operation: an operation defined in clause 10 of ISO 10303-22.

3.1.6.12 shallow copy: a copy of an instance of a class in which member objects pointed to are not copied.

3.1.6.13 widen: to assign an object reference of a given class to an object reference of a base class of the given class.

3.2 Abbreviations

For the purposes of this part of ISO 10303, the following abbreviations apply.

CPP	C++
IDL	Interface Definition Language
Repo	Repository
RO	Read-Only
RW	Read-Write
SDAI	Standard Data Access Interface

3.3 Typographical conventions

For the purposes of this part of ISO 10303, C++ language statements shall be presented as 10 point Courier font. The C++ language statements are presented as a syntactically complete mapping of SDAI elements. However, this part of ISO 10303 is not designed to be a compilable document. C++ language statements specified in this part of ISO 10303 designate required functionality and should not be interpreted as restricting implementation techniques. C++ language statements are generally separated from text; however this is only for the purpose of clarity. The notation “<T>” shall be used to represent an arbitrary data type “T”.

EXAMPLE - 1 This example illustrates a C++ language statement in 10 point Courier font.

```
class foo { };
```

4 Overall requirements

4.1 Characteristics of the binding functions

If a parameter to a function has an invalid value, the behaviour is not defined in this part of ISO 10303.

EXAMPLE - 2 This example describes possible invalid values.

An invalid value may correspond to values such as a value outside the domain of the function, a pointer outside the address space of the program, a null pointer, or an index out of range for an aggregate.

The SDAI implementation need not implement the type declarations given in this part of ISO 10303 identically. However, the SDAI implementation shall present types which shall be equivalent, from the point of view of the user of the type, with the types prescribed in this part of ISO 10303.

For each function specification in this part of ISO 10303 which maps to an SDAI operation, the correspondence between the operation's SDAI input/output, as specified in ISO 10303-22, and its C++ language equivalent are provided.

Table 1 - Arguments and return values

SDAI/EXPRESS/IDL	In argument	Out argument	Return value
INTEGER	Integer	Integer&	Integer
REAL	Real	Real&	Real
BOOLEAN	sdaiBoolean	sdaiBoolean&	sdaiBoolean
LOGICAL	sdaiLogical	sdaiLogical&	sdaiLogical
ENUMERATION	<Enum> where <Enum> is the name of an enumeration	<Enum>&	<Enum>
STRING	const char*	char*&	char*
Aggregate <S>, BINARY	const S_var&	S_var&	S_var
SELECT <S>	const S_var&	S_var&	S_var
ANY	const Any_var&	Any_var&	Any_var
ENTITY <E>	const E_ptr&	E_ptr&	E_ptr

Arguments to functions are classified as 'in' arguments and 'out' arguments. 'In' arguments are read-only with respect to the function; 'out' arguments may be read-write with respect to the function. Except where otherwise noted, the data types which are used for arguments and return values are given in table 1.

EXPRESS TYPE types are treated according to the method prescribed for the underlying type.

4.2 Language specific features

4.2.1 Names of types and operations

Except where otherwise noted, the bindings of the names of operations and types to names in the C++ SDAI implementation shall be accomplished as follows. EXPRESS names for data types including SELECT data types, ENUMERATION data types, ENTITY data types, aggregate data types, and defined types shall be mapped to C++ names as follows: the C++ name shall be exactly the same as the EXPRESS name except that the first character of the C++ name shall be uppercase and all other characters shall be lowercase. This mapping shall also apply to EXPRESS schema names. The mapping of EXPRESS attribute names shall be exactly the same as the EXPRESS name except that all characters shall be lowercase and the name shall be followed by an underscore ("_"). In the case where the EXPRESS ENTITY data type <T> ends with the suffix "_ptr" or "_var", the first letter of the suffix shall be uppercase in the corresponding C++ class name.

EXAMPLE 3 - This example illustrates a mapping from an EXPRESS type name to a C++ type name:

in EXPRESS:
`tHis_IS_a_sAMPLE_Name`
in C++:
`This_is_a_sample_name`

in EXPRESS:
`special_case_var`
in C++:
`Special_case_Var`

4.2.2 Memory management

This part of ISO 10303 does not specify the implementation of memory management in an SDAI implementation. The term 'release' is defined with respect to an object's handle or containing aggregate. When an object is released, either explicitly or implicitly, through its handle or containing aggregate, the SDAI implementation may behave as if the object will no longer be addressed through that handle or aggregate. Beyond any specified errors, the behaviour of operations on released objects is not specified in this part of ISO 10303.

4.3 Binding styles

This document describes three binding styles for a C++ language binding to the SDAI: a) early binding only, b) late binding only, and c) mixed binding.

Early binding functionality is to be provided by translating application schemas into the C++ classes specified in this binding and linking the resulting code into the SDAI implementation. This instantiation shall also include schema-dependent classes for `Schema_instance`, `Model_contents`, and `Application_instance`. The portions of the binding designated by the following late binding compiler directive:

```
#define SDAI_CPP_LATE_BINDING
```

need not be included in the early bound SDAI implementation. The classes based on the `SDAI_dictionary_schema` in ISO 10303-22 need not be instantiated.

Late binding functionality shall be provided by implementing the functions designated in this binding by the late-binding compiler directive. Dictionary classes shall be instantiated. Application schemas need not be translated into classes and linked into the SDAI implementation. Schema-dependent classes for `Schema_instance`, `Model_contents`, and `Application_instance` need not be provided.

Mixed binding functionality shall be provided by translating application schemas to the C++ classes specified in this binding and linking the resulting code into the SDAI implementation. Schema-dependent class for `Schema_instance`, `Model_contents`, and `Application_instance` shall be provided. The functionality designated by the late-binding compiler directive shall be included in the SDAI implementation. Dictionary classes shall be instantiated.

5 Constants and data type definitions and global functions

5.1 SDAI namespace

For implementations where `namespace` is available, all declarations of C++ classes and types described in this part of ISO 10303 shall be enclosed in an outermost C++ language `namespace` named `SDAI`. For implementations where `namespace` is not available, the name scoping provided by:

```
namespace SDAI {
  class T { };
};
```

may be provided by

```
class SDAI_T { };
```

Each application schema shall be mapped to a C++ `namespace` whose name shall be the name of the EXPRESS SCHEMA mapped according to 4.2.1. All declarations of C++ classes and types derived from the EXPRESS application SCHEMA shall be contained in this `namespace`. The application schema `namespace` shall not be nested in the `SDAI namespace`.

If the implementation supports only a single application protocol, and therefore, does not require `namespace`, it need not be used.

5.2 EXPRESS built-in constants

The EXPRESS built-in constants for attribute values shall be represented by the C++ language constants which are given in table 2.

Table 2 - EXPRESS built-in constants

EXPRESS	Constant name	C++ data type
FALSE	SDAI::sdaiFALSE	SDAI::sdaiBOOL
TRUE	SDAI::sdaiTRUE	SDAI::sdaiBOOL
UNKNOWN	SDAI::sdaiUNKNOWN	SDAI::sdaiLOGICAL
PI	SDAI::PI	SDAI::Real
CONST_E	SDAI::E	SDAI::Real

5.3 EXPRESS attribute data types

The data types of EXPRESS shall be represented by C++ data types as described below.

5.3.1 Primitive data types

The `SDAI` namespace shall contain the following C++ types for the primitive data types.

5.3.1.1 Integer

The following C++ types shall represent the integer values:

Type Name	Range
Integer	-2 ** 31 to 2 ** 31 -1
ULong	0 to 2 ** 32 -1
Short	-2 ** 15 to 2 ** 15 -1
UShort	0 to 2 ** 16 -1

There shall also be a type `Octet`, that represents an 8-bit quantity.

The EXPRESS INTEGER data type shall be represented by the C++ type called `Integer`. The value of 2 ** 31 - 1 shall designate an unset EXPRESS INTEGER. This representation does not prohibit the type `Integer` as a C++ class.

5.3.1.2 Real

The EXPRESS REAL data type shall be represented by a C++ type called `Real`. The C++ type `Real` shall be equivalent in functionality to the IEEE double precision floating point number. The value of NaN shall designate an unset EXPRESS REAL. This representation does not prohibit the definition of the type `Real` as a C++ class.

NOTE - To determine whether a Real number is equal to NaN, a method which looks like the following may be implemented:

```
int isNaN(double d) { return (d != d); }
```

5.3.1.3 Logical

The EXPRESS LOGICAL data type shall be represented by the following C++ enumeration:

```
enum sdaiLogical  { LFalse, LTrue, LUnset, LUnknown };
```

The enumerators of the enum sdaiLogical shall correspond to the states EXPRESS FALSE, EXPRESS TRUE, unset, and EXPRESS UNKNOWN respectively.

The C++ class `sdaiLOGICAL` shall implement the behaviour of the EXPRESS LOGICAL type, including support for the relational operators. The following class definition specifies the behaviour of the `LOGICAL` type:

```
class sdaiBOOL;
class sdaiLOGICAL {
public:
sdaiLOGICAL();
```

The default constructor shall create an instance whose value is unset.

```
sdaiLOGICAL(sdaiLogical state);
```

This function shall construct an instance whose state is specified by the argument `state`.

```
sdaiLOGICAL(const sdaiLOGICAL& source);
```

This function shall construct a copy from the `sdaiLOGICAL source`.

```
sdaiLOGICAL(int I);
```

This function shall construct an instance of `sdaiLOGICAL` from an `int` argument. If the value of `I` is 0, the value of the new instance shall be FALSE; otherwise, the value of the instance shall be TRUE.

```
sdaiLOGICAL(const sdaiBOOL& boo);
```

This function shall construct an instance of `sdaiLOGICAL` from an instance of `sdaiBOOL`. If the value

of boo is SDAI::sdaiTRUE, the new value shall be LTrue; if the value of boo is SDAI::sdaiFALSE, the new value shall be LFalse; if the value of boo is unset, the new value shall be LUnset.

```
operator int() const;
```

This operator function shall convert the receiver to a C++ int. If the value of the receiver is FALSE, the function shall return 0, otherwise the function shall return 1. If the value of the receiver is unset, the function shall set the error code:

```
sdaiVA_NSET             // value unset
```

If the value of the receiver is UNKNOWN, the function shall set the error code:

```
sdaiVT_NVLD             // invalid value type
```

```
operator sdaiLogical() const;
```

This operator shall convert from an instance of class sdaiLOGICAL to an instance of enum sdaiLogical.

```
sdaiLOGICAL& operator=(const sdaiLOGICAL& t);
```

This function shall assign a value for a variable of type sdaiLOGICAL or sdaiBOOL or int to the receiver. For an *int* argument, the function shall follow the rules of the int user-defined conversion method.

```
sdaiLOGICAL operator==(const sdaiLOGICAL& t) const;
sdaiLOGICAL operator!=(const sdaiLOGICAL& t) const;
sdaiLOGICAL operator< (const sdaiLOGICAL& t) const;
sdaiLOGICAL operator> (const sdaiLOGICAL& t) const;
sdaiLOGICAL operator<=(const sdaiLOGICAL& t) const;
sdaiLOGICAL operator>=(const sdaiLOGICAL& t) const;
```

These relational operators shall obey the rules for the comparison of EXPRESS LOGICAL types as defined in 8.1.4 of ISO 10303-11. If either operand is unset, the return value shall be UNKNOWN.

```
sdaiLOGICAL operator&&(const sdaiLOGICAL& t) const; // EXPRESS AND operator
sdaiLOGICAL operator||(const sdaiLOGICAL& t) const; // EXPRESS OR operator
sdaiLOGICAL operator^(const sdaiLOGICAL& t) const; // EXPRESS XOR operator
sdaiLOGICAL operator!() const; // EXPRESS NOT operator
```

These logical operators shall obey the rules for the logical operators as defined in 12.4 of ISO 10303-11. If either operand is unset, the return value shall be UNKNOWN.

```
int exists() const;
```

This function shall test whether the receiver has been set. If the receiver is unset, it shall return 0; otherwise, it shall return 1.

```
void nullify();
```

This function shall change the receiver to an unset state.

```
}; // end of class definition for sdaiLOGICAL
const sdaiLOGICAL UNKNOWN(LUnknown);
```

This declaration shall create the global constant UNKNOWN instance.

5.3.1.4 Boolean

The EXPRESS BOOLEAN data type shall be represented by the following C++ enumeration:

```
enum sdaiBool { BFalse, BTrue, BUnset };
```

The enumerators of the enum `sdaiBool` shall correspond to the states EXPRESS FALSE, EXPRESS TRUE, and unset respectively.

The C++ class `sdaiBOOL` shall implement the behaviour of the EXPRESS BOOLEAN type--including support for the relational operators. The following class definition specifies the behaviour of the `BOOLEAN` type:

```
class sdaiBOOL  {
public:
sdaiBOOL();
```

The default constructor shall create an instance whose state is unset.

```
sdaiBOOL(sdaiBool state);
```

This function shall construct an instance whose state is specified by the argument `state`.

```
sdaiBOOL(const sdaiBOOL& source);
```

This function shall construct a copy from the `sdaiBOOL`.

```
sdaiBOOL(int I);
```

This function shall construct an instance of `sdaiBOOL` from an `int` argument. If the value of `I` is 0, the value of the new instance shall be `BFalse`; otherwise, the value of the instance shall be `BTrue`.

```
sdaiBOOL(const sdaiLOGICAL& logical);
```

This function shall construct an instance of `sdaiBOOL` from a `sdaiLOGICAL` argument. If the value of `logical` is TRUE, the value of the new instance shall be TRUE. If the value of `logical` is FALSE, the value of the new instance shall be FALSE. If the value of `logical` is unset, the value of

the new instance is unset. If the value of `logical` is UNKNOWN, the value of the new instance is unset, and the function shall set the error code:

```
sdaiVT_NVLD             // Invalid value type.
```

```
operator int() const;
```

This function shall convert the receiver to a C++ `int`. If the value of the receiver is FALSE, the function shall return 0, otherwise the function shall return 1. If the value of the receiver is unset, the function shall set the error code:

```
sdaiVA_NSET                    // value unset
```

```
operator sdaiBool() const;
```

This operator shall convert an instance of class `sdai`BOOL to an instance of enum `sdai`Bool.

`sdai`BOOL& operator=(const sdaiLOGICAL& t);

This function shall assign a value for a variable of type `sdaiLOGICAL` or `sdai`BOOL or int to the receiver. For an `int` argument, the function shall follow the rules of the `int` user-defined conversion. If the argument is UNKNOWN, the function shall set the error code:

```
sdaiVT_NVLD             //Invalid value type.
```

```
sdaiLOGICAL operator==(const sdaiLOGICAL& t) const;
sdaiLOGICAL operator!=(const sdaiLOGICAL& t) const;
sdaiLOGICAL operator< (const sdaiLOGICAL& t) const;
sdaiLOGICAL operator> (const sdaiLOGICAL& t) const;
sdaiLOGICAL operator<= (const sdaiLOGICAL& t) const;
sdaiLOGICAL operator>= (const sdaiLOGICAL& t) const;
```

These relational operators shall obey the rules for the comparison of EXPRESS LOGICAL types as defined in 8.1.4 of ISO 10303-11. If either operand is unset, the return value shall be UNKNOWN.

```
// EXPRESS AND operator
 sdaiLOGICAL operator&&(const sdaiLOGICAL& t) const;
// EXPRESS OR operator
sdaiLOGICAL operator||(const sdaiLOGICAL& t) const;
// EXPRESS XOR operator
sdaiLOGICAL operator^(const sdaiLOGICAL& t) const;
// EXPRESS NOT operator
sdaiLOGICAL operator!() const;
```

These logical operators shall obey the rules for the logical operators as defined in 12.4 of ISO 10303-11. If either operand is unset, the return value shall be UNKNOWN.

```
int exists() const;
```

This function shall test whether the receiver has been set. If the receiver is unset, the function shall return 0; otherwise, the function shall return 1.

```
void nullify();
```

This function shall change the receiver to an unset status.

```
}; // end of class definition for sdaiBOOL
const sdaiBOOL FALSE(BFalse);
const sdaiBOOL TRUE(BTrue);
const sdaiBOOL UNSET(BUnset);
```

These declarations shall create the global constants `sdai`FALSE, `sdai`TRUE and `sdai`UNSET.

5.3.1.5 String

The EXPRESS STRING data type shall be represented by the C++ class `String_var`, which shall contain a `char*` value which shall point to a null-terminated string. The single character C++ string consisting of the character '\xFF' shall represent an unset string. The class definition shall be as follows:

```
class String_var {

public:
String_var();
```

This function shall create an instance which points to a C++ string whose `sizeof` is zero.

```
String_var(char* s);
```

This function shall create an instance which points to the `char*` referenced by `s`.

```
String_var(const String_var& s);
```

This function shall create an instance which points to a copy of the C++ string in `s`.

```
~String_var();
```

This function shall destroy the C++ string referenced by the receiver.

```
String_var &operator=(char* s);
```

This function shall release the C++ string referenced by the receiver, and assign the `char* s` to the receiver.

```
String_var &operator=(const String_var& s);
```

This function shall release the C++ string referenced by the receiver, and assign a copy of the C++ string `s` to the receiver.

```
operator char*();
```

This function shall return a pointer to the `char*` referenced by the receiver.

```
operator const char*() const;
```

This function shall return a read-only pointer to the `char*` referenced by the receiver.

```
char &operator[](ULong I);
```

This function shall return a reference to the character at position `I`. The behaviour is not specified in this part of ISO 10303 if `I` is out of range.

```
char operator[](ULong I) const;
};
```

This function shall return a copy of the character at position `I`. The behaviour is not specified in this part of ISO 10303 if `I` is out of range.

NOTE - For a description of how this class is used in CORBA implementations, see [1], section 16.7.

The C++ class `String` shall implement the EXPRESS STRING data type. The following class definition specifies the behaviour of the `String` type:

```
class String : public String_var {

public:
String();
```

This function shall create an instance of the class `String`. This class shall contain a C++ string whose `sizeof` is zero.

```
String(char* p);
```

This function shall create an instance of class `String`. This class shall contain the `char* p`.

```
String(const String& s);
```

This function shall create an instance of class `String` with the same C++ string value of `s`. The function shall perform a deep copy so that subsequent changes to the new instance shall not be reflected in the original String `s`.

```
~String();
```

This function shall destroy the receiver and the underlying C++ string.

```
String& operator=(char* p);
```

This operator shall first release any contained C++ string and then assign the `char* p` to the receiver. The receiver shall contain the `char* p`.

```
String& operator=(const String& s);
};
```

This operator shall first release any contained C++ string and then assign a copy of the C++ string contained in `s` to the receiver. Subsequent changes to the receiver shall not be reflected in the original String `s`.

5.3.1.6 Binary

The EXPRESS BINARY data type shall be represented by a C++ class called `Binary`. A `Binary` whose first octet value is zero shall represent an unset `Binary`. If the first octet value is not zero, it shall signify the number of bits in the last octet which contain data for the `Binary`. The maximum size of a instance of `Binary` is the maximum number of octets possible in the instance excluding the first octet. The length of an instance of `Binary` is the number of octets currently in the instance excluding the first octet. The class definition shall be as follows:

```
class Binary {

public:
Binary();
```

This function shall create an instance which points to an empty binary.

```
Binary(ULong max);
```

This function shall create an instance in which the initial maximum size is `max`.

```
Binary(ULong max, ULong length, Octet* value, sdaiBool release = BFalse);
```

This function shall create an instance in which the initial maximum size is `max`, the length is `length`, the contents of `Binary` are contained in the buffer `value`. The concept of release is described in 4.2.2.

```
Binary (const Binary& b);
```

This function shall create an instance which points to a copy of the `Binary b` which has the same maximum and length as `b`.

```
~Binary();
```

This function shall release the `Binary` referenced by the receiver.

```
Binary& operator=(const Binary& b);
```

This function shall release the `Binary` referenced by the receiver and shall assign a copy of the `Binary b` to the receiver.

```
ULong maximum() const;
```

This function shall return the maximum size of the receiver that can be used without reallocation.

```
void length(ULong);
```

This function shall set the number of `Octets` of the receiver.

```
ULong length() const;
```

This function shall return the number of `Octets` of the receiver.

```
sdaiLogical operator==(const Binary& b) const;
```

This function shall return TRUE if the receiver and the Binary argument `b` are equal; FALSE if they are not equal; UNKNOWN if either is unset.

```
sdaiLogical operator!=(const Binary& b) const;
```

This function shall return FALSE if the receiver and the Binary argument `b` are equal; TRUE if they are not equal; UNKNOWN if either is unset.

```
Octet& operator[](ULong index);
```

This function shall return a reference to the Octet at position `I`. The behaviour is not specified in this part of ISO 10303 if `I` is out of range.

```
const Octet& operator[](ULong index) const;
}; // end of class definition for Binary
```

This function shall return a copy of the Octet at position `I`. The behaviour is not specified in this part of ISO 10303 if `I` is out of range.

NOTE - For a description of how the Octet class is used in CORBA implementations, see [1], section 3.8.1.

5.3.2 Aggregate and iterator data types

The EXPRESS aggregate data types, whether bounded or unbounded, shall be represented by a C++ class. The name of the class shall be the concatenation of the aggregate element type, two underscores ("__"), and the name of the aggregate type with the name of the aggregate type all lower case. In the case of an aggregate element type which is itself an aggregate, the aggregate element type name shall be the name of the contained aggregate. This aggregate class shall accommodate any EXPRESS data type and shall be used for persistent as well as non-persistent aggregates. Each aggregate class shall be provided with an "_var" handle type. Access to the aggregate class instances shall be restricted to the "_var" handle. If the "_var" returns TRUE to the `is_nil` function, then the aggregate is unset.

A non-persistent aggregate shall be deleted by means of its destructor. The effect of the deletion upon the aggregates resources is described in the construction paragraphs below. This shall support the SDAI operation Delete non-persistent list as well as the deletion of non-persistent aggregates of type SET, BAG, and ARRAY. The deletion of a non-persistent aggregate shall have no effect on its contents.

Each aggregate class shall have the following methods:

— Default constructor: this function shall create a new empty aggregate.

— Maximum size constructor: this function shall create a new empty aggregate. The maximum size value shall indicate the number of elements that the aggregate accommodates without memory reallocation.

— Data constructor: this function shall create a new aggregate with a designated length, maximum size and release flag. It shall point to a buffer of elements provided as an argument. The parameters shall appear in the following order: maximum size, designated length, buffer of elements, and the release flag. The release flag shall indicate the manner in which the aggregate manages it resources. If the release flag is set to TRUE, the aggregate shall release its resources upon reallocation of the elements buffer and upon destruction of the aggregate. Upon assignment of a new element value, it shall release the resources associated with that element. If the aggregate contains other aggregates as its elements, the contained aggregates shall release their resources when the containing aggregate releases its resources. If the release flag is set to FALSE, the aggregate shall not release its resources.

— Copy constructor: this function shall create a new aggregate with the same maximum size, length and release flag as the given aggregate. The new aggregate shall contain a deep copy of the given aggregate's elements, where the elements are other aggregates, SELECT types, or primitive values and shall follow the entity copy behaviour described in 5.3.9.3 for the aggregate elements which are ENTITY types.

— Destructor: this function shall destroy the aggregate but not its contents.

— Assignment operator: this function shall assign an aggregate with the same maximum size, length and release flag as the given aggregate. The new aggregate shall be assigned a deep copy of the aggregate elements of the given aggregate. It shall release its current resources if the current release flag is set to TRUE.

— Subscript operators: these functions shall return the element at the given index. The `non-const` version of the operator shall return an lvalue. The `const` version of the operator shall allow read-only access to the element.

— `ULong maximum() const` function: this function shall return the number of elements the aggregate can contain without reallocation.

— `void length(ULong)` function: this function shall set the length of the aggregate.

— `ULong length() const` function: this function shall return the number of elements in the aggregate.

EXAMPLE 4 - This example illustrates the C++ class for an EXPRESS aggregate:

In EXPRESS:

```
edge : LIST OF point;
```

In C++:

```
class Point__list{

public:
Point__list();
Point__list(ULong max);
```

```
Point__list(ULong max, ULong len, Point* value, sdaiBool release =
 sdaiFALSE);
Point__list(const Point__list&);
~Point__list();

Point__list& operator=(const Point__list&);

Point& operator[](ULong index);
const Point& operator[](ULong index) const;

ULong maximum() const;

void length(ULong);
ULong length() const;
};
```

Each SDAI iterator type shall be represented by a C++ class. The name of the class shall be a concatenation of the name of the aggregate element type and "__iterator". Each SDAI iterator type shall iterate over only an "_var" aggregate handle type.

A non-persistent aggregate of type <A> shall be created by invoking the constructor of class <A> and assigning a pointer to the new instance (of type <A>*) to an instance of type <A>_ptr. This shall support the SDAI operation Create non-persistent list as well as the creation of non-persistent aggregates of type SET, BAG, and ARRAY. These non-persistent aggregates may be assigned to aggregate-valued attributes in Application_instances.

The member functions for these aggregate classes and iterator classes are specified in 7.6.

5.3.3 Enumeration data type

Each EXPRESS ENUMERATION data type shall be represented by a C++ enum. The mapping of each EXPRESS ENUMERATION data type shall be accomplished as follows. Each element name in the C++ enumeration shall be represented as a concatenation of the C++ mapping of the EXPRESS enumeration identifier, two underscores ("__"), and the EXPRESS element name with the EXPRESS element name all lowercase. Each C++ enumeration shall contain an element name which signifies the unset state. The unset element name shall follow the last EXPRESS element name and shall be represented by the concatenation of the EXPRESS enumeration identifier and "_unset".

EXAMPLE 5 - This example illustrates a mapping from EXPRESS to C++:

```
In EXPRESS:
TYPE
traffic_light = ENUMERATION_OF ( red, yellow, green );
END_TYPE;

In C++:
enum Traffic_light { Traffic_light__red, Traffic_light__yellow,
Traffic_light__green, Traffic_light_unset };
```

The order of the C++ element names shall be the same as the order of the EXPRESS element names. The values of the elements shall begin at zero and increase by one as the EXPRESS declaration is read from left to right.

NOTE - 1 For a description of how this type is used in CORBA implementations, see [1], section 16.11.

For each C++ `enum` representing an EXPRESS ENUMERATION there shall be an associated C++ class. Its name shall be the concatenation of the name of the corresponding C++ `enum` and "_var". The following operations are defined for this data type:

— default construction of an unset instance;

— copy construction from an existing instance;

— initialization by an appropriate `enum` element name;

— assignment from an appropriate `enum` element name;

— automatic conversion between the class type and the C++ `enum`.

Each enumeration class shall be a derived class of the C++ class `Enum`.

NOTE - 2 Enum is defined in order to collect behaviour associated with all EXPRESS enumerations.

```
class Enum {
public:
Enum();
```

This function shall construct an unset instance of the enumeration.

```
Enum(const Enum& e);
```

This function shall create an instance of class `Enum` with the same enumeration value as the `Enum e`.

```
operator ULong() const;
```

This function shall return the integer equivalent of the enumeration's enumerator.

```
int exists() const;
```

This function shall test whether the receiver has been set. If the receiver is unset, it shall return 0; otherwise, it shall return 1;

```
void nullify();
};
```

This function shall change the receiver to an unset state.

EXAMPLE 6 -This example illustrates the mapping of an EXPRESS enumeration to C++, including the minimum interface to the class definition.

In EXPRESS:

```
TYPE
    an_enum = ENUMERATION OF(a,b);
END_TYPE;
```

In C++:

```
enum An_enum { An_enum__a, An_enum__b, An_enum_unset };

// the class definition
class An_enum_var : public Enum {

public:

An_enum_var();
An_enum_var(const An_enum_var&);
An_enum_var(An_enum);

An_enum_var& operator=(const An_enum);
operator An_enum() const;
};
```

5.3.4 Select data type

Each EXPRESS SELECT data type shall be represented by a C++ class, which implements a discriminated union, and a C++ `enum` for its discriminant. The mapping for the name of the C++ class is described in 4.2.1. The discriminant of the union shall be represented by a C++ `enum` whose identifier is a concatenation of the C++ mapping of the SELECT type name and "_sdaiselect". Each enumerator shall be represented by a concatenation of the C++ mapping of the SELECT type name, two underscores ("__"), and the C++ mapping of the SELECT type member name; the enumerators shall appear in the same order as their respective SELECT type members. The SELECT instance shall be unset if the underlying instance is unset.

5.3.4.1 Basic SELECT classes

Each class for each EXPRESS SELECT shall contain the following member functions:

— default constructor, which shall create a new uninitialized select instance. If an application accesses this class before setting it, the behaviour is not specified in this part of ISO 10303;

— copy constructor, which shall perform a deep copy of its parameter if the parameter is an aggregate, SELECT type, or primitive value and shall follow the entity copy behaviour described in 5.3.9.3 if the parameter is an ENTITY type;

— destructor, which shall release all resources owned by the receiver;

— assignment operator, which shall release all resources owned by the receiver if necessary. It shall
then perform a deep copy of its parameter;

— discriminant accessor member functions, which shall be named `_d`. The fetch function shall be a `const` member function and shall return a value of the type of the select's discriminant. The set function shall have a `void` return type and shall take an argument of the type of the select's discriminant;

— value accessor member functions, which shall automatically set the discriminant and release the resources associated with the previous value. There shall be a set of functions for each underlying type of the select. There shall be one fetch function which takes no arguments and returns an instance of the underlying type. There shall be one or more set functions which shall have return type `void` and which shall take an argument for the underlying type. Each value accessor member function shall be named `c<n>`, where n is the position of the underlying type in the EXPRESS SELECT statement, beginning with one. The following value accessor member functions have the following additional requirements:

— for underlying members of type STRING, there shall be three set functions. The first shall take an argument of type `char*`; the select class shall assume ownership of the resources of the string. The second shall take an argument of type `const char*`; the select class shall contain a copy of the string. The third shall take an argument of type `String_var&`; the select class shall contain a copy of the string;

— for underlying members of ENTITY type <T>, the set function shall take an argument of "`<T>_ptr`". The fetch function shall return a variable of "`<T>_ptr`";

— for all other types, the set function shall take an argument of that type.

EXAMPLE 7 - This example illustrates a mapping from EXPRESS to C++:

```
In EXPRESS:

TYPE
my_string = STRING;
END_TYPE;

TYPE
my_real = REAL;
END_TYPE;

TYPE
arbitrary_choice = SELECT( my_string, my_real, point );
END_TYPE;

ENTITY point;
END_ENTITY;

In C++:

typedef String_var My_string;

typedef double My_real;

enum Arbitrary_choice_sdaiselect {Arbitrary_choice__My_string,
Arbitrary_choice__My_real, Arbitrary_choice__Point};

class Arbitrary_choice {
```

```
public:
Arbitrary_choice();
Arbitrary_choice(const Arbitrary_choice&);
~Arbitrary_choice();

Arbitrary_choice& operator=(const Arbitrary_choice&);

void _d(Arbitrary_choice_sdaiselect);
Arbitrary_choice_sdaiselect _d() const;

void c1(char*);
void c1(const char*);
void c1(const My_string&);
const char* c1() const;

void c2(My_real);
My_real c2() const;

void c3(Point_ptr);
Point_ptr c3() const;
};
```

NOTE - For a description of how this class is used in CORBA implementations, see [1], section 16.10.

5.3.4.2 Extended SELECT classes

For each EXPRESS SELECT data type there shall be a C++ class which implements the behaviour of the EXPRESS SELECT. The name of this class shall be the concatenation of "`Select`", two underscores ("__"), and the name of the SELECT data type. This class shall support the following functionality:

— automatic type conversion; for each of the underlying types, there shall be a conversion function that transforms an instance of the underlying type to an instance of the select type. The instance of the select type shall refer to the identical instance of the underlying type. The input parameter for this function shall be a handle for entity or aggregate classes and shall be an instance of a primitive type;

— automatic conversion from select type to underlying type; for each of the immediate underlying types and for each of the leaf underlying types, there shall be a conversion operator that transforms the select type to the underlying type. This function shall return a handle for entity or aggregate classes and shall return an instance for primitive types. If the conversion fails because the underlying type does not match the conversion operator, the function shall return an undefined value and shall set the error code `sdaiVT_NVLD` (Invalid value type). If an immediate underlying type has the same name as the leaf underlying type, the function shall return the immediate underlying type;

— underlying type member functions; the select type shall be able to receive the public member functions (excluding constructors, destructors, and operators) of each underlying entity type and shall respond to the invocation of the member function as if it were a handle to the underlying type. The name of the select type's member function shall be identical to the member function of the underlying type, except in the following case. If the member functions of any underlying types result in a name clash in the select type, the name of the member function in the select type shall be a concatenation of the underlying type's name, two underscores ("__"), and the underlying

type's member function name with the underlying type's member function name all lower case. If the member function cannot be invoked because the underlying type does not implement the member function, the function shall set the error code `sdaiVT_NVLD` (Invalid value type).

— assignment operator; for each of the underlying types, there shall be an assignment operator (=) which supports the assignment of an instance of the underlying type to an instance of the select type. The instance of the select type shall refer to the identical instance of the underlying type. For entity or aggregate type instances the assignment operator shall take a handle as its argument. For primitive type instances, the assignment operator shall take a primitive instance as its argument.

— underlying type verification; for each of the underlying types, there shall be a `const` member function that tests the type of the underlying instance. The function shall return a `sdaiLogical`; a return value of TRUE shall indicate that the underlying type is of the tested type, a return value of FALSE shall indicate that the underlying type is not of the tested type, a return value of UNKNOWN shall indicate that the underlying instance is not set. The name of the member function shall be an underscore, followed by the underlying type name.

— underlying type; there shall be a member function of the select type which shall return a `String` that represents the type name of the underlying type. The member function shall be a `const` member function called `Underlying_typeName`, which takes no arguments. If the SDAI data dictionary is supported, there shall be a member function of the select type called `Underlying_type`, which takes no arguments and shall return a `const` handle to a `Named_type` entity for the underlying type.

— set the underlying type; for the case of a nested SELECT which contains at least two underlying types which ultimately resolve to the same EXPRESS type, there shall be a member function to set the type of the select's immediate underlying type. The constructors and operators described need only be provided for the common underlying type.

— select type default constructor; there shall be a default constructor for the select type which shall construct an instance whose value is unset.

EXAMPLE 8 - This example illustrates the interface for the select class for a simple example:

In EXPRESS:

```
TYPE person = SELECT(employee,student);

ENTITY employee;
ssn : STRING;
END_ENTITY;

ENTITY student;
id : INTEGER;
END_ENTITY;
```

In C++:

```
typedef class Employee* Employee_ptr;
```

```
class Employee {
. . .
public:
char* ssn_();
void ssn_(const char*);
. . .
};

typedef class Student* Student_ptr;
class Student {
. . .
public:
Integer id_();
void id_(Integer);
};

//      THE SELECT CLASS:
class Select__Person : public Select {
. . .
public:
//   1) Automatic conversion from underlying type to select type

Select__Person(Employee_ptr&);
Select__Person(Student_ptr&);

//   2)  Automatic conversion from select type to underlying type

operator Employee_ptr();
operator Student_ptr();

//    3)  Underlying type member functions

char* ssn_();
void ssn_(const char*);
Integer id_();
void id_(Integer);

//    4) Assignment operator

Select__Person& operator=(Employee_ptr&);
Select__Person& operator=(Student_ptr&);

//    5) Underlying type verification

sdaiLogical _Student() const;
sdaiLogical _Employee() const;

//    6)  Underlying type

String_var  Underlying_typeName() const;
#ifdef SDAI_CPP_LATE_BINDING
const Named_type_Ptr Underlying_type() const;
#endif

//   8) Select type default constructor

Select__Person();
};
```

5.3.4.2.1 The Select abstract base class

Each select class shall be a derived class of the C++ class `Select`.

```
class Select {
public:
Select();
```

This function is a default constructor which shall create a new uninitialized select instance.

```
Select(const Select&);
```

This function is a copy constructor which shall perform a deep copy of its parameter.

```
char*  Underlying_typeName() const;
```

This function shall return the name of the type of the instance which is currently contained in the select instance.

```
#ifdef SDAI_CPP_LATE_BINDING
const Named_type_ptr Underlying_type() const;
#endif
```

This function shall return a constant handle to the `Named_type` instance of the underlying type of the select instance.

```
int exists() const;
```

This function shall test whether the receiver has been set. If the receiver is unset, it shall return 0; otherwise, it shall return 1.

```
void nullify();
```

This function shall change the receiver to an unset state.

```
void SetUnderlying_type(const char* typename);
};
```

The `SetUnderlying_type` function shall set the type of the select's immediate underlying type. This function shall be used only for the special case of a select which contains at least two underlying types which ultimately resolve to the same EXPRESS type.

Possible error indicators

sdaiFN_NAVL	// Operation not implemented
sdaiSYS_ERR	// Underlying system error

Origin

Convenience function

5.3.5 Number data type

The EXPRESS NUMBER data type shall be represented by a C++ class called `Number` with a discriminator enumeration called `Number_discriminant`. The EXPRESS NUMBER data type shall be represented as if it were declared in EXPRESS as:

```
TYPE number_integer = INTEGER; END_TYPE;
TYPE number_real = REAL; END_TYPE;
TYPE NUMBER = SELECT(number_integer, number_real); END_TYPE;
```

5.3.6 Handles

In addition to the C++ types described in this part of ISO 10303, the following EXPRESS data types shall be provided with handle types. The arbitrary EXPRESS ENTITY data type named <T> shall be provided with handle types named <T>_ptr and <T>_var. Mutual assignment of <T>_ptr and <T>_var handle types shall be supported -- without explicit operations or casts. The EXPRESS aggregate data type named <A> shall be provided with the handle type named <A>_var. The EXPRESS BINARY data type shall be provided with a handle type Binary_var. Each EXPRESS SELECT data type <S> shall be provided with a handle type <S>_var. Each EXPRESS TYPE data type <S> which refers to a data type which has been provided with a handle type shall be provided with a handle type <S>_var. The SDAI data type `primitive`, described in 9.3 in ISO 10303-22, which is represented by the C++ class `Any`, shall be provided with the handle type `Any_var`. Handle types need only be accessible by means of the `->` operator.

Each C++ class for each EXPRESS ENTITY data type need only be accessible through its handle, that is, the handle is an opaque handle. If the application addresses the C++ class with a C++ pointer, a C++ reference, a derived class or an instance of the class, then the behaviour is not specified in this part of ISO 10303. The `_ptr` handle shall provide the semantics of a C++ pointer -- except as follows: conversion to `void*`, arithmetic operations, relational operations (including less than, greater than, equal to). The `_ptr` handle shall provide the same implicit widening semantics as a C++ pointer. The `_ptr` handle shall support the same semantics as a C++ pointer for explicit narrowing. Explicit narrowing shall be performed by the static member function `_narrow` that is found in each entity class. This function shall return a narrowed object reference in those cases where narrowing would have been meaningful for the respective C++ pointer types; otherwise, the narrowing shall return a nil object reference. The `_ptr` shall support the C++ pointer semantics for nil object references. Each entity class shall provide a static member function, `_nil`, which shall return a nil object reference of that type.

The `_var` handle shall support the same semantics as the `_ptr` handle, except in the following cases: The `_var` handle shall be used to signal the release of the resources of the type the handle points to. The SDAI implementation shall release the resources of the referenced object upon deallocation or assignment of a new object to the `_var`. For handles to C++ classes representing EXPRESS ENTITY data types A and B, where A is a subtype of B, the following implicit widening operation shall be supported: A_var to B_ptr. An implementation shall not support implicit widening of A_var to B_var. The assignment of an object of type A_var to an object of type B_var shall cause a compile error. Widening of A_var to B_var shall require an explicit call to the `_duplicate` member function contained in each entity class.

NOTE - For a description of how handles are used in CORBA implementations, see[1], section 16.3.

5.3.7 TYPE data type

There are five categories of defined types in EXPRESS:

— simple types;

— aggregate types;

— select types;

— enumeration types;

— defined types.

Enumeration types and select types are described in 5.3.3 and 5.3.4 respectively. The remaining defined types shall, at a minimum, provide the functionality provided by a C++ `typedef` of the form `typedef old-name new-name`. The mapping of both the old-name and the new-name is specified in 4.2.1. This does not prohibit the implementation of defined types as C++ classes.

If the original type is provided with one or more handle types, the new type shall also be provided with handle types. The name of each new handle type shall be derived from the new type name as specified in 5.3.6.

The new type shall respond correctly to all of the operations defined for the original type and shall be automatically converted to and from the original type. An instance of the new type shall be determined to be unset if the original type is determined to be unset.

When a WHERE clause appears in the EXPRESS TYPE declaration, the SDAI implementation shall support the validation of the type's constraints in the case in which the type occurs as an attribute domain in an entity which is a (conceptual) subtype of `Application_instance`. Support for validation of the constraint shall be by means of the `ValidateWhere_rule` function for the `Application_instance`.

EXAMPLES

9 - This example illustrates one possible implementation of a defined type for a simple underlying type:

```
TYPE volume = REAL;
END_TYPE;

typedef Real Volume;
```

10 - This example illustrates one possible implementation of a defined type for an aggregate underlying type:

```
TYPE pointset = SET OF point;
END_TYPE;

typedef Point__set Pointset;
typedef Point__set_var Pointset_var;
```

11 - This example illustrates one possible implementation of a defined type for an underlying type which is itself a defined type:

```
TYPE unit = INTEGER;
END_TYPE;

TYPE measure = unit;
END_TYPE;
```

```
typedef Integer Unit;
typedef Unit Measure;
```

5.3.8 ENTITY data type

The EXPRESS ENTITY data types, described in the application schemas, shall be represented as C++ classes. The mapping from the entity names to the C++ class names is specified in 4.2.1. Each EXPRESS explicit attribute shall be mapped to a pair of C++ (get and set) accessor member functions. For each EXPRESS explicit attribute, the type of the return value of the get function and the type of the argument of the set function shall be the C++ type which represents the EXPRESS type of the attribute. Each EXPRESS inverse attribute shall be mapped to a C++ (get) accessor member function. For each EXPRESS inverse attribute of type <T>, the type of the return value of the get function shall be <T>__list_var. The mapping for derived attributes for each EXPRESS ENTITY is described in 7.5.5. The names of the accessor member functions shall be the mapping of the EXPRESS attribute name as specified in 4.2.1. If the application schema contains classes that 1) are ancestors of a multiply-inherited derived class and 2) have identical attribute names which result in a name clash in the derived class, then the name of the accessor member functions in the ancestor class shall be as specified in 4.2.1.

An EXPRESS subtype shall be mapped to a C++ class which is derived from the C++ class mapped from the EXPRESS supertype.

Each class shall contain the static member function `_nil` which shall return a nil object reference of the same type as the class.

Each class shall contain the static member function `_duplicate` which shall return a new handle of the same type as the class pointing to the same object as its argument. It shall take an argument of type _ptr for the class. If the argument is nil, the function shall return the nil object reference.

Each class shall contain the static member function `_narrow` which shall return a new handle given an existing handle. It shall take an argument of type _ptr for the class. If the type of the argument can be narrowed to the requested type, the function shall return a valid handle; otherwise it will return a nil object reference. If this argument is nil, the function shall return the nil object reference.

EXAMPLE 12 - This example illustrates the mapping from EXPRESS ENTITYs to C++ classes:

In EXPRESS:

```
ENTITY A;
attr1 : REAL;
attr2 : INTEGER;
END_ENTITY;
```

```
ENTITY B
SUBTYPE OF(A);
attr3 : C;
END_ENTITY;

ENTITY C;
attr1 : INTEGER;
attr4 : A;
INVERSE
attr5 :  B for attr3;
END_ENTITY;

ENTITY D
SUBTYPE OF(B, C);
END_ENTITY;

ENTITY E;
attr1 : BOOLEAN;
END_ENTITY;
```

In C++:

```
typedef class A* A_ptr;
typedef A_ptr A_var;

class A {
public:

static A_ptr _duplicate(A_ptr);
static A_ptr _narrow(Object_ptr);
static A_ptr _nil();

Real attr1_() const;
void attr1_(Real);

Integer attr2_() const;
void attr2_(Integer);
};

class C;
typedef class C* C_ptr;
typedef C_ptr C_var;

typedef class B* B_ptr;
typedef B_ptr B_var;

class B : public A {
public:

static B_ptr _duplicate(B_ptr);
static B_ptr _narrow(Object_ptr);
static B_ptr _nil();

C_ptr attr3_() const;
void attr3_(const C_ptr);
};

class C {
public:
```

```
    static C_ptr _duplicate(C_ptr);
    static C_ptr _narrow(Object_ptr);
    static C_ptr _nil();

    Integer attr1_() const;
    void attr1_(Integer);

    A_ptr attr4_() const;
    void attr4_(const A_ptr);

    B__list_var attr5_() const;
    };

    typedef class D* D_ptr;
    typedef D_ptr D_var;

    class D : public B, public C {
    public:

    static D_ptr _duplicate(D_ptr);
    static D_ptr _narrow(Object_ptr);
    static D_ptr _nil();
    };

    typedef class E* E_ptr;
    typedef E_ptr E_var;

    class E {
    public:

    static E_ptr _duplicate(E_ptr);
    static E_ptr _narrow(Object_ptr);
    static E_ptr _nil();

    sdaiBool attr1_() const;
    void attr1_(sdaiBool);
    };
```

Base classes in the application schema need not be virtual base classes; however, no multiply inherited derived class shall contain multiple instantiations of any of its base classes. This requirement applies to all classes, including classes used to instantiate complex instances.

5.3.9 Entity instance

The EXPRESS ENTITY `entity_instance` from the SDAI parameter data schema in 9.4.2 of ISO 10303-22, shall be mapped to the C++ class `Object`. An Object instance for which the `is_nil` function returns TRUE is an unset entity_instance.

```
class Object
{
public:
 static Object_ptr _duplicate(Object_ptr);
 static Object_ptr _nil();
};
```

The class `Object` shall be accessed through the handle types `Object_ptr` and `Object_var`.

SDAI implementations shall support the following functions:

```
void release(Object_ptr obj);
```

The `release` function indicates that the caller will no longer access the reference `obj`.

```
sdaiBool is_nil(Object_ptr obj);
```

The `is_nil` function shall return TRUE if `obj` is a nil object reference; otherwise it returns FALSE. An SDAI application shall not invoke an operation through a nil object reference. The nil object reference may be implemented as a NULL pointer. The `is_nil()` operation shall be the only way an object reference can be checked whether it is nil.

5.3.9.1 Common static member functions for object derived classes

For each class <T> which is a descendent of the class `Object` there shall be three static member functions. The first member function is `_duplicate`. This function shall take a <T>_ptr as an argument, and it shall return a <T>_ptr, which refers to a shallow copy of the argument. The second member function is `_narrow`. This function shall take an `Object_ptr` as an argument, and it shall return a <T>_ptr, which refers to the argument. The third member function is `_nil`. This function shall take no arguments, and it shall return a <T>_ptr, which shall be a nil object reference.

5.3.9.2 Create Entity Instance

For each class named <T> which represents an EXPRESS ENTITY, there shall be a C++ class named <T>Factory, which shall be a derived class of `DAObjectFactory`. Each class shall have a member function called `create_`<T>, which shall return a new instance of type <T> in a <T>_ptr return value and which shall take no arguments. The `create_`<T> function shall create a new instance of type <T> which shall have the same instance values as if it were created with <T>'s default constructor.

EXAMPLE 13 - This example illustrates the mapping from EXPRESS ENTITYs to C++ classes.

In EXPRESS:

```
ENTITY my_entity;
  c : STRING;
END_ENTITY;

ENTITY derived_entity;
SUBTYPE OF( my_entity);
  a  :  REAL;
  b  :  INTEGER;
END_ENTITY;
```

In C++:

```
class My_entity {
  public:
  My_entity();
  My_entity(My_entity&);
};
```

```
class Derived_entity {
  public:
  Derived_entity();
  Derived_entity(Derived_entity&);
};

class My_entityFactory : public virtual DAObjectFactory {
  ....
  My_entity_ptr create_My_entity();
};

class Derived_entityFactory : public virtual DAObjectFactory {
  ...
  Derived_entity_ptr create_Derived_entity();
};
```

NOTE - The application may create new instances of derived classes of Application_instance only by using the Factory, the CreateEntityInstance operation, or the sdaiCreate macro. A macro is used to support a variety of implementations. The syntax of the sdaiCreate macro is designed to resemble the syntax of the new operator with placement.

There shall be an `sdaiCreate` macro of the form:

```
sdaiCreate(placement, (constructor_args), (classNames));
```

where the macro shall return a `_ptr` to the new entity instance; `placement` shall be a `_ptr&` to the `Model_contents` in which the new instance is to be placed, `constructor_args` shall be the, possibly empty, list of arguments to a constructor of the entity class and shall be enclosed in a single pair of parentheses, and `className` shall be one or more C++ class names of the entity. The `classNames` shall be provided in alphabetical order.

EXAMPLE 14 - This example illustrates how a new instance of class Foo is created.

```
Model_contents_ptr smh;
...
Foo_ptr sf = sdaiCreate(smh,(),(Foo));
```

The `sdaiCreate` macro shall implement the SDAI operation specified in 10.7.9 of ISO 10303-22, Create entity instance. An implicit `AddInstance` function with the new entity instance as its input parameter shall be invoked on the `Model_contents` placement.

5.3.9.3 Constructors

For the class `Object` as well as each of its derived classes, the following constructors shall be specified:

— a default constructor, which shall take no arguments and shall construct each of its attributes as unset;

— a copy constructor, which shall take a `const` reference to an instance of the class itself and shall perform a shallow copy.

5.3.10 Persistent data and persistent object identifiers

The EXPRESS ENTITY `application_instance` from the SDAI parameter data schema ISO 10303-22 as specified by 9.4.3, shall be mapped to the C++ class `Application_instance`. The class `Application_instance` shall be a derived class of the C++ class `DAObject_SDAI`.

```
typedef char* DAObjectID;
```

5.3.10.1 PID

The `PID` class maintains the persistent object identifier for every persistent object, objects of class `DAObject`, and objects of any class derived directly or indirectly from `DAObject`.

```
class PID : public virtual Object
{
public:
static PID_ptr _duplicate(PID_ptr);
static PID_ptr _narrow(Object_ptr);
static PID_ptr _nil();
char* Datastore_type() const;
void Datastore_type(char*);
```

The `Datestore_type` attribute shall identify the type of the data access interface of the underlying datastore. It shall be set to the value "SDAI" for all `Model_contents` stored in an SDAI implementation.

```
char* get_PIDString();
};
```

This function shall return the string representation of the persistent object identifier for the object referred to by this PID. The contents of the `PIDString` is not specified in this part of ISO 10303. The persistent object identifier shall consist of identifiers from the PID class itself as well as from its derived classes. The PID class shall maintain the persistent identifier for the repository in which the persistent object resides.

5.3.10.2 PID_SDAI

The `PID_SDAI` class maintains the persistent object identifier for a `Model_contents` object.

```
class PID_SDAI : public virtual PID
{
public:
static PID_SDAI_ptr _duplicate(PID_SDAI_ptr);
static PID_SDAI_ptr _narrow(Object_ptr);
static PID_SDAI_ptr _nil();

char* Modelid();
```

This function shall return the string representation of the persistent identifier of the cluster of data for the `Model_contents` referred to by this PID.

```
void Modelid(char*);
};
```

This function shall set the string representation of the persistent identifier of the cluster of data for the `Model_contents` referred to by this object.

5.3.10.3 PID_DA

The `PID_DA` class maintains the persistent object identifier for an application `DAObject`.

```
class PID_DA : public virtual PID
{
public:
static PID_DA_ptr _duplicate(PID_DA_ptr);
static PID_DA_ptr _narrow(Object_ptr);
static PID_DA_ptr _nil();

DAObjectID oid();
```

This function shall return the string representation of the persistent identifier, local to the `Model_contents`, of the `DAObject` referred to by this object.

```
void oid(DAObjectID );
};
```

This function shall set the string representation of the persistent identifier, local to the `Model_contents`, of the `DAObject` referred to by this object.

5.3.10.4 DAObjectFactory

The `DAObjectFactory` class may create instances of `DAObjects`.

```
class DAObjectFactory : public virtual Object
{
public:
DAObject_ptr create_DAObject();
};
```

5.3.10.5 DAObject

The `DAObject` interface shall be the base class for application and dictionary entities.

```
class DAObject : public virtual Object
{
public:
static DAObject_ptr _duplicate(DAObject_ptr);
static DAObject_ptr _narrow(Object_ptr);
static DAObject_ptr _nil();

sdaiBool dado_same(DAObject_ptr obj);
```

This function shall return TRUE if `obj` points to the same object as the receiver.

```
PID_DA dado_oid();
```

The `dado_oid` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.6, Get persistent label.

SDAI input/output

SDAI	C++
Object	receiver
Label	return value

```
void dado_remove();
```

This function shall delete the object from the persistent store and releases its resources. The function need not delete the aggregate-valued attributes of the object.

```
void dado_free();
};
```

This function shall inform the implementation that it is free to move the object back to the persistent store.

5.3.10.6 DAObject_SDAI

The `DAObject_SDAI` interface shall implement the operations from the SDAI entities entity_instance and application_instance that apply to application and dictionary entities.

```
class DAObject_SDAI : virtual public DAObject {

protected:
DAObject_SDAI();
DAObject_SDAI(const DAObject_SDAI&);

public:
static DAObject_SDAI_ptr _duplicate(DAObject_SDAI_ptr);
static DAObject_SDAI_ptr _narrow(Object_ptr);
static DAObject_SDAI_ptr _nil();
```

5.3.10.6.1 Find entity instance SDAI-model

The `OwningModel` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.3, Find entity instance SDAI-model.

```
Model_ptr OwningModel() const;
```

SDAI input/output

SDAI	C++
Object	receiver
Model	return value

Additional possible error indicator

FN_NAVL	This function is not supported by this implementation.

5.3.10.6.2 Find entity instance users

The `FindUsers` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.8, Find entity instance users.

```
Object__list_var FindUsers(const Schema_instance__list_var& aggr) const;
```

SDAI input/output

SDAI	C++
Instance	receiver
Domain	aggr
Result	return value

5.3.10.6.3 Find entity instance usedin

The `FindUsedin` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.9, Find entity instance usedin.

```
#ifdef SDAI_CPP_LATE_BINDING
Object__list_var FindUsedin(const Attribute_ptr& attr, const
Schema_instance__list_var& aggr) const;
#endif
```

SDAI input/output

SDAI	C++
Instance	receiver
Role	attr
Domain	aggr
Result	return value

5.3.10.6.4 Find instance roles

The `FindRoles` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.11, Find instance roles.

```
#ifdef SDAI_CPP_LATE_BINDING
Object__list_var FindRoles(const Schema_instance__list_var& aggr) const;
#endif
```

SDAI input/output

SDAI	C++
Instance	receiver
Domain	aggr
Result	return value

5.3.10.6.5 Find instance data types

The `FindDataTypes` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.12, Find instance data types.

```
#ifdef SDAI_CPP_LATE_BINDING
Object__list_var FindDataTypes(const Schema_instance__list_var& aggr) const;
#endif
String__list_var FindDataTypes(const Schema_instance__list_var& aggr) const;
```

SDAI input/output

SDAI	C++
Instance	receiver
Result	return value

5.3.10.6.6 Is same

The `IsSame` function shall determine whether the parameter `otherEntity` refers to the same entity instance as the receiver. This function shall implement the behaviour of the EXPRESS instance comparison operator entity instance equal (:=:).

```
sdaiLogical IsSame(const Object_ptr& otherEntity) const;
```

Output

This function shall return TRUE if the two instances are the same. This function shall return FALSE if the two instances are not the same. This function shall return UNKNOWN if either instance is unset.

Possible error indicators

```
sdaiMX_NDEF             // SDAI-model access not defined
sdaiEI_NEXS             // Entity instance does not exist
sdaiTY_NDEF             // Type not defined
sdaiSY_ERR              // Underlying system error
```

Origin

Convenience function

5.3.10.6.7 Get attribute

The `GetAttr` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.1, Get attribute.

```
#ifdef SDAI_CPP_LATE_BINDING
Any_var GetAttr(const Attribute_ptr& attDef);
Any_var GetAttr(const char* attName);
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Attribute	attDef, attName
Value	return value

5.3.10.6.8 Get instance type

The `GetInstanceType` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.4, Get instance type.

```
#ifdef SDAI_CPP_LATE_BINDING
const Entity_ptr GetInstanceType() const;
#endif
char* GetInstanceTypeName() const;
```

SDAI input/output

SDAI	C++
Object	receiver
Type	return value

5.3.10.6.9 Is instance of

The `IsInstanceOf` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.5, Is instance of.

```
sdaiBool IsInstanceOf(const char* typeName) const;
#ifdef SDAI_CPP_LATE_BINDING
sdaiBool IsInstanceOf(const Entity_ptr& otherEntity) const;
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Type	typeName, otherEntity
Result	return value

5.3.10.6.10 Is kind of

The `IsKindOf` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.6, Is kind of.

```
sdaiBool IsKindOf(const char* typeName) const;
#ifdef SDAI_CPP_LATE_BINDING
sdaiBool IsKindOf(const Entity_ptr& theType) const;
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Type	typeName, theType
Result	return value

5.3.10.6.11 Is SDAI kind of

The `IsSDAIKindOf` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.7, Is SDAI Kind Of.

```
sdaiBool IsSDAIKindOf(const char* typeName) const;
#ifdef SDAI_CPP_LATE_BINDING
sdaiBool IsSDAIKindOf(const Entity_ptr& theType) const;
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Type	typeName, theType
Result	return value

5.3.10.6.12 Test attribute

The `TestAttr` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.2, Test attribute.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiBool TestAttr(const Attribute_ptr& attDef) const;
sdaiBool TestAttr(const char* attName) const
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Attribute	attDef, attName
Result	return value

5.3.10.6.13 Get attribute value bound

The `GetAttrValueBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.10, Get attribute value bound.

```
#ifdef SDAI_CPP_LATE_BINDING
Integer GetAttrValueBound(const Attribute_ptr& attr) const;
#endif
};
```

SDAI input/output

SDAI	C++
Instance	receiver
Attribute	attr
Value	return value

5.3.11 Domain equivalence for early binding

Early binding support for domain equivalence as specified in ISO 10303-22 as described by A2, shall be provided as follows. The SDAI implementation shall contain a namespace for the native schema and for each `external_schema`. There shall be a class declaration for every `domain_equivalent_type` as well as for every native entity data type to which it is declared to be domain equivalent. The C++ class which represents a `domain_equivalent_type` shall be assignable to the C++ class which represents the native entity data type to which it is domain equivalent.

EXAMPLE 15 - This example illustrates one possible implementation for early bound domain equivalence:

```
SCHEMA X;

ENTITY  A;
END_ENTITY;

ENTITY B;
END_ENTITY;

ENTITY C;
END_ENTITY;

END_SCHEMA;

SCHEMA Y;

ENTITY  A;
END_ENTITY;
```

```
ENTITY B;
END_ENTITY;

ENTITY C;
END_ENTITY;

END_SCHEMA;
```

The domain equivalence table for `SCHEMA X` is:

Native	Foreign
A	A
C	C

The domain equivalence table for `SCHEMA Y` is:

Native	Foreign
B	B
C	C

```
namespace SDAI_INTEROPERABLE {

  class A {   };
  class B {   };
  class C {   };
}

namespace X {
   typedef  SDAI_INTEROPERABLE::A A;
   class B : public SDAI_INTEROPERABLE { };
   typedef  SDAI_INTEROPERABLE::C C;
};

namespace Y {
   class A : public SDAI_INTEROPERABLE::A  {   };
   typedef  SDAI_INTEROPERABLE::B B;
   typedef  SDAI_INTEROPERABLE::C C;
};
```

5.3.12 Application instance

All classes created as a result of this part of ISO 10303, which are defined in or interfaced into an application schema, shall be a derived class, directly or indirectly, from `Application_instance`.

5.3.12.1 Class definition

The `Application_instance` class shall implement the SDAI entity application_instance specified in ISO 10303-22 as described by 9.4.3.

```
class Application_instance : virtual public DAObject_SDAI {

public:
Application_instance();
Application_instance(const Application_instance&);

virtual ~Application_instance();
```

```
private:
void operator delete(void*);
```

The `operator delete` function shall be private so that `Application_instance` instances may be deleted only by the `Model::DeleteApplication_instance` function.

5.3.12.1.1 Unset attribute value

The `UnsetAttr` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.4, Unset attribute value.

```
public:
#ifdef SDAI_CPP_LATE_BINDING
void UnsetAttr(const char* attName);
void UnsetAttr(const Explicit_attr_ptr& attDef);
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Attribute	attName, attDef

5.3.12.1.2 Validate required explicit attributes assigned

The `ValidateRequiredExplicitAttrsAssigned` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.10, Validate required explicit attributes assigned.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiBool ValidateRequiredExplicitAttrsAssigned (Attribute__list& aggr)
const;
#endif
sdaiBool ValidateRequiredExplicitAttrsAssigned (String__list& aggr)
const;
```

SDAI input/output

SDAI	C++
Object	receiver
NonConf	aggr
Result	return value

5.3.12.1.3 Validate inverse attributes

The `ValidateInverseAttrs` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.11, Validate inverse attributes.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiLogical ValidateInverseAttrs(Inverse_attribute__list& aggr) const;
#endif
sdaiLogical ValidateInverseAttrs(String__list& aggr) const;
```

SDAI input/output

SDAI	C++
Object	receiver
NonConf	aggr
Result	return value

5.3.12.1.4 Copy application instance

The `Copy` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.1, Copy application instance.

```
Application_instance_ptr Copy (const Model_contents_ptr& targetModel) const;
```

SDAI input/output

SDAI	C++
Object	receiver
TargetModel	targetModel
NewObject	return value

5.3.12.1.5 Put attribute

The `PutAttr` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.3, Put attribute.

```
#ifdef SDAI_CPP_LATE_BINDING
void PutAttr(const char* attrName, const Any_var& attrVal);
void PutAttr(const Explicit_attribute_ptr& attrDef, const Any_var& attrVal);
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Attribute	attrName, attrDef
Value	attrVal

5.3.12.1.6 Validate where rule

The `ValidateWhereRule` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.9, Validate where rule.

```
sdaiLogical ValidateWhereRule(const char* ruleName) const;
#ifdef SDAI_CPP_LATE_BINDING
sdaiLogical ValidateWhereRule(const Where_rule_ptr& ruleDef) const;
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Rule	ruleName,ruleDef
Result	return value

5.3.12.1.7 Validate explicit attributes references

The `ValidateExplicitAttrsReferences` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.12, Validate explicit attributes references.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiLogical ValidateExplicitAttrsReferences(const Attribute__list_var& aggr)
const;
#endif
sdaiLogical  ValidateExplicitAttrsReferences(const  String__list_var&  aggr)
const;
```

SDAI input/output

SDAI	C++
Object	receiver
NonConf	aggr
Result	return value

5.3.12.1.8 Validate aggregates size

The `ValidateAggrSize` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.13, Validate aggregates size.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiLogical ValidateAggrSize(const Attribute__list_var& aggr) const;
#endif
sdaiLogical ValidateAggrSize(const String__list_var& aggr) const;
```

SDAI input/output

SDAI	C++
Object	receiver
NonConf	aggr
Result	return value

5.3.12.1.9 Validate aggregates uniqueness

The `ValidateAggrUniqueness` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.14, Validate aggregates uniqueness.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiLogical ValidateAggrUniqueness(const Attribute__list_var& aggr) const;
#endif
sdaiLogical ValidateAggrUniqueness(const String__list_var& aggr) const;
```

SDAI input/output

SDAI	C++
Object	receiver
NonConf	aggr
Result	return value

5.3.12.1.10 Validate array not optional

The `ValidateArrayNotOptional` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.15, Validate array not optional.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiLogical ValidateArrayNotOptional(const Attribute__list_var& aggr) const;
#endif
sdaiLogical ValidateArrayNotOptional(const String__list_var& aggr) const;
```

SDAI input/output

SDAI	C++
Object	receiver
NonConf	aggr
Result	return value

5.3.12.1.11 Create aggregate instance

The `CreateAggrInstance` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.5, Create aggregate instance.

```
void CreateAggrInstance(const char* attrName);
#ifdef SDAI_CPP_LATE_BINDING
void CreateAggrInstance(const Explicit_attribute_ptr& attrDef);
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Attribute	attrName, attrDef
Aggregate	(not supported in this operation)

5.3.12.1.12 Get persistent label

The `GetPersistentLabel` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.6, Get persistent label.

```
PID_DA_ptr GetPersistentLabel() const;
```

SDAI input/output

SDAI	C++
Object	receiver
Label	return value

5.3.12.1.13 Get description

The `GetDescription` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.8, Get description.

```
char* GetDescription() const;
```

SDAI input/output

SDAI	C++
Object	receiver
Label	return value

5.3.12.1.14 Is scope owner

The `IsScopeOwner` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.8.2, Is scope owner.

```
sdaiLogical IsScopeOwner() const;
```

SDAI input/output

SDAI	C++
Object	receiver
Result	return value

5.3.12.1.15 Get scope

The GetScope function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.8.3, Get scope.

```
Scope_ptr GetScope() const;
```

SDAI input/output

SDAI	C++
Object	receiver
Result	return value

5.3.12.1.16 Add to scope

The AddToScope function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.8.1, Add to scope.

```
void AddToScope(const Application_instance_ptr& theInstance) const;
```

SDAI input/output

SDAI	C++
Object	receiver
Target	theInstance

5.3.12.1.17 Remove from scope

The `RemoveFromScope` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.8.4, Remove from scope.

```
void RemoveFromScope(Scope_ptr& scope) const;
```

SDAI input/output

SDAI	C++
Object	receiver
Target	scope

5.3.12.1.18 Add to export list

The `AddToExportList` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.8.5, Add to export list.

```
void AddToExportList(Scope_ptr& scope) const;
```

SDAI input/output

SDAI	C++
Object	receiver
Target	scope

5.3.12.1.19 Remove from export list

The `RemoveFromExportList` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.8.6, Remove from export list.

```
void RemoveFromExportList(Scope_ptr&) const;
```

SDAI input/output

SDAI	C++
Object	receiver
Target	scope

5.3.12.1.20 Validate scope

The `ValidateScope` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.8.9, Validate scope reference restrictions.

```
sdaiLogical ValidateScope() const;
```

SDAI input/output

SDAI	C++
Object	receiver
Result	return value

5.3.12.1.21 Validate string width

The `ValidateStringWidth` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.16, Validate string width.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiLogical ValidateStringWidth(const Attribute__list_var& aggr) const;
#endif
sdaiLogical ValidateStringWidth(const String__list_var& aggr) const;
```

SDAI input/output

SDAI	C++
Object	receiver
NonConf	aggr
Result	return value

5.3.12.1.22 Validate binary width

The `ValidateBinaryWidth` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.17, Validate binary width.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiLogical ValidateBinaryWidth(const Attribute__list_var& aggr) const;
#endif
sdaiLogical ValidateBinaryWidth(const String__list_var& aggr) const;
```

SDAI input/output

SDAI	C++
Object	receiver
NonConf	aggr
Result	return value

5.3.12.1.23 Validate real precision

The `ValidateRealPrecision` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.18, Validate real precision.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiLogical ValidateRealPrecision(const Attribute__list_var& aggr) const;
#endif
sdaiLogical ValidateRealPrecision(const String__list_var& aggr) const;
};
```

SDAI input/output

SDAI	C++
Object	receiver
NonConf	aggr
Result	return value

5.3.12.2 Schema-dependent application_instance factory

Within each C++ namespace which corresponds to an application schema, there shall be a C++ class `App_inst_<schema_name>Factory`, where <schema_name> is the name of the EXPRESS SCHEMA in all lower case. The class shall contain a "`create_<T>`" function for each application entity type in the schema. The "`create_<T>`" function shall return an `Application_instance` handle which shall reference the instance referenced by the function's argument.

EXAMPLE 16 - This example illustrates the implementation of a schema-dependent application_instance factory.

In EXPRESS:

```
SCHEMA example_schema;

ENTITY first_entity;
END_ENTITY;

ENTITY second_entity;
END_ENTITY;

END_SCHEMA;
```

In C++:

```
namespace Example_schema {

  typedef First_entity* First_entity_ptr;
  class First_entity {   };

  typedef Second_entity* Second_entity_ptr;
  class Second_entity {  };

  class App_inst_example_schemaFactory {

    Application_instance_ptr create_First_entity(const
      First_entity_ptr&);
    Application_instance_ptr create_Second_entity(const
      Second_entity_ptr&);
  };
};
```

As a consequence of the conversion constructors, the C++ class `Application_instance` is conceptually an ancestor of each class generated from each application schema. However, an SDAI implementation need not implement application entity data types as derived classes of `Application_instances`.

5.3.12.3 Early binding accessor member functions

Although the late binding accessor member functions for the manipulation of attributes including get attribute, put attribute, test attribute and unset attribute, are explicitly declared for the classes defined in this binding, member functions with this functionality are required for each class that represents an application entity data type in an early-binding SDAI implementation.

The prototypes for the get and put accessor member functions for an early-binding SDAI implementation are described in 5.3.8. For each explicit attribute, the Test attribute function shall be accomplished by accessing the attribute value and testing whether it is equal to the designated unset value for its type. For each inverse attribute, the Test attribute function shall be accomplished by accessing the attribute value testing whether the resulting list is unset. For each explicit attribute, the Unset attribute function shall be accomplished by setting the attribute to the designated unset value for its type.

5.3.13 Binding-specific data types

5.3.13.1 Access type data type

`Access_type` shall be used to specify the access mode for an SDAI-model or transaction. It shall consist of the values, read-only and read-write, and shall be defined as:

```
enum Access_type {sdaiRO, sdaiRW, Access_type_unset};
```

5.3.13.2 Primitive_type

The `SDAI` namespace shall contain the C++ class `TypeCode`, the C++ data type `TypeCode_ptr`, the C++ `enum` `TCKind` and a set of constants of type `TypeCode_ptr`. The type of a value, given for or

expected from, a particular attribute or aggregate element shall specify these types. The class definition for `Typecode` and declarations for `TypeCode_ptr` and `TCKind` shall be:

```
enum TCKind { tk_null, tk_void, tk_short, tk_long, tk_ushort, tk_ulong,
tk_float, tk_double, tk_boolean, tk_char, tk_octet, tk_any, tk_TypeCode,
tk_Principal, tk_objref, tk_struct, tk_union, tk_enum, tk_string, tk_sequence,
tk_array };

typedef class TypeCode* TypeCode_ptr;
class TypeCode
{
 public:
  TCKind kind() const;
  Long param_count() const;
  Any parameter(Long) const;
  sdaiBool equal(TypeCode_ptr) const;
};
```

The TypeCode shall consist of an attribute of type `TCKind` and a parameter list of type `Any`. The underlying types shall vary according to the TypeCode's `kind` value. The following describes contents of the parameter lists for the various values of `kind`:

— for TypeCode with the `kind` value `tk_enum`, the parameter list shall consist of strings for the enumeration type name and the enumerators;

— for TypeCode with the `kind` value `tk_string`, the parameter list shall consist of the integer value of the maximum string length or zero for an unbounded string;

— for TypeCode with the `kind` value `tk_sequence`, the parameter list shall consist of a TypeCode representing the underlying type and the integer value of the maximum sequence length or zero for an unbounded sequence;

— for TypeCode with the `kind` value `tk_objref`, the parameter list shall consist of a string value containing the name of the object class;

— for TypeCode with the `kind` value `tk_union`, the parameter list shall consist of a string value containing the name of the union, the TypeCode representing the union's discriminant, and a set of triples, one for each member of the union containing a string value representing the union label name, a string value representing the union member name, and a TypeCode representing the union member type.

The relationship between the C++ classes and their corresponding TypeCode values are given in table 3.

Table 3 - Mapping of EXPRESS types to TypeCode values

C++ data type / EXPRESS	TypeCode: TC_kind and {parameters}
Integer / INTEGER	tk_long { }
Real / REAL	tk_double { }
`sdai`Bool / BOOLEAN	tk_enum {`sdai`Bool, BFalse, BTrue, BUnset}
sdaiLogical / LOGICAL	tk_enum {sdaiLogical, LFalse, LTrue, LUnknown, LUnset}
(unbounded) String, / STRING (bounded) String, / STRING Fixed_string_<n> / STRING	tk_string {0} tk_string {maxlen-integer} tk_string {<n>}
Binary / BINARY Bounded_Binary_<n> / BINARY Fixed_Binary_<n> / BINARY	tk_sequence {tk_octet, 0} tk_sequence {tk_octet, <n>} tk_sequence {tk_octet, <n>}
Enumeration / ENUMERATION	tk_enum {enum-name, enumerator1...}
Array, Bag, List, Set of T / ARRAY,BAG,LIST,SET Bounded <n> Array, Bag, List, Set of T / ARRAY,BAG,LIST,SET	tk_sequence {T, 0} tk_sequence {T, <n>}
Entity_instance / ENTITY	tk_objref {entity-name}
Number / NUMBER	tk_union {Number, Number_discriminant, Number_discriminant__integer, c1, tk_long, Number_discriminant__real, c2, tk_double}
Select / SELECT	tk_union {union-name, switch-TypeCode, label-value, member-name, TypeCode (repeat triples)}

For each EXPRESS primitive, aggregate, enumeration, select, type and entity data type, the SDAI implementation shall provide a constant of type `TypeCode_ptr`. The name of the constant shall be "`_tc_<typename>`", where <typename> is the name of the type. Each constant shall be defined at the same scoping level as its corresponding type.

5.3.13.3 Error code data type

`Error_id` shall be the type used to represent error conditions. Its values are defined in clause 6.

5.3.13.4 Time_stamp data type

`Time_stamp` shall be the type used to represent the EXPRESS TYPE `time_stamp` in the SDAI session schema from ISO 10303-22 as described by 7.3.3.

```
typedef String_var Time_stamp;
```

5.3.13.5 Schema_name data type

Schema_name shall be the type used to represent the EXPRESS TYPE `schema_definition` in the SDAI population schema from ISO 10303-22 as described by 8.3.1.

```
typedef String_var Schema_name;
```

5.3.13.6 Entity_name data type

`Entity_name` shall be the type used to represent the EXPRESS TYPE `entity_definition` in the SDAI population schema from ISO 10303-22 as described by 8.3.2.

```
typedef String_var Entity_name;
```

5.3.13.7 Transaction commit mode data type

`Commit_mode` shall be used to specify whether a particular transaction shall be committed or not. It shall consist of two values, commit and abort, and shall be defined as:

```
enum Commit_mode {sdaiCOMMIT, sdaiABORT};
```

5.3.13.8 SDAI primitive

The primitive data type in the SDAI parameter data schema from ISO 10303-22 as described by 9 shall be represented in the `SDAI` namespace as the C++ classes `Any` and `Any_var`. The class `Any` shall contain an untyped pointer to a value, a value, a TypeCode indicating the type of the value and a `TypeCode_ptr` which references its TypeCode. The class `Any` shall provide the following member functions:

```
class Any
{
 public:
  Any();
```

This function shall create a new `Any` instance with a `TypeCode tk_null` and no value.

```
Any(const Any& a);
```

This function shall create a new `Any` instance with a copy of the `TypeCode` of `a` and shall contain a deep copy of its parameter value.

```
Any(TypeCode_ptr tc, void* val, sdaiBool release = BFalse);
```

This function shall create a new Any instance which contains a copy of the TypeCode tc and the value val. If release is set to TRUE, the Any instance shall release its resources upon assignment of a new value or when the Any's destructor is called. If release is set to TRUE, the value of the val pointer after this function is called is not specified in this part of ISO 10303. If release is set to FALSE, the Any instance shall point the value at val; the resources of the Any instance shall not be released.

```
~Any();
```

This function shall release the TypeCode, the value and any other resources owned by the instance.

```
Any& operator=(const Any&);
```

This function shall release the resources of the instance, including the TypeCode and the value, if the receiver was created with release set to TRUE. The instance shall then contain a copy of the TypeCode of the parameter and a deep copy of the value of the parameter.

```
void operator<<=(Short);
void operator<<=(UShort);
void operator<<=(Integer);
void operator<<=(ULong);
void operator<<=(Float);
void operator<<=(Real);
void operator<<=(const sdaiLOGICAL&);
void operator<<=(const sdaiBOOL&);
void operator<<=(const Any&);
void operator<<=(const char*);
```

For each EXPRESS primitive, aggregate, enumeration, select, type and entity data type, there shall be a C++ operator named "<<=" which inserts a value of that type into the instance and automatically sets the TypeCode to the value that represents that type. The Any instance may not store its value as a reference or pointer to the parameter. For a value of type <T>, the parameter shall be of type <T>, except for entities, which shall be of type T_ptr, and strings, which shall be of type const char*. If the receiver was created with release set to TRUE, the instance shall release the existing value.

```
sdaiBool operator>>=(Short&) const;
sdaiBool operator>>=(UShort&) const;
sdaiBool operator>>=(Integer&) const;
sdaiBool operator>>=(ULong&) const;
sdaiBool operator>>=(Float&) const;
sdaiBool operator>>=(Real&) const;
void operator>>=(const sdaiLOGICAL&);
void operator>>=(const sdaiBOOL&);
sdaiBool operator>>=(Any&) const;
sdaiBool operator>>=(char*&) const;
```

For each EXPRESS primitive, aggregate, enumeration, select, type and entity data type, there shall be an operator named ">>=" which extracts a typed value from the Any. This function shall return a value of type sdaiBool, which shall be sdaiTRUE if the Any contains a value of the correct type; otherwise, it shall return sdaiFALSE. For a value of type T&, the parameter of the argument shall be T& for primitive types (except strings), enumerations and entities. It shall be T*& for strings, selects and Any. If the extraction is successful and the receiver was created with release set to sdaiTRUE, the

caller's pointer will point to resources managed by the Any. If the Any's resources are released, the behaviour is not defined in this part of ISO 10303.

```
void replace(TypeCode_ptr tc, void* value, sdaiBool release = BFalse);
```

This function shall release the current TypeCode and insert a copy of the TypeCode tc. If release is set to sdaiTRUE, the Any shall release any resources associated with the current value. The function shall assign the new value to the receiver. If release is set to sdaiTRUE, the Any instance shall release its resources upon assignment of a new value or when the Any goes out of scope. If release is set to sdaiTRUE, the value of the val pointer after this function is called is not specified in this part of ISO 10303. If release is set to sdaiFALSE, the Any instance shall point the value at val; the Any instance shall not release its resources.

```
TypeCode_ptr type() const;
```

This function returns the current TypeCode.

```
const void* value() const;
};
```

This function returns a pointer to the current value stored in the receiver. An implementation of this part of ISO 10303 shall support the casting of returned void* if the value is of one of the following types: Octet, Short, UShort, Integer, ULong, Double, Any, Object_ptr, TypeCode_ptr. The implementation need not support casting to any other types.

5.3.13.9 Complex instances

A complex entity data type shall represent an intersecting set of subtypes of a supertype, which are not explicitly defined as entities in the schema. A complex entity data type shall be a set of single entity data types which serve as subtype leaves in the particular subtype/supertype instance graph.

Complex entity data types may be created either at compile-time (early binding) or at run-time using the dictionary (late binding).

In the early-bound implementations complex entity type data names shall be constructed using the algorithm described in ISO 10303-22 as described by annex A. The set of valid early-bound complex entity data types shall be specified in an addendum to the EXPRESS file. The interface for defining the list of valid complex entity names is defined as follows:

```
<complex_data_types>   :=   'COMPLEX;'   <complex_type_identifier>   ';'   {
<complex_type_identifier> ';' } 'END_COMPLEX;'
```

EXAMPLE 17 - This example illustrates the logical results of specifying complex entity data types:

In EXPRESS:

```
ENTITY person;
END_ENTITY;
```

```
ENTITY student
SUBTYPE OF(person);
END_ENTITY;

ENTITY employee
SUBTYPE OF(person);
END_ENTITY;

ENTITY manager
SUBTYPE OF(person);
END_ENTITY;

COMPLEX;
employee+student;
employee+manager;
END_COMPLEX;
```

The name of the complex class shall be the concatenation of each component type name in the order specified in ISO 10303-22 as described by A.1.3. The first letter of each component class name shall be uppercase and all remaining letters shall be lowercase. The result of this addendum to the EXPRESS file is that the following declarations are added to the resulting SDAI implementation:

```
class  EmployeeStudent :  public Employee,  public Student {   };

class EmployeeManager :  public Employee,  public Manager {   };
```

In the late bound implementation, the complex entity data types need not be specified until runtime. In this implementation, data types are specified when a new entity instance is created with the `Create entity instance` operation.

Handles to complex instances shall be mutually assignable regardless of whether the instance was created using the early bound or late bound implementation.

5.3.13.10 Bound_instance_value

`Bound_instance_value` shall be the type used to represent the EXPRESS TYPE `bound_instant_value` in the SDAI parameter data schema from ISO 10303-22 as described by 9.3.11.

```
typedef Integer bound_instant_value;
```

5.3.13.11 Query source

The class `Query_source` shall represent the EXPRESS TYPE query_source in the SDAI parameter data schema from ISO 10303-22 as described by 9.3.12.

```
class Query_source  :  public Select  {

protected:
Query_source();
Query_source(const Query_source_ptr&);
Query_source(const Aggregate_instance_ptr&);
Query_source(const Model_ptr&);
Query_source(const Repository_ptr&);
```

```
Query_source(const Schema_instance_ptr&);

operator Aggregate_instance_ptr();
operator Model_ptr();
operator Repository_ptr();
operator Schema_instance_ptr();

Query_source_ptr& operator=(Aggregate_instance_ptr&);
Query_source_ptr& operator=(Model_ptr&);
Query_source_ptr& operator=(Repository_ptr&);
Query_source_ptr& operator=(Schema_instance_ptr&);

public:
operator const Aggregate_instance_ptr() const;
operator const Model_ptr() const;
operator const Repository_ptr() const;
operator const Schema_instance_ptr() const;

private:
sdaiLogical _Aggregate_instance() const;
sdaiLogical _Model() const;
sdaiLogical _Repository() const;
sdaiLogical _Schema_instance() const;

public:
static Query_source_ptr _duplicate(Query_source_ptr);
static Query_source_ptr _narrow(Object_ptr);
static Query_source_ptr _nil();

// Underlying type member functions

// Aggregate_instance, Model, Repository, Schema_instance

Integer SDAIQuery(const char* expression, const Object_ptr& theEntity,
T__sdailist_var& destination) const;


// Model, Repository, Schema_instance
char* name_() const;

// Model
const Model_contents_ptr Model__contents_() const;
Model_contents_ptr Model__contents_();
#ifdef SDAI_CPP_LATE_BINDING
const Schema_ptr underlying_schema_() const;
#endif
Schema_name underlying_schema_name_() const;
const Repository_ptr repository_() const;
Time_stamp change_date_() const;
Access_type mode_() const;
Schema_instance__list_var associated_with_() const;
void RenameModel(const char* modelName);
void CloseModel();
void CloseModel() const; Model_ptr PromoteModelToReadWrite() const;
void SaveChanges();
void UndoChanges();
#ifdef SDAI_CPP_LATE_BINDING
const Entity_ptr GetEntityDefinition(const char* entityName) const;
#endif

// Repository
```

```
const Repository_contents_ptr Repository__contents_() const;
char* description_() const;
Session__list_var session_() const;
Model_ptr CreateModel(const char* modelName, const char* schemaName);
#ifdef SDAI_CPP_LATE_BINDING
Model_ptr CreateModel(const char* modelName, const Schema_ptr& schema);
#endif
const Model_ptr GetModel(const char* modelName) const;
Model_ptr GetModelRW(const char* modelName);
void DeleteModel(Model_ptr& model);DAObject_ptr GetSessionIdentifier(const
PID_DA_ptr& label) const;
#ifdef SDAI_CPP_LATE_BINDING
Schema_instance_ptr CreateSchemaInstance(const char* sname, const Schema_ptr&
schemaDef);
#endif
Schema_instance_ptr CreateSchemaInstance(const char* sname, const char*
schemaName);
void DeleteSchemaInstance(Schema_instance_ptr& inst);

// Schema_instance
const Model__set_var associated_models_() const;
Schema_name native_schema_() const;
const Repository_ptr repository_() const;
Time_stamp change_date_() const;
Time_stamp validation_date_() const;
sdaiLogical validation_result_() const;
Integer validation_level_() const;
void Rename(const char* name);
void AddModel(const Model_ptr& model);
void RemoveModel(const Model_ptr& model);
virtual sdaiLogical ValidateGlobalRule(const char* ruleName,
String__list_var& aggr) const;
#ifdef SDAI_CPP_LATE_BINDING
virtual sdaiLogical ValidateGlobalRule(const Global_rule_ptr& globalRule,
Where_rule__list_var& aggr) const;
#endif
virtual sdaiLogical ValidateUniquenessRule(const char* ruleName, Object_ptr&
theEntity) const;
#ifdef SDAI_CPP_LATE_BINDING
virtual sdaiLogical ValidateUniquenessRule(const Uniqueness_rule_ptr& theRule,
Object_ptr& theEntity) const;
#endif
#ifdef SDAI_CPP_LATE_BINDING
virtual sdaiLogical ValidateInstanceReferenceDomain(const
 Application_instance_ptr& app, Attribute__list_var& aggr) const;
#endif
virtual sdaiLogical ValidateInstanceReferenceDomain(const
 Application_instance_ptr& app, String__list_var& aggr) const;
virtual sdaiLogical ValidateSchemaInstance() const;
virtual sdaiLogical IsValidationCurrent() const;
};
```

5.4 EXISTS functions

The SDAI namespace shall contain the following convenience functions for the testing of the existence of instances of every data type specified in this part of ISO 10303.

`sdai`Bool EXISTS(const Object_ptr& ob);

This function shall return `BFalse` if the `ob->is_nil()` operation returns 0 and shall return `BTrue` if the `ob->is_nil()` operation returns 1.

```
sdaiBool EXISTS(Integer ob);
sdaiBool EXISTS(Real ob);
sdaiBool EXISTS(BOOL ob);
sdaiBool EXISTS(sdaiLOGICAL ob);
sdaiBool EXISTS(const char* ob);
sdaiBool EXISTS(const Binary_var& ob);
sdaiBool EXISTS(const Enum& ob);
sdaiBool EXISTS(const Select_var& ob);
sdaiBool EXISTS(const Aggregate_instance_var& ob);
```

These functions shall return `BFalse` if `ob` is unset, otherwise it shall return `BTrue`.

```
sdaiBool EXISTS(const Any_var& ob);
```

This function shall return the return value that results from invoking the EXISTS function on the underlying value of the `Any`.

6 Error handling

The SDAI namespace shall contain the following C++ exception classes: `Exception`, `UserException`, and `SDAIException`.

```
class Exception
{
  public:
  Exception(const Exception&);
  ~Exception();
  Exception& operator=(const Exception&);

  protected:
  Exception();
};

class UserException : public Exception
{
  public:
  UserException();
  UserException(const UserException&);
  ~UserException();
  UserException& operator=(const UserException&);
};

class SDAIException : public UserException
{
  public:
  SDAIException();
  SDAIException(const SDAIException&);
  SDAIException(ULong minor);
  ~SDAIException();
```

```
  SDAIException& operator=(const UserException&);

  ULong Minor() const;
  void Minor(ULong);
};
```

NOTE - C++ environments that do not support real C++ exception handling may implement error handling according to the method described in [1], Appendix D.

The implementation shall provide user error handling which may be called by the application after the SDAI enters an error state and has updated the SDAI event log (if the event logging is on).

The following standard error codes shall be defined as global symbolic constants. These values may be used to set and test the error code value, `minor`, in `SDAIException`. These values shall be conformant with the C++ type `ULong`:

```
enum Error_id {
sdaiNO_ERR = 0 ,     // No error
sdaiSS_OPN = 10,     // Session open
sdaiSS_NAVL = 20,    // Session not available
sdaiSS_NOPN = 30,    // Session is not open
sdaiRP_NEXS = 40,    // Repository does not exist
sdaiRP_NAVL = 50,    // Repository not available
sdaiRP_OPN = 60,     // Repository open
sdaiRP_NOPN = 70,    // Repository is not open
sdaiTR_EAB = 80,     // Transaction ended abnormally
sdaiTR_EXS = 90,     // Transaction exists
sdaiTR_NAVL = 100,   // Transaction not available
sdaiTR_RW = 110,     // Transaction read-write
sdaiTR_NRW = 120,    // Transaction not read-write
sdaiTR_NEXS = 130,   // Transaction does not exist
sdaiMO_NDEQ = 140,   // SDAI-model not domain equivalent
sdaiMO_NEXS = 150,   // SDAI-model does not exist
sdaiMO_NVLD = 160,   // SDAI-model invalid
sdaiMO_DUP = 170,    // SDAI-model duplicate
sdaiMX_NRW = 180,    // SDAI-model access not read-write
sdaiMX_NDEF = 190,   // SDAI-model access not defined
sdaiMX_RW = 200,     // SDAI-model access read-write
sdaiMX_RO = 210,     // SDAI-model access read-only
sdaiSD_NDEF = 220,   // Schema definition not defined
sdaiED_NDEF = 230,   // Entity definition not defined
sdaiED_NDEQ = 240,   // Entity definition not domain equivalent
sdaiED_NAVL = 250,   // Entity definition not available
sdaiRU_NDEF = 260,   // Rule not defined
sdaiEX_NSUP = 270,   // Expression evaluation not supported
sdaiAT_NVLD = 280,   // Attribute invalid
sdaiAT_NDEF = 290,   // Attribute not defined
sdaiSI_DUP = 300,    // Schema instance duplicate
sdaiSI_NEXS = 310,   // Schema instance does not exist
sdaiEI_NEXS = 320,   // Entity instance does not exist
sdaiEI_NAVL = 330,   // Entity instance not available
sdaiEI_NVLD = 340,   // Entity instance invalid
sdaiEI_NEXP = 350,   // Entity instance not exported
sdaiSC_NEXS = 360,   // Scope does not exist
sdaiSC_EXS = 370,    // Scope exists
sdaiAI_NEXS = 380,   // Aggregate instance does not exist
sdaiAI_NVLD = 390,   // Aggregate instance invalid
sdaiAI_NSET = 400,   // Aggregate instance is empty
sdaiVA_NVLD = 410,   // Value invalid
```

```
sdaiVA_NEXS = 420,  // Value does not exist
sdaiVA_NSET = 430,  // Value not set
sdaiVT_NVLD = 440,  // Value type invalid
sdaiIR_NEXS = 450,  // Iterator does not exist
sdaiIR_NSET = 460,  // Current member is not defined
sdaiIX_NVLD = 470,  // Index invalid
sdaiER_NSET = 480,  // Event recording not set
sdaiOP_NVLD = 490,  // Operator invalid
sdaiFN_NAVL = 500,  // Function not available
sdaiSY_ERR = 1000   // Underlying system error
};
```

Upon an error condition in any SDAI operation, the SDAI implementation shall set the `minor` value of the SdaiException to the `Error_id` enumerator which represents the appropriate error indicator from the list of possible error indicators specified in ISO 10303-22 for that SDAI operation.

6.1 Event

The `Event` class shall implement the SDAI entity `event` specified in ISO 10303-22 as described by 7.4.6.

```
typedef Any Error_base;

class Event : public Session_instance {

friend class Session;
friend class Error_event;

public:
Event();
Event(const Event&);

char* function_id_() const;
Time_stamp time_() const;

static Event_ptr _duplicate(Event_ptr);
static Event_ptr _narrow(Object_ptr);
static Event_ptr _nil();

private:
void function_id_(const char*);
void time_(const Time_stamp&);
};
```

6.2 Error event

The `Error_event` class shall implement the SDAI entity `error_event` specified in ISO 10303-22 as described by 7.4.7.

```
class Error_event : public Event {

public:
Error_event();
Error_event(const Error_event&);

public:
```

```
Error_id error_() const;
char* description_() const;
const Error_base_var base_() const;

static Error_event_ptr _duplicate(Error_event_ptr);
static Error_event_ptr _narrow(Object_ptr);
static Error_event_ptr _nil();

private:
void error(Error_id);
void description_(const char*);
void base_(const Error_base_var&)
};
```

7 C++ binding of the SDAI operations

The SDAI entity data types specified in ISO 10303-22 for the SDAI session schema (with the exception of the entity data types for error handling), the SDAI population schema, and the SDAI dictionary schema shall be implemented by the following C++ classes.

7.1 Session classes

The SDAI entity data types specified in ISO 10303-22 for the SDAI session schema (with the exception of the entity data types for error handling) shall be implemented by the following C++ classes.

7.1.1 Session_instance

The `Session_instance` class shall implement the SDAI entity `session_instance` specified in ISO 10303-22 as described by 9.4.6.

```
class Session_instance : public Object {

public:
Session_instance();
Session_instance(const Session_instance&);
virtual ~Session_instance();

static Session_instance_ptr _duplicate(Session_instance_ptr);
static Session_instance_ptr _narrow(Object_ptr);
static Session_instance_ptr _nil();
```

7.1.1.1 Find entity instance users

The `FindUsers` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.8, Find entity instance users.

```
Object__list_var FindUsers(const Schema_instance__list_var& aggr) const;
```

SDAI input/output

SDAI	C++
Instance	receiver
Domain	aggr
Result	return value

7.1.1.2 Find entity instance usedin

The `FindUsedin` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.9, Find entity instance usedin.

```
#ifdef SDAI_CPP_LATE_BINDING
Object__list_var FindUsedin(const Attribute_ptr& attr, const
Schema_instance__list_var& aggr) const;
#endif
```

SDAI input/output

SDAI	C++
Instance	receiver
Role	attr
Domain	aggr
Result	return value

7.1.1.3 Find instance roles

The `FindRoles` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.11, Find instance roles.

```
#ifdef SDAI_CPP_LATE_BINDING
Object__list_var FindRoles(const Schema_instance__list_var& aggr) const;
#endif
```

SDAI input/output

SDAI	C++
Instance	receiver
Domain	aggr
Result	return value

7.1.1.4 Find instance data types

The `FindDataTypes` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.12, Find instance data types.

```
#ifdef SDAI_CPP_LATE_BINDING
Object__list_var FindDataTypes(const Schema_instance__list_var& aggr) const;
#endif
String__list_var FindDataTypes(const Schema_instance__list_var& aggr) const;
```

SDAI input/output

SDAI	C++
Instance	receiver
Result	return value

7.1.1.5 Is same

The `IsSame` function shall determine whether the parameter `otherEntity` refers to the same entity instance as the receiver. This function shall implement the behaviour of the EXPRESS ENTITY instance comparison operator (:=:).

```
sdaiLogical IsSame(const Object_ptr& otherEntity) const;
```

Output

This function shall return TRUE if the two instances are the same. This function shall return FALSE if the two instances are not the same. This function shall return UNKNOWN if either instance is unset.

Possible error indicators

```
sdaiMX_NDEF             // SDAI-model access not defined
sdaiEI_NEXS             // Entity instance does not exist
sdaiTY_NDEF             // Type not defined
sdaiSY_ERR              // Underlying system error
```

Origin

Convenience function

7.1.1.6 Get attribute

The `GetAttr` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.1, Get attribute.

```
#ifdef SDAI_CPP_LATE_BINDING
Any_var GetAttr(const Attribute_ptr& attDef);
Any_var GetAttr(const char* attName);
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Attribute	attDef, attName
Value	return value

7.1.1.7 Get instance type

The `GetInstanceType` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.4, Get instance type.

```
#ifdef SDAI_CPP_LATE_BINDING
const Entity_ptr GetInstanceType() const;
#endif
char* GetInstanceTypeName() const;
```

SDAI input/output

SDAI	C++
Object	receiver
Type	return value

7.1.1.8 Is instance of

The `IsInstanceOf` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.5, Is instance of.

```
sdaiBool IsInstanceOf(const char* typeName) const;
#ifdef SDAI_CPP_LATE_BINDING
sdaiBool IsInstanceOf(const Entity_ptr& otherEntity) const;
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Type	typeName, otherEntity
Result	return value

7.1.1.9 Is kind of

The `IsKindOf` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.6, Is kind of.

```
sdaiBool IsKindOf(const char* typeName) const;
#ifdef SDAI_CPP_LATE_BINDING
sdaiBool IsKindOf(const Entity_ptr& theType) const;
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Type	typeName, theType
Result	return value

7.1.1.10 Is SDAI kind of

The `IsSDAIKindOf` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.7, Is SDAI Kind Of.

```
sdaiBool IsSDAIKindOf(const char* typeName) const;
#ifdef SDAI_CPP_LATE_BINDING
sdaiBool IsSDAIKindOf(const Entity_ptr& theType) const;
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Type	typeName, theType
Result	return value

7.1.1.11 Test attribute

The TestAttr functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.2, Test attribute.

```
#ifdef SDAI_CPP_LATE_BINDING
sdaiBool TestAttr(const Attribute_ptr& attDef) const;
sdaiBool TestAttr(const char* attName) const
#endif
```

SDAI input/output

SDAI	C++
Object	receiver
Attribute	attDef, attName
Result	return value

7.1.1.12 Get attribute value bound

The `GetAttrValueBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.10.10, Get attribute value bound.

```
#ifdef SDAI_CPP_LATE_BINDING
Integer GetAttrValueBound(const Attribute_ptr& attr) const;
#endif
};
```

SDAI input/output

SDAI	C++
Instance	receiver
Attribute	attr
Value	return value

7.1.2 Session

The `Session` class shall implement the SDAI entity `sdai_session` specified in ISO 10303-22 as described by 7.4.1.

```
class Session : public Session_instance {

friend class Repository;
```

NOTE - The Repository is a friend to Session so that models may be added to the set of active models from within the Repository's member functions.

```
protected:
Session();
Session(const Session&);

public:
const Implementation_ptr sdai_implementation_() const;
sdaiBool recording_active_() const;
const Error_event__list_var errors_() const;
const Repository__set_var known_servers_() const;
const Repository__set_var active_servers_() const;
const Model__set_var active_models_() const;
const Schema_instance_ptr data_dictionary_() const;
const Transaction__list_var active_transaction_() const;

static Session_ptr _duplicate(Session_ptr);
static Session_ptr _narrow(Object_ptr);
static Session_ptr _nil();

private:
void sdai_implementation_(const Implementation_ptr&);
void recording_active_(sdaiBool);
void errors_(const Error_event__list_var&);
void known_servers_(const Repository__set_var&);
void active_servers_(const Repository__set_var&);
void active_models_(const Model__set_var&);
void data_dictionary_(const Schema_instance_ptr&);
```

7.1.2.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS `ENTITY` `sdai_session` shall be implemented by the following C++ member functions.

7.1.2.1.1 Errors

The `Errors` function shall provide read-only public access to the list of errors in the `errors` attribute of the class `Session`.

```
public:
const Error_event__list_var Errors();
```

Output

The `Errors` function shall return a read-only list of the error events associated with the session. If event recording is not on, the function shall return an empty list.

Possible error indicators

```
sdaiSS_NOPN             // Session not open
sdaiSY_ERR              // Underlying system error
```

Origin

Convenience function

7.1.2.1.2 Open repository

The `OpenRepo` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.5, Open repository.

```
void OpenRepo(Repository_ptr& repo);
```

SDAI input/output

SDAI	C++
Session	receiver
Repository	repo

7.1.2.1.3 Close repository

The `CloseRepo` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.5.3, Close repository.

```
void CloseRepo(Repository_ptr& repo);
```

SDAI input/output

SDAI	C++
Repository	repo

7.1.2.1.4 Get attribute definition

The `GetAttrDefinition` functions shall retrieve a handle to an `Attribute` instance from the SDAI dictionary schema for a specified instance of `Entity`. The `Entity` shall be specified in either of two ways. The `Entity` may be specified directly by the parameter `attrDef` or the `Entity` may be specified indirectly by the parameters `entityName` and `schemaName`. The `schemaName` parameter shall specify the SDAI dictionary schema which contains the `schemaName` parameter. The `Attribute` retrieved shall be specified by the `attrName` parameter.

```
#ifdef SDAI_CPP_LATE_BINDING
const Attribute_ptr GetAttrDefinition(const Entity_ptr& attrDef, const
char* attrName) const;
const Attribute_ptr GetAttrDefinition(const char* entityName, const
char* schemaName, const char* attrName) const;
#endif
```

Output

This function shall return an `Attribute` handle. Otherwise, this function shall return a null handle.

Possible error indicators

```
sdaiMX_NDEF             // SDAI-model is in undefined access mode
sdaiEI_NEXS             // Entity instance does not exist
sdaiAT_NDEF             // Attribute name unknown
sdaiVA_NSET             // Value not set
sdaiSY_ERR              // Underlying system error
```

Origin

Convenience function

7.1.2.1.5 Open session

The `OpenSession` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.3.1, Open session.

```
static Session_ptr OpenSession();
```

SDAI input/output

SDAI	C++
Session	return value

7.1.2.1.6 Delete aggregate

The `DeleteAggr` function shall delete a persistent `Aggregate_instance`. `Entity_instances` having attribute values which refer to the deleted `Aggregate_instance` need not be automatically updated.

```
void DeleteAggr(Aggregate_instance_var& aggr);
```

Possible error indicators

```
sdaiMX_NRW        // SDAI-model access not read-write
sdaiAG_NEXS       // Aggregate does not exist
sdai SY_ERR       // Underlying system error
```

Origin

Convenience function

7.1.2.1.7 Record error

The `RecordError` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.1, Record error.

```
void RecordError(const char* functionID, Error_id error, const char* descpt);
```

SDAI input/output

SDAI	C++
Session	receiver
Function_id	functionID
Error	error
Description	descpt

7.1.2.1.8 Is recording on

The `IsRecordingOn` function shall indicate whether event recording is currently enabled, disabled, or not supported by the SDAI implementation.

```
sdaiLogical IsRecordingOn() const;
```

Output

This function shall return TRUE if event recording is enabled, FALSE if disabled, and UNKNOWN if not supported.

Possible error indicators

```
sdaiSS_NOPN             // Session not open
sdaiSY_ERR              // Underlying system error
```

Origin

Convenience function

7.1.2.1.9 Start event recording

The `StartEventRecording` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.2, Start event recording.

```
sdaiBool StartEventRecording();
```

SDAI input/output

SDAI	C++
Session	receiver
Result	return value

7.1.2.1.10 Stop event recording

The `StopEventRecording` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.3, Stop event recording.

```
sdaiBool StopEventRecording();
```

SDAI input/output

SDAI	C++
Session	receiver
Result	return value

7.1.2.1.11 Close session

The `CloseSession` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.4, Close session.

```
void CloseSession();
```

SDAI input/output

SDAI	C++
Session	receiver

7.1.2.1.12 Start transaction

The `StartTransaction` function shall implement the SDAI operations specified in ISO 10303-22 as described by 10.4.6 and ISO 10303-22 as described by 10.4.7, Start transaction read-write access and Start transaction read-only access. The `mode` attribute of the new `Transaction` instance shall be set to the value of `theMode`.

```
Transaction_ptr StartTransaction(Access_type theMode);
```

SDAI input/output

SDAI	C++
Session	receiver
Transaction	return value

7.1.2.1.13 Commit

The `Commit` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.8, Commit.

```
void Commit(const Transaction_ptr& xaction);
```

SDAI input/output

SDAI	C++
Transaction	xaction

7.1.2.1.14 Abort

The `Abort` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.9, Abort.

```
void Abort(const Transaction_ptr& xaction);
```

SDAI input/output

SDAI	C++
Transaction	xaction

7.1.2.1.15 End transaction

The `EndTransaction` function shall implement the SDAI operations specified in ISO 10303-22 as described by 10.4.10 and ISO 10303-22 as described by 10.4.11, End transaction access and commit and End transaction access and abort. If the parameter `mode` is set to `sdaiCOMMIT`, this function shall implement the End transaction access and commit operation. If the mode parameter is set to `sdaiABORT`, this function shall implement the End transaction access and abort operation.

```
void EndTransaction(Transaction_ptr& xaction, Commit_mode mode);
};
```

SDAI input/output

SDAI	C++
Transaction	xaction

7.1.3 Implementation

The `Implementation` class shall implement the SDAI entity `Implementation` specified in ISO 10303-22 as described by 7.4.2.

```
class Implementation : public Session_instance {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of Implementation instances only by member functions of Session.

```
protected:
Implementation();
Implementation(const Implementation&);

public:
char* name_() const;
char* level_() const;
char* sdai_version_() const;
char* binding_version_() const;
Integer implementation_class_() const;
Integer transaction_level_() const;
Integer expression_level_() const;
Integer recording_level_() const;
Integer scope_level_() const;
Integer domain_equivalence_level_() const;

static Implementation_ptr _duplicate(Implementation_ptr);
static Implementation_ptr _narrow(Object_ptr);
static Implementation_ptr _nil();

private:
void name_(const char*);
void level_(const char*);
void sdai_version_(const char*);
void binding_version_(const char*);
void implementation_class_(Integer);
void transaction_level_(Integer);
void expression_level_(Integer);
void recording_level_(Integer);
void scope_level_(Integer);
void domain_equivalence_level_(Integer);
void conforms_to_(Implementation_class);
};
```

7.1.4 Transaction

The `Transaction` class shall implement the SDAI entity `sdai_transaction` specified in ISO 10303-22 as described by 7.4.5.

```
class Transaction : public Session_instance  {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of Transaction instances only by member functions of Session.

```
protected:
Transaction();
Transaction(const Transaction&);

public:
Access_type mode_() const;
Session_ptr owning_session_() const;

static Transaction_ptr _duplicate(Transaction_ptr);
static Transaction_ptr _narrow(Object_ptr);
static Transaction_ptr _nil();

private:
void mode_(Access_type);
void owning_session_(const Session_ptr&);
};
```

7.2 Schema_instance

The `Schema_instance` class shall implement the SDAI entity `schema_instance` specified in ISO 10303-22 as described by 8.4.1.

```
class Schema_instance : public Session_instance {

friend class Session;
```

NOTE - Session is friend so that Session may construct and access the private members of Schema_instance instances only by member functions of Session.

```
protected:
Schema_instance();
Schema_instance(const Schema_instance&);
~Schema_instance();

public:
char* name_() const;
const Model__set_var associated_models_() const;
Schema_name native_schema_() const;
const Repository_ptr repository_() const;
Time_stamp change_date_() const;
Time_stamp validation_date_() const;
sdaiLogical validation_result_() const;
Integer validation_level_() const;

static Schema_instance_ptr _duplicate(Schema_instance_ptr);
static Schema_instance_ptr _narrow(Object_ptr);
static Schema_instance_ptr _nil();

private:
void name_(const char*);
void associated_models_(const Model__set_var&);
void native_schema_(const Schema_name&);
void repository_(const Repository_ptr&);
void change_date_(const Time_stamp&);
void validation_date_(const Time_stamp&);
```

```
void validation_result_(sdaiLogical);
void validation_level_(Integer);
```

7.2.1 Rename schema instance

The `Rename` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.6.2, Rename schema_instance.

```
void Rename(const char* name);
```

SDAI input/output

SDAI	C++
Instance	receiver
Name	name

7.2.2 Add SDAI-model

The `AddModel` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.6.3, Add SDAI-Model.

```
void AddModel(const Model_ptr& model);
```

SDAI input/output

SDAI	C++
Instance	receiver
Model	model

7.2.3 Remove SDAI-model

The `RemoveModel` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.6.4, Remove SDAI-model.

```
void RemoveModel(const Model_ptr& model);
```

SDAI input/output

SDAI	C++
Instance	receiver
Model	model

7.2.4 Validate global rule

The `ValidateGlobalRule` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.6.5, Validate global rule.

```
virtual sdaiLogical ValidateGlobalRule(const char* ruleName,
String__list_var& aggr) const;
#ifdef SDAI_CPP_LATE_BINDING
virtual sdaiLogical ValidateGlobalRule(const Global_rule_ptr& globalRule,
Where_rule__list_var& aggr) const;
#endif
```

SDAI input/output

SDAI	C++
Instance	receiver
Rule	ruleName, globalRule
NonConf	aggr
Result	return value

7.2.5 Validate uniqueness rule

The `ValidateUniquenessRule` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.6.6, Validate uniqueness rule.

```
virtual sdaiLogical ValidateUniquenessRule(const char* ruleName, Object_ptr&
theEntity) const;
#ifdef SDAI_CPP_LATE_BINDING
virtual sdaiLogical ValidateUniquenessRule(const Uniqueness_rule_ptr& theRule,
Object_ptr& theEntity) const;
#endif
```

SDAI input/output

SDAI	C++
Instance	receiver
Rule	ruleName, theRule
NonConf	theEntity
Result	return value

7.2.6 Validate instance reference domain

The `ValidateInstanceReferenceDomain` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.6.7, Validate instance reference domain.

```
#ifdef SDAI_CPP_LATE_BINDING
virtual sdaiLogical ValidateInstanceReferenceDomain(const
 Application_instance_ptr& app, Attribute__list_var& aggr) const;
#endif
virtual sdaiLogical ValidateInstanceReferenceDomain(const
 Application_instance_ptr& app, String__list_var& aggr) const;
```

SDAI input/output

SDAI	C++
Instance	receiver
Object	app
NonConf	aggr
Result	return value

7.2.7 Validate schema instance

The `ValidateSchemaInstance` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.6.8, Validate schema instance.

```
virtual sdaiLogical ValidateSchemaInstance() const;
```

SDAI input/output

SDAI	C++
Instance	receiver
Result	return value

7.2.8 Is validation current

The `IsValidationCurrent` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.6.9, Is validation current.

```
virtual sdaiLogical IsValidationCurrent() const;
```

SDAI input/output

SDAI	C++
Instance	receiver
Result	return value

7.2.9 SDAI query

The `SDAIQuery` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.14, SDAI query.

```
Integer SDAIQuery(const char* expression, const Object_ptr& theEntity,
T__sdailist_var& destination) const;
};
```

SDAI input/output

SDAI	C++
Source	receiver
Where	expression
Entity	theEntity
Result	destination
Count	return value

7.3 Schema-specific schema_instance (early binding)

For each application schema in the early-binding implementation, there shall be a class whose name shall be the concatenation of “Schema_instance_” and the name of the schema with the name of the schema all lower case. This class shall be a public derived class from the C++ class `Schema_instance`.

7.4 Repository classes

The SDAI entity data types specified in ISO 10303-22 for sdai_repository_contents and sdai_repository shall be implemented by the following C++ classes.

7.4.1 Repository_contents

The `Repository_contents` class shall implement the SDAI entity `sdai_repository_contents` specified in ISO 10303-22 as described by 7.4.4.

```
class Repository_contents : public Session_instance {

friend class Session;
```

NOTE - Session is a friend so that Repository_contents instances may be constructed only by the member functions of Session.

```
protected:
Repository_contents();
Repository_contents(const Repository_contents&);

public:
const Model__set_var models_() const;
const Schema_instance__set_var schemas_() const;
Repository__list_var repository_() const;

static Repository_contents_ptr _duplicate(Repository_contents_ptr);
static Repository_contents_ptr _narrow(Object_ptr);
static Repository_contents_ptr _nil();

private:
void models_(const Model__set_var&);
void schemas_(const Schema_instance__set_var&);
};
```

7.4.2 Repository

The `Repository` class shall implement the SDAI entity `sdai_repository` specified in ISO 10303-22 as described by 7.4.3.

```
class Repository : public Session_instance {

friend class Session;
```

NOTE - Session is a friend so that Repository instances may be constructed only by member functions of Session.

```
protected:
Repository();
Repository(const Repository&);

public:
char* name_() const;
const Repository_contents_ptr contents_() const;
char* description_() const;
Session__list_var session_() const;

static Repository_ptr _duplicate(Repository_ptr);
static Repository_ptr _narrow(Object_ptr);
static Repository_ptr _nil();

private:
void name_(const char*);
void contents_(const Repository_contents_ptr&);
void description_(const char*);
```

7.4.2.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS `ENTITY` `sdai_repository` shall be implemented by the following C++ member functions.

7.4.2.1.1 Create SDAI-model

The `CreateModel` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.5.1, Create SDAI-model.

```
public:
Model_ptr CreateModel(const char* modelName, const char* schemaName);
#ifdef SDAI_CPP_LATE_BINDING
Model_ptr CreateModel(const char* modelName, const Schema_ptr& schema);
#endif
```

SDAI input/output

SDAI	C++
Repository	receiver
ModelName	modelName
Schema	schemaName, schema
Model	return value

7.4.2.1.2 Get model

The `GetModel` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.7.2, Start read-only access. The `GetModelRW` function shall implement the SDAI operations specified in ISO 10303-22 as described by 10.7.6 and ISO 10303-22 as described by 10.7.4, Start read-write access and Promote SDAI-model to read-write. The model shall be specified indirectly by the `modelName` parameter.

```
const Model_ptr GetModel(const char* modelName) const;
Model_ptr GetModelRW(const char* modelName);
```

SDAI input/output

SDAI	C++
Model	return value

7.4.2.1.3 Delete SDAI-model

The `DeleteModel` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.7.1, Delete SDAI-model.

```
void DeleteModel(Model_ptr& model);
```

SDAI input/output

SDAI	C++
Model	model

7.4.2.1.4 Get session identifier

The `GetSessionIdentifier` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.7, Get session identifier.

```
DAObject_ptr GetSessionIdentifier(const PID_DA_ptr& label) const;
```

SDAI input/output

SDAI	C++
Label	label
Repository	receiver
Object	return value

7.4.2.1.5 Create schema instance

The `CreateSchemaInstance` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.5.2, Create schema instance. The early-binding version of the function shall create a schema-specific `Schema_instance` as specified in 7.3, and the late-binding version of the function shall create an instance of `Schema_instance`.

```
#ifdef SDAI_CPP_LATE_BINDING
Schema_instance_ptr CreateSchemaInstance(const char* sname, const Schema_ptr&
schemaDef);
#endif
Schema_instance_ptr CreateSchemaInstance(const char* sname, const char*
schemaName);
```

SDAI input/output

SDAI	C++
Name	sname
Schema	schemaDef
Repository	receiver
Instance	return value

7.4.2.1.6 Delete schema instance

The `DeleteSchemaInstance` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.6.1, Delete schema instance.

```
void DeleteSchemaInstance(Schema_instance_ptr& inst);
```

SDAI input/output

SDAI	C++
Instance	inst

7.4.2.1.7 SDAI query

The `SDAIQuery` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.14, SDAI query.

```
Integer SDAIQuery(const char* expression, const Object_ptr& theEntity,
T__sdailist_var& destination) const;
};
```

SDAI input/output

SDAI	C++
Source	receiver
Where	expression
Entity	theEntity
Result	destination
Count	return value

7.5 Model classes

The SDAI entity data types specified in ISO 10303-22 for `sdai_model`, `sdai_model_contents`, `entity_extent`, and `scope` shall be implemented by the following C++ classes.

7.5.1 Model

The `Model` class shall implement the SDAI entity `sdai_model` specified in ISO 10303-22 as described by 8.4.2.

```
class Model : public Session_instance {

friend class Repository;
```

NOTE - Repository is a friend so that Model instances may be constructed and their private members may be accessed by member functions of Repository.

```
protected:
Model();
Model(const Model&);

public:
char* name_() const;
const Model_contents_ptr contents_() const;
Model_contents_ptr contents_();
#ifdef SDAI_CPP_LATE_BINDING
const Schema_ptr underlying_schema_() const;
#endif
Schema_name underlying_schema_name_() const;
const Repository_ptr repository_() const;
Time_stamp change_date_() const;
Access_type mode_() const;
Schema_instance__list_var associated_with_() const;

static Model_ptr _duplicate(Model_ptr);
static Model_ptr _narrow(Object_ptr);
static Model_ptr _nil();

private:
void name_(const char*);
void contents_(const Model_contents_ptr&);
#ifdef SDAI_CPP_LATE_BINDING
void underlying_schema_(const Schema_ptr&);
#endif
void underlying_schema_name_(const Schema_name&);
void repository_(Repository_ptr&);
```

```
void change_date_(const Time_stamp&);
void mode_(Access_type);
```

7.5.1.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS `ENTITY sdai_model` shall be implemented by the following C++ member functions.

7.5.1.1.1 Rename SDAI-model

The `RenameModel` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.7.2, Rename SDAI-model.

```
public:
void RenameModel(const char* modelName);
```

SDAI input/output

SDAI	C++
Model	receiver
ModelName	ModelName

7.5.1.1.2 Close model

The `CloseModel` functions shall implement the SDAI operations specified in ISO 10303-22 as described by 10.7.5 and ISO 10303-22 as described by 10.7.7, End read-only access and End read-write access.

```
void CloseModel();
void CloseModel() const;
```

SDAI input/output

SDAI	C++
Model	receiver

7.5.1.1.3 Promote SDAI-model to read/write

The `PromoteModelToReadWrite` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.7.4, Promote SDAI-model to read-write.

```
Model_ptr PromoteModelToReadWrite() const;
```

SDAI input/output

SDAI	C++
Model	receiver

7.5.1.1.4 Save changes

The `SaveChanges` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.7.11, Save changes.

```
void SaveChanges();
```

SDAI input/output

SDAI	C++
Model	receiver

7.5.1.1.5 Undo changes

The `UndoChanges` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.7.10, Undo changes.

```
void UndoChanges();
```

SDAI input/output

SDAI	C++
Model	receiver

7.5.1.1.6 Get entity definition

The `GetEntityDefinition` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.7.8, Get entity definition.

```
#ifdef SDAI_CPP_LATE_BINDING
const Entity_ptr GetEntityDefinition(const char* entityName) const;
#endif
```

SDAI input/output

SDAI	C++
Model	receiver
EntityName	entityName
Entity	return value

7.5.1.1.7 SDAI query

```
Integer SDAIQuery(const char* expression, const Object_ptr& theEntity,
T__sdailist_var& destination) const;
};
```

The `SDAIQuery` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.14, SDAI query.

SDAI input/output

SDAI	C++
Source	receiver
Where	expression
Entity	theEntity
Result	destination
Count	return value

7.5.2 Model_contents_instances

The class `Model_contents_instances` shall implement convenience functions required by Model_contents in this part of ISO 10303.

```
class Model_contents_instances : public DAObject {

public:
DAObject__set_var contents_();
const DAObject__set_var contents_() const;
};
```

This function shall the return the set of `DAObjects` contained in the receiver.

7.5.3 Model_contents

The `Model_contents` class shall implement the SDAI entity `sdai_model_contents` specified in ISO 10303-22 as described by 8.4.3.

```
class Model_contents : public Session_instance {

friend class Model;
```

NOTE - Model is a friend so that Model may access the contents of the Entity_extents folders.

```
protected:
Model_contents();
Model_contents(const Model_contents&);

public:
const Model_contents_instances_ptr instances_() const;
Model_contents_instances_ptr instances_();
const Entity_extent__set_var folders_() const;
Entity_extent__set_var folders_();
const Entity_extent__set_var populated_folders_() const;
Entity_extent__set_var populated_folders_();

const Model_ptr parent_model_() const;
```

This function shall return a reference to the `Model` which the `Model_content` is related to.

Possible error indicators

```
sdaiSS_NOPN     // Session not open
sdaiRP_NOPN     // Repository not open
sdaiMO_NEXS     // SDAI-model does not exist
sdaiSY_ERR      // Underlying system error
```

```
static Model_contents_ptr _duplicate(Model_contents_ptr);
static Model_contents_ptr _narrow(Object_ptr);
static Model_contents_ptr _nil();

PID_DA_ptr get_object_pid(const DAObject_ptr& ) const;
```

This function shall return the persistent object identifier for a `DAObject`.

Possible error indicators

```
sdaiSS_NOPN     // Session not open
sdaiRP_NOPN     // Repository not open
sdaiMO_NEXS     // SDAI-model does not exist
sdaiED_NDEF     // Entity definition unknown in this model
sdaiSY_ERR      // Underlying system error

DAObject_ptr lookup(const PID_DA_ptr& p) const;
```

This function shall return the `DAObject` for the persistent object identifier `p`.

Possible error indicators

```
sdaiSS_NOPN     // Session not open
sdaiRP_NOPN     // Repository not open
sdaiMO_NEXS     // SDAI-model does not exist
sdaiED_NDEF     // Entity definition unknown in this model
sdaiSY_ERR      // Underlying system error

DAObjectFactory_ptr find_factory(const char* TypeName) const;
```

This function shall return the `DAObjectFactory` for the type named `TypeName`.

Possible error indicators

```
sdaiSS_NOPN     // Session not open
sdaiRP_NOPN     // Repository not open
sdaiED_NDEF     // Entity definition unknown in this model
sdaiSY_ERR      // Underlying system error
```

7.5.3.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS `ENTITY` `sdai_model_contents` shall be implemented by the following C++ member functions.

7.5.3.1.1 Delete application instance

The `DeleteInstance` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.2, Delete application instance.

```
void DeleteInstance(DAObject_sdai_ptr& appInst);
```

SDAI input/output

SDAI	C++
Object	appInst

7.5.3.1.2 Add application instance

The `AddInstance` function shall add an application instance to the receiver.

```
void AddInstance(DAObject_sdai_ptr& appInst);
```

Possible error indicators

```
sdaiSS_NOPN             // Session not open
sdaiRP_NOPN             // Repository not open
sdaiED_NDEF             // Entity definition unknown in this model
sdaiSY_ERR              // Underlying system error
```

Origin

Convenience function

7.5.3.1.3 Get entity extent

The `GetEntity_extent` functions shall return a reference to a read-only entity folder from the `folders` attribute within the `contents` attribute of the receiver. This folder shall contain all of the instances of a particular entity and its derived types within the model. The entity may be specified indirectly by the `entityName` parameter or directly by the entity.

```
#ifdef SDAI_CPP_LATE_BINDING
DAObject__set_var GetEntity_extent(const Entity_ptr&);
const DAObject__set_var GetEntity_extent(const Entity_ptr&) const;
#endif
DAObject__set_var GetEntity_extent(const char* entityName);
const DAObject__set_var GetEntity_extent(const char* entityName) const;
```

Output

This function shall return an aggregate handle. Otherwise, this function shall return a null handle.

Possible error indicators

```
sdaiSS_NOPN             // Session not open
sdaiRP_NOPN             // Repository not open
sdaiMO_NEXS             // SDAI-model does not exist
sdaiED_NDEF             // Entity definition unknown in this model
sdaiSY_ERR              // Underlying system error
```

Origin

Convenience function

7.5.3.1.4 Create entity instance

The `CreateEntityInstance` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.7.9, Create entity instance.

```
DAObject_ptr CreateEntityInstance(const char* Type);
#ifdef SDAI_CPP_LATE_BINDING
DAObject_ptr CreateEntityInstance(const Entity_ptr& Type);
#endif
};
```

SDAI input/output

SDAI	C++
Type	Type
Model	receiver
Object	return value

7.5.4 Entity extent

The `Entity_extent` class shall implement the SDAI entity `entity_extent` specified in ISO 10303-22 as described by 8.4.4.

```
class Entity_extent : public Session_instance {

friend class Model_contents;
```

NOTE - The Model_content class is a friend so that Model_content may set the private access member functions of Entity_extents.

```
protected:
Entity_extent();
Entity_extent(const Entity_extent&);
```

```
public:
Entity_name definition_name_() const;
#ifdef SDAI_CPP_LATE_BINDING
const Entity_ptr definition_() const;
#endif
DAObject__set_var instances_();
const DAObject__set_var instances_() const;
Model_contents__list_var owned_by_() const;

static Entity_extent_ptr _duplicate(Entity_extent_ptr);
static Entity_extent_ptr _narrow(Object_ptr);
static Entity_extent_ptr _nil();

private:
#ifdef SDAI_CPP_LATE_BINDING
void definition_(const Entity_ptr&);
#endif
void definition_name_(const Entity_name&)
void instances_(const DAObject__set_var&);
};
```

7.5.5 Model contents by schema

Within the namespace associated with an EXPRESS schema there shall be a C++ class, whose name shall be a concatenation of “Model_contents_” and the schema name with the schema name all lower case. For each ENTITY data type in the schema, this class shall contain a member function which returns the entity extents for that type. For an ENTITY named <T>, the return type of the function shall be `DAObject__set` and the name of the function shall be <T>`_get_extents`, where <T> is the name of the C++ class for that type. The class shall be a public derived class from the C++ class `Model_contents`.

For each derived attribute in the schema, there shall be a `const` accessor member function in this class. The return type shall be the C++ type which represents the EXPRESS type of the derived attribute. The name of the function shall be <T>_<D>, where <T> is the name of the EXPRESS ENTITY, mapped according to 4.2.1, which contains the derived attribute, and <D> is the name of the derived attribute with the name of the derived attribute all lower case. The function shall take one argument whose type shall be the C++ type which represents the EXPRESS ENTITY which contains the derived attribute.

For each derived attribute in the schema, there shall be a `const` accessor member function in the class that represents the EXPRESS entity which contains the derived attribute. The return type shall be the C++ type which represents the EXPRESS type of the derived attribute. The name of the function shall be the name of the derived attribute mapped in the same fashion as explicit EXPRESS attributes as described in 4.2.1. The function shall take no arguments.

EXAMPLE - 20 This example illustrates the mapping of an EXPRESS entity which contains a derived attribute to the C++ programming language.

In EXPRESS:

```
SCHEMA my_schema;
```

```
ENTITY my_entity;
DERIVE
my_derived :   REAL :=  PI;
END_ENTITY;

END_SCHEMA;
```

In C++:

```
namespace My_schema {

  class My_entity  {
    .....
  };

  class Model_contents_my_schema  : Model_contents {

    DAObject__set  My_entitity_get_extents();

    Real  My_entity_my_derived(My_entity_ptr);

  };
};
```

7.5.6 Scope

The Scope class shall implement the SDAI entity scope specified in ISO 10303-22 as described by 8.4.5.

```
class Scope  :  public Session_instance  {

friend class Application_instance;

protected:
Scope();
Scope(const Scope&);

public:
const Application_instance_ptr owner_() const;
const Application_instance__set_var owned_() const;
const Application_instance__set_var export_list_() const;

static Scope_ptr _duplicate(Scope_ptr);
static Scope_ptr _narrow(Object_ptr);
static Scope_ptr _nil();

private:
void owner_(const Application_instance_ptr&);
void owned_(const Application_instance__set_var&);
void export_list_(const Application_instance__set_var&);
```

7.5.6.1 SDAI operation declaration

The SDAI operations specified in ISO 10303-22 for the EXPRESS ENTITY scope shall be implemented by the following C++ member functions.

7.5.6.1.1 Scoped delete

The `ScopedDelete` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.8.7, Scoped delete.

```
public:
void ScopedDelete();
```

SDAI input/output

SDAI	C++
Object	receiver

7.5.6.1.2 Scoped copy

The `ScopedCopy` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.8.8, Scoped copy.

```
Scope_ptr ScopedCopy(Model_contents_ptr& model);
};
```

SDAI input/output

SDAI	C++
Object	receiver
TargetModel	model
NewObject	return value

7.6 Aggregate and iterator classes

The SDAI aggregate data types and their operations specified in ISO 10303-22 shall be implemented by the C++ classes in this subclause. Within this subclause, when `<T>` represents an object of type ENTITY, the type `<T>` is used to represent `T_ptr` for each argument or return value in any member function. When `<T>` represents an instance of type `String_var`, `Binary_var`, `Number_var`, or `Select_var`, the type `<T>` is used to represent `T_var` for each argument or return value in any member function. When `<T>` represents an instance of type `Integer`, `Enumeration`, `Real`, `sdaiBool`, or `sdaiLogical`, the type `<T>` is used to represent `<T>` for each argument or return value in any member function.

7.6.1 Aggregate instance

The `Aggregate_instance` class shall implement the base class for the aggregate classes specified in this subclause.

```
class Aggregate_instance {
```

7.6.1.1 SDAI operation declaration

The SDAI operations specified in ISO 10303-22 for the SDAI `ENTITY` `aggregate_instance` shall be implemented by the following C++ member functions.

7.6.1.1.1 SDAI query

The `SDAIQuery` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.4.14, SDAI query.

```
public:
Integer SDAIQuery(const char* expression, const Object_ptr& theEntity,
T__sdailist_var& destination) const;
};
```

SDAI input/output

SDAI	C++
Source	receiver
Where	expression
Entity	theEntity
Result	destination
Count	return value

7.6.2 Array

The following class specifies the interface to an ARRAY with elements of type <T>.

```
class T__sdaiarray : public Aggregate_instance {

public:
T__sdaiarray();
```

This constructor shall create a new array with a lower index of zero and an upper index of zero.

```
T__sdaiarray(const T__sdaiarray&);
```

This constructor shall be equivalent in functionality to the constructor `T__array(const T__array&)`.

```
T__sdaiarray(Integer Upper, Integer Lower);
```

This constructor shall create a new array and set the lower and upper index values to agree with the EXPRESS bounds. The array shall be filled with unset values of type `<T>`.

```
T_sdaiarray(T__array& source);
```

This constructor shall create a new `T_sdaiarray` from `source`. The new `T__sdaiarray` shall point to the same objects as in `source`.

```
operator T__array() const;
```

This operator shall create a new `T__array` which references the same objects as those contained in the receiver.

```
private:
virtual ~T__sdaiarray();

public:
sdaiBool TestLower() const;
sdaiBool TestUpper() const;
private:
void lower_(Integer);
void upper_(Integer);
```

7.6.2.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS `ENTITY ARRAY` shall be implemented by the following C++ member functions.

7.6.2.1.1 Create aggregate instance by index

The `CreateAggrInstanceByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.11.5, Create aggregate instance by index.

```
public:
void CreateAggrInstanceByIndex(Integer ind, Aggregate_instance_var& newAggr);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind
NewAggregate	newAggr

7.6.2.1.2 Get member count

The `GetMemberCount` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.1, Get member count.

```
Integer GetMemberCount() const;
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Result	return value

7.6.2.1.3 Is member

The `IsMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.2, Is member.

```
sdaiBool IsMember(const <T>& anItem) const;
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	anItem
Result	return value

7.6.2.1.4 Get by index

The `GetByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.15.1, Get by index.

```
<T> GetByIndex(Integer ind);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind
Value	return value

7.6.2.1.5 Reindex array

The `Reindex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.18.3, Reindex array.

```
void Reindex();
```

SDAI input/output

SDAI	C++
Array	receiver

7.6.2.1.6 Reset array index

The `ResetIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.18.4, Reset array index.

```
void ResetIndex(Integer lower, Integer upper);
```

SDAI input/output

SDAI	C++
Array	receiver
Lower	lower
Upper	upper

7.6.2.1.7 Put by index

The `PutByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.16.1, Put by index.

```
void PutByIndex(Integer ind,const <T>& ele);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind
Value	ele

7.6.2.1.8 Test by index

The `TestByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.17.1, Test by index.

```
sdaiBool TestByIndex(Integer ind) const;
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind
Result	return value

7.6.2.1.9 Unset value by index

The `UnsetValueByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.18.1, Unset value by index.

```
void UnsetValueByIndex(Integer ind);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind

7.6.2.1.10 Unset value current member

The `UnsetValueCurrentMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.18.2, Unset value current member.

```
void UnsetValueCurrentMember(T__iterator_ptr& anIter);
```

SDAI input/output

SDAI	C++
Iterator	anIter

7.6.2.1.11 Create iterator

The `CreateIterator` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.3, Create iterator.

```
const T__iterator_ptr CreateIterator() const;
T__iterator_ptr CreateIterator();
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Iterator	return value

7.6.2.1.12 Delete iterator

The `DeleteIterator` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.4, Delete iterator.

```
void DeleteIterator(T__iterator_ptr& iter);
```

SDAI input/output

SDAI	C++
Iterator	iter

7.6.2.1.13 Get lower bound

The `GetLowerBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.9, Get lower bound.

```
Integer GetLowerBound();
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	return value

7.6.2.1.14 Get upper bound

The `GetUpperBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.10, Get upper bound.

```
Integer GetUpperBound();
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	return value

7.6.2.1.15 Get value bound by index

The GetValueBoundByIndex function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.15.4, Get value bound by index.

```
Integer GetValueBoundByIndex(Integer index);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	index
Value	return value

7.6.2.1.16 Get upper index

The `GetUpperIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.17.4, Get upper index.

```
Integer GetUpperIndex();
```

SDAI input/output

SDAI	C++
Array	receiver
Value	return value

7.6.2.1.17 Get lower index

The `GetLowerIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.17.3, Get lower index.

```
Integer GetLowerIndex();
};
```

SDAI input/output

SDAI	C++
Array	receiver
Value	return value

7.6.3 List

The following class specifies the interface to an EXPRESS LIST with elements of type <T>. An unset value shall be used to indicate open limits for an upper bound.

```
class T__sdailist : public Aggregate_instance {

public:
T__sdailist();
```

This constructor shall be equivalent in functionality to the constructor `T__list()`.

```
T__sdailist(const T__sdailist&);
```

This constructor shall be equivalent in functionality to the constructor `T__list(const T__list&)`.

```
T__sdailist(Integer Upper, Integer Lower);
```

This constructor shall be provide the functionality of the default constructor and set the lower and upper bound values to agree with the EXPRESS bounds.

```
T__sdailist(Integer Lower);
```

This constructor shall create an empty `T_sdailist` with lower bound set to `Lower`.

```
T_sdailist(T__list& source);
```

This constructor shall create a new `T_sdailist` from `source`. The new `T__sdailist` shall reference the same objects as in source.

```
operator T__list() const;
```

This operator shall create a new `T__list` which references the same objects as those contained in the receiver.

```
private:
virtual ~T__sdailist();

public:
sdaiBool TestUpper() const;
private:
void lower_(Integer);
void upper_(Integer);
```

7.6.3.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS `ENTITY LIST` shall be implemented by the following C++ member functions.

7.6.3.1.1 Create aggregate instance by index

The `CreateAggrInstanceByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.16.2, Create aggregate instance by index.

```
public:
void CreateAggrInstanceByIndex(Integer ind, Aggregate_instance_var&
 newAggr);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind
NewAggregate	newAggr

7.6.3.1.2 Get member count

The `GetMemberCount` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.1, Get member count.

```
Integer GetMemberCount() const;
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Result	return value

7.6.3.1.3 Is member

The `IsMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.2, Is member.

```
sdaiBool IsMember(const <T>& anItem) const;
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	anItem
Result	return value

7.6.3.1.4 Get by index

The `GetByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.15.1, Get by index.

```
<T> GetByIndex(Integer ind);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind
Value	return value

7.6.3.1.5 Put by index

The `PutByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.16.1, Put by index.

```
void PutByIndex(Integer ind,const <T>& ele);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind
Value	ele

7.6.3.1.6 Add by index

The `AddByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.19.3, Add by index.

```
void AddByIndex(Integer ind,const <T>& ele);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind
Value	ele

7.6.3.1.7 Add aggregate instance by index

The `AddAggregateInstanceByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.19.6, Add aggregate instance by index.

```
void AddAggregateInstanceByIndex(Integer ind,const Aggregate_instance_var&
aggr);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind
NewAggregate	aggr

7.6.3.1.8 Remove by index

The `RemoveByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.19.7, Remove by index.

```
void RemoveByIndex(Integer ind);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	ind

7.6.3.1.9 Create iterator

The `CreateIterator` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.3, Create iterator.

```
const T__iterator_ptr CreateIterator() const;
T__iterator_ptr CreateIterator();
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Iterator	return value

7.6.3.1.10 Delete iterator

The `DeleteIterator` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.4, Delete iterator.

```
void DeleteIterator(T__iterator_ptr& iter);
```

SDAI input/output

SDAI	C++
Iterator	iter

7.6.3.1.11 Get lower bound

The `GetLowerBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.9, Get lower bound.

```
Integer GetLowerBound();
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	return value

7.6.3.1.12 Get upper bound

The `GetUpperBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.10, Get upper bound.

```
Integer GetUpperBound();
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	return value

7.6.3.1.13 Get value bound by index

The `GetValueBoundByIndex` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.15.4, Get value bound by index.

```
Integer GetValueBoundByIndex(Integer index);
};
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Index	index
Value	return value

7.6.4 Set

The following class specifies the interface to an EXPRESS SET with elements of type <T>. When <T> represents an object of type ENTITY, the type T_ptr shall be used for each argument or return value in any member function. An unset value shall be used to indicate open limits for an upper bound.

```
class T__sdaiset : public Aggregate_instance {

public:
T__sdaiset();
```

This constructor shall be create an empty T_sdaiset.

```
T__sdaiset(const T__sdaiset&);
```

This constructor shall be equivalent in functionality to the constructor T__set(const T__set&).

```
T__sdaiset(Integer Lower, Integer Upper);
```

This constructor shall be create an empty T__sdaiset and set the lower and upper bound values to agree with the EXPRESS bounds.

```
T__sdaiset(Integer Lower);
```

This constructor shall create an empty T__sdaiset and set the lower bound value to agree with the EXPRESS lower bound. This constructor shall apply to unbounded sets only.

```
T_sdaiset(T__set& source);
```

This constructor shall create a new T_sdailset from source. The new T__sdaiset shall reference the same objects as in source.

```
operator T__set() const;
```

This operator shall create a new T__set which references the same objects as those contained in the receiver.

```
private:
virtual ~T__sdaiset();

public:
sdaiBool TestUpper() const;
private:
void lower_(Integer);
void upper_(Integer);
```

7.6.4.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS ENTITY SET shall be implemented by the following C++ member functions.

7.6.4.1.1 Create aggregate instance unordered

The `CreateAggrInstance` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.14.2, Create aggregate instance unordered.

```
public:
void CreateAggrInstance(Aggregate_instance_var& newAggr);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
NewAggregate	newAggr

7.6.4.1.2 Get member count

The `GetMemberCount` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.1, Get member count.

```
Integer GetMemberCount() const;
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Result	return value

7.6.4.1.3 Is member

The `IsMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.2, Is member.

```
sdaiBool IsMember(const <T>& anItem) const;
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	anItem
Result	return value

7.6.4.1.4 Add unordered

The `Add` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.14.1, Add unordered.

```
sdaiBool Add(const <T>& anItem);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	anItem

7.6.4.1.5 Remove unordered

The `Remove` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.14.3, Remove unordered.

`sdai`Bool Remove(const <T>& anItem);

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	anItem

7.6.4.1.6 Create iterator

The `CreateIterator` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.3, Create iterator.

```
const T__iterator_ptr CreateIterator() const;
T__iterator_ptr CreateIterator();
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Iterator	return value

7.6.4.1.7 Delete iterator

The `DeleteIterator` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.4, Delete iterator.

```
void DeleteIterator(T__iterator_ptr& iter);
```

SDAI input/output

SDAI	C++
Iterator	iter

7.6.4.1.8 Get lower bound

The `GetLowerBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.9, Get lower bound.

```
Integer GetLowerBound();
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	return value

7.6.4.1.9 Get upper bound

The `GetUpperBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.10, Get upper bound.

```
Integer GetUpperBound();
};
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	return value

7.6.5 Bag

The following class specifies the interface to an unbounded EXPRESS BAG with elements of type <T>. An unset value shall be used to indicate open limits for an upper bound.

```
class T__sdaibag : public Aggregate_instance {

public:
T__sdaibag();
```

This constructor shall create an empty `T__sdaibag`.

```
T__sdaibag(const T__sdaibag&);
```

This constructor shall be equivalent in functionality to the constructor `T__bag(const T__bag&)`.

```
T__sdaibag(Integer Lower, Integer Upper);
```

This constructor create an empty bag and set the lower and upper bound values to agree with the EXPRESS bounds.

```
T__sdaibag(Integer Lower);
```

This constructor create an empty `T__sdaibag` and set the lower bound value to agree with the EXPRESS lower bound. This constructor shall apply to unbounded bags only.

```
T_sdaibag(T__bag& source);
```

This constructor shall create a new `T_sdailist` from `source`. The new `T__sdailist` shall reference the same objects as in `source`.

```
operator T__sdaibag() const;
```

This operator shall create a new `T__bag` which references the same objects as those contained in the receiver.

```
private:
virtual ~T__sdaibag();

public:
sdaiBool TestUpper() const;
private:
void lower_(Integer);
void upper_(Integer);
```

7.6.5.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS `ENTITY BAG` shall be implemented by the following C++ member functions.

7.6.5.1.1 Create aggregate instance unordered

The `CreateAggrInstance` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.14.2, Create aggregate instance unordered.

```
public:
void CreateAggrInstance(Aggregate_instance_var& newAggr);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
NewAggregate	newAggr

7.6.5.1.2 Get member count

The `GetMemberCount` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.1, Get member count.

```
Integer GetMemberCount() const;
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Result	return value

7.6.5.1.3 Is member

The `IsMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.2, Is member.

```
sdaiBool IsMember(const <T>& anItem) const;
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	anItem
Result	return value

7.6.5.1.4 Add unordered

The `Add` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.14.1, Add unordered.

```
sdaiBool Add(const <T>& anItem);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	anItem

7.6.5.1.5 Remove unordered

The `Remove` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.14.3, Remove unordered.

```
sdaiBool Remove(const <T>& anItem);
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	anItem

7.6.5.1.6 Create iterator

The `CreateIterator` functions shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.3, Create iterator.

```
const T__iterator_ptr CreateIterator() const;
T__iterator_ptr CreateIterator();
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Iterator	return value

7.6.5.1.7 Delete iterator

The `DeleteIterator` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.4, Delete iterator.

```
void DeleteIterator(T__iterator_ptr& iter);
```

SDAI input/output

SDAI	C++
Iterator	iter

7.6.5.1.8 Get lower bound

The `GetLowerBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.9, Get lower bound.

```
Integer GetLowerBound();
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	return value

7.6.5.1.9 Get upper bound

The `GetUpperBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.10, Get upper bound.

```
Integer GetUpperBound();
};
```

SDAI input/output

SDAI	C++
Aggregate	receiver
Value	return value

7.6.6 Iterator

The following illustrates the interface to an iterator with elements of type <T>. When <T> represents an object of type ENTITY, the type T_ptr shall be used for each argument or return value in any member function.

```
class T__iterator {

protected:
T__iterator ();
public:
T__iterator (const T__iterator&);

Aggregate_instance_var subject_();
void subject_(Aggregate_instance_var);
<T> current_member_();
private:
void current_member_(const <T>&);
```

7.6.6.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS ENTITY iterator shall be implemented by the following C++ member functions.

7.6.6.1.1 Create aggregate instance as current member

The CreateAggrInstanceAsMember function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.13.1, Create aggregate instance as current member.

```
public:
Aggregate_instance_var CreateAggrInstanceAsCurrentMember();
```

SDAI input/output

SDAI	C++
Iterator	receiver
NewAggregate	return value

7.6.6.1.2 Create aggregate instance before current member

The CreateAggrInstanceBeforeCurrentMember function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.19.4, Create aggregate instance before current member.

```
Aggregate_instance_var CreateAggrInstanceBeforeCurrentMember();
```

SDAI input/output

SDAI	C++
Iterator	receiver
NewAggregate	return value

7.6.6.1.3 Create aggregate instance after current member

The `CreateAggrInstanceAfterCurrentMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.19.5, Create aggregate instance after current member.

```
Aggregate_instance_var CreateAggrInstanceAfterCurrentMember();
```

SDAI input/output

SDAI	C++
Iterator	receiver
NewAggregate	return value

7.6.6.1.4 Beginning

The `Beginning` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.5, Beginning.

```
void Beginning();
```

SDAI input/output

SDAI	C++
Iterator	receiver

7.6.6.1.5 Next

The `Next` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.6, Next.

```
sdaiBool Next();
```

SDAI input/output

SDAI	C++
Iterator	receiver
Result	return value

7.6.6.1.6 Get current member

The `GetCurrentMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.7, Get current member.

```
<T> GetCurrentMember();
```

SDAI input/output

SDAI	C++
Iterator	receiver
Value	return value

7.6.6.1.7 Put current member

The `PutCurrentMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.13.2, Put current member.

```
void PutCurrentMember(const <T>& anItem);
```

SDAI input/output

SDAI	C++
Iterator	receiver
value	return value

7.6.6.1.8 Remove current member

The `RemoveCurrentMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.13.3, Remove current member.

```
sdaiBool RemoveCurrentMember();
```

SDAI input/output

SDAI	C++
Iterator	receiver
Result	return value

7.6.6.1.9 Previous

The `Previous` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.15.3, Previous.

```
sdaiBool Previous();
```

SDAI input/output

SDAI	C++
Iterator	receiver
Result	return value

7.6.6.1.10 End

The `End` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.15.2, End.

```
void End();
```

SDAI input/output

SDAI	C++
Iterator	receiver

7.6.6.1.11 Add before current member

The `AddBeforeCurrent` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.19.1, Add before current member.

```
void  AddBeforeCurrentMember(<T>& anItem);
```

SDAI input/output

SDAI	C++
Iterator	receiver
Value	anItem

7.6.6.1.12 Add after current member

The `AddAfterCurrentMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.19.2, Add after current member.

```
void AddAfterCurrentMember(<T>& anItem);
```

SDAI input/output

SDAI	C++
Iterator	receiver
Value	anItem

7.6.6.1.13 Get value bound by iterator

The `GetValueBound` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.12.8, Get value bound by iterator.

```
Integer GetValueBound();
```

SDAI input/output

SDAI	C++
Iterator	receiver
Value	return value

7.6.6.1.14 Test current member

The `TestCurrentMember` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.17.2, Test current member.

```
sdaiBool TestCurrentMember();
};
```

SDAI input/output

SDAI	C++
Iterator	receiver
Result	return value

7.7 Dictionary Classes

The following classes implement the EXPRESS ENTITYs and TYPE data types specified in ISO 10303-22 as described by 6.3.

```
#ifdef SDAI_CPP_LATE_BINDING

typedef String_var Express_id;
typedef String_var Info_object_id;
```

7.7.1 Named_type

The `Named_type` class shall implement the SDAI entity `named_type` specified in ISO 10303-22 as described by 6.4.10.

```
class Named_type : public Dictionary_instance {

friend class Session;
```

NOTE Session is a friend so that Named_type instances may be constructed only by member functions of Session.

```
protected:
Named_type();
Named_type(const Named_type&);

public:
virtual TypeCode_ptr Type() const = 0;
Express_id name_() const;
const Where_rule__list_var where_rules_() const;
const Schema_ptr parent_schema_() const;
```

```
static Named_type_ptr _duplicate(Named_type_ptr);
static Named_type_ptr _narrow(Object_ptr);
static Named_type_ptr _nil();

private:
void name_(const Express_id&);
void where_rules_(const Where_rule__list_var&);
void parent_schema_(const Schema_ptr&);
};
```

7.7.2 Dictionary_instance

The `Dictionary_instance` class shall implement the base class for all C++ classes which represent entity data types from the SDAI dictionary schema.

```
class Dictionary_instance : public DAObject_SDAI {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of Dictionary_instance instances.

```
protected:
Dictionary_instance();
Dictionary_instance(const Dictionary_instance&);

public:
static Dictionary_instance_ptr _duplicate(Dictionary_instance_ptr);
static Dictionary_instance_ptr _narrow(Object_ptr);
static Dictionary_instance_ptr _nil();
};
```

7.7.3 Schema_definition

The `Schema` class shall implement the SDAI entity `schema_definition` specified in ISO 10303-22 as described by 6.4.1.

```
class Schema : public Dictionary_instance {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Schema instances.

```
protected:
Schema();
Schema(const Schema&);

public:
const Express_id name_() const;
const Info_object_id identification_() const;
const Entity__set_var entities_() const;
const Defined_type__set_var types_() const;
const Global_rule__set_var global_rules_() const;
const External_schema__set_var external_schemas_() const;
```

```
static Schema_ptr _duplicate(Schema_ptr);
static Schema_ptr _narrow(Object_ptr);
static Schema_ptr _nil();

private:
void name_(const Express_id&);
void identification_(const Info_object_id&);
void entities_(const Entity__set_var&);
void types_(const Defined_type__set_var&);
void global_rules_(const Global_rule__set_var&);
void external_schemas_(const External_schema__set_var&);
```

7.7.3.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS `ENTITY schema_definition` shall be implemented by the following C++ member functions.

7.7.3.1.1 NType

The `NType` function shall return the handle to the type and entity in the schema for the Named_type with the name `typeName`.

```
public:
const Named_type_ptr NType(const char* typeName) const;
```

Output

This function shall return the `Named_type` handle for the `typeName` in the receiver. Otherwise the function shall return a null handle.

Possible error indicators

```
sdaiTY_NDEF             // Type undefined
sdaiFN_NAVL             // Function not implemented
sdaiSY_ERR              // Underlying system error
```

Origin

Convenience function

7.7.3.1.2 Is valid data type

The `IsValidDataType` function shall determine if all types specified in `typeSet` are defined in the receiver.

```
sdaiBool IsValidDataType(const Entity_type_set_ptr& typeSet) const;
```

Output

This function shall return TRUE if all types specified by `typeSet` are defined in the receiver. This function shall return FALSE if one or more are not.

Possible error indicators

```
sdaiTY_NDEF             // Type unknown
sdaiSY_ERR              // Underlying system error
```

Origin

Convenience function

7.7.3.1.3 Get complex entity definition

The `GetComplexEntity` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.9.1, Get complex entity definition.

```
const Entity_ptr GetComplexEntity(const Entity__list_var& typeSet) const;
};
```

SDAI input/output

SDAI	C++
Types	typeSet
Complex	return value

7.7.4 Defined_type

The `Defined_type` class shall implement the SDAI entity `defined_type` specified in ISO 10303-22 as described by 6.4.11.

```
class Defined_type : public Named_type {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Defined_type instances.

```
protected:
Defined_type();
Defined_type(const Defined_type&);

public:
virtual TypeCode_ptr Type() const;
const Underlying_type_ptr domain_() const;

static Defined_type_ptr _duplicate(Defined_type_ptr);
static Defined_type_ptr _narrow(Object_ptr);
static Defined_type_ptr _nil();

private:
void domain_(const Underlying_type_ptr&);
};
```

7.7.5 Entity_definition

The `Entity` class shall implement the SDAI entity `entity_definition` specified in ISO 10303-22 as described by 6.4.12.

```
class Entity : public Named_type {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Entity instances.

```
protected:
Entity();
Entity(const Entity&);

public:
virtual TypeCode_ptr Type() const;
const Entity__list_var supertypes_() const;
sdaiBool complex_() const;
sdaiBool instantiable_() const;
sdaiBool independent_() const;
const Attribute__set_var attributes_() const;
const Uniqueness_rule__set_var uniqueness_rules_() const;
const Global_rule__set_var global_rules_() const;

static Entity_ptr _duplicate(Entity_ptr);
static Entity_ptr _narrow(Object_ptr);
static Entity_ptr _nil();

private:
void supertypes_(const Entity__list_var&);
void complex_(sdaiBool);
void instantiable_(sdaiBool);
void independent_(sdaiBool);
void attributes_(const Attribute__set_var&);
void uniqueness_rules_(const Uniqueness_rule__set_var&);
void global_rules_(const global_rule__set_var&);
```

7.7.5.1 SDAI operation declarations

The SDAI operations specified in ISO 10303-22 for the EXPRESS `ENTITY` `entity_definition` shall be implemented by the following C++ member functions.

7.7.5.1.1 Is domain equivalent with

The `IsDomainEquivalentWith` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.9.4, Is domain equivalent with.

```
public:
sdaiBool IsDomainEquivalentWith(const Entity_definition_ptr& anEnt) const;
```

SDAI input/output

SDAI	C++
Type	receiver
CompType	anEnt
Result	return value

7.7.5.1.2 Is subtype of

The `IsSubtypeOf` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.9.2, Is subtype of.

```
sdaiBool IsSubtypeOf(const Entity_definition_ptr& theEntity) const;
```

SDAI input/output

SDAI	C++
Type	receiver
CompType	theEntity
Result	return value

7.7.5.1.3 Is SDAI subtype of

The `IsSDAISubtypeOf` function shall implement the SDAI operation specified in ISO 10303-22 as described by 10.9.3, Is SDAI subtype of. If the type defined by the receiver is an application entity and the type defined by `theEntity` is an application instance, this function shall return TRUE; TY_SCHEMA shall not be indicated even though the types are not in the same schema.

```
sdaiBool IsSDAISubtypeOf(const Entity_definition_ptr& theEntity) const;
};
```

SDAI input/output

SDAI	C++
Type	receiver
CompType	theEntity
Result	return value

7.7.6 Attribute

The `Attribute` class shall implement the SDAI entity `attribute` specified in ISO 10303-22 as described by 6.4.13.

```
class Attribute : public Dictionary_instance {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Attribute instances.

```
protected:
Attribute();
Attribute(const Attribute&);

public:
Express_id name_() const;
const Entity__list_var parent_entity_();

static Attribute_ptr _duplicate(Attribute_ptr);
static Attribute_ptr _narrow(Object_ptr);
static Attribute_ptr _nil();

private:
void name_(const Express_id&);
void parent_entity_(const Entity__list_var&);
};
```

7.7.7 Derived_attribute

The `Derived_attribute` class shall implement the SDAI entity `derived_attribute` specified in ISO 10303-22 as described by 6.4.14.

```
class Derived_attribute : public Attribute {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Derived_attribute instances.

```
protected:
Derived_attribute();
Derived_attribute(const Derived_attribute&);

public:
const Base_type_ptr domain_() const;
const Explicit_or_derived_ptr redeclaring_() const;

static Derived_attribute_ptr _duplicate(Derived_attribute_ptr);
static Derived_attribute_ptr _narrow(Object_ptr);
static Derived_attribute_ptr _nil();

private:
void redeclaring_(Explicit_or_derived_ptr&);
void domain_(const Base_type_ptr&);
};
```

7.7.8 Explicit_attribute

The `Explicit_attribute` class shall implement the SDAI entity `explicit_attribute` specified in ISO 10303-22 as described by 6.4.15.

```
class Explicit_attribute : public Attribute {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Explicit_attribute instances.

```
protected:
Explicit_attribute();
Explicit_attribute(const Explicit_attribute&);

public:
const Base_type_ptr domain_() const;
const Explicit_attribute_ptr redeclaring_() const;
sdaiBool optional_flag_() const;

static Explicit_attribute_ptr _duplicate(Explicit_attribute_ptr);
static Explicit_attribute_ptr _narrow(Object_ptr);
static Explicit_attribute_ptr _nil();

private:
void domain_(const Base_type_ptr&);
void redeclaring_(const Explicit_attribute_ptr&);
void optional_flag_(sdaiBool);
};
```

7.7.9 Inverse_attribute

The `Inverse_attribute` class shall implement the SDAI entity `inverse_attribute` specified in ISO 10303-22 as described by 6.4.16.

```
class Inverse_attribute : public Attribute {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Inverse_attribute instances.

```
protected:
Inverse_attribute();
Inverse_attribute(const Inverse_attribute&);

public:
const Entity_ptr domain_();
const Inverse_attribute_ptr redeclaring_() const;
const Explicit_attribute_ptr inverted_attr_() const;
Integer min_cardinality_() const;
Integer max_cardinality_() const;
sdaiBool duplicates_() const;

static Inverse_attribute_ptr _duplicate(Inverse_attribute_ptr);
static Inverse_attribute_ptr _narrow(Object_ptr);
static Inverse_attribute_ptr _nil();

private:
void domain_(const Entity_ptr&);
void redeclaring_(const Inverse_attribute_ptr&);
void inverted_attr_(const Explicit_attribute_ptr&);
void min_cardinality_(Integer);
```

```
void max_cardinality_(Integer);
void duplicates_(sdaiBool);
};
```

7.7.10 Uniqueness_rule

The `Uniqueness_rule` class shall implement the SDAI entity `uniqueness_rule` specified in ISO 10303-22 as described by 6.4.17.

```
class Uniqueness_rule : public Dictionary_instance {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Uniqueness_rule instances.

```
protected:
Uniqueness_rule();
Uniqueness_rule(const Uniqueness_rule&);

public:
const Express_id label_() const;
const Attribute__list_var attributes_() const;
const Entity__list_var parent_entity_() const;

static Uniqueness_rule_ptr _duplicate(Uniqueness_rule_ptr);
static Uniqueness_rule_ptr _narrow(Object_ptr);
static Uniqueness_rule_ptr _nil();

private:
void label_(const Express_id&);
void attributes_(const Attribute__list_var&);
void parent_entity_(const Entity__list_var&);
};
```

7.7.11 Where_rule

The `Where_rule` class shall implement the SDAI entity `where_rule` specified in ISO 10303-22 as described by 6.4.18.

```
class Where_rule : public Dictionary_instance {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Where_rule instances.

```
protected:
Where_rule();
Where_rule(const Where_rule&);

public:
Express_id label_() const;
const Type_or_rule_var parent_item_() const;
```

```
static Where_rule_ptr _duplicate(Where_rule_ptr);
static Where_rule_ptr _narrow(Object_ptr);
static Where_rule_ptr _nil();

private:
void label_(const Express_id&);
void parent_item_(const Type_or_rule_var&);
};
```

7.7.12 Enumeration_type

The `Enumeration_type` class shall implement the SDAI entity `enumeration_type` specified in ISO 10303-22 as described by 6.4.28.

```
class Enumeration_type : public Dictionary_instance {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Enumeration_type instances.

```
protected:
Enumeration_type();
Enumeration_type(const Enumeration_type&);

public:
virtual TypeCode_ptr Type() const;
const Express_id__list_var elements_() const;

static Enumeration_type_ptr _duplicate(Enumeration_type_ptr);
static Enumeration_type_ptr _narrow(Object_ptr);
static Enumeration_type_ptr _nil();

private:
void elements_(const Express_id__list_var&);
};
```

7.7.13 Select_type

The `Select_type` class shall implement the SDAI entity `select_type` specified in ISO 10303-22 as described by 6.4.29.

```
class Select_type : public Dictionary_instance {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Select_type instances.

```
protected:
Select_type();
Select_type(const Select_type&);
```

```
public:
virtual TypeCode_ptr Type() const;
const Named_type__set_var selections_() const;

static Select_type_ptr _duplicate(Select_type_ptr);
static Select_type_ptr _narrow(Object_ptr);
static Select_type_ptr _nil();

protected:
void selections_(const Named_type__set_var&);
};
```

7.7.14 Global_rule

The `Global_rule` class shall implement the SDAI entity `global_rule` specified in ISO 10303-22 as described by 6.4.19.

```
class Global_rule : public Dictionary_instance {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Global_rule instances.

```
protected:
Global_rule();
Global_rule(const Global_rule&);

public:
Express_id name_() const;
const Entity__set_var entities_() const;
const Where_rule__list_var where_rules_() const;
const Schema_ptr parent_schema_() const;

static Global_rule_ptr _duplicate(Global_rule_ptr);
static Global_rule_ptr _narrow(Object_ptr);
static Global_rule_ptr _nil();

private:
void name_(const Express_id&);
void entities_(const Entity__set_var&);
void where_rules_(const Where_rule__list_var&);
void parent_schema_(const Schema_ptr&);
};
```

7.7.15 Simple_type

The `Simple_type` class shall implement the SDAI entity `simple_type` specified in ISO 10303-22 as described by 6.4.20.

```
class Simple_type : public Dictionary_instance {

friend class Session;
```

NOTE - Session is a friend so that Session may construct and access the private members of the Simple_type instances.