ESPRIT Project 2072 ECIP2

European CAD Integration Project

Issued by the WP1 Team

Prepared by
Wolfgang Mueller, Cadlab,
Georg Lehrenfeld, Paderborn University,
Norbert Wiechers, Cadlab

Validation of EXPRESS Models (Preliminary Report)

Deliverable R:C1.6.2.51.B (Row 19)

ECIP2/PU/017-1 July 31, 1993

Title of Deliverable	Validation of EXPRESS Models (Preliminary Report)
Deliverable No.	R:C1.6.2.51.B (Row 19)
Brief Description	This report gives first an introduction to ISO10303-11 and ISO10303-21 dedicated to the problem domain of conformance checking. Different classes of conformance checks are identified in the second part. The last part discusses the implementation of a checker.
Partner(s) Responsible	Cadlab - Paderborn University
Estimated work content	1.75 man-month
Date	July 7, 1993

Workpackage Partner	Name of reviewer	Review Comments

Contents

1 Introduction

EXPRESS is an object oriented data specification language which is currently under standardisation by the ISO. It has currently been accepted as a Draft International Standard (DIS) [ISO10303-11] by the beginning of 1993. The final agreement on the standard is expected by the beginning of 1994.

Typical EXPRESS information models cover 50 up to 300 entity specifications defining very complex super/subtype inheritance structures. Though inheritance eases the writing of models, it is not always obvious whether an EXPRESS model serves its intended data specification. We denote the checking of EXPRESS models with respect to its intended specification as *validation* in this report. Concerning validation of EXPRESS models we can classify two kinds of validation techniques:

1. solely checking of EXPRESS models
2. defining instances for EXPRESS models

The first technique is a preliminary check concerning EXPRESS specifications only. This is mainly to evaluate the super/subtype hierarchy in order to get a "flattened" view of EXPRESS models. This permits a first visual inspection of the compositions of attributes for each entity.

```
TYPE date = ARRAY[1:3]  OF INTEGER; END_TYPE;

TYPE hair_type = ENUMERATION OF (blonde,brown); END_TYPE;

ENTITY person;
    first_name : STRING;
    birthdate  : date;
    hair       : OPTIONAL hair_type;
    children   : SET [0:?] OF person;
END_ENTITY;
```

Figure 1: EXPRESS Specification

When a set of real instances exists, the user check them when evaluating the correspondence with the EXPRESS specification.

We can distinguish

- interactive checkers
- batch checkers

Interactive checkers provide forms which can be filled in by a user. Forms can be represented by widgets within an X11 window environment, for instance (see [Sauder93]). Interactive validation means to perform checks incrementally on a particular EXPRESS model. Interactive definition of instances are best suited to

apply simple checks that do not concern consistency checks between instances. This is because the definition of an instance might result in an inconsistent state due to cardinality constraints and local/global consistency rules, for instance. Thus, it is more appropriate to perform full conformance checks by *batch checkers* which run their validation on an existing set of instances.
There are two possibilities to define instances in terms of lexical descriptions. The first format is the so-called *Physical File Format* which has already been accepted as a DIS [ISO10303-21]. The second format is called *EXPRESS-I* [Wilson92] which has been introduced to ISO 10303 as a Working Document in 1992. The main difference between both is that ISO 103033-21 presumes a flattened super/subtype relationship. Additionally, in this format the attribute names are omitted in order to reduce the size of files. EXPRESS-I is a direct representation of the hierarchical super/subtype structure containing also the identifiers of the corresponding attributes. Given the EXPRESS specification in Figure 1, Figure 2 depicts the corresponding EXPRESS-I definition and the equivalent ISO10303-21 instance. In this report we ignore EXPRESS-I since we want to concentrate on the validation in the context of product data exchange.

```
/* ISO 10303-21 Instance for ENTITY person */
                      #2=PERSON('W.',(11,10,60),.blonde.,(#4,#5,#6));

(* EXPRESS-I Instance for ENTITY person *)
                      d1 = date{(11,10,60)};
                      e1 = hair_type{!blonde};

                      p2 = person{ first_name --> 'W.';
                                   birthdate  --> @d1;
                                   hair       --> @e1;
                                   children   --> (@p4,@p5,@p6);};
```

Figure 2: ISO 10303-21 vs. EXPRESS-I

In the following sections we briefly introduce EXPRESS especially dedicated to the evaluation of super/subtype relationships. Additionally, we introduce the STEP physical file format as it is defined in [ISO10303-21]. Thereafter we identify the different classes of validation checks which have to be performed by a full conformance test. These tests are explained giving EXPRESS and ISO 10303-21 instances presuming an already flattened super/subtype structure. Finally we give some preliminary outlines on the implementation of a validation suite.

2 EXPRESS

The EXPRESS language is an object-oriented data specification language. An EXPRESS model consists of a set of schema declarations. A schema defines a common scope for a collection of data type declarations. A data type may be a simple type (number, real, integer, boolean, logical, string), an enumeration type, a select type, a user defined type, or an entity type. An entity type definition is a set of attribute definitions which include the definition of relationships. EXPRESS provides the definition of constraints on attribute values, on entities, and between entities by the means of a procedural language. Cardinality constraints may be defined on all kind of data types. Local rules define constraints local to the scope of data types. Global rules define constraints between multiple entities. In addition, attributes may be functionally related by so-called derived attribute declarations.

Entities may inherit attributes and constraints from other entities that are defined as their supertypes. On this super/subtype hierarchy the specification of constraints (*ONEOF*, *AND*, *ANDOR*) on combinations of subtypes is allowed. A supertype may be defined to be *ABSTRACT*. This means that no instance of this entity type is allowed.

We give some outlines on how to evaluate super/subtype compositions by the following example which is a simplified excerpt from the EXPRESS DIS language reference manual [ISO10303-11]. Figure 3 gives the corresponding EXPRESS-G diagram[1].

```
SCHEMA example;
   TYPE hair_type = ENUMERATION OF (blonde, black, brown, white); END_TYPE;
   TYPE date      = ARRAY [1:3] OF INTEGER; END_TYPE;

   ENTITY person SUPERTYPE OF (ONEOF(female, male));
     first_name : STRING;
     last_name  : STRING;
     nickname   : OPTIONAL STRING;
     birth_date : date;
     hair       : hair_type;
     age        : INTEGER;
     children   : SET [0:?] OF person;
   INVERSE
     parents    : SET [0:2] OF person FOR children;
   END_ENTITY;

   ENTITY female SUBTYPE OF (person);
     husband     : OPTIONAL male;
     maiden_name : OPTIONAL STRING;
   END_ENTITY;

   ENTITY male SUBTYPE OF (person);
     wife : OPTIONAL female;
   END_ENTITY;
END_SCHEMA;
```

[1]Note here, that in Figure 3 the enumeration items are represented explicitly which is an extension of ISO 10303-11 Appendix D

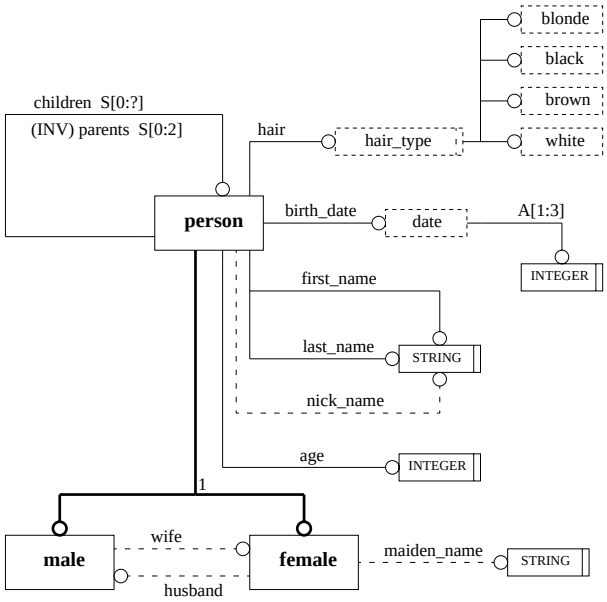

Figure 3: EXPRESS-G EXAMPLE

The above example specifies a very simple super/subtype hierarchy. Thus, the intended composition of attributes is directly visible. The algorithm how to compose inherited attributes is described in the ISO/DIS 10303-11 appendix B[2]. For the above example this algorithm evaluates the following set of allowed instantiations: $R = [person, person\&female, person\&male]$. This means, that entity *person* can be instantiated on its own. Furthermore, an instantiation of entity $female$ and entity *male* is allowable. When instantiating $female$ or *male* respectively, their private attributes are appended to the (inherited) attributes of entity *person* which is denoted by the "&".
Considering this simple example, it is obvious which attributes are inherited and which instantiations are allowable. But concerning complex hierarchical structures, containing *abstract supertypes*, multiple inheritance and the use of three different kinds of constraints ($ONEOF$, AND, $ANDOR$), constraints may cause completely unexpected results. Given a complex EXPRESS model and the algorithm defined by ISO/DIS 10303-11 Appendix B, it is not possible to check without the support of tools whether the resulting EXPRESS specification meets the intended specification. This is mainly, since the algorithm is described in terms of partly complex computation steps. These complex steps result from the complex semantics of super/subtype constraints.
We sketch the problems by giving some examples. First consider the following example.

[2] The inheritance defined in ISO/DIS 10303-11 is still subject of discussion. WG5 has decided to modify appendix B (Meeting of ISO TC184/SC4/WG5 Languages Project, Atlanta, June 1993).

```
ENTITY a SUPERTYPE OF(ONEOF(b, c)); END_ENTITY;
ENTITY b SUBTYPE OF(a);             END_ENTITY;
ENTITY c SUBTYPE OF(a);             END_ENTITY;
ENTITY d SUBTYPE OF(b, c);          END_ENTITY;
```

The allowable set of instantiations is: $R_1 = [a, a\&b, a\&c, a\&b\&c\&d]$. Please note here, that the instantiation of $a\&b\&c\&d$ is valid though the modeller's intend might have been not to allow the attribute set of b and the attribute set of c included within one instantiation: `SUPERTYPE OF(ONEOF(b, c))`.
Changing the supertype constraint in the above example from `ONEOF` to `ANDOR` results in $R_2 = [a, a\&b, a\&c, a\&b\&c, a\&b\&c\&d]$ and changing `ONEOF` to `AND` results in a different set of instantiations: $R_3 = [a, a\&b\&c, a\&b\&c\&d]$. Defining entity a as an abstract supertype removes a from R_2 or from R_1 respectively.

Our above example depicts the problem of unexpected results concerning a very simple example covering 4 entities only. But concerning real application this might cause real problems. Taking the example 144 from [ISO10303-11] covering 9 entities (see Figure 4) for instance, 19 different compositions of these entity attributes are allowed. This shows that a visual inspection, even if it is aided by tools, is not easy to perform. Thus, sometimes it seems to be more appropriate to apply samples of real instances to the EXPRESS model which is discussed in the remainder of this paper.

```
ENTITY a SUPERTYPE OF(ONEOF(b, c)AND (d ANDOR f);END_ENTITY;
ENTITY b SUBTYPE OF(a);                          END_ENTITY;
ENTITY c SUBTYPE OF(a);                          END_ENTITY;
ENTITY d ABSTRACT SUPERTYPE OF(ONEOF(k, l));     END_ENTITY;
ENTITY f SUBTYPE OF(a, z);                       END_ENTITY;
ENTITY k SUBTYPE OF(d);                          END_ENTITY;
ENTITY l SUBTYPE OF(d);                          END_ENTITY;
ENTITY y SUBTYPE OF(z);                          END_ENTITY;
ENTITY z;                                        END_ENTITY;
```

Figure 4: EXAMPLE 144 of [ISO10303-11]

3 The Physical File Format

Before discussing the classification of full conformance tests including some examples we first give a brief introduction to the physical file format the later checks are based on.
The specification of the STEP physical file format (or short: STEP file) is defined in [ISO10303-21]. Each STEP file refers to one EXPRESS schema. A STEP file contains an enumeration of instances. Each instance has a unique identifier. This identifier is an integer preceded by a "#". Valid identifiers range from 1 to 999.999.999. Each instance definition is a type specification delimited by a semicolon. The type identifier has to match an entity identifier of the corresponding EXPRESS schema. The entity type specification includes the enumeration of items which may be instance identifiers, simple items (integer, real, etc.), select type specifications, and lists of items, which can be an enumeration of lists or items again. Additionally there are two particular symbols, one for indicating an empty optional value ($) and a placeholder for attributes which are redeclared when inherited (*). Figure 5 gives a list of valid STEP symbols.

	Value Items
Instance Identifier	#3
Type Specification	MY_TYPE(...)
List	(a1, A2, a3)
Integer	1234
Real	9.432
String	'ABC'
Boolean Values	.F., .T.
Logical Values	.F., .T., .U.
Binary	"2EC"
Enumeration Item	.BLONDE.
Empty value	$
Redefined Attribute	*

Figure 5: ISO 10303-21 Basic Components

The listing of values has to keep a particular order. This order is given by the order of attribute declarations within the corresponding entity specification of a corresponding EXPRESS schema. Thus, for example the first item within the enumerated items of the instance corresponds to the first declared attribute of an EXPRESS entity, the second item to the second attribute declaration, etc.. This implies that the number of enumerated values within one instance has to be equal to the number of declared attributes within the corresponding entity. For this reason the language introduces a special symbol ($) to represent empty values in the case where the attribute is declared to be optional.
EXPRESS aggregates, like sets, bags, arrays, and lists, are all represented by

lists. Derived, inverse attributes as well as all executable specifications like rules, procedures, and functions are not mapped to the STEP file.
Inherited attributes appear prior to the explicit attributes of the entity if there exists a *ONEOF* relationship to the supertype. This is called *internal mapping* by [ISO10303-21]. We exploit this by the following examples. In the case of internal mapping consider the following EXPRESS specification.

```
TYPE hair_type = ENUMERATION OF (blonde, brown); END_TYPE;

ENTITY person SUPERTYPE OF (ONEOF(female, male));
  name       : STRING;
  hair       : hair_type;
  children   : SET [0:?] OF person;
END_ENTITY;

ENTITY female SUBTYPE OF (person);
  husband     : OPTIONAL male;
END_ENTITY;

ENTITY male SUBTYPE OF (person);
  wife : OPTIONAL female;
END_ENTITY;
```

The following samples are valid instances due to the above EXPRESS specification. In order to make the interpretation given by the above EXPRESS specification visible, we have included comments which denote the corresponding attribute name.

```
/*-------------------------------------------*/
/*            name      hair    children     */
/*              v         v         v         */
#1 = PERSON('A.Maier',.blonde.,    ()   );
#2 = PERSON('B.Mayer',.brown., (#11,#20));
/*-------------------------------------------*/
/*           Internal Mappings               */
/*-------------------------------------------*/
/*       wife    name      hair    children   */
/*         v      v         v          v      */
#10 = MALE($,   'C.Mayar',.blonde.,    ()   );
#11 = MALE(#21, 'D.Mayer',.brown.,     (#10));
/*-------------------------------------------*/
/*      husband    name      hair    children */
/*          v      v         v          v     */
#20 = FEMALE($,   'E.Mayer',.blonde.,    ()    );
#21 = FEMALE(#11, 'F.Mayar',.brown.,    (#10)  );
```

Attributes which are inherited via an *AND* or an *ANDOR* constraint may be composed by so-called *external mappings* if the order for composing the attributes is not unique. Otherwise they are composed by an *internal mapping*. In the case of an *external mapping* the instance is a set of several entity type specifications. The order in which these specifications are enumerated is not important.

For external mapping consider the following EXPRESS specification.

```
ENTITY person SUPERTYPE OF (employee ANDOR student);
  name       : STRING;
END_ENTITY;

ENTITY employee SUBTYPE OF (person);
  salary     : integer;
END_ENTITY;

ENTITY student SUBTYPE OF (person);
  s_id       : OPTIONAL Integer;
END_ENTITY;
```

The following samples are valid external and internal mappings corresponding to the above EXPRESS specification.

```
/*-------------------*/
/*            name   */
/*              v    */
#1 = PERSON('A.Maier');
/*----------------------------------------*/
/*          Internal Mappings             */
/*----------------------------------------*/
/*            name       s_id             */
/*              v         v               */
#2 = STUDENT('D.Meiar',332219);
/*----------------------------------------*/
/*            name      salary            */
/*              v         v               */
#3 = EMPLOYEE('E.Meiar',332219);
/*-------------------------------------------------*/
/*                  External Mappings              */
/*-------------------------------------------------*/
/*            name             salary       s_id    */
/*              v                v            v     */
#4 = (PERSON('B.Meier')EMPLOYEE(12000)STUDENT($));
/*-------------------------------------------------*/
/*          s_id              name            salary */
/*            v                v                v    */
#5 = (STUDENT(221334)PERSON('C.Mayer')EMPLOYEE(12000));
```

4 Classification of Conformance Tests

In this section we give a classification of conformance tests in order to check the validity of a given set of ISO 10303-21 instances with respect to a given EXPRESS model. We classify these tests in order to separate them into independent subchecks which can be performed in parallel.

We can distinguish three basic classes of validation checks:

- Class A: Type Checks,
- Class B: Simple Rule Checks, and
- Class C: Complex Rule Checks.

These basic classes can be partitioned again into subclasses as it is depicted by Figure 6.

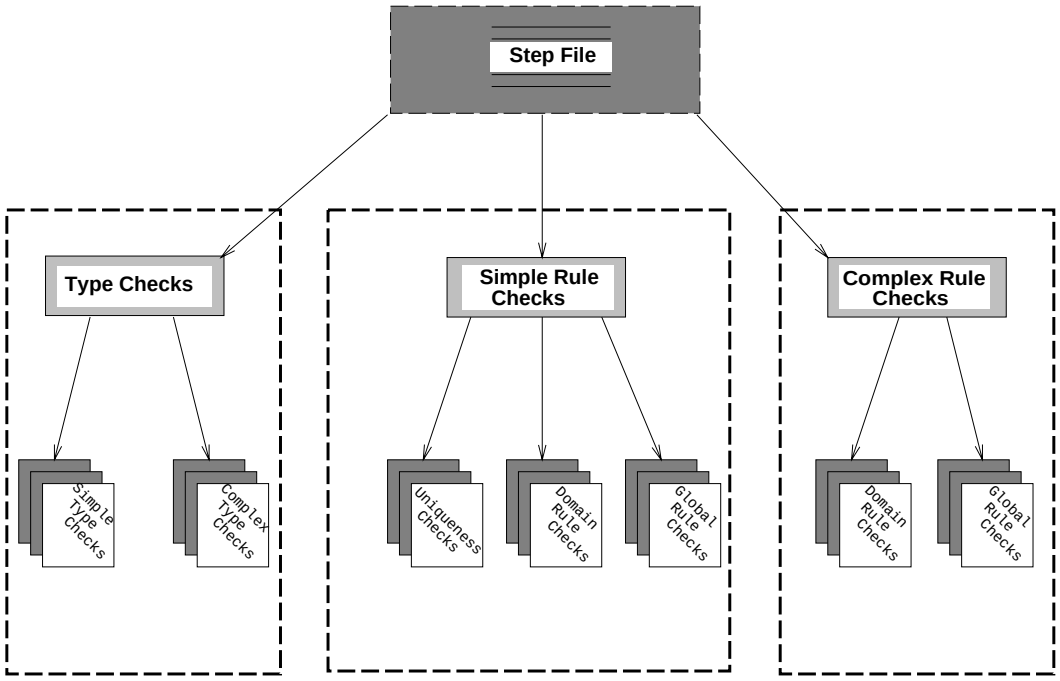

Figure 6: Partitioning a Full Validation into Subchecks

Concerning the following checks, we roughly distinguish simple and complex validation checks. Simple checks are checks which require a 1-pass analysis only. Complex checks require a 2-pass analysis of the STEP file which mainly results in a highly increased total number of data that have to be processed as a whole, e.g. cross reference checks.

We give some examples concerning the individual checks. For this we presume the following EXPRESS specification.

```
01          TYPE date = ARRAY[1:3]  OF INTEGER;
02               WHERE
03                  { 1 <= SELF[1] <= 31 };
04          END_TYPE;
05
06          TYPE hair_type = ENUMERATION OF (blonde,brown); END_TYPE;
07
08          ENTITY person SUPERTYPE OF (student ANDOR ONEOF(male, female));
09              name      : STRING (10) FIXED;
10              birthdate : date;
11              hair      : OPTIONAL hair_type;
12              children  : SET [1:?] OF person;
13          INVERSE
14              parents   : SET [0:2] OF person for children;
13          UNIQUE
14              un1: name, birthdate;
15          END_ENTITY;
16
17          ENTITY student SUBTYPE OF (person);
18              SELF\person.hair : hair_type;
19          END_ENTITY;
20
21          ENTITY female SUBTYPE OF (person);
22              husband     : OPTIONAL male;
23              maiden_name : OPTIONAL STRING;
24          WHERE
25            w1: (EXISTS(maiden_name) AND EXISTS(husband)) OR
26                NOT EXISTS(maiden_name);
27          END_ENTITY;
28
29          ENTITY male SUBTYPE OF (person);
30              wife : OPTIONAL female;
31          END_ENTITY;
32
33          RULE married FOR (female, male);
34          (* checks pairwise relationship between male and female *)
35          WHERE
36            r1 : SIZEOF(
37                   QUERY(tf <* female | EXISTS(tf.husband) AND
38                         (tf.husband.wife :<>: tf))
39                 ) = 0;
40          END_RULE;
```

4.1 Class A: Type Checks

We can identify simple type checks and complex type checks.

4.1.1 Simple Type Checks

Given an EXPRESS specification, simple type checking denotes the checking when concerning one instance at one time only. In details these are the following checks which are referred to as rules in the next example:

1. The number of STEP items has to match the number of declared EXPRESS attributes.
2. An external mapping has to be a valid composition of entity types.
3. Each value has to match the specified type (value/type checks) including that the empty value ($) is alternatively allowed in the enumeration

(required/option check) and the placeholders for redeclared attributes (*) appears at the right position.

4. Select attributes case-B&C have to be checked for their type compatibility (for a detailed introduction to case-A&B&C see [ISO10303-21]).

5. The number of list elements has to match the cardinality constraint of the corresponding EXPRESS aggregate definition.

6. Checking the size of fixed strings, fixed binaries, precision of reals. This also includes a check for the maximum string length of 32767 characters (see [ISO10303-21]).

7. Some lists have to be checked for the uniqueness of their elements. Lists of unique elements are "ARRAY/LIST OF UNIQUE" and "SET" specifications.

Examples of Class A.1 Checks:

Considering the following set of instances:

```
#1 = PERSON('Paul Maier',(30,5,60),$,(#2),$);
#2 = (FEMALE((),$)
      STUDENT('Paula Mair',(30,9,60),*,(#6),.blonde.));
#3 = PERSON(5,(30,7,60),.blonde.,(#4));
#4 = PERSON($,(30,8,60),.blonde.,(#5));
#5 = STUDENT('Paul Maier',(30,9,60),.blonde.,(#6),*);
#6 = PERSON('Paul Maier',(30,1,60),.blonde.,());
#7 = PERSON('A.Maier',(30,2,60),.blonde.,(#2));
#8 = PERSON('Paul Maier',(30,3,60),.blonde.,(#2,#3,#3));
#9 = STUDENT('Paul Maier',(30,4,60),*,(#6,#2),.brown.);
```

Instance #1 violates rule 1 since the instance has 5 elements but the EXPRESS model specifies 4 attributes only. Instance #2 is an invalid external mapping (rule 2) since an instance of person has to be included. The first value given in instance #3 is not a string (rule 3). Instance #4 is invalid since the first element is not defined to be optional (rule 3). In instance #5 the symbols ".blonde." and "*" have to be exchanged in order to be valid (rule 3). Instance #6 is in conflict with the cardinality constraint (rule 5). This means, the list of children has to contain at least 1 element. Instance #7 has an invalid first element since only a constant string with 10 characters (including blanks) is allowed (rule 6). Instance #8 conflicts with rule 7 since the list of children is defined to be a set which requires unique elements. Instance #9 is a valid instantiation due to the above EXPRESS specification.

4.1.2 Complex Type Checks

Complex type checking denotes the checking in the case that the check ranges over a set of instances. In details these are the following checks:

1. Uniqueness of instance identifiers

2. Type checks on explicit cross references between entities have to be performed. This is, whether references to other instances are type compatible to the attribute declaration.

3. Select attribute case-A checks (see [ISO10303-21]).

4. Type cardinality checks of INVERSE relationships. When the INVERSE relationship is defined to be a set this also includes the check whether the referred elements are unique. This uniqueness check has only to be performed if the relationship is not defined as "SET" or "ARRAY/LIST OF UNIQUE" otherwise the uniqueness of set elements of inverse relationships is correct anyway.

Examples of Class A.2 Checks:
Considering the following set of instances:

```
#2 = PERSON('Paul Maier',(30,5,62),.blonde.,(#9));
#2 = PERSON('Paul Maier',(30,5,60),.brown. ,(#5));
#3 = PERSON('Anna Maier',(30,4,60),.blonde.,(#5));
#4 = PERSON('Anna Maier',(30,3,60),.blonde.,(#5));
#5 = PERSON('Paul Maier',(30,3,81),.blonde.,());
#9 = GARAGE(30,4);
```

It is not allowed that an instance identifier is defined twice. In the above case this is #2. The first #2 Instance does not match the EXPRESS specification in addition since the referred instance #9 is of type "garage" but "person" is required. Furthermore, the three references of instance #2, #3, and #4 to instance #5 are in conflict to the EXPRESS specification that a maximum of two parents is allowed.

4.2 Class B: Simple Rule Checks

We can distinguish uniqueness, simple domain and global rule checks. Domain and global rules are called simple if no attribute qualifier is used in the rule body or in the function/procedure body of a called function/procedure. The same holds for used derived attributes. Otherwise rules are called complex. To restrict checks to simple checks only means to avoid the evaluation of cross references to other entities.

4.2.1 Uniqueness Checks

Uniqueness checks check a set of local uniqueness constraints.

Concerning the above EXPRESS model there is one uniqueness specification in line 14. It defines the tupel "(name,birthdate)" as unique within all instances of entity person and all related subtypes.

4.2.2 Simple Domain Rule Checks

The set of independent domain rules of an entity or domain rules over the type of an instance element have to be checked. Each simple domain rule check applies the check on all instances of one entity type.

Concerning the above EXPRESS model there are two simple domain rules specified. The specification in line 3 constraints the first component of array "date". The specification of "w1" in line 25 and 26 constraint the local attributes of entity female.

4.2.3 Simple Global Rule Checks

These checks perform similar to the simple domain rule checks but each check may range over all instances of a set of entity types.

Concerning the above EXPRESS model there is no simple global rule defined. The global rule "married" is a complex global rule since within this rule cross references between females and males have to be checked ("tf.husband.wife").

Examples of Class B Checks:

Considering the following set of instances:

```
#1 = PERSON('Paul Maier',(30,5,60),.blonde.,(#2));
#2 = PERSON('Paul Maier',(30,5,60),.brown. ,(#5));
#3 = FEMALE('Anna Maier',(30,5,61),.brown. ,(), $, 'Mayer');
#4 = MALE ('Paul Maier',(35,15,61),.brown. ,(), $);
```

Instance #1 and instance #2 do not fulfil the uniqueness constraint "un1" line 14 of the above EXPRESS specification since they both have the same name and the same birthdate. Instance #3 is in conflict with rule "w1" since a maiden_name is specified but no husband. Instance #4 conflicts the domain rule of type date since the first component is larger than 31.

4.3 Class C: Complex Rule Checks

We can distinguish two kinds of complex checks: *Complex Domain Rule Checks* and *Complex Global Rule Checks.* Since these checks concern attribute qualifiers this check can range over a quite complex set of instances. These references by qualified attributes first have to be resolved by being replaced by their actual value(s) before the basic checks can be applied. The basic checks can then be performed according to the simple rule checks. Concerning the above EXPRESS specification the rule "married" is a complex global rule since the attribute qualifier "tf.husband.wife" ranges over instances of type male and type female.

Examples of Class C Checks:

Considering the following set of instances:

```
#2 = FEMALE('Anna Maier',(30,5,61),.brown. ,(), #5, 'Mayer');
#3 = FEMALE('Anna Maier',(31,5,61),.brown. ,(), #5, 'Mayer');
#4 = MALE  ('Paul Maier',(30,12,61),.brown. ,(),#2);
#5 = MALE  ('Paul Maier',(31,12,61),.brown. ,(),#3);
```

Due to the given EXPRESS specification the instances #2 and #3 conflict the constraint specified by the rule "married". This is because

```
SIZEOF( QUERY(#3 :<>: #2)) = 1
```

since #2 refers to #5 but #5 does not refer to #2 accordingly.

5 Implementation Issues

5.1 C++

NIST has established the National PDES Testbed (NPT) for STEP validation testing in 1989 [Clark90]. The recent release of this testbed provides a set of integrated tools, namely the Data Probe [Sauder93]. The Data Probe performs a simple validation by validating attribute values against the attribute's type and checks attribute values to be optional and required, i.e. very basic parts of Class A.1 simple type checks.
The toolset includes an EXPRESS compiler which generates C++ code and additional structures in order to support some dedicated checks. The EXPRESS model can be found hard coded in the generated C++ program. The generated code is linkable to existing libraries (Working Forms). The resulting executable provides an X11 based browser which allows to read and store ISO10303-21 instances. When browsing through the instances only those instances are accepted whose values are compliant with the EXPRESS attribute declaration.
The C++ generator only translates the basic entity structures and attributes to C++. This means that all executable specifications (functions, procedures) are ignored. The same holds for all rules but uniqueness constraints. This check as well as the optional check is coded in the attribute descriptor of the corresponding EXPRESS attribute. At present the C++ generator ignores different super/subtype constraints. This means, AND, ANDOR, and ONEOF constraints are equally mapped to C++. The basic concept of this implementation is that schemas, entities, and types are registered as global objects. The corresponding attributes are stored as local nodes of these global objects. The super/subtype relationship is represented by a local sub- or supertype object which is linked to the corresponding global entity objects. The inheritance between EXPRESS entities is realised by a direct translation of sub-supertype relations to a C++ class hierarchy.
In the remainder of this paragraph we give an overview on how the generated C++ code is organised when considering the following EXPRESS schema.

```
SCHEMA my_1stExample;
   TYPE date = ARRAY[1:3]  OF INTEGER;
   END_TYPE;


   ENTITY person SUPERTYPE OF (ONEOF(male,female));
      name: STRING;
      birth_date: date;
      children: SET [0:?] of person;
   END_ENTITY;

   ENTITY male SUBTYPE OF (person);
      wife: OPTIONAL female;
   END_ENTITY;


   ENTITY female SUBTYPE OF (person);
      maiden_name: OPTIONAL STRING;
```

```
  END_ENTITY;
END_SCHEMA;
```

Feeding the above schema to the C++ generator `fedex_plus`. It generates 5 files: schema.cc, schema.h, s_MY_1STEXAMPLE.init.cc, s_MY_1STEXAMPLE.h, and s_MY_1STEXAMPLE.cc. The first two files are equal for each schema and contain methods and data structures in order to initialize the validation system. The basic structure of the generated files are sketched below.

s_MY_1STEXAMPLE.h:

First for each type, entity, and attribute EXPRESS declaration one pointer variable is defined. The objects are created by a function in file s_MY_1STEX AM PLE.init.cc. One C++ class is declared for each entity. The EXPRESS super/subtype hierarchy is mapped to a C++ class hierarchy. The following example lists the class of entity person. The basic methods which return the value of entity attributes are defined to be protected within each entity class.

```
...
10  //      ************** TYPES
11  TypeDescriptor   *MY_1STEXAMPLEt_DATE;
12  //      ************** ENTITIES
13  EntityDescriptor *MY_1STEXAMPLEe_PERSON;
14  AttrDescriptor   *a_0NAME;
15  AttrDescriptor   *a_1BIRTH_DATE;
16  AttrDescriptor   *a_2CHILDREN;
17  class s_Person : public STEPentity {
18    protected:
...
34  };
...
40  class s_Female  :    public s_Person {
41    protected:
...
51  };
...
56  class s_Male  :    public s_Person {
57    protected:
...
67  };
...
```

s_MY_1STEXAMPLE.init.cc:

Concerning the automatically generated file we have slightly changed the order of the statements in the following listing in order to outline the basic concepts only.
The generated file basically contains a function to initialise the validation system. Within this function first an object for the schema and objects for the types and entities are created. These objects are registered in a second step. Thereafter the objects for the attributes of each entity are created. Optional and unique attributes are denoted by a particular coding of the parameters. These objects are stored as explicit attribute types in terms of nodes. Finally, the entities are connected by nodes which represent the super/subtype relationship.

```
s_MY_1STEXAMPLEInit (Registry& reg)
{
   z_MY_1STEXAMPLE       = new SchemaDescriptor("My_1stexample");
   MY_1STEXAMPLEt_DATE   = new TypeDescriptor (
                           "Date",                    // Name
                           AGGREGATE_TYPE,            // FundamentalType
                           "ARRAY [1:3] OF INTEGER");// Description
   MY_1STEXAMPLEe_PERSON = new EntityDescriptor("Person",z_MY_1STEXAMPLE,F);
   ...
   reg.AddSchema (*z_MY_1STEXAMPLE);
   reg.AddType   (*MY_1STEXAMPLEt_DATE);
   reg.AddEntity (*MY_1STEXAMPLEe_PERSON);
   ...
   a_0NAME       = new AttrDescriptor("name",t_STRING_TYPE,F,F,F,*MY_1STEXAMPLEe_PERSON);
   a_1BIRTH_DATE = new AttrDescriptor("birth_date",MY_1STEXAMPLEt_DATE,F,F,F,
                                      *MY_1STEXAMPLEe_PERSON);
   t_0           = new TypeDescriptor;
   t_0->FundamentalType(AGGREGATE_TYPE);
   t_0->Description("SET [0:?] OF Person");
   t_0->ReferentEntity(MY_1STEXAMPLEe_PERSON);
   a_2CHILDREN   = new AttrDescriptor("children",t_0,F,F,F,*MY_1STEXAMPLEe_PERSON);

   MY_1STEXAMPLEe_PERSON->ExplicitAttr().AddNode(a_0NAME);
   MY_1STEXAMPLEe_PERSON->ExplicitAttr().AddNode(a_1BIRTH_DATE);
   MY_1STEXAMPLEe_PERSON->ExplicitAttr().AddNode(a_2CHILDREN);
   MY_1STEXAMPLEe_PERSON->NewSTEPentity = (Creator) create_s_Person;
   ...
   MY_1STEXAMPLEe_MALE->Supertypes().AddNode(MY_1STEXAMPLEe_PERSON);
   MY_1STEXAMPLEe_PERSON->Subtypes().AddNode(MY_1STEXAMPLEe_MALE);
   ...
}
```

s_MY_1STEXAMPLE.cc:

This file contains the implementation of the constructor, the empty destructor, and methods which return the attributes of the particular entity. Since the order of attribute definitions within the EXPRESS specification is relevant, each attribute is pushed onto a local stack.

```
...
8  s_Person::s_Person( )
9  {
10         STEPattribute * a;
11         EntityDescriptor = MY_1STEXAMPLEe_PERSON;
12         extern AttrDescriptor *a_0NAME;
13         a = new STEPattribute (*a_0NAME, &_name);
14         a -> set_null ();
15         attributes.push (*a);
16         extern AttrDescriptor *a_1BIRTH_DATE;
17         a = new STEPattribute (*a_1BIRTH_DATE, &_birth_date);
18         a -> set_null ();
19         attributes.push (*a);
20         extern AttrDescriptor *a_2CHILDREN;
21         a = new STEPattribute (*a_2CHILDREN, &_children);
22         a -> set_null ();
23         attributes.push (*a);
24 }
25 s_Person::~s_Person () {  }
26 s_String
27   s_Person::name()
28         { return (s_String ) _name; }
29 void
30   s_Person::name (s_String  x)
31         { _name = x; }
32 IntAggregate&
```

```
33    s_Person::birth_date()
34          { return (IntAggregate&) _birth_date; }
35  void
36    s_Person::birth_date (IntAggregate& x)
37          { _birth_date = x; }
38  EntityAggregate&
39    s_Person::children()
40          { return (EntityAggregate&) _children; }
42    s_Person::children (EntityAggregate& x)
43          { _children = x; }
...
```

5.2 Alternative Approaches

5.2.1 Prolog

We started implementing a simple checker using a Prolog environment since Prolog provides an easily implementable framework for optional/required, type, and cardinality checks, i.e. some Class A simple type checks (see [LehMue91, LehMue92]). The detailed translation rules from STEP files and parts of EXPRESS models to Prolog can be found in [LehMue92]. A demonstration prototype based on EXPRESS N496 has been implemented. In this approach we also have shown that it is possible to translate executable EXPRESS specifications (functions, procedures, rules) to Prolog. But this needs further investigation since this is not a 1:1 mapping from EXPRESS to Prolog. Additionally to this in the following paragraph we sketch that nearly the same rules can be applied when mapping STEP files and the same parts of EXPRESS models to a strongly typed, functional language such as SML (Standard Meta Language).

5.2.2 SML

ML has been developed in the 1980s for the use as a meta language in a verification system. A language standard has emerged, namely SML [MiToHa90]. ML is the most prominent of a new group of strongly typed, functional languages. Meanwhile, SML is a quite popular language. It is educated at a majority of universities. For further introduction the reader is referred to [Paulson91]. The following examples presume the Standard ML of New Jersey, Version 75. This is an interpreter in the public domain which has been developed at AT&T Labatories, NJ, USA.

The big advantage using SML is that simple type checks can easily be performed by the use of the interpreter since SML is strongly typed. The following table depicts that the same simple 1:1 translations rules which have been developed for Prolog in [LehMue92] can be applied to SML as well. First, we give an ad-hoc mapping from EXPRESS to SML. Thereafter, we sketch how a more advanced mapping to SML can be realised.

	STEP	Prolog	SML
Integer	1234	1234	1234
Real	9.432	real(9.432)	9.432
String	'ABC'	'ABC'	"ABC"
Boolean	.F., .T.	e_F_e, e_T_e	False, True
Logical	.F., .T., .U.	e_F_e, e_T_e, e_U_e	False, True, Unknown
Binary	"2EC"	binary('2EC')	binary("2EC")
Enumerations	.BLONDE.	e_BLONDE_e	BLONDE
Optional value	$	_	OPTIONAL
Instance Id	#3	id(3)	id(3)
Name	aBCd	abcd	abcd
List	(a1, A2, a3)	[a1, a2, a3]	[a1, a2, a3]
Delimiter	;	.	;

An application of the above rules is given by the following example.

STEP	SML
#4=PERSON('Phil',(25,5,89),(#5,#6));	person("Phil",[25,5,89], [id(5), id(6)]);

Concerning the translation rules the same holds for the basic EXPRESS types and identifiers which is outlined by the following table.

	EXPRESS	Prolog	SML
Base types	string	string	string
	integer	integer	int
	real	real	real
	boolean	boolean	bool
	logical	logical	SML datatype
	binary	binary	SML datatype
Defined types	myTypeIdentifier	myTypeIdentifier	myTypeIdentifier

The following simple example sketches how SML can be used for the checking of simple types of an EXPRESS model.

EXPRESS	SML
ENTITY person;	datatype identifier = id of int and
name : STRING;	persontype = person of string *
birth_date: ARRAY [1:3] OF INTEGER;	int list *
children : SET [0:?] OF PERSON;	identifier list;
END_ENTITY;	

Thus, entering the instance "`person("Phil",[25,5,89], [id (5), id(6)]);`" is accepted by SML but entering the instance "`person(11,[25,5,89], []);`" results in the following error message.

```
Error: operator and operand don't agree (tycon mismatch)
       operator domain: string * int list * identifier list
       operand:         int    * int list * identifier list
```

This indicates that the first field of the operand contains an integer type but a string type is required.

The above translation is a first approach on how to make use of a strongly typed functional language for STEP file simple type checking. We think SML is also a potential language for the efficient implementation of more advanced checks. This is because SML provides efficient means to implement operations on sets and lists which are fundamental for domain as well as global EXPRESS rules. But this surely needs further investigation since this requires the translation of EXPRESS models to more advances SML programs.
We sketch the potential use of SML by giving a final example of a more advanced translation. Consider the following EXPRESS model.

```
TYPE hair_type = ENUMERATION OF (blonde,brown); END_TYPE;

ENTITY person;
    name      : STRING (3) FIXED;
    hair      : hair_type;
    children  : SET [0:?] OF person;
END_ENTITY;
```

The above enumeration type can be directly mapped to an SML datatype. Each entity can be mapped into an SML function which returns a record that covers the values of each instance (`returninst`). The type check can be implemented by specifying a type for each parameter of this function. The first parameter can be used to store the instance identifier. The consistency constraints can be implemented in the declaration part of the SML expression. In the following example this is to check whether the name is exactly 3 characters long. In the case that this check fails an exception can be raised (`stringerror`).

```
exception stringerror;

datatype hair_type  = blonde | brown;
datatype identifier = id of int;


fun person(pid       : identifier,
           pname     : string,
           phair     : hair_type,
           pchildren : identifier list) =
let
       val bo         = if size pname = 3
                        then "STRING OK"
                        else raise stringerror
           and
           returninst = { instid   = pid,
                          name     = pname,
                          hair     = phair,
                          children = pchildren}
in
   returninst
end;
```

Using the above mapping from EXPRESS to SML results in a slightly different translation from STEP files to SML. The following sample is a valid SML instance for STEP corresponding to the above SML program. The returned record of instance 6 is finally stored in i'6.

```
val i'6 = person(id(6),"PMU",brown,[id(11),id(12)]);
```

5.3 Conclusion

Translating EXPRESS to a programming language which has to be done when implementing a validation suite raises several problems. First, the scope of EXPRESS variables and types are quite different from the scope of all other known programming languages. This makes the generated programs not easy to read if a full EXPRESS description including executable parts is translated into a programming language. Second, the EXPRESS super/subtype structure can not directly be mapped to the class structure of an object-oriented language since the inheritance defined in EXPRESS is different from that of object-oriented programming languages. Finally, there can be a problem for real industrial applications when processing an immense amount of data. An upper limit on the number of instances is given by ISO10303-21, namely $10^9 - 1$. Thus, approaching this limit means to process over 1 Gigabyte of product data. Processing these immense amount of data in order to validate them, results in severe problems in run-time and in the required main memory. Present implementations of STEP validation tools do not concern the processing of these huge file sizes. The critical checks regarding to this problem are the checks which require cross-reference checks over a huge set of instances. In this context the parallel processing of subchecks needs further investigation.

Concerning the validation of medium sized STEP files only, the most elegant and efficient way to automatically generate a conformance checker seems to be the use of SML. Unfortunately, SML is a language which has not been succeeded to be frequently used in industry so far and thus available SML compilers/interpreters cannot be presumed though the SML/NJ system is public. Taking only these industrial constraints into account, we recommend a programming language which is widely available, i.e. C or C++.

References

[ISO10303-11] ISO/DIS 10303-11, EXPRESS Language Reference Manual, ISO TC184/SC4, August 31, 1992.

[ISO10303-21] ISO/DIS 10303-21, Product Data Representation and Exchange - Part 21: Clear Text Encoding of the Exchange Structure, ISO TC184/SC4/WG5, January 13, 1993.

[Clark90] Clark, S.N., An Introduction to The NIST PDES Toolkit, NISTIR 4336, NIST, Gaithersburg, MD, May 1990.

[LehMue91] Lehrenfeld, G., Mueller, W., Validation of EXPRESS Models Using Prolog, EXPRESS User's Group Conference, Houston, TX, October 1991.

[LehMue92] Lehrenfeld, G., Mueller, W., Translation of EXPRESS to a Logical Programming Language, ECIP2 Deliverable R-C-1-6-2-39-A, Cadlab, Paderborn, August 24, 1992.

[MiToHa90] Milner, R., Tofte, M., Harper, R., The Definition of Standard ML, The MIT Press, Cambridge, MA, 1990.

[Sauder93] Sauder, D.A. , Data Probe User's Guide, NISTIR 5141, NIST, Gaithersburg, MD, March 1993.

[Paulson91] Paulson, L.C., ML for the Working Programmer,Cambridge University Press, Cambridge, MA, 1991.

[Wilson92] Wilson, Peter R., EXPRESS-I Language Reference Manual, ISO WD 10303-12: 1992 (E), Reference Documents, Second Annual EXPRESS User's Group EUG'92, Westin Galleria, Dallas, TX, October 17-18, 1992.